Java File Structure

Introduction

1. A java Program can contain any no. Of classes but at most one class can be declared as public.
2. "If there is a public class the name of the Program and name of the public class must be matched otherwise we will get compile time error".
3. If there is no public class then any name we can give for java source file.

```
class A{}
class B{}
class C{}
```

Case 1:

- If there is no public class then we can use any name for java source file there are no restrictions.

Case 2:

- If class B declared as public then the name of the Program should be B.java otherwise we will get compile time error saying "class B is public, should be declared in a file named B.java".

Case 3:

- If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".
- It is highly recommended to take only one class for source file and name of the Program (file) must be same as class name. This approach improves readability and understandability of the code.

- We can compile a java Program but not java class. In that Program for every class one dot class file will be created.
- We can run a java class but not java source file. Whenever we are trying to run a class the corresponding class main method will be executed.
- If the class won't contain main method then we will get runtime exception saying "NoSuchMethodError: main".
- If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying "NoClassDefFoundError: Sample".

Import Statements

Types of Import Statements:

There are 2 types of import statements.

1) Explicit class import

Import java.util.Scanner

- This type of import is highly recommended to use because it improves readability of the code.

2) Implicit class import.

import java.util.;*

- It is not recommended to use because it reduces readability of the code.

Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not require to use fully qualified name.

- We may get the ambiguity problem because it may be available in multiple packages.
- While resolving class names compiler will always gives the importance in the following order.

  1. Explicit class import
  2. Classes present in current working directory.
  3. Implicit class import.
- Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes. Explicit import is required.
- In any java Program the following 2 packages are not require to import.

  1. java.lang package
  2. default package(current working directory)

"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

- In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
- But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or "load-on -demand" or "load-on-fly".

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

```
System.out.println(Math.sqrt(4));
System.out.println(Math.max(10,20));
System.out.println(Math.random());
```

