Module 10

I/O Fundamentals

Objectives

Upon completion of this module, you should be able to:

- Write a program that uses command-line arguments and system properties
- Examine the `Properties` class
- Construct node and processing streams, and use them appropriately
- Serialize and deserialize objects
- Distinguish readers and writers from streams, and select appropriately between them

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition.* Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, First Edition*. O'Reilly Media. 2003.

Command-Line Arguments

When a Java technology program is launched from a terminal window, you can provide the program with zero or more *command-line arguments*. These command-line arguments allow the user to specify the configuration information for the application. These arguments are strings: either standalone tokens, such as `arg1`, or quoted strings, such as `"another arg"`.

The sequence of arguments follows the name of the program class and is stored in an array of `String` objects passed to the static `main` method; Code 10-1 shows an example:

Code 10-1 Testing the Command-Line Arguments

```
1   public class TestArgs {
2      public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++) {
4            System.out.println("args[" + i + "] is: " + args[i]);
5         }
6      }
7   }
```

This program displays each command-line argument that is passed to the `TestArgs` program. For example:

```
java TestArgs arg1 arg2 "another arg"
args[0] is: arg1
args[1] is: arg2
args[2] is: another arg
```

Note – If an application requires command-line arguments other than type `String`, for example numeric values, the application should convert the `String` arguments to their respective types using the wrapper classes, such as `Integer`. `parseInt` method, which can be used to convert the `String` argument that represents the numeric integer to type `int`.

System Properties

System properties are another mechanism used to parameterize a program at runtime. A *property* is a mapping between a property name and its value; both are strings. The `Properties` class represents this kind of mapping. The `System.getProperties` method returns the system properties object. The `System.getProperty(String)` method returns the string value of the property named in the `String` parameter. There is another method, `System.getProperty(String, String)`, that enables you to supply a default string value (the second parameter), which is returned if the named property does not exist.

Note – Every JVM implementation must supply a default set of properties. (Refer to the documentation for the `System.getProperties` method for details.) Moreover, a particular JVM implementation vendor can supply others.

There are also static methods in the wrapper classes that perform conversion of property values: `Boolean.getBoolean(String)`, `Integer.getInteger(String)`, and `Long.getLong(String)`. The string argument is the name of the property. If the property does not exist, then `false` or `null` (respectively) is returned.

The Properties Class

An object of the Properties class contains a mapping between property names (String) and values (String). It has two main methods for retrieving a property value: getProperty(String) and getProperty(String, String); the latter method provides the capability of specifying a default value that is returned if the named property does not exist.

You can iterate through the complete set of property names using the propertyNames method. By calling getProperty on each name, you can retrieve all of the values.

Finally, property sets can be stored and retrieved from any I/O stream using the store and load methods.

The Code 10-2 program lists the complete set of properties that exist when the program executes:

Code 10-2 Retrieving the System Properties

```
1   import java.util.Properties;
2
3   public class TestProperties {
4      public static void main(String[] args) {
5         Properties props = System.getProperties();
6         props.list(System.out);
7      }
8   }
```

Line 6 retrieves the set of system properties and Line 7 prints the properties using the list method of the Properties class.

```
java -DmyProp=theValue TestProperties
```

The following is an excerpt of the output:

```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

I/O Stream Fundamentals

A *stream* is a flow of data from a *source* to a *sink*. Typically, your program is one end of that stream, and some other node (for example, a file) is the other end.

Sources and sinks are also called *input streams* and *output streams*, respectively. You can read from an input stream, but you cannot write to it. Conversely, you can write to an output stream, but you cannot read from it.Table 10-1 shows the fundamental stream classes.

Table 10-1 Fundamental Stream Classes

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

Data Within Streams

Java technology supports two types of data in streams: raw bytes or Unicode characters. Typically, the term *stream* refers to byte streams and the terms *reader* and *writer* refer to character streams.

More specifically, character input streams are implemented by subclasses of the `Reader` class and character output streams are implemented by subclasses of the `Writer` class. Byte input streams are implemented by subclasses of the `InputStream` class and byte output streams are implemented by subclasses of the `OutputStream` class.

Byte Streams

The following sections describe the fundamental byte streams.

The `InputStream` Methods

The following three methods provide access to the data from the input stream:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

The first method returns an `int`, which contains either a byte read from the stream, or a `-1`, which indicates the end of file condition. The other two methods read the stream into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a sub range in the target array that needs to be filled.

Note – For efficiency, always read data in the largest practical block, or use buffered streams.

