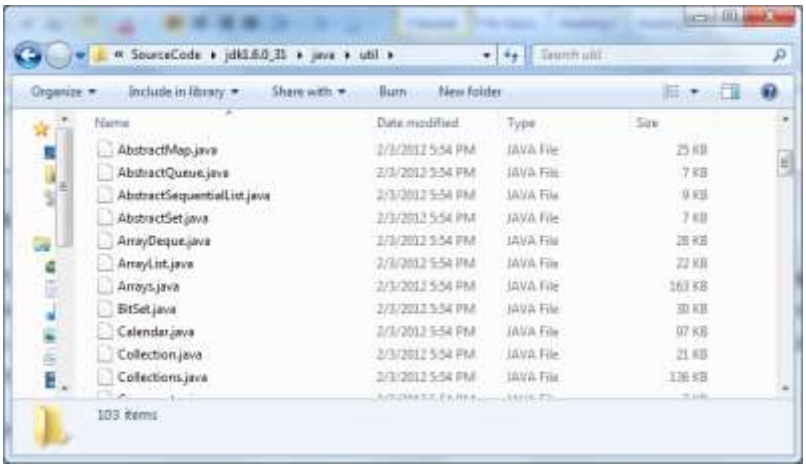
What are packages?

Packages are directory-based structures that group some related source files together. For example, the `java.util` package in JDK groups all interfaces and classes in the Collections Framework such as `Collection`, `List`, `ArrayList`, `Map`, `Set`, `HashMap`, etc. The following screenshot shows a part of this package in Windows Explorer program:



To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

Why are Packages?

- **Using packages allows us to avoid naming collisions:**

  Imagine a situation in which two programmers write two classes that have same name, let's say - `Dog`. If these two classes are used in a program, how to identify which `Dog` is which? So packages come to rescue: the 1st programmer puts his `Dog` class under a package called `john.animal`; and the 2nd programmer puts his `Dog` under `tom.pets` package.

  When accessing these classes, we use their fully qualified names: `john.animal.Dog` and `tom.pets.Dog`.

  In JDK, you can find some classes that have same name but in different packages, e.g. `java.util.Date` and `java.sql.Date`.

- **Packages facilitate the encapsulation feature in Java:**

  Think packages like directories that isolate some classes from classes outside. In Java, we can use access modifiers to restrict access to some classes in a certain package. For example, the default access modifier (when no explicit access modifier is used) makes a class accessible only by others in the same package. Whereas the `public` access modifier makes a class visible and accessible by all classes regardless of packages.

- **Packages allow us to group some related classes together for better organization and management:**

For example, the `java.util` package contains only interfaces and classes which belong to the Collections framework; The `javax.swing` package contains only interfaces and classes which are related to Graphical User Interface (GUI) components.

In practice, we tend to organize a complex application into packages for better organization and management, for example:

- `com.mycompany.model`: contains entity classes.
- `com.mycompany.business`: contain business classes.
- `com.mycompany.gui`: contains GUI classes.

etc.

Packages

- A package does not mean only predefined classes; a package may contain user defined classes also.
- Packages can be compressed into JAR files for fast traversal in a network or to download from Internet
- With a single import statement, all the classes and interfaces can be obtained into our program
- Avoids namespace problems. Two classes of the same name cannot be put in the same package but can be placed in two different packages
- Access between the classes can be controlled. Using packages, restrictions can be imposed on the access of other package classes. Access specifiers work on package boundaries (between the classes of other packages)
- Packages and sub-packages are the easiest way to organize the classes

Importing All/Single Class

- Packages have an advantage over header files of C-lang. A package allows importing a single class also instead of importing all. C-lang does not have this ease of getting one function from a header file

```
import java.net.*;                  // imports all the classes and interfaces
import java.awt.event.*;            // imports all the classes and interfaces

import java.net.Socket;             // imports only Socket class
import java.awt.event.WindowEvent; // imports only WindowEvent class
```

- Note: While importing a single class, asterisk (*) should not be used

Fully Qualified Class Name

- By placing the same class in two different packages, which Java permits, namespace problems can be solved. Namespace is the area of execution of a program in RAM
- The Date class exists in two packages –java.util and java.sql.
- Importing these two packages in a program gives ambiguity problem to the compiler. In the following program compiler gets ambiguity problem and is solved with fully-qualified name
- Note: When fully qualified name is included, importing the package is not necessary

Program N0.

