PREDEFINED PACKAGES

Predefined packages are those which are developed by SUN micro systems and supplied as a part of JDK (Java Development Kit) to simplify the task of java programmer.
NOTE:
Core packages of java starts with java. (For example: java.lang.*) and Advanced packages of java starts with javax. (For example: java.sql.*)

TYPES of predefined packages:

As a part of J2SE we have nine predefined packages which are given in the following table:

java.lang.*
This package is used for achieving the language functionalities such as convertion of data from string to fundamental data, displaying the result on to the console, obtaining the garbage collector. This is the package which is by default imported for each and every java program.

java.io.*
This package is used for developing file handling applications, such as, openingthe file in read or write mode, reading or writing the data, etc.

java.awt.*(abstract windowing toolkit)
This package is used for developing GUI (Graphic Unit Interface) components such as buttons, check boxes, scroll boxes, etc.

java.awt.event.*
Event is the sub package of awt package. This package is used for providing the functionality to GUI components, such as, when button is clicked or when check box is checked, when scroll box is adjusted either vertically or horizontally.

java.applet.*
This package is used for developing browser oriented applications. In other words this package is used for developing distributed programs. An applet is a java program which runs in the context of www or browser.

java.net.*
This package is used for developing client server applications.

java.util.*
This package is used for developing quality or reliable applications in java or J2EE. This package contains various classes and interfaces which improves the performance of J2ME applications. This package is also known as collection framework (collection framework is the standardized mechanism of grouping of similar or different type of objects into single object. This single object is known as collection object).

java.text.*
This package is used for formatting date and time on day to day business operations.

java.lang.reflect.*
Reflect is the sub package of lang package. This package is basically used to study runtime information about the class or interface. Runtime information represents data members of the class or interface, Constructors of the class, types of methods of the class or interface.

java.sql.*
This package is used for retrieving the data from data base and performing various operations on data base.

The java.io Package

The `java.io` package contains a relatively large number of classes, but, as you can see from Figure 11-1 and Figure 11-2, the classes form a fairly structured hierarchy. Most of the package consists of byte streams--subclasses of `InputStream` or `OutputStream` and (in Java 1.1) character streams--subclasses of `Reader` or `Writer`. Each of these stream types has a specific purpose, and, despite its size, `java.io` is a straightforward package to understand and to use.

Before we consider the stream classes in the package, let's examine the important non-stream classes. `File` represents a file or directory name in a system-independent way and provides methods for listing directories, querying file attributes, and renaming and deleting files. `FilenameFilter` is an interface that defines a method that accepts or rejects specified filenames. It is used by `java.awt.FileDialog` and `File` to specify what types of files should be included in directory listings. `RandomAccessFile` allows you to read from or write to arbitrary locations of a file. Often, though, you'll prefer sequential access to a file and should use one of the stream classes.

`InputStream` and `OutputStream` are abstract classes that define methods for reading and writing bytes. Their subclasses allow bytes to be read from and written to a variety of sources and sinks. `FileInputStream` and `FileOutputStream` read from and write to files. `ByteArrayInputStream` and `ByteArrayOutputStream` read from and write to an array of bytes in memory. `PipedInputStream` reads bytes from a `PipedOutputStream`, and `PipedOutputStream` writes bytes to a `PipedInputStream`. These classes work together to implement a *pipe* for communication between threads.

`FilterInputStream` and `FilterOutputStream` are special; they filter input and output bytes. When you create a `FilterInputStream`, you specify an `InputStream` for it to filter. When you call the `read()` method of a `FilterInputStream`, it calls the `read()` method of its `InputStream`, processes the bytes it reads, and returns the filtered bytes. Similarly, when you create a `FilterOutputStream`, you specify an `OutputStream` to be filtered. Calling the `write()` method of a `FilterOutputStream` causes it to process your bytes in some way and then pass those filtered bytes to the `write()` method of its `OutputStream`.