```
void close()
```

When you have finished with a stream, close it. If you have a *stack* of streams, use filter streams to close the stream at the top of the stack. This operation also closes the lower streams.

```
int available()
```

This method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

```
long skip(long n)
```

This method discards the specified number of bytes from the stream.

```
boolean markSupported()
void mark(int readlimit)
void reset()
```

You can use these methods to perform *push-back* operations on a stream, if supported by that stream. The `markSupported()` method returns `true` if the `mark()` and `reset()` methods are operational for that particular stream. The `mark(int)` method indicates that the current point in the stream should be noted and a buffer big enough for at least the specified argument number of bytes should be allocated. The parameter of the `mark(int)` method specifies the number of bytes that can be re-read by calling `reset()`. After subsequent `read()` operations, calling the `reset()` method returns the input stream to the point you marked. If you read past the marked buffer, `reset()` has no meaning.

The `OutputStream` Methods

The following methods write to the output stream:

```
void write(int)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

As with input, always try to write data in the largest practical block.

```
void close()
```

You should close output streams when you have finished with them. Again, if you have a stack and close the top one, this closes the rest of the streams.

```
void flush()
```

Sometimes an output stream accumulates writes before committing them. The `flush()` method enables you to force writes.

Character Streams

The following sections describe the fundamental character streams.

The `Reader` Methods

The following three methods provide access to the character data from the reader:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

The first method returns an `int`, which contains either a Unicode character read from the stream, or a `-1`, which indicates the end of file condition. The other two methods read into a character array and return the number of bytes read. The two `int` arguments in the third method indicate a sub range in the target array that needs to be filled.

Note – Use the largest practical block for efficiency.

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

These methods are analogous to the input stream versions.

The `Writer` Methods

The following methods write to the writer:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

Similar to output streams, writers include the `close` and `flush` methods.

```
void close()
void flush()
```

Node Streams

In the Java JDK, there are three fundamental types of nodes (Table 10-2):

- Files
- Memory (such as arrays or `String` objects)
- Pipes (a channel from one process or thread [a light-weight process] to another; the output pipe stream of one thread is attached to the input pipe stream of another thread)

It is possible to create new node stream classes, but it requires handling native function calls to a device driver, and this is non-portable. Table 10-2 shows the node streams.

Table 10-2 Types of Node Streams

Type	Character Streams	Byte Streams
File	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Memory: array	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Memory: string	`StringReader` `StringWriter`	N/A
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

A Simple Example

Code 10-3 reads characters from a file named by the first command-line argument and writes the character out to a file named by the second command-line argument. Thus, it copies the file. This is how the program might be invoked:

```
java TestNodeStreams file1 file2
```

Code 10-3 The `TestNodeStreams` Program

```
1   import java.io.*;
2
3   public class TestNodeStreams {
4      public static void main(String[] args) {
```

```
5         try {
6            FileReader input = new FileReader(args[0]);
7            try {
8               FileWriter output = new FileWriter(args[1]);
9               try {
10                 char[]     buffer = new char[128];
11                 int        charsRead;
12
13                 // read the first buffer
14                 charsRead = input.read(buffer);
15                 while ( charsRead != -1 ) {
16                    // write buffer to the output file
17                    output.write(buffer, 0, charsRead);
18
19                    // read the next buffer
20                    charsRead = input.read(buffer);
21                 }
22
23              } finally {
24                 output.close();}
25           } finally {
26              input.close();}
27        } catch (IOException e) {
28           e.printStackTrace();
29        }
30     }
31  }
```

As easy as this was, handling the buffer is tedious and error-prone. It turns out that there are classes that handle the buffering for you and present you with the capability to read a stream a *line at a time*. It is called a `BufferedReader` and is a type of a stream called a *processing stream*.

Buffered Streams

The Code 10-4 performs the same function as the program in Code 10-3 on page 10-11 but uses `BufferedReader` and `BufferedWriter`.

