Java Programming

Packages & File I/O

Packages

Packages

- Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.
- A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.
- Some of the existing packages in Java are:
  - **java.lang** - bundles the fundamental classes
  - **java.io** - classes for input , output functions are bundled in this package
  - **java.util** - contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
    - ***Note:*** This is the package used for the **Scanner** object that we use frequently, as well as the **List** and **Map** objects.

Package Creation

- When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

- The **package** statement should 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 a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Package Example

- It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

```
/* File name : Animal.java */
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

- Now, put an implementation in the same package *animals*:

```
package animals;

/* File name : MammalInt.java */
public class MammalInt implements Animal{
    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();

        m.eat();
        m.travel();
    }
}
```

Screen:

```
C:\> javac animals/MammalInt.java
C:\> java animals.MammalInt
Mammal eats
Mammal travels
```

or

```
C:\> javac animals/MammalInt.java
C:\> java animals/MammalInt
Mammal eats
Mammal travels
```

Import

- If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax. (i.e. the Employee class)

```
package payroll;

public class Boss {
   public void payEmployee(Employee e) {
      e.mailCheck();
   }
}
```

- What happens if Employee is not in the payroll package?
  - The Boss class must then use one of the following techniques for referring to the Employee class in a different package.
    - The fully qualified name of the class can be used. For example:
      `payroll.Employee`
    - The package can be imported using the import keyword and the wild card (*). For example:
      `import payroll.*;`
    - The class itself can be imported using the import keyword. For example:
      `import payroll.Employee;`

Streams, Files, & I/O

Streams, Files, & I/O

- The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
- All these streams represent an input source and an output destination.
- The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.
- A stream can be defined as a sequence of data.
  - The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Byte Streams

- Java byte streams are used to perform input and output of 8-bit bytes.
- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.
- Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Character Streams

- Java Byte streams are used to perform input and output of 8-bit bytes, where as Java Character streams are used to perform input and output for 16-bit unicode.
- Though there are many classes related to character streams but the most frequently used classes are , FileReader and FileWriter.
- Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.
- We can re-write the previous example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Standard Streams

- All programming languages usually provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen.
- C or C++ programming languages use three standard devices: STDIN, STDOUT and STDERR.
- Similar way Java provides the following three standard streams:
  - **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
  - **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as System.out.
  - **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

Standard Streams Example

- Following is a simple program which creates InputStreamReader to read standard input stream until the user types a "q“:

```
import java.io.*;

public class ReadConsole {
   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char)cin.read();
            System.out.print(c);
         } while(c != 'q');
      } finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

Screen:

```
C:\> javac ReadConsole.java
C:\> java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

FileInputStream

- The FileInputStream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

- The following constructor takes a file name as a string to create an input stream object to read the file:

```
InputStream f = new FileInputStream("C:/java/hello");
```

- The following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

FileInputStream Methods

- **public void close() throws IOException{}**
  - This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
- **protected void finalize() throws IOException {}**
  - This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
- **public int read(int r) throws IOException{}**
  - This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
- **public int read(byte[] r) throws IOException{}**
  - This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
- **public int available() throws IOException{}**
  - Gives the number of bytes that can be read from this file input stream. Returns an int.

FileOutputStream

- The FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

- Here are two constructors which can be used to create a FileOutputStream object.

- The following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/java/hello");
```

- The following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

FileOutputStream Methods

- **public void close() throws IOException{}**
  - This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
- **protected void finalize()throws IOException {}**
  - This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
- **public void write(int w)throws IOException{}**
  - This methods writes the specified byte to the output stream.
- **public void write(byte[] w)**
  - Writes w.length bytes from the mentioned byte array to the OutputStream.

InputStream & OutputStream Example

- The below code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

```
import java.io.*;

public class fileStreamTest {

   public static void main(String args[]) {

   try {
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

File Navigation and I/O

- There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.
  - File Class
  - FileReader Class
  - FileWriter Class

Creating Directories

- There are two useful File utility methods, which can be used to create directories:
  - The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
  - The **mkdirs()** method creates both a directory and all the parents of the directory.

- The following example creates the "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // Create directory now.
      d.mkdirs();
   }
}
```

Listing Directories

- You can use the **list()** method provided by the **File** object to list all the files and directories available in a directory as follows:

```
import java.io.File;

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

      File file = null;
      String[] paths;

      try {
         // create new file object
         file = new File("/tmp");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

Screen:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```