With static import:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
System.out.println(sqrt(4));
System.out.println(max(10,20));
System.out.println(random());
```

Packages In Java

Packages

- A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations, annotations and sub-packages) providing access protection and name space management.
- Packages are use to control access of classes, interface, enumeration etc. and avoid namespace collision.
- There can not be two classes with same name in a same Package
- But two packages can have a class with same name.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

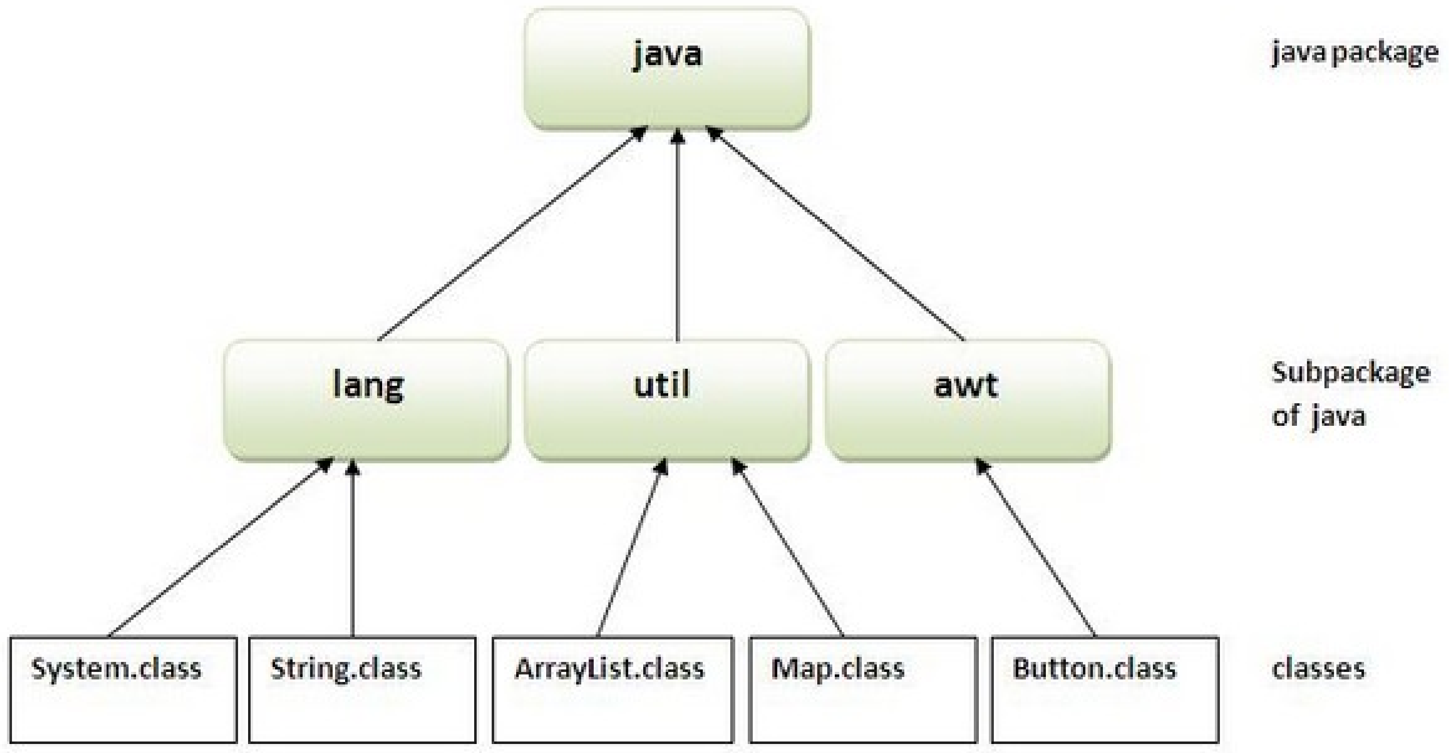
Packages

- Package names are dot separated, e.g., java.lang.
- Package names have a correspondence with the directory structure.
- Exact Name of the class is identified by its package structure.
- << Fully Qualified Name>>
- java.lang.String ;
- java.util.Arrays;
- java.io.BufferedReader ;
- java.util.Date

Advantage of Java Package

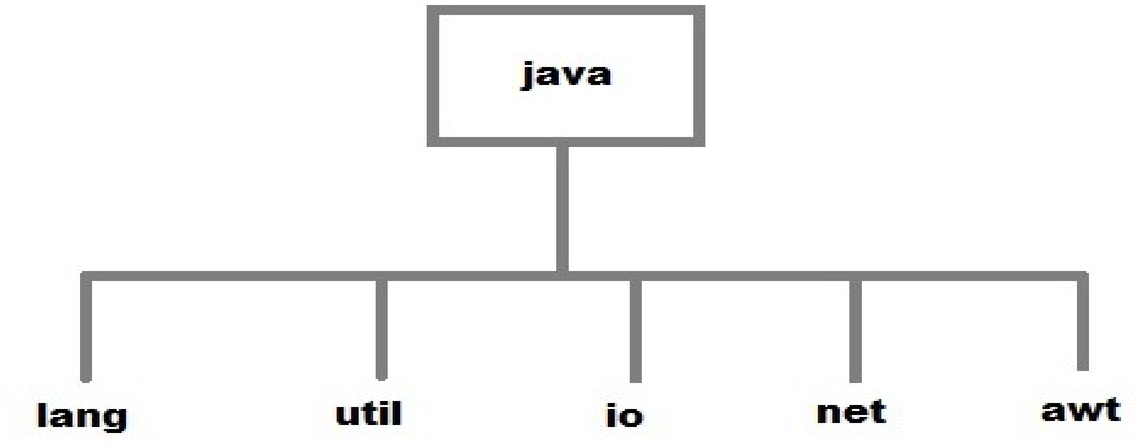
- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Package Hierarchy



Con’t