Code 10-4 The `TestBufferedStreams` Program

```
1   import java.io.*;
2   public class TestBufferedStreams {
3      public static void main(String[] args) {
4         try {
5            FileReader input = new FileReader(args[0]);
6            BufferedReader bufInput = new BufferedReader(input);
7            try {
8               FileWriter output = new FileWriter(args[1]);
9               BufferedWriter bufOutput= new BufferedWriter(output);
10              try {
11                 String line;
12                 // read the first line
13                 line = bufInput.readLine();
14                 while ( line != null ) {
15                    // write the line out to the output file
16                    bufOutput.write(line, 0, line.length());
17                    bufOutput.newLine();
18                    // read the next line
19                    line = bufInput.readLine();
20                 }
21              } finally {
22                 bufOutput.close();
23              }
24           } finally {
25              bufInput.close();
26           }
27        } catch (IOException e) {
28           e.printStackTrace();
29        }
30     }
31  }
```

The flow of this program is the same as before. Instead of reading a buffer, this program reads a *line at a time* using the `line` variable to hold the `String` returned by the `readLine` method (Lines 14 and 20), which provides greater efficiency. Line 7 chains the file reader object within a buffered reader stream. You manipulate the outer-most stream in the chain (`bufInput`), which manipulates the inner-most stream (`input`).

I/O Stream Chaining

A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. Figure 10-1 demonstrates an example input stream; in this case, a file stream is *buffered* for efficiency and then converted into data (Java primitives) items.

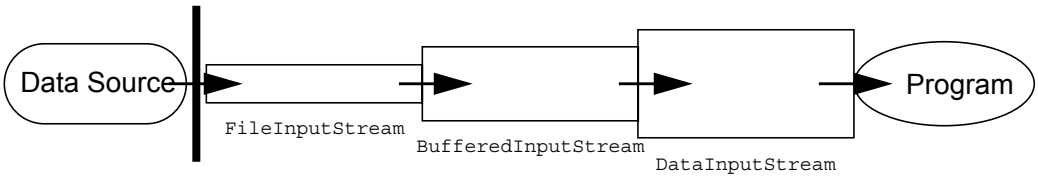

Figure 10-1 An Input Stream Chain Example

Figure 10-2 demonstrates an example output stream; in this case, data is written, then buffered, and finally written to a file.

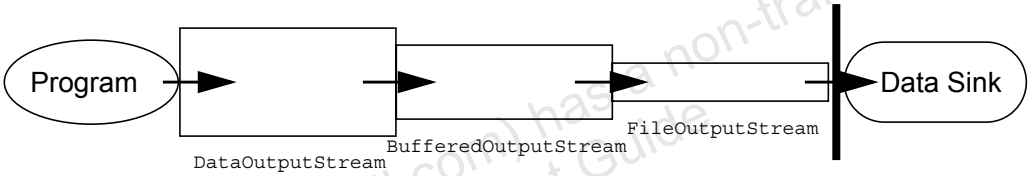

Figure 10-2 An Output Stream Chain Example

Processing Streams

A *processing stream* performs some sort of conversion on another stream. Processing streams are also known as *filter streams*. A filter input stream is created with a connection to an existing input stream. This is done so that when you try to read from the filter input stream object, it supplies you with characters that originally came from the other input stream object. This enables you to convert the raw data into a more usable form for your application. Table 10-3 lists the built-in processing streams that are included in the `java.io` package.

Table 10-3 List of Processing Streams by Type

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	`FilterReader` `FilterWriter`	`FilterInputStream` `FilterOutputStream`
Converting between bytes and character	`InputStreamReader` `OutputStreamWriter`	
*Performing object serialization		`ObjectInputStream` `ObjectOutputStream`
Performing data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

Note – The `Filter`*`Xyz`* streams are abstract classes and cannot be used directly. You can subclass them to implement your own processing streams.

It is easy to create new processing streams. This is described in the next section.

Note – Performing object serialization is described later in this module.

Basic Byte Stream Classes

Figure 10-3 illustrates the hierarchy of input byte stream classes in the `java.io` package. Some of the more common byte stream classes are described in the following sections.

