

Packages

Unit 5

Objectives

- Name clashes
- Packages
- class path and CLASSPATH environmental variables
- Importing a package
- Default access specifier
- JAR files
- Lib/ext JAR files

Name clashes

```
class Connection {  
    public void connectToOracle() { }  
}
```

Connect to Oracle ,
developed by team X

```
class Connection {  
    public void connectToSybase() { }  
}
```

Connect to Sybase ,
developed by team Y

```
class DatabaseWriter{  
    public static void main(String[ ] args)  
    {  
        Connection connection = new Connection();  
        connection.connectToOracle();  
    }  
}
```

//Error

Name clashes

- In the earlier example two classes with the same name Connection existed, one to connect to Oracle database and other to Sybase database.
- Since the class names were same there is no way to specify which of the two classes to use.
- In general, while an application is being developed, several classes are written by different programmers.
 - Care should also be taken to ensure that 're-declarations' do not happen when these classes are integrated
- **How would you solve the problem of name clashes?**

What is a Package?

- To make types (classes and interfaces) easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups related types into *packages*.
- A *package* is a grouping of related classes, interfaces etc, providing access protection and namespace management.
- Uses of Package
 - Organization of related classes in to units.
 - Reduce Naming conflicts
 - Protection of variables, methods and classes.
 - Can contain further subpackages.
 - A package is both a directory as well as a library

Creating Packages

- To create a package, first choose the name of the package.
- To include a class or interface to be a part of a package, put a *package* statement with the name of package at the top of *every source file* that contains the classes or interface.

```
package instruments;  
public interface Instrument  
{  
    .....  
}
```

```
package instruments;  
public class Sitar implements  
    Instrument  
{  
    .....  
}  
public class Violin implements  
    Instrument  
{  
    .....  
}
```

Rules for creating Packages

- The package statement 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.
- If there are multiple classes or interface in the same source file, only one can be public, and it must have the same name as the source file.
- Only the public type will be accessible from outside of the package, and all the non-public types will be private to the package.

Rules for creating Packages

```
package instruments;  
  
public class Violin implements Instrument  
{  
  
    public void play() { }  
  
}
```

- A physical folder with the same name as the package has to be created and all the .class files belonging to that package have to be included in it.
- Example
On compilation of the above code we would get a Violin.class file, this file should be included in a directory instruments.

C:/myproject/instruments/Violin.class

Nested packages

```
package instruments.western;  
  
public class Guitar implements  
           instruments.Instrument  
{  
  
    public void play() { }  
  
}
```

- A package can be nested inside another package.
- Here on compilation of the above code we would get a Guitar.class file, this file should be included in a directory instruments/western.

C:/myproject/instruments/western/Guitar.class

Package naming convention

- With programmers worldwide writing classes and interfaces using Java, it is likely that many programmers will use the same name for different types and would lead to name clashes when shared.
- This calls for not only the types to be defined in a package to avoid name clashes, but also the package names need to be unique.
- Since the Internet domain names are unique for a company, the reversed domain name is used to name a package.
 - Ex `com.pratian.javatraining` for a package `javatraining`
- Package names are written in lowercase to avoid conflict with the names of classes or interfaces.
- Packages in the Java language itself begin with `java.` or `javax.`

Accessing package members

- The types that comprise a package are known as the package members.
- A public package member can be accessed from outside its package by,
 - Refer to the member by its fully qualified name
 - Import the package member
 - Import the member's entire package