- Package are categorized into two forms:
  - **Built-in** java package : java.lang, java.util etc.



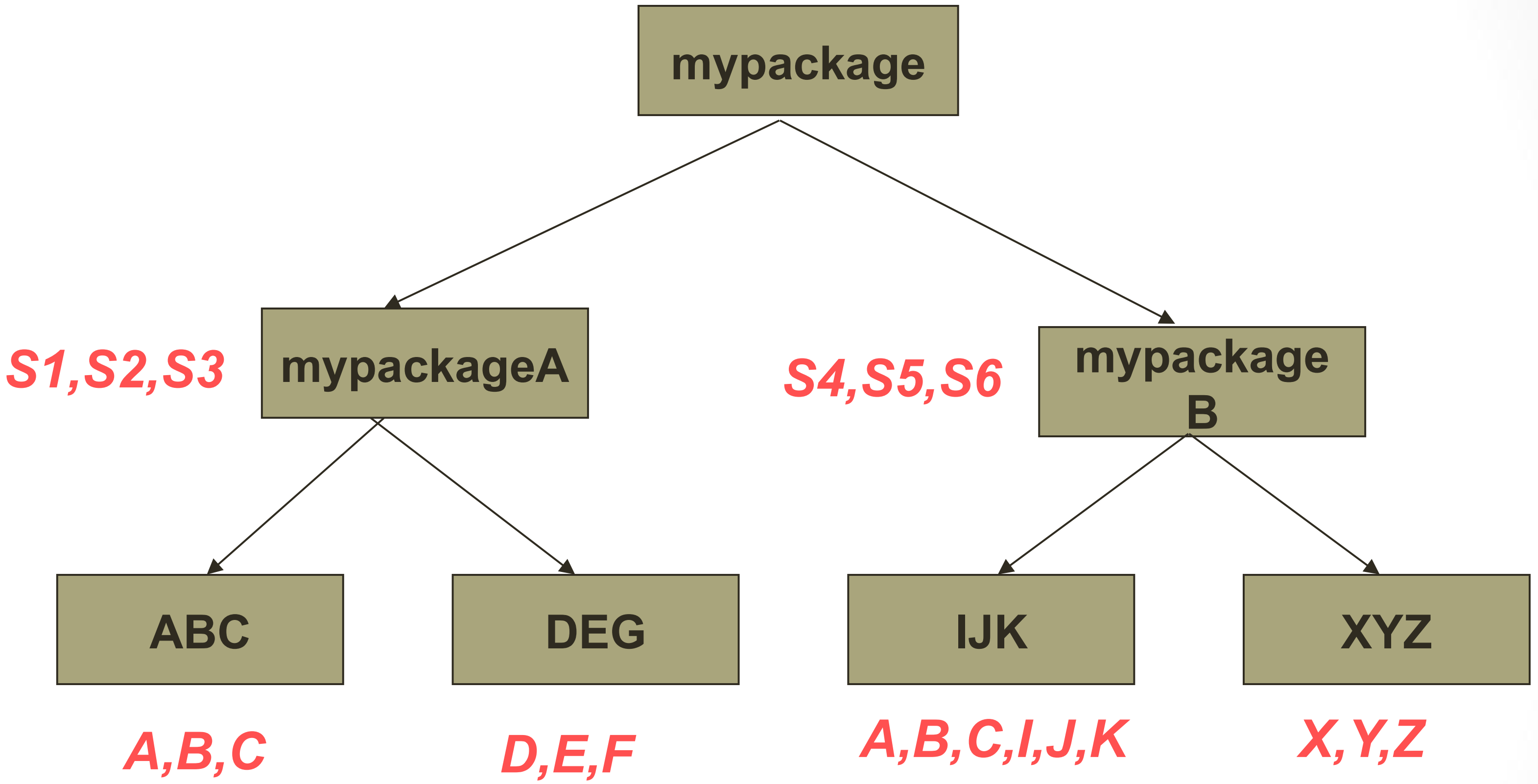
  - **User-defined** packages : Java Package created by user to categorize classes and interface.

How To Create a Package

- To create a package, First we have to create a directory /directory structure that matches the package hierarchy.
- To make a class belongs to a particular package include the package statement as the first statement of source file.
- There can be only one package statement in each source file, and it applies to all types in the file.

- I:\5\mypack\Student.java
- In mypack folder, file Student.java (we need to be in - I:\5)
  - Compile: javac mypack/Student.java (compiler operates on files)
  - Run : java mypack.Student (interpreter loads a class)

Creating Packages



- Package ABC and IJK have classes with same name.
- A class in ABC has name mypackage.mypackageA.ABC.A
- A class in IJK has name mypackage.mypackageB.IJK.A

How to make a class Belong to a Package

- Include a proper package statement as first line in source file

Make class S1 belongs to mypackageA