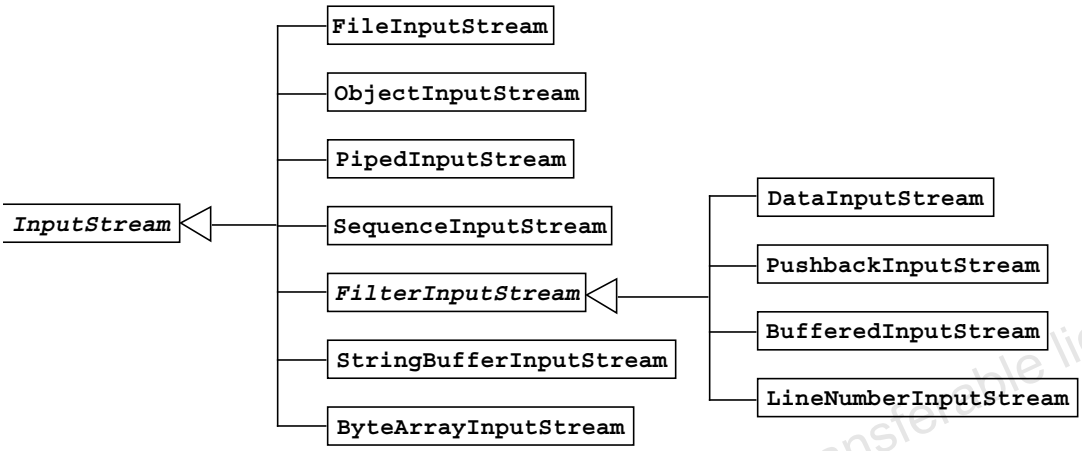

Figure 10-3 The `InputStream` Class Hierarchy

Figure 10-4 illustrates the hierarchy of the output byte stream classes in the java.io package.

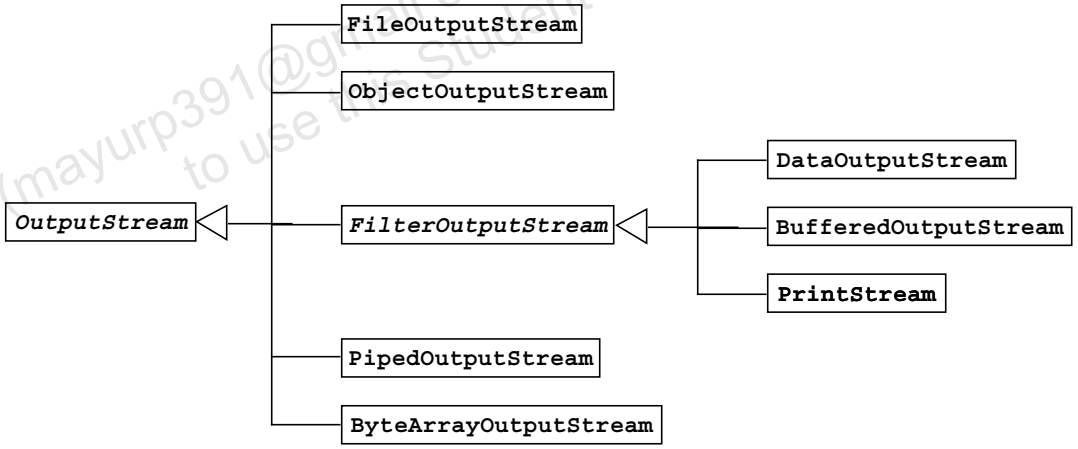

Figure 10-4 The `OutputStream` Class Hierarchy

The FileInputStream and FileOutputStream Classes

The FileInputStream and FileOutputStream classes are node streams and, as the name suggests, they use disk files. The constructors for these classes enable you to specify the path of the file to which they are connected. To construct a FileInputStream, the associated file must exist and be readable. If you construct a FileOutputStream, the output file is overwritten if it already exists.

```
FileInputStream infile
  = new FileInputStream("myfile.dat");

FileOutputStream outfile
  = new FileOutputStream("results.dat");
```

The BufferedInputStream and BufferedOutputStream Classes

Use the BufferedInputStream and BufferedOutputStream class filter streams to increase the efficiency of I/O operations.