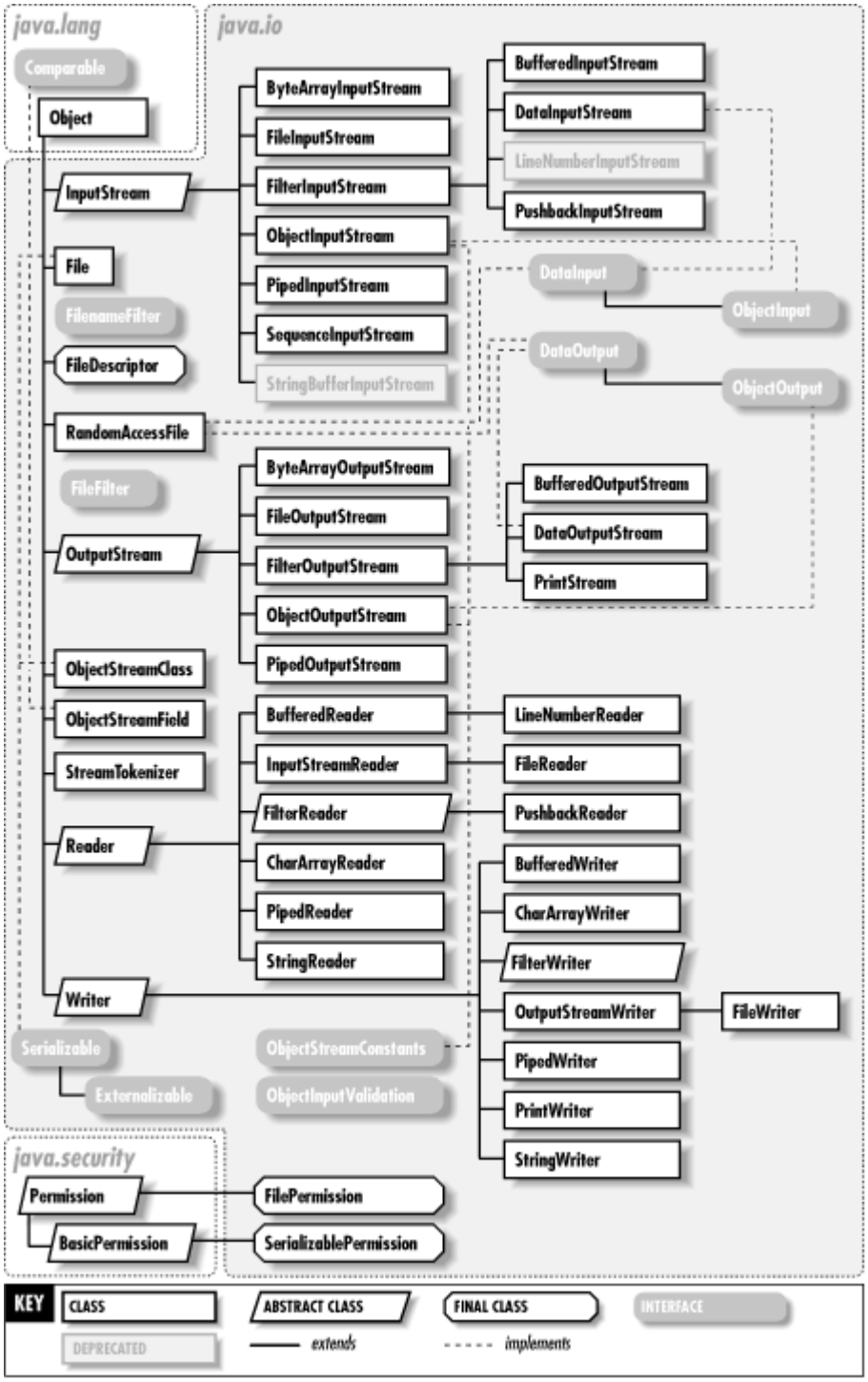

Figure 11-1. The java.io package

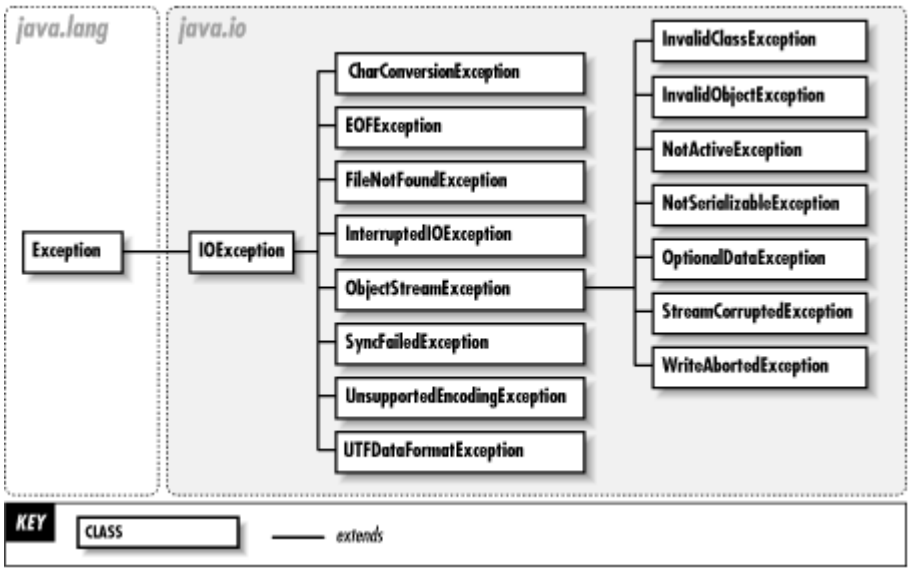

Figure 11-2. The exception classes of the java.io package

`FilterInputStream` and `FilterOutputStream` do not perform any filtering themselves; this is done by their subclasses. `BufferedInputStream` and `BufferedOutputStream` provide input and output buffering and can increase I/O efficiency. `DataInputStream` reads raw bytes from a stream and interprets them in various binary formats. It has various methods to read primitive Java data types in their standard binary formats. `DataOutputStream` allows you to write Java primitive data types in binary format.

In Java 1.1 and later, the byte streams I just described are complemented by an analogous set of character input and output streams. `Reader` is the superclass of all character input streams, and `Writer` is the superclass of all character output streams. These character streams supersede the byte streams for all textual I/O. They are more efficient than the byte streams, and they correctly handle the conversion between local encodings and Unicode text, making them invaluable for internationalized programs. Most of the `Reader` and `Writer` streams have obvious byte-stream analogs. `BufferedReader` is a commonly used stream; it provides buffering for efficiency and also has a `readLine()` method to read a line of text at a time. `PrintWriter` is another very common stream; its methods allow output of a textual representation of any primitive Java type or of any object (via the object's `toString()` method).

The `ObjectInputStream` and `ObjectOutputStream` classes are special. These byte-stream classes are new as of Java 1.1 and are part of the Object Serialization API.

CharConversionException **Java 1.1**

java.io ***serializable checked PJ1.1***

Signals an error when converting bytes to characters or vice versa.

```
public class CharConversionException extends IOException {
// Public Constructors
   public CharConversionException ();
   public CharConversionException (String s);
}
```

Hierarchy: `Object-->Throwable(Serializable)-->Exception-->IOException-->CharConversionException`

EOFException **Java 1.0**

java.io ***serializable checked PJ1.1***

An `IOException` that signals the end-of-file.