```
package mypackage.mypackageA;
   public class S1    {
      public S1( )        {
     System.out.println("This is Class S1");
      }
}
```

Name the source file as S1.java and compile it and store the S1.class file in mypackageA directory

Make class S2 belongs to mypackageA

```
package mypackage.mypackageA;
   public class S2
   {
      public S2( )
      {
         System.out.println("This is Class S2");
       }
   }
```

Name the source file as S2.java and compile it and store the S2.class file in mypackageA directory

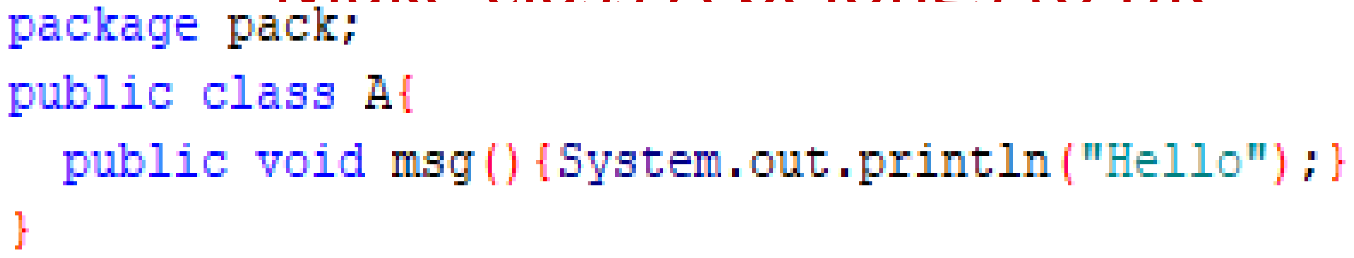
Make class A belongs to IJK

```
package mypackage.mypackageB.IJK;
    public class A
    {
        public A( )
        {
            System.out.println("This is Class A in IJK");
        }
}
```

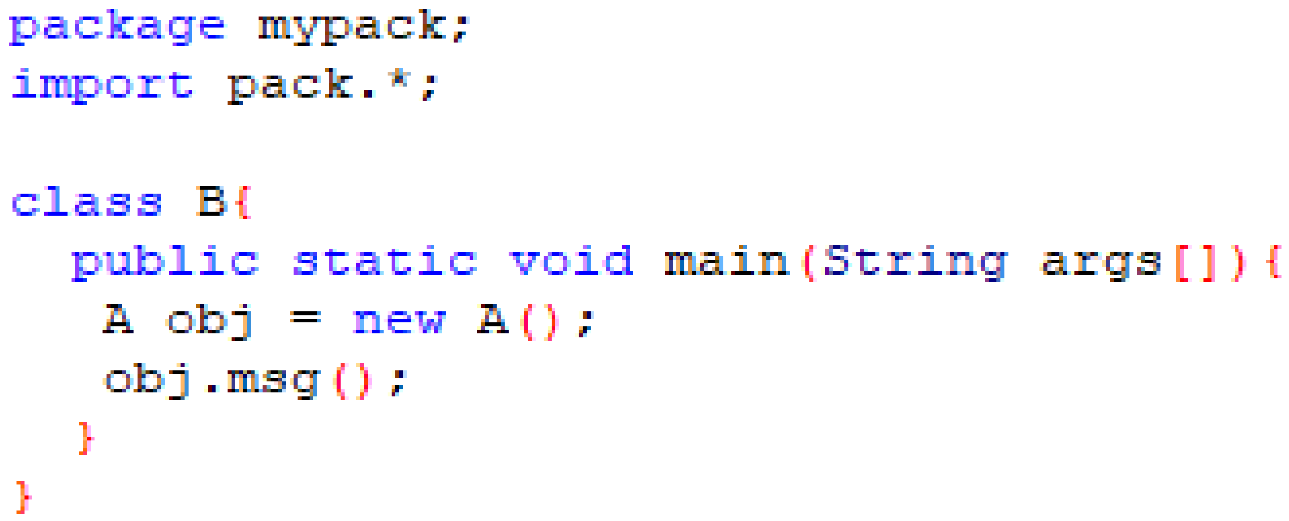
Name the source file as A.java and compile it and store the A.class file in IJK directory

<< Same Procedure For all classes>>

Make class A belongs to UK