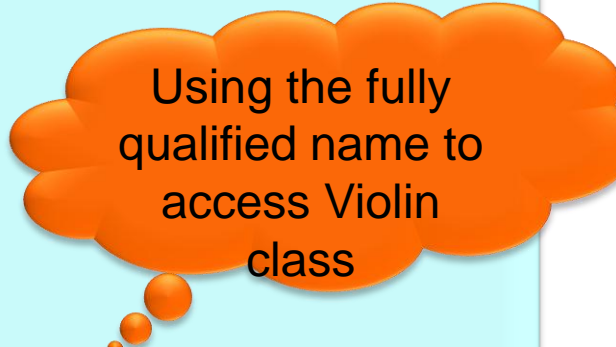
Accessing package members

- Referring to the member by its fully qualified name
 - A class or interface belonging a package can be referred by using a fully qualified name.

```
package com.classical.instruments;  
  
public class Violin implements Instrument  
{  
    public void play() { }  
}
```

Accessing package members

```
package com.myorchestra;  
  
public class Orchestra  
{  
    public static void main(String [ ] arg)  
    {  
        com.classical.instruments.Violin violin = new  
            com.classical.instruments.Violin();  
  
        violin.play();  
    }  
}
```



Using the fully
qualified name to
access Violin
class

Accessing package members

- Import the package member
 - To import a specific member into the current file, put an **import statement** at the beginning of the file before any type definitions but after the **package statement**, if there is one.
 - This approach works well if just few classes are used. But to use many types then the entire package has to be imported.


Using the import to access the Violin class

```
package com.myorchestra;  
  
import com.classical.instruments.Violin;  
  
public class Orchestra {  
    public static void main(String [ ] arg) {  
        Violin violin = new Violin();  
        violin.play();  
    }  
}
```

Accessing package members

- Import all the members of a package
 - To import all the types contained in a particular package, use the import statement with the asterisk(*) wildcard character.

```
package com.myorchestra;  
  
import com.classical.instruments.*;  
  
public class Orchestra {  
    public static void main(String [ ] arg) {  
        Violin violin = new Violin();  
        violin.play();  
    }  
}
```



Using the import with *, to access all members of the package

classpath

```
package com.pratian.mypackage;  
  
public class MyClass  
{  
    .....  
}
```

- When the above source file is compiled, the compiler generates a MyClass.class file, this file can be located at,
**<path to parent directory of the output file>
com\pratian\mypackage\MyClass.class**
- Here the .class file should be in a series of directories that reflect the package name.

classpath

- Now, when the package `com.pratian.mypackage` is imported in a source file, the JVM needs the full path to the `com` folder to include the classes in the package. This *full path* to the `com` folder is must be included in the **CLASSPATH** environment variable.

- For example, if

`C:\JavaApplication\classes` is in the classpath,
`com.pratian.mypackage` is the package name,
then the compiler will look for the `MyClass.class` in,
`C:\JavaApplication\classes\com\pratian\mypackage`

Setting CLASSPATH system variable

- To display the current CLASSPATH variable on Windows, use the commands

```
C:\set CLASSPATH
```

- To set the CLASSPATH variable, use the commands

```
C:\set CLASSPATH=C:\JavaApplication\classes;%classpath%
```

- The CLASSPATH variable, can as well be set in Windows by changing the System Properties, Advanced settings, which will be demonstrated.

Running an application

- Once the class path is set, the application can be run by giving a fully qualified name of the class with the main.
- Example

```
package instruments.classical;  
  
public class Orchestra {  
    public static void main(String [ ] arg) {  
        Violin violin = new Violin();  
        violin.play();  
    }  
}
```

- To run the above application from the console, say,
java instruments.classical.Orchestra

Default Access Specifiers

- A class member can be public, private, protected or of default access.
- All members of the class with default access are accessible only by the members of the package and are not exposed outside the package.
- Protected members of a class are accessible by members of the package and by subclasses of the class belonging to other package.
- A class within a package can be public or default access, only public are accessible outside the package.