The PipedInputStream and PipedOutputStream Classes

You use piped streams for communicating between threads. A PipedInputStream object in one thread receives its input from a complementary PipedOutputStream object in another thread. The piped streams must have both an input side and an output side to be useful.

The DataInputStream and DataOutputStream Classes

The DataInputStream and DataOutputStream called filter streams enable reading and writing of Java primitive types and some special formats using streams. The following methods are provided for the different primitives.

The DataInputStream Methods

The DataInputStream methods are as follows:

```
byte readByte()
long readLong()
double readDouble()
```

The DataOutputStream Methods

The DataOutputStream methods are as follows:

```
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

The methods of DataInputStream are paired with the methods of DataOutputStream.

These streams have methods for reading and writing strings but do not use these methods. They have been deprecated and replaced by readers and writers that are described later.

The ObjectInputStream and ObjectOutputStream Classes

The ObjectInputStream and ObjectOutputstream classes enable reading from and writing Java Objects to streams.

Writing an object to a stream primarily involves writing the values of all the fields of the object. If the fields are objects themselves, these objects should also be written to the stream.

Note – If the fields are declared as transient or static, their values are not written to the stream. This is discussed in the next section.

Reading an object from the stream involves, reading the object type, creating the blank object of that type and filling it with the data that was written.

Persistent storage of objects can be accomplished if files (or other persistent storages) are used as the stream.

The Java API provides a standard mechanism that completely automates the process of writing and reading objects from streams.

Note – If the stream is a network socket stream, the objects are serialized before sending and deserialized after receiving on another host or process.