```
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```



```
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Import keyword

- A class can use all classes from its own package and all public classes from other packages.

3 ways to refer to a class present in different package:

- Using fully qualified name
  - Class MyDate extends java.util.Date { … }
- Import the only class you want to use
  - import java.util.date;
  - class MyDate extends Date{ … }
- Import all the classes from the particular package
  - import java.util.*;
  - class MyDate extends Date{ … }
- Import statement is kept after package statement
  - E.g. package mypack;

    import java.util.*;

Importing the Package

- import statement allows the importing of package
- Library packages are automatically imported irrespective of the location of compiling and executing program
- JRE looks at two places for user created packages

  (i) Under the current working directory

  (ii) At the location specified by CLASSPATH environment variable
- Most ideal location for compiling/executing a program is immediately above the package structure.

Employee.java

Boss.java

Example importing

```
import mypackage.mypackageA.ABC;
import mypackage.mypackageA.ABC.*;
class packagetest
{
     public static void main(String args[])
     {
        B b1 = new B();
        C c1 = new C();
     }
}
```

<< packagetest.java>>

```
This is Class B
This is Class C
```

<< Store it in location above the package structure. Compile and Execute it from there>>

```
import mypackage.mypackageA.ABC.*;
import mypackage.mypackageB.IJK.*;
class packagetest
{
        public static void main(String args[])
        {
                A a1 = new A();
        }
}
```

<< What’s Wrong Here>>