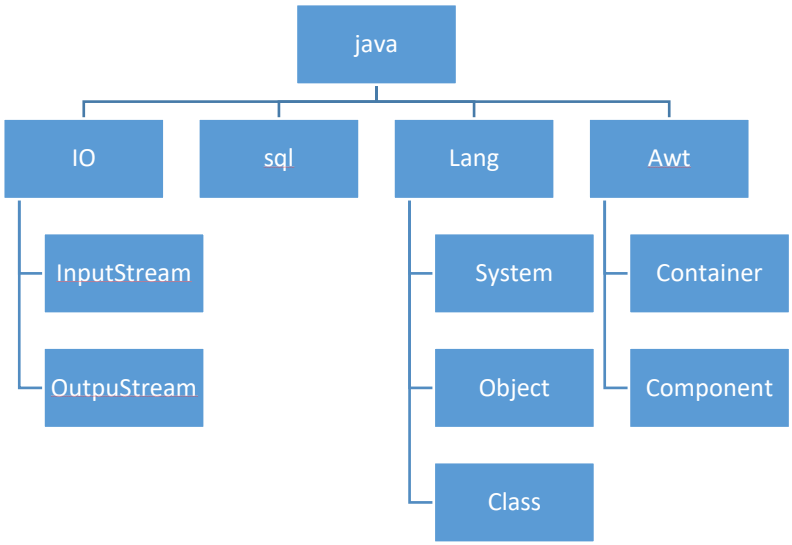
Fully Qualified names

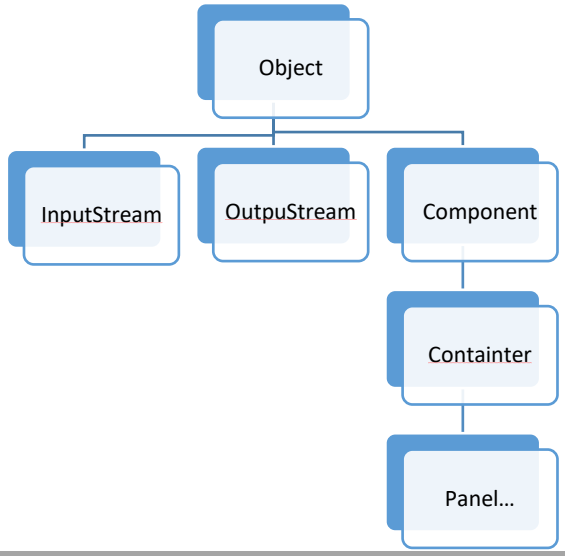
```
class PackageDemo1
{
        public static void main(String args[])
```

```
        {
                System.out.println("\n\t Using util Package....");
                java.util.Date d1 = new java.util.Date();

                System.out.println("\n\t Using sql Package....");

                java.sql.Date d2 = new java.sql.Date(1);

        }
}
```

Java Class Libraries – Java API

- All the classes and interfaces that come with the installation of JDK are put together are known as Java API (Application Programming Interface)
- All the Java API packages are prefixed with java or javax.





Package Name	Example Classes	Functionality (Purpose)
java.lang	System, String, Object, Thread, Exception etc.	These classes are indispensable for every Java program. For this reason, even if this package is not imported, JVM automatically imports
java.util	Scanner	These are called as utility (service) classes and are used very frequently in coding.
java.io	FileInputStream, FileOutputStream, FileReader, FileWriter, RandomAccessFile, BufferedReader, BufferedWriter etc.	These classes are used in all I/O operations including keyboard input.
java.net	URL, ServerSocket, Socket, DatagramPacket, DatagramSocket etc.	Useful for writing socket programming (LAN communication).
java.applet	AppletContext, Applet, AudioStub, AudioClip etc	Required for developing applets that participate on client-side in Internet (Web) programming.

java.awt	Button, Choice, TextField, Frame, List, Checkbox etc.	Essential for developing GUI applications.
java.awt.event	MouseListener, ActionListener, ActionEvent, WindowAdapter etc.	Without these classes, it is impossible to handle events generated by GUI components

Program N0.