Figure 10-5 provides an overview illustration of the possible input stream and reader classes you could chain together.

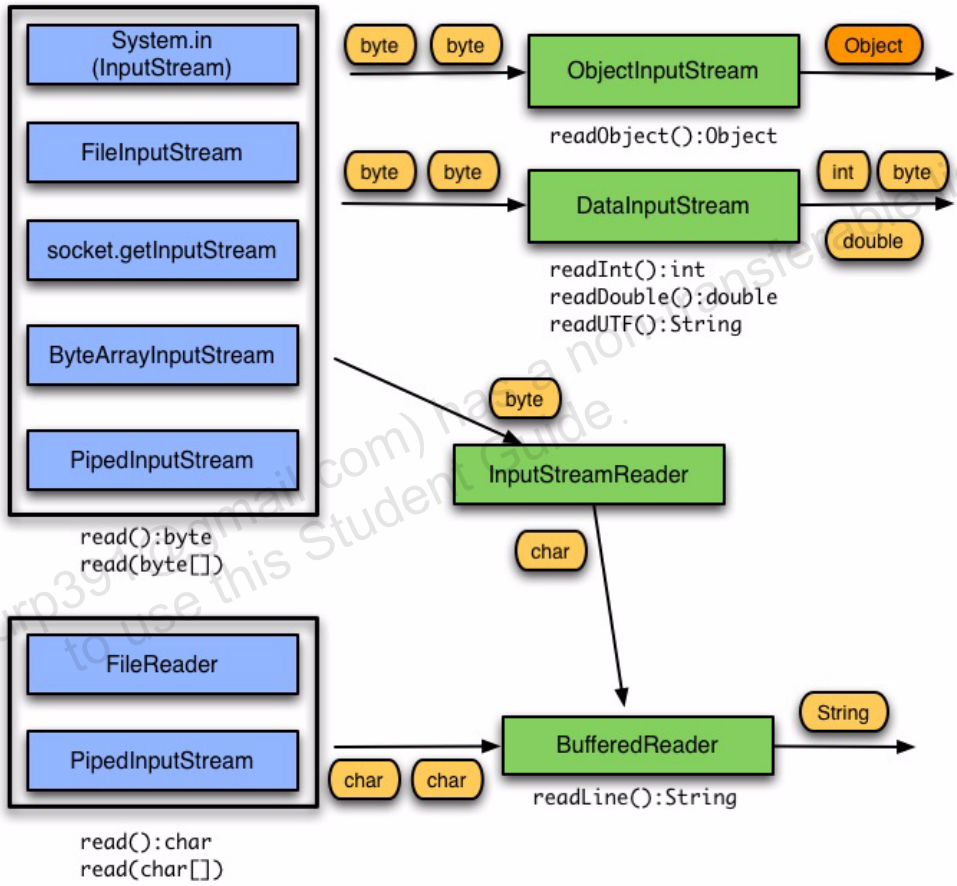

Figure 10-5 Input Chaining Combinations: A Review

Figure 10-6 provides an overview illustration of the possible output stream and writer classes you could chain together.

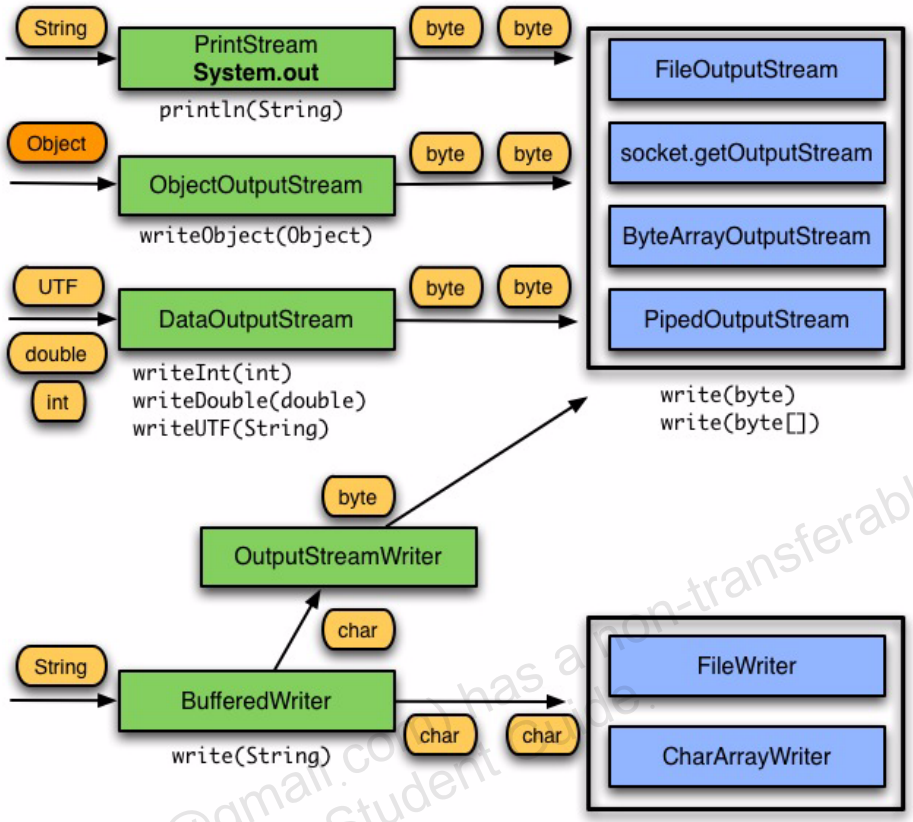

Figure 10-6 Output Chaining Combinations: A Review

Serialization

Saving an object to some type of permanent storage is called *persistence*. An object is said to be *persistent-capable* when you can store that object on a disk or tape or send it to another machine to be stored in memory or on disk. The non-persisted object exists only as long as the Java Virtual Machine is running.

Serialization is a mechanism for saving the objects as a sequence of bytes and later, when needed, rebuilding the byte sequence back into a copy of the object.

For objects of a specific class to be serializable, the class must implement the `java.io.Serializable` interface. The `Serializable` interface has no methods and only serves as a *marker* that indicates that the class that implements the interface can be considered for serialization.

Serialization and Object Graphs

When an object is serialized, only the fields of the object are preserved; methods and constructors are not part of the serialized stream. When a field is a reference to an object, the fields of that referenced object are also serialized if that object's class is serializable. The tree, or structure of an object's fields, including these sub-objects, constitutes the object *graph*.

Some object classes are not serializable because the data they represent contain references to transient operating system resources. For example, `java.io.FileInputStream` and `java.lang.Thread` classes. If a serializable object contains a reference to a non-serializable element, the entire serialization operation fails and a `NotSerializableException` is thrown.

If the object graph contains a non-serializable object reference, the object can still be serialized if the reference is marked with the `transient` keyword.

Code 10-5 shows an example of a class that implements the `Serializable` interface.

Code 10-5 The `Serializable` Example