```
mypackage.mypackageA.ABC.A a1 = new
mypackage.mypackageA.ABC.A();
                              OR
mypackage.mypackageB.IJK.A a1 = new mypackage.mypackageB.IJK.A();
```

<< class A is present in both the imported packages ABC and IJK. So A has to be fully qualified in this case>>

CLASSPATH Environmental Variables

- CLASSPATH Environmental Variable lets you define path for the location of the root of the package hierarchy.
- Consider the following statement :

  package mypack;

  What should be true in order for the program to find mypack.

  (i) Program should be executed from the location immediately above mypack.

  - I:\5> javac mypackage/mypackageA/ABC/A.java
  - I:\5> java mypackage.mypackageA.ABC.A

OR

CLASSPATH Environmental Variables

(ii) mypack should be listed in the set of directories for CLASSPATH.

- I:\5\mypack>javac Student.java
- I:\5\mypack>set classpath = .;I:\5\;
- I:\5\mypack>java mypack.Student

- I:\5\mypackage\mypackageA\ABC>javac A.java
- I:\5\mypack>set classpath = .;I:\5\;
- I:\5\mypackage\mypackageA\ABC>java mypackage.mypackageA.ABC.A

The Directory Structure of Packages:

- In general, a company uses its reversed Internet domain name for its package names.
  - Eg: A company's Internet domain name is apple.com, then all its package names would start with com.apple.
- Example: The company had a com.apple.computers package that contained a Dell.java source file.
  - // File Name: Dell.java
  - package com.apple.computers;
  - public class Dell{ }
  - class Ups{ }

What is jar files? How to create it?

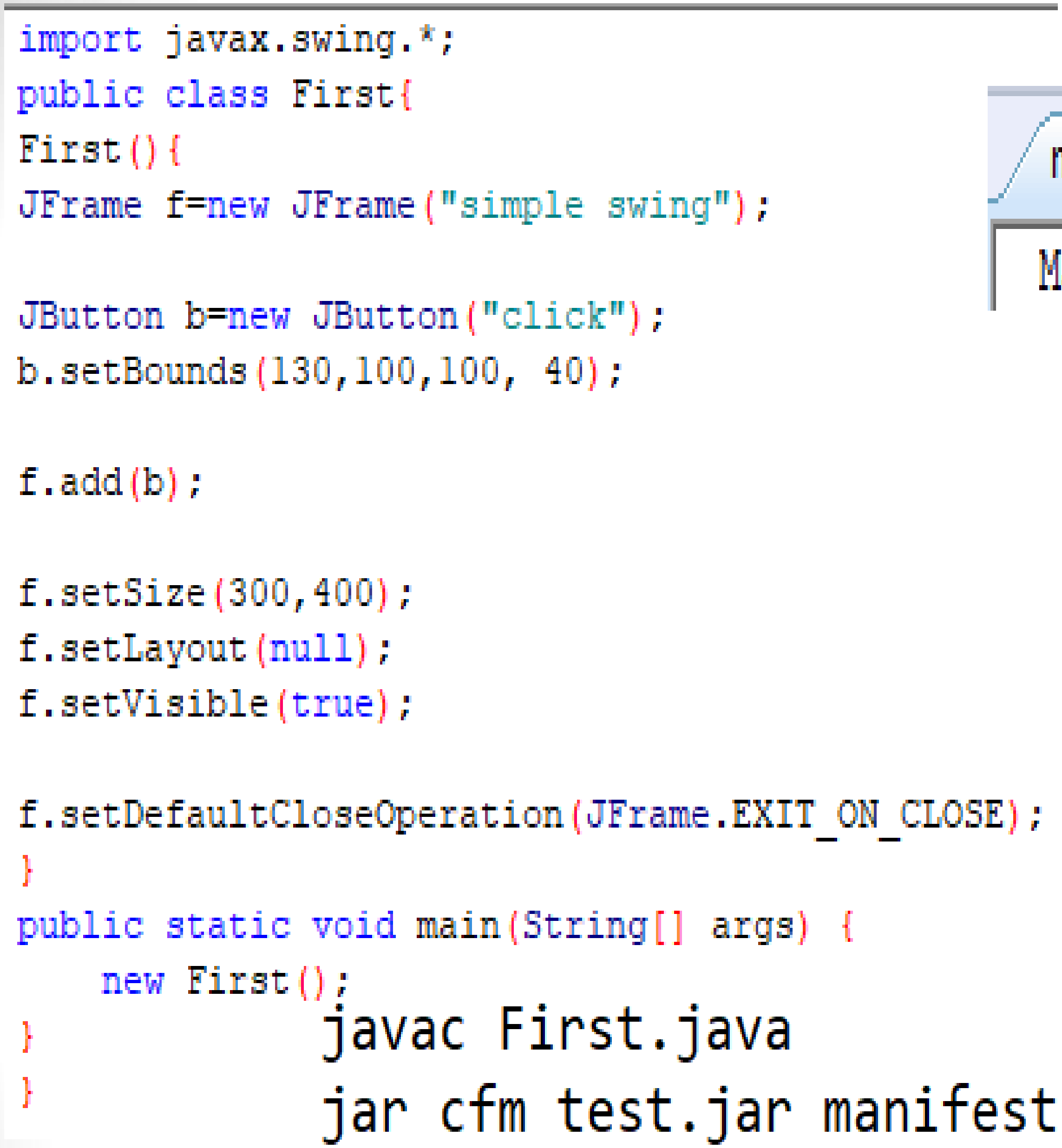
- JAR : It is a java archive file used to package classes, files etc. as single file.
  - This is similar to zip file in windows.
- To create a jar file with all class files under the current directory.
  - jar –cvf jarfilename.jar *.class
  - c - to *create* a JAR file.
  - f - the output goes to a *file* rather than to stdout.
  - v -Produces *verbose* output on stdout while the JAR file is being built.
    - The verbose output tells you the name of each file as it's added to the JAR file.

What is jar files? How to create it?

- Can jar code be retrieved from byte code?
    - Yes, class files can be decompiled using utilities like JAD(JAva Decompiler). This takes the class files as an input and generates a java file.

How to run?

- You need to specify a Main-Class in the jar file manifest.
- You need two files: java/class files to be included in jar file and manifest.mf file to indicate the main class.
- Note that 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.
  - javac Test.java
  - jar cfm test.jar manifest.mf Test.class
    - -m includes manifest information from the given mf file.
  - java -jar test.jar



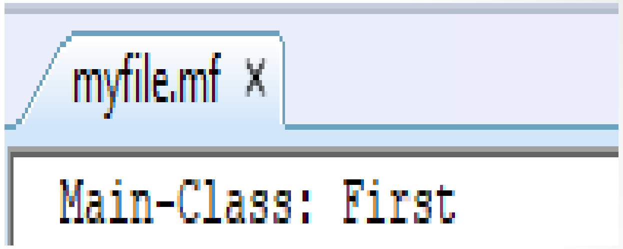
```java
import javax.swing.*;
public class First{
First(){
JFrame f=new JFrame("simple swing");