Date Class

```
import java.util.Date;

public class PackageDemo_154
{
     public static void main(String args[])
     {
         Date today = new Date();
      System.out.println("Today Particulars: " + today);
      System.out.println("Hours Part of today: " + today.getHours());
      System.out.println("Minutes part of today: " + today.getMinutes());
      System.out.println("Seconds part of today: " + today.getSeconds());
      System.out.println("Month part of today: " + today.getMonth());
      System.out.println("Date part of today: "  + today.getDate());
      System.out.println("Day part of today: " + today.getDay());
      System.out.println("Year part of today: " + today.getYear());
      System.out.println("Milliseconds representation of today (from epoch
time 01-01-1900: " + today.getTime());

     }
}
```

Math Class

- The java.lang.Math class includes many methods with which simple arithmetic operations can be done like finding the square root, rounding, trigonometric and logarithm functions.
- As these methods are defined as static, they can be used directly by the programmer without creating an object
- **Note** : Java.lang.Math class is final

Program N0.

Math class

```
import java.lang.Math;

public class MathDemo
{
       public static void main(String[] args)
       {
              System.out.println("\n\t Square Root of  100 :" +
                                   Math.sqrt(100));
              System.out.println("\n\t Power of 2^3= " + Math.pow(2, 3));
       }
```

```
}
```

Static Imports

- Many methods of classes like Math and Character are static.
- If the variables and methods of these classes are used very often, all are must be prefixed with Math or Character
- Normal import statement imports all classes and interfaces of a package. But static import imports only static members of a single class.
- It avoids usage of the class name multiple times in coding. The above statement avoids Math class name to prefix every variable or method used
- Ex. ceil(), floor(), pow are the static methods of Math class and PI is a static variable. All these are used without using prefix Math name.

Program N0.

Static import

```
import static java.lang.Math.*;

public class PackageDemo_155
{
     public static void main(String args[])
     {
             int radius = 10;
              System.out.println("Perimeter: " + ceil(2 * PI * radius));
              System.out.println("Area: " + floor(PI * pow(radius, 2)));
             System.out.println("Raised 3 times: " + pow(radius, 3));

     }
}
```

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a `package` statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note: If you put multiple types in a single source file, only one can be `public`, and it must have the same name as the source file. For example, you can define `public class Circle` in the file `Circle.java`, define `public interface Draggable` in the file `Draggable.java`, define `public enum Day` in the file `Day.java`, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be package private.

- **Note:**

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Creating Own Package

- Java permits to create our own packages and use in programming

Steps of creating User Defined Packages Java and using them.

1. Create a package with a .class file
2. set the classpath from the directory from which you would like to access. It may be in a different drive and directory. Let us call it as a target directory.
3. Write a program and use the file from the package.

Let us create a package called forest and place a class called Tiger in it. Access the package from a different drive and directory.

1st Step: Create a package (forest) and place Tiger.class in it.

Let us assume C:\snr is the current directory where we would like to create the package.

```
C:\snr > notepad Tiger.java
```

```
package forest;
import java.util.*;
public class Tiger
{
  public void getDetails(String nickName, int weight)
  {
    System.out.println("Tiger nick name is " + nickName);
    System.out.println("Tiger weight is " + weight);
  }
}
```

When the code is ready, the next job is compilation. We must compile with package notation. Package notation uses –d compiler option as follows.

```
C:\snr > javac -d . Tiger.java
```

The **–d compiler** option creates a new folder called forest and places the Tiger.class in it. The dot (.) is an operating system's environment variable that indicates the current directory. It is an instruction to the OS to create a directory called forest and place the Tiger.class in it.

2nd step: Set the classpath from the target directory.

Let us assume `D:\ javaprg` is the target directory. Let us access Tiger.class in forest package from here.

From the target directory set the classpath following way.

```
D:\javaprg> set classpath=C:\snr;%classpath%;
```

`classpath` is another environment variable which gives the address of the forest directory to the OS. **`%classpath%`** informs the OS to append the already existing **`classpath`** to the current **`classpath`** that is right now set.

3rd Step: Now finally, write a program from the target directory D:/sumathi and access the package.

```
D:\javaprg> notepad Animal.java
```

The above statement creates a file called Animal.java and write the code in it, say, as follows

```
import forest.Tiger;
public class Animal
{
  public static void main(String args[])
  {
    Tiger t1 = new Tiger ();
    t1.getDetails("Everest", 50);
  }
}
```

The compilation and execution is as usual as follows.

```
D:\sumathi> javac Animal.java
D:\sumathi> java Animal
```

Program N0.