Exercise

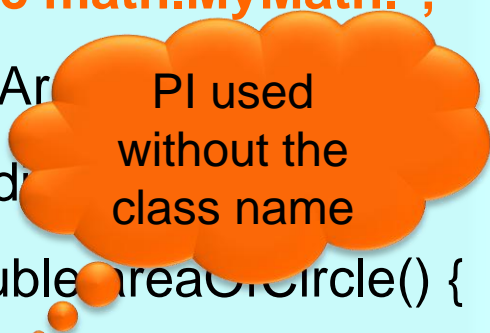
- Write a Printer class in the package `com.pratian.myprinter` with the data member `boolean : inUse` and the following methods,
`setInUse(boolean)` : to set the status of the printer .
`getStatus()` : returns the status of the printer.
`free()` : to free the printer after use
`print(String)` : to print a String on the console
- In the `myprinter` package ensure that an object of the Printer class can be created only by members of the same package, but the same object can be used outside the package as well.
- Write a `PrinterManager` class in the package `com.pratian.myprinter` with an instance of Printer class as a data member.
The class has a method `getPrinter()` which returns the printer object to a client if not in use and has a method `releasePrinter()` which in turn calls the method `free()` on the printer object.
- Import the above package in your program and write a client to create an instance of the `PrinterManager` and in turn get the printer and write on to the screen.

Static Import

- There are many situations which requires frequent access to static final fields (constants) and static methods from one or more classes.
- Prefixing the name of these classes over and over can result in cluttered code.
- The *static import* statement can be used to import constants and methods that used without prefixing the class name.

```
package math;  
  
public class MyMath {  
    public static final double  
        PI = 3.14;  
}
```

```
import static math.MyMath.*;  
  
public class AreaOfCircle {  
    double radius;  
    public double areaOfCircle() {  
        return PI * radius * radius ;  
    }  
}
```



PI used
without the
class name

JAR Files

- The Java Archive (JAR) file format enables to bundle multiple files into a single archive file.
- Need for JAR files
 - **Compression** – The JAR format allows compression of files for efficient storage.
 - **Decreased download time** – If a library is bundled in a JAR file, the resources can be downloaded in a single transaction, without the need for opening a new connection for each file.
 - **Package Sealing** - Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.
 - **Package Versioning** - A JAR file can hold data about the files it contains, such as vendor and version information.
 - **Portability** - The mechanism for handling JAR files is a standard part of the Java platform's core API.

Creating a JAR File

- The basic format of the command for creating a JAR file is:
jar cvf jar-file input-file(s)
- The options and arguments used in this command are:
 - The **c** option indicates that you want to *create* a JAR file.
 - The **f** option indicates that you want the output to go to a *file* rather than to stdout.
 - **jar-file** is the name that you want the resulting JAR file to have. By convention, JAR filenames are given a .jar extension, though this is not required.
 - The **input-file(s)** argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file(s) argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.
 - The **c** and **f** options can appear in either order, but there must not be any space between them.
- This command will generate a compressed JAR file and place it in the current directory.

Viewing contents of a JAR File

- The basic format of the command for viewing the contents of a JAR file is:

jar tvf jar-file

- The options and arguments used in this command are:
 - The **t** option indicates that you want to view the *table* of contents of the JAR file.
 - The **f** option indicates that the JAR file whose contents are to be viewed is specified on the command line.
 - The **jar-file** argument is the path and name of the JAR file whose contents you want to view.
 - The **t** and **f** options can appear in either order, but there must not be any space between them.
- This command will display the JAR file's table of contents to stdout.

Extracting contents of a JAR File

- The basic command to use for extracting the contents of a JAR file is:

jar xvf jar-file [archived-file(s)]

- The options and arguments used in this command are:
 - The **x** option indicates that you want to *extract* files from the JAR archive.
 - The **f** options indicates that the JAR *file* from which files are to be extracted is specified on the command line, rather than through stdin.
 - The **jar-file** argument is the filename (or path and filename) of the JAR file from which to extract files.
 - **archived-file(s)** is an optional argument consisting of a space-separated list of the files to be extracted from the archive. If this argument is not present, the Jar tool will extract all the files in the archive.
 - The order in which the **x** and **f** options appear in the command doesn't matter, but there must not be a space between them.
- When extracting files, the Jar tool makes copies of the desired files and writes them to the current directory, reproducing the directory structure that the files have in the archive. The original JAR file remains unchanged.