```
1   public class MyClass implements Serializable {
2      public transient Thread myThread;
```

```
3        private String customerID;
4        private int total;
5    }
```

The field access modifier (`public`, `protected`, *default*, and `private`) has no effect on the data field being serialized. Data is written to the stream in byte format and with strings represented as file system safe universal character set transformation format (UTF) characters. The `transient` keyword prevents the data from being serialized.

```
1    public class MyClass implements Serializable {
2       public transient Thread myThread;
3       private transient String customerID;
4       private int total;
5    }
```

Note – The values stored in static fields are not serialized. When the object is deserialized the values of the static fields are set to the values stored in the corresponding class variables.

Writing and Reading an Object Stream

Writing and reading an object to a stream is a simple process. This section provides examples of writing to and reading from an object stream.

Writing

The code fragment contained in Code 10-6 sends an instance of a `java.util.Date` object to a file.

Code 10-6 The `SerializeDate` Class

```
1    import java.io.*;
2    import java.util.Date;
3
4    public class SerializeDate {
5
6      SerializeDate() {
7        Date d = new Date ();
8
9        try {
10         FileOutputStream f =
11             new FileOutputStream ("date.ser");
```

```
12        ObjectOutputStream s =
13            new ObjectOutputStream (f);
14        s.writeObject (d);
15        s.close ();
16      } catch (IOException e) {
17        e.printStackTrace ();
18      }
19    }
20
21    public static void main (String args[]) {
22      new SerializeDate();
23    }
24  }
```

The serialization starts at Line 14 when `writeObject()` method is invoked.

Reading

Reading the object is as simple as writing it, but with one caveat—the `readObject()` method returns the stream as an `Object` type, and it must be cast to the appropriate class name before methods on that class can be executed. The Code 10-7 illustrates how to deserialize data from a stream.

Code 10-7 The `DeSerializeDate` Class

```
1   import java.io.*;
2   import java.util.Date;
3
4   public class DeSerializeDate {
5
6     DeSerializeDate () {
7       Date d = null;
8
9       try {
10        FileInputStream f =
11            new FileInputStream ("date.ser");
12        ObjectInputStream s =
13            new ObjectInputStream (f);
14        d = (Date) s.readObject ();
15        s.close ();
16      } catch (Exception e) {
17        e.printStackTrace ();
18      }
19
```