Package Student

```
package javaprg;

public class Student
{
	public int roll;
	public String name;
	public float avg;

	public Student()
	{
		System.out.println("\n\t Welcome in Student Class");
	}

	public Student(int r,String nm, float a)
	{
		roll = r;
		name= nm;
		avg = a;
	}

	public void showStudent()
	{
		System.out.println("\n\t Roll Number = " + roll);
```

```
                System.out.println("\n\t Name = "+ name);
                System.out.println("\n\t Avg Marks = " + avg);
        }
}
```

Order of Package Statement

```
package student;
import java.util.*;
public class Student ….
```

- package is a keyword of Java followed by the package name. Just writing the package statement followed by the name creates a new package
- If exists, the package statement must be first one in the program
- If exists, the import statement must be the second one
- Our class declaration is the third

Program NO.

Package Example

```
import javaprg.*;

class StudentDemo1
{
        public static void main(String args[])
        {
                Student s1 = new Student();
                Student s2 = new Student(101,"Niraja",45.65f);
                s2.showStudent();
        }
}
```

Mastering Javac Command

`javac.exe` is the Java compiler program. It compiles Java source files (`.java`) into bytecode class files (`.class`). The tool is located under `JDK_HOME\bin` directory. So make sure you included this directory in the `PATH` environment variable so it can be accessed anywhere in command line prompt.

Syntax of this command is:

```
javac [options] [source files]
```

Type javac -help to view compiler options, and type javac -version to know current version of the compiler. By default, the generated .class files are placed under the same directory as the source files.

1. Compiling a single source file

```
javac HelloWorld.java
```

2. Compiling multiple source files

1. Compile three source files at once, type:

```
javac Program1.java Program2.java Program3.java
```

2. Compile all source files whose filenames start with *Swing*

```
javac Swing*.java
```

3. Compile all source files:

```
javac *.java
```

3. Compiling a source file which has dependencies

It's very common that a Java program depends on one or more external libraries (jar files). Use the flag `-classpath` (or `-cp`) to tell the compiler where to look for external libraries (by default, the compiler is looking in bootstrap classpath and in CLASSPATH environment variable).

1. Compile a source file which depends on an external library:

```
javac -classpath mail.jar EmailSender.java
javac -cp mail.jar EmailSender.java
```

2. Compile a source file which depends on multiple libraries:

```
javac -cp lib1.jar; lib2.jar; lib3.jar MyProgram.java
javac -cp *; MyProgram.java
```

4. Specifying destination directory

Use the `-d directory` option to specify where the compiler puts the generated .class files. For example:

```
javac -d classes MyProgram.java
```

NOTE:

- The compiler will complain if the specified directory does not exist, and it won't create one.
- If the source file is under a package, the compiler will create package structure in the destination directory.

5. Specifying source path directory

We can tell the compiler where to search for the source files by using the `-sourcepath directory` option. For example

```
javac -sourcepath src MyProgram.java
```

6. Specifying source compatibility version

We can tell the compiler which Java version applied for the source file, by using the `-source release` option. For example

```
javac -source 1.5 MyProgram.java
```

That will tell the compiler using specific language features in Java 1.5 to compile the source file. The valid versions are: 1.3, 1.4, 1.5 (or 5), 1.6 (or 6) and 1.7 (or 7)

Mastering the jar tool

`jar` is the **Java archive** tool that packages (and compresses) a set of files into a single archive. The archive format is ZIP but the file name usually has .jar extension. This tool is used for creating, updating, extracting and viewing content of jar files.

The executable file of this tool can be located under the `JDK_HOME\bin` directory (jar.exe on Windows), so make sure you include this path in the PATH environment variable in order to run this tool anywhere from the command line prompt.

1. Creating normal jar file

A normal jar file is the non-executable one, such as a library jar file or an applet jar file. The following command put all files under the build\classes directory into a new jar file called SwingEmailSender.jar:

```
jar cfv SwingEmailSender.jar -C build\classes .
```

Note that there is a dot (.) at the end which denotes all files. The `c` option is to create, the `f` is to specify jar file name, the `v` is to generate verbose output, and the `-C` is to specify the directory containing the files to be added.