Adding a JAR file to class path

- While adding JAR files to the classpath, it is necessary to add, not just the location, but also the file name of the JAR file.
- E.g. – Consider a JAR file called MyLib.jar located in

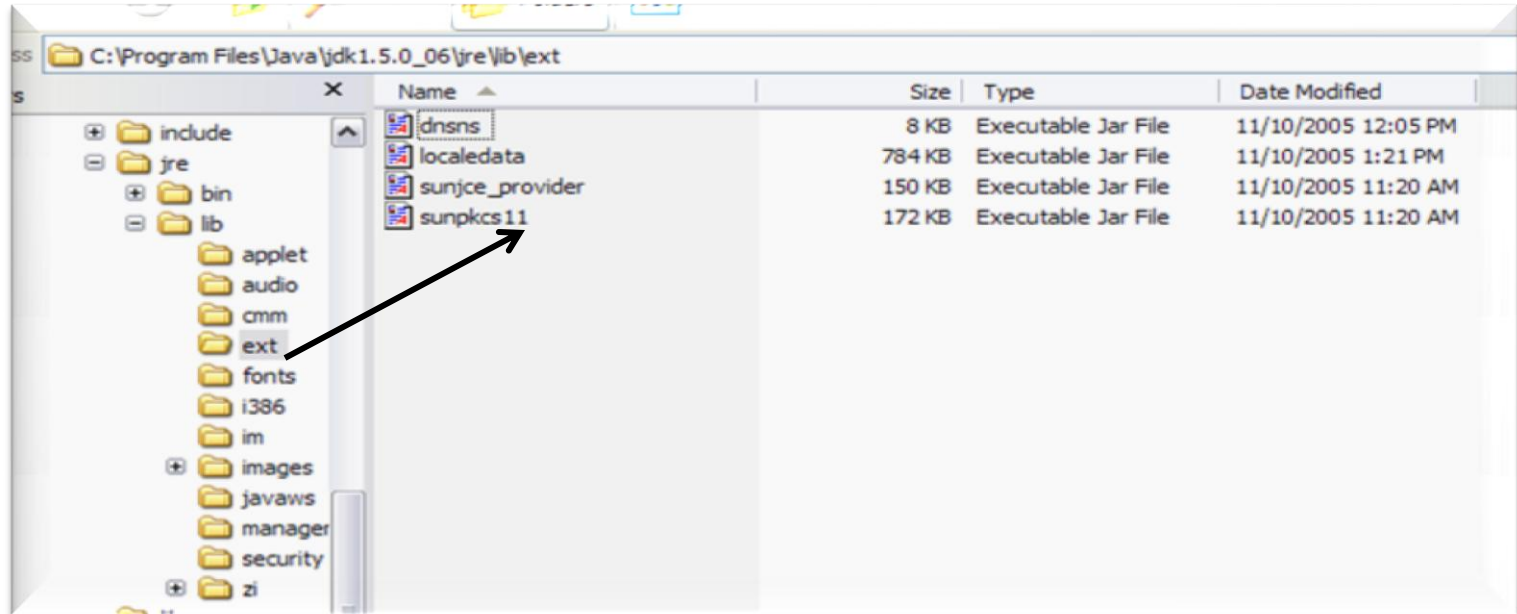
C:\MyJava\Project\MyLib.jar

set classpath=C:\MyJava\Project\MyLib.jar

And not

set classpath=C:\MyJava\Project\

Lib/ext JAR files



- JAR files placed in `<jrehome>/lib/ext` folder need not be explicitly included in the class path, they are as if automatically in the class path.

Question time



Please try to limit the questions to the topics discussed during the session.

Participants are encouraged to discuss other issues during the breaks.

Thank you.