```
public class EOFException extends IOException {
// Public Constructors
   public EOFException ();
   public EOFException (String s);
}
```

Hierarchy: `Object-->Throwable(Serializable)-->Exception-->IOException-->EOFException`

FileNotFoundException **Java 1.0**

java.io ***serializable checked PJ1.1(opt)***

An `IOException` that signals that a specified file cannot be found.

```
public class FileNotFoundException extends IOException {
// Public Constructors
   public FileNotFoundException ();
   public FileNotFoundException (String s);
}
```

Hierarchy: `Object-->Throwable(Serializable)-->Exception-->IOException-->FileNotFoundException`

Thrown By: `FileInputStream.FileInputStream(), FileOutputStream.FileOutputStream(), FileReader.FileReader(), RandomAccessFile.RandomAccessFile()`

InterruptedIOException **Java 1.0**

java.io ***serializable checked PJ1.1***

An `IOException` that signals that an input or output operation was interrupted.
The `bytesTransferred` field contains the number of bytes read or written before the operation was interrupted.

```
public class InterruptedIOException extends IOException {
// Public Constructors
   public InterruptedIOException ();
   public InterruptedIOException (String s);
```

```
// Public Instance Fields
   public int bytesTransferred ;
}
```

Hierarchy: `Object-->Throwable(Serializable)-->Exception-->IOException-->InterruptedIOException`

InvalidClassException **Java 1.1**

java.io ***serializable checked PJ1.1***

Signals that the serialization mechanism has encountered one of several possible problems with the class of an object that is being serialized or deserialized. The `classname`field should contain the name of the class in question, and the `getMessage()` method is overridden to return this class name with the message.

```
public class InvalidClassException extends ObjectStreamException {
// Public Constructors
   public InvalidClassException (String reason);
   public InvalidClassException (String cname, String reason);
// Public Methods Overriding Throwable
   public String getMessage ();
// Public Instance Fields
   public String classname ;
}
```

Hierarchy: `Object-->Throwable(Serializable)-->Exception-->IOException-->ObjectStreamException-->InvalidClassException`