2. Including/Excluding manifest file

By default, the `jar` tool automatically creates a manifest file when generating a new jar file. If we don't want to have the manifest created, use the `M` option as in the following example:

```
jar cfvM SwingEmailSender.jar -C build\classes .
```

In case we want to manually add an external manifest file, use the `m` option as in the following example:

```
jar cfm SwingEmailSender.jar manifest.txt -C build\classes .
```

Here, content of the `manifest.txt` is copied to the generated manifest file inside the jar file.

Java Command

`java` is the Java application launcher tool which is used to execute programs written in Java programming language and compiled into bytecode class files. Its executable file can be found under `JDK_HOME\bin` directory (`java.exe` on Windows and `java` on Linux), so make sure you include this path to the PATH environment variable in order to invoke the program anywhere in command line prompt.

```
java [options] file.class [arguments...]

java [options] -jar file.jar [arguments... ]
```

The first syntax is for executing a class file, and the second one is for executing a JAR file.

Type `java -help` to consult the available options or browse Oracle's Java documentation for detailed description and explanation of the options. The arguments, if specified, will be passed into the running program.

NOTES:

- A Java class must have the `public static void main(String[] args)` method in order to be executed by the Java launcher.
- An executable JAR file must specify the startup class in by the `Main-Class` header in its manifest file

1. Running a Java program from a class file

Run a simple class:

If you have a source file called `MyProgram.java` and it is compiled into `MyProgram.class` file, type the following command:

```
java MyProgram
```

Run a class which is declared in a package:

If the class `MyProgram.java` is declared in the package `net.deesha`, change the working directory so that it is parent of the `net\deesha` directory, then type

```
java net.deesha.MyProgram
```

Run a class which has dependencies on jar files:

If we have a Java Mail-based program that depends on `mail.jar` library. Assuming the jar file is at the same directory as the class file, type:

```
java -cp mail.jar;. PlainTextEmailSender
```

NOTES: There must be a dot (.) after the semicolon

If the jar file is inside a directory called `lib`:

```
java -cp lib/mail.jar;. PlainTextEmailSender
```

If the program depends on more than one jar files:

```
java -cp mail.jar;anotherlib.jar;. MyProgram
```

We can use wildcard character to refer to all jar files:

```
java -cp *;. MyProgram
```

Or:

```
java -cp lib/*;. MyProgram
```

Passing arguments to the program:

The following example passes two arguments “code” and “java” into the MyProgram:

```
java MyProgram code java
```

If the argument has spaces, we must enclose it in double quotes, for example:

```
java MyProgram "Welcome" 2013
```

That will pass two arguments “welcome” and “2013”

2. Running a Java program from an executable jar file

Run a standalone jar file:

```
java -jar MyApp.jar
```

Here the `MyApp.jar` file must define the main class in the header `Main-Class` of its manifest file `MANIFEST.MF`. The header is usually created by the `jar` tool.

NOTE: if the jar file depends on other jar files, the reference jar files must be specified in the header `Class-Path` of the jar’s manifest file. The `-cp` option will be ignored when using `-jar` flag.

Passing arguments:

Pass two arguments “code” and “java” to the program:

```
java -jar MyApp.jar Welcome to java
```

3. Specifying splash screen

For Swing-based application, we can use the `-splash:imagePath` flag to show a splash screen at program's startup. For example:

```
java -splash:SplashScreen.png MyProgram
```

Here the image `SplashScreen.png` is loaded as splash screen at startup.

4. Setting system properties

We can use the `-Dproperty=value` option to specify a system property when running a program:

Specify a single property:

```
java -Dupload.dir=D:\Uploads MyProgram
```

if the property's value contains spaces, enclose it in double quotes:

```
java -Dupload.dir="D:\My Uploads" MyProgram
```

Specify multiple properties:

```
java -Dupload.dir=D:\Uploads -Ddownload.dir=D:\Downloads MyProgram
```

Override predefined property:

We can override the predefined system properties. For example, the following command overrides the system property `java.io.tmpdir`:

```
java -Djava.io.tmpdir=E:\Temp MyProgram
```

5. Specifying memory constraints

When launching a Java program, we can specify initial size and maximum size of the heap memory:

- `-Xms<size>:` specifies initial heap size
- `-Xmx<size>:` specifies maximum heap size.