JButton b=new JButton("click");
b.setBounds(130,100,100, 40);

f.add(b);

f.setSize(300,400);
f.setLayout(null);
f.setVisible(true);

f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public static void main(String[] args) {
    new First();
}
}
```



```
Main-Class: First
```

```
javac First.java
jar cfm test.jar manifest.mf First.class
java -jar test.jar
```

static import

- To import static member of class.
- Using static import it is possible to refer static member without even using class name.
- Import static package.class-name.static-member-name;
    - E.g. import static java.lang.Math.sqrt;
- Import static package.class-type-name.*;
    - E.g. import static java.lang.Math.*;

- Test.java

What's the use of static imports?

- Static imports are used to save your time and typing.
- If you hate to type same thing again and again then you may find such imports interesting.
- Lets understand this with the help of below examples:
- **Example 1: Without Static Imports**
- class Demo1{

```
        public static void main(String args[]) {
            double var1= Math.sqrt(5.0);
            double var2= Math.tan(30);
            System.out.println("Square of 5 is:"+ var1);
            System.out.println("Tan of 30 is:"+ var2);
        }
  }
```

- **Output:**
  - Square of 5 is:2.23606797749979 Tan of 30 is:-6.405331196646276

What’s the use of static imports?

- **Example 2: Using Static Imports**
- import static java.lang.System.out;
- import static java.lang.Math.*;

```
class Demo2{
      public static void main(String args[]) {
            //instead of Math.sqrt need to use only sqrt
            double var1= sqrt(5.0); //instead of Math.tan need to use only tan
            double var2= tan(30); //need not to use System in both the below statements
            out.println("Square of 5 is:"+var1);
            out.println("Tan of 30 is:"+var2);
      }
}
```

Output:

```
Square of 5 is:2.23606797749979 Tan of 30 is:-6.405331196646276
```

When to use static imports?

- If you are going to use static variables/methods a lot then it’s fine to use static imports.
- For example if you want to write a code with lot of mathematical calculations then you may want to use static import.
- **Drawbacks**
- It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it.
- You can also use wildcard(*) imports.