```
20      System.out.println(
21        "Deserialized Date object from date.ser");
22      System.out.println("Date: "+d);
23    }
24
25    public static void main (String args[]) {
26      new DeSerializeDate();
27    }
28  }
```

The object deserialization occurs at Line 14, when the `readObject()` method is invoked.

Basic Character Stream Classes

Figure 10-7 illustrates the hierarchy of `Reader` character stream classes in the `java.io` package. Some of the more common character stream classes are described in the following sections.

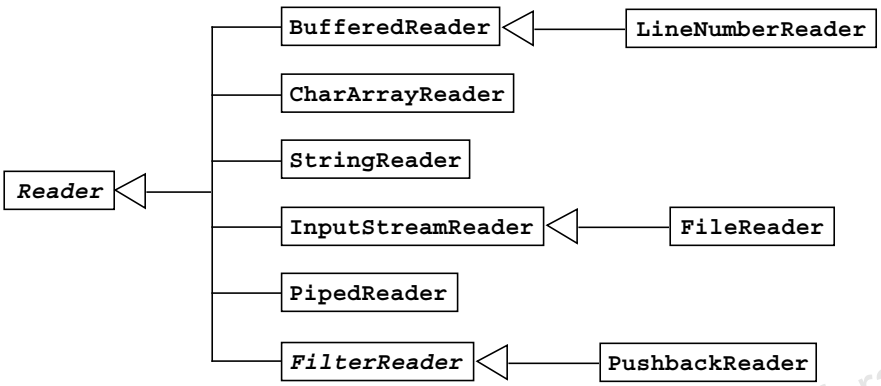

Figure 10-7 The `Reader` Class Hierarchy

Figure 10-8 illustrates the hierarchy of the `Writer` character stream classes in the `java.io` package.

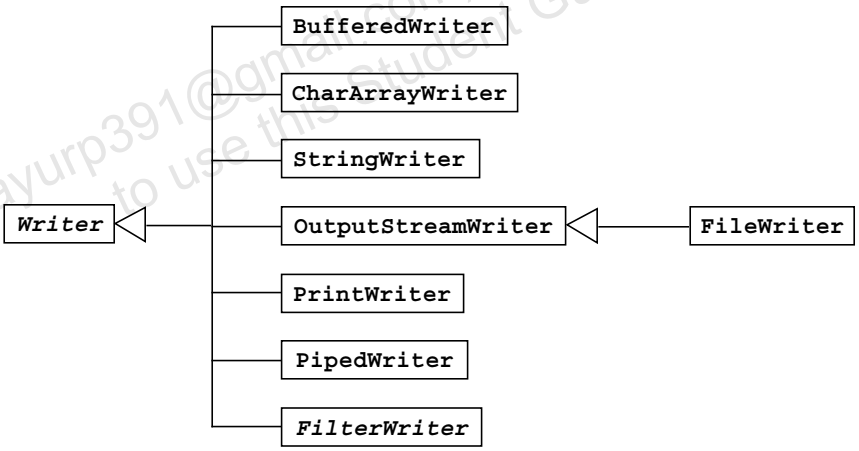

Figure 10-8 The `Writer` Class Hierarchy

The `InputStreamReader` and `OutputStreamWriter` Methods

The most important versions of readers and writers are `InputStreamReader` and `OutputStreamWriter`. These classes are used to interface between byte streams and character readers and writers.

When you construct an `InputStreamReader` or `OutputStreamWriter`, conversion rules are defined to change between 16-bit Unicode and other platform-specific representations.

Byte and Character Conversions

By default, if you construct a reader or writer connected to a stream, the conversion rules change between bytes using the default platform character encoding and Unicode. In English-speaking countries, the byte encoding used is *International Organization for Standardization (ISO) 8859-1*.

Specify an alternative byte encoding by using one of the supported encoding forms. If you have the documentation installed, you can find a list of the supported encoding forms in the documentation found in the following URL:
`http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html`.

Using this conversion scheme, Java technology uses the full flexibility of the local platform character set while still retaining platform independence through the internal use of Unicode.

Using Other Character Encoding

If you need to read input from a character encoding that is not your local one (for example, reading from a network connection with a different type of machine), you can construct the `InputStreamReader` with an explicit character encoding, such as:

```
InputStreamReader ir
  = new InputStreamReader(System.in, "ISO-8859-1");
```

Note – If you are reading characters from a network connection, use this form. If you do not, your program will always attempt to convert the characters it reads as if they were in the local representation, which is probably not correct. ISO 8859-1 is the Latin-1 encoding scheme that maps onto ASCII.

The `FileReader` and `FileWriter` Classes

The `FileReader` and `FileWriter` classes are node streams that are the Unicode character analogues of the `FileInputStream` and `FileOutputStream` classes.

The `BufferedReader` and `BufferedWriter` Classes

Use the `BufferedReader` and `BufferedWriter` class filter character streams to increase the efficiency of I/O operations.

The `StringReader` and `StringWriter` Classes

The `StringReader` and `StringWriter` classes are node character streams that *read* from or *write* to Java technology `String` objects.

Suppose that you wrote a set of report classes that contains methods that accept a `Writer` parameter (the destination of the report text). Because the method makes calls against a generic interface, the program can pass in a `FileWriter` object or a `StringWriter` object; the method code does not care. You use the former object to write the report to a file. You might use the latter object to write the report into memory within a `String` to be displayed within a GUI text area. In either case, the report writing code remains the same.

The `PipedReader` and `PipedWriter` Classes

You use piped streams for communicating between threads. A `PipedReader` object in one thread receives its input from a complementary `PipedWriter` object in another thread. The piped streams must have both an input side and an output side to be useful.