The size is measured in bytes. It must be multiple of 1024 and is greater than 1MB for initial size and 2MB for maximum size. Append `k` or `K` to indicate kilobytes; `m` or `M` to indicate megabytes. For example, the following command launches a program with initial heap size 32MB and maximum heap size 1024MB:

```
java -Xms32M -Xmx1024M MyProgram
```

Working with Manifest Files

JAR files support a wide range of functionality, including electronic signing, version control, package sealing, and others

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

Default Manifest

When you create a JAR file, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it always has the pathname

```
META-INF/MANIFEST.MF
```

When you create a JAR file, the default manifest file simply contains the following:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
```

These lines show that a manifest's entries take the form of "header: value" pairs. The name of a header is separated from its value by a colon. The default manifest conforms to version 1.0 of the manifest specification and was created by the 1.7.0_06 version of the JDK.

The manifest can also contain information about the other files that are packaged in the archive. Exactly what file information should be recorded in the manifest depends on how you intend to use the JAR file. The default manifest makes no assumptions about what information it should record about other files

Setting an Application's Entry Point

If you have an application bundled in a JAR file, you need some way to indicate which class within the JAR file is your application's entry point. You provide this information with the `Main-Class` header in the manifest, which has the general form:

```
Main-Class: classname
```

The value `classname` is the name of the class that is your application's entry point.

Recall that the entry point is a class having a method with signature `public static void main(String[] args).`

After you have set the `Main-Class` header in the manifest, you then run the JAR file using the following form of the `java` command:

```
java -jar JAR-name
```

The `main` method of the class specified in the `Main-Class` header is executed.

An Example

We want to execute the `main` method in the class `MyClass` in the package `MyPackage` when we run the JAR file.

We first create a text file named `Manifest.txt` with the following contents:

```
Main-Class: MyPackage.MyClass
```

Warning: The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

We then create a JAR file named `MyJar.jar` by entering the following command:

```
jar cfm MyJar.jar Manifest.txt MyPackage/*.class
```

This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Main-Class: MyPackage.MyClass
```

When you run the JAR file with the following command, the `main` method of `MyClass` executes:

```
java -jar MyJar.jar
```

Setting an Entry Point with the JAR Tool

The 'e' flag (for 'entrypoint') creates or overrides the manifest's `Main-Class` attribute. It can be used while creating or updating a JAR file. Use it to specify the application entry point without editing or creating the manifest file.
For example, this command creates `app.jar` where the `Main-Class` attribute value in the manifest is set to `MyApp`:

```
jar cfe app.jar MyApp MyApp.class
```

You can directly invoke this application by running the following command:

```
java -jar app.jar
```

If the entrypoint class name is in a package it may use a '.' (dot) character as the delimiter. For example, if `Main.class` is in a package called `foo` the entry point can be specified in the following ways:

```
jar cfe Main.jar foo.Main foo/Main.class
```

Adding Classes to the JAR File's Classpath

You may need to reference classes in other JAR files from within a JAR file.

For example, in a typical situation an applet is bundled in a JAR file whose manifest references a different JAR file (or several different JAR files) that serves as utilities for the purposes of that applet.

You specify classes to include in the `Class-Path` header field in the manifest file of an applet or application. The `Class-Path` header takes the following form:

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

By using the `Class-Path` header in the manifest, you can avoid having to specify a long `-classpath` flag when invoking Java to run the your application.

Note: The `Class-Path` header points to classes or JAR files on the local network, not JAR files within the JAR file or classes accessible over Internet protocols. To load classes in JAR files within a JAR file into the class path, you must write custom code to load those classes. For example, if `MyJar.jar` contains another JAR file called `MyUtils.jar`, you cannot use the `Class-Path` header in `MyJar.jar's` manifest to load classes in `MyUtils.jar` into the class path.

An Example

We want to load classes in `MyUtils.jar` into the class path for use in `MyJar.jar`. These two JAR files are in the same directory.

We first create a text file named `Manifest.txt` with the following contents:

```
Class-Path: MyUtils.jar
```

Warning: The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

We then create a JAR file named `MyJar.jar` by entering the following command:

```
jar cfm MyJar.jar Manifest.txt MyPackage/*.class
```

This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Class-Path: MyUtils.jar
Created-By: 1.7.0_06 (Oracle Corporation)
```

The classes in `MyUtils.jar` are now loaded into the class path when you run `MyJar.jar`.