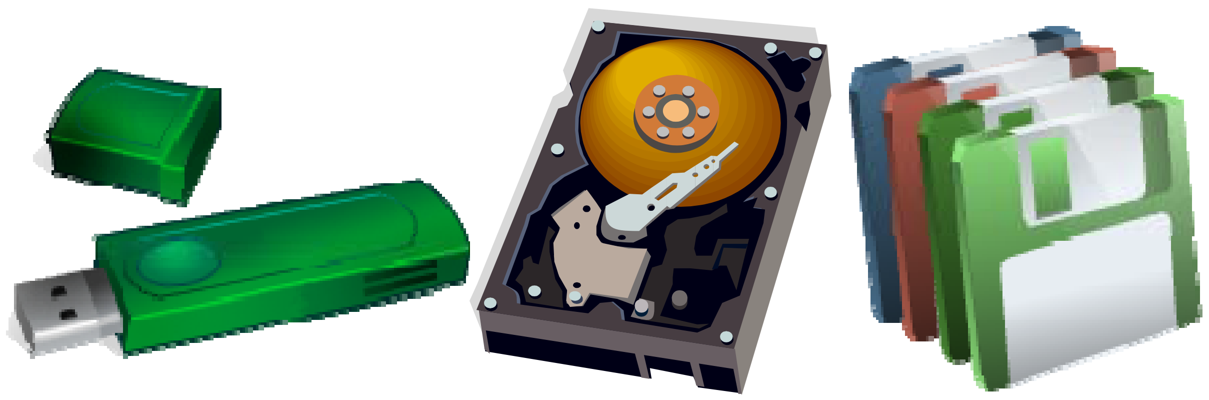
Chapter 11

Saving and Loading Information

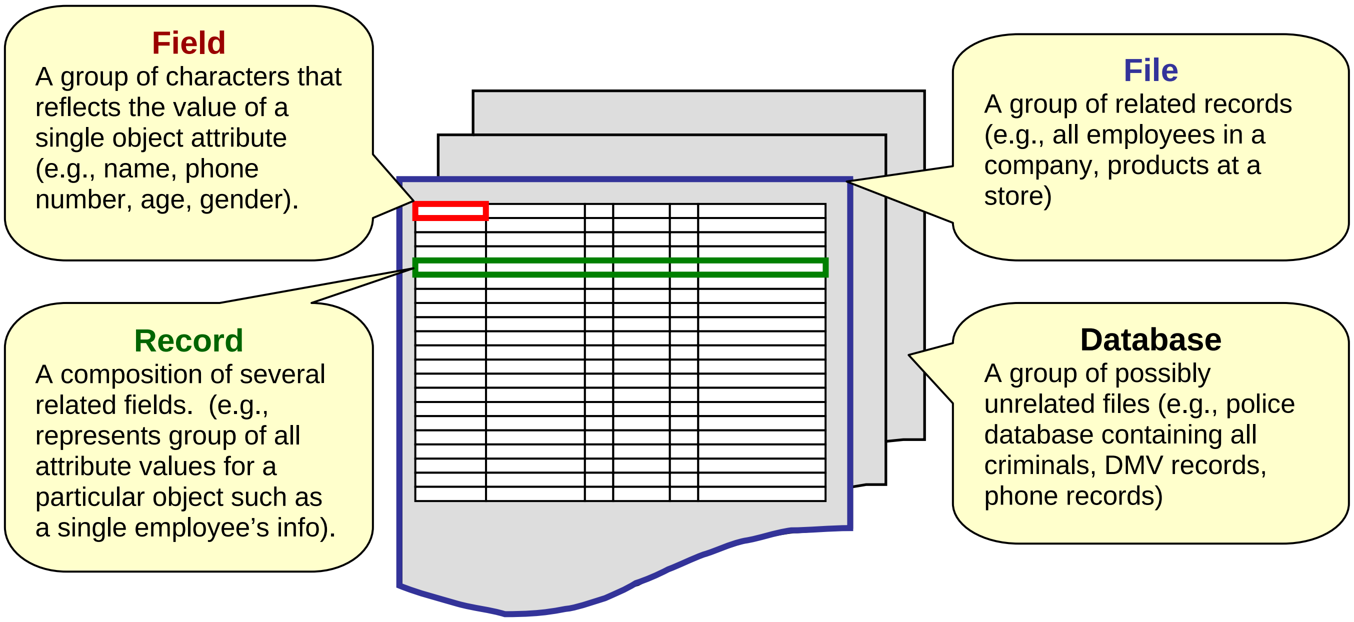
What is in This Chapter ?

In computer science, all data eventually gets stored onto storage devices such as hard drives, diskettes, USB flash drives, CDs, DVDs, etc... This set of notes explains how to save information from your program to a file that sits on one of these backup devices. It also discusses how to load that information back into your program. The saving/loading of data from files can be done using different formats. We discuss here the notion of **text** vs. **binary** formats. Note as well that the techniques presented here also apply to sending and receiving information from **Streams** (e.g., networks). We will look at the way in which **Stream** objects are used to do data I/O in JAVA. We will also look at how to use **ObjectStreams** to read/write entire objects easily and finally investigate the **File** class which is useful for querying files and folders on your computer.



11.1 Introduction to Files and Streams

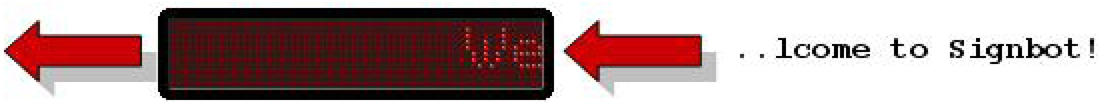
File processing is very important since eventually, all data must be stored externally from the machine so that it will not be erased when the power is turned off. Here are some of the terms related to file processing:


Field
A group of characters that reflects the value of a single object attribute (e.g., name, phone number, age, gender).
File
A group of related records (e.g., all employees in a company, products at a store)
Record
A composition of several related fields. (e.g., represents group of all attribute values for a particular object such as a single employee's info).
Database
A group of possibly unrelated files (e.g., police database containing all criminals, DMV records, phone records)

In JAVA, we can store information from our various objects by extracting their attributes and saving these to the file. To use a file, it must be first ***opened***. When done with a file, it MUST be ***closed***. We use the terms ***read*** to denote getting information *from* a file and ***write*** to denote saving information *to* a file. The contents of a file is ultimately reduced to a set of numbers from 0 to 255 called **bytes**.

In JAVA, files are represented as ***Stream*** objects. The idea is that data "streams" (or flows) to/from the file … similar to the idea of streaming video that you may have seen online. Streams are *objects* that allow us to send or receive information in the form of bytes. The information that is put into a stream, comes out in the same order.

It is similar to those scrolling signs where the letters scroll from right to left, spelling out a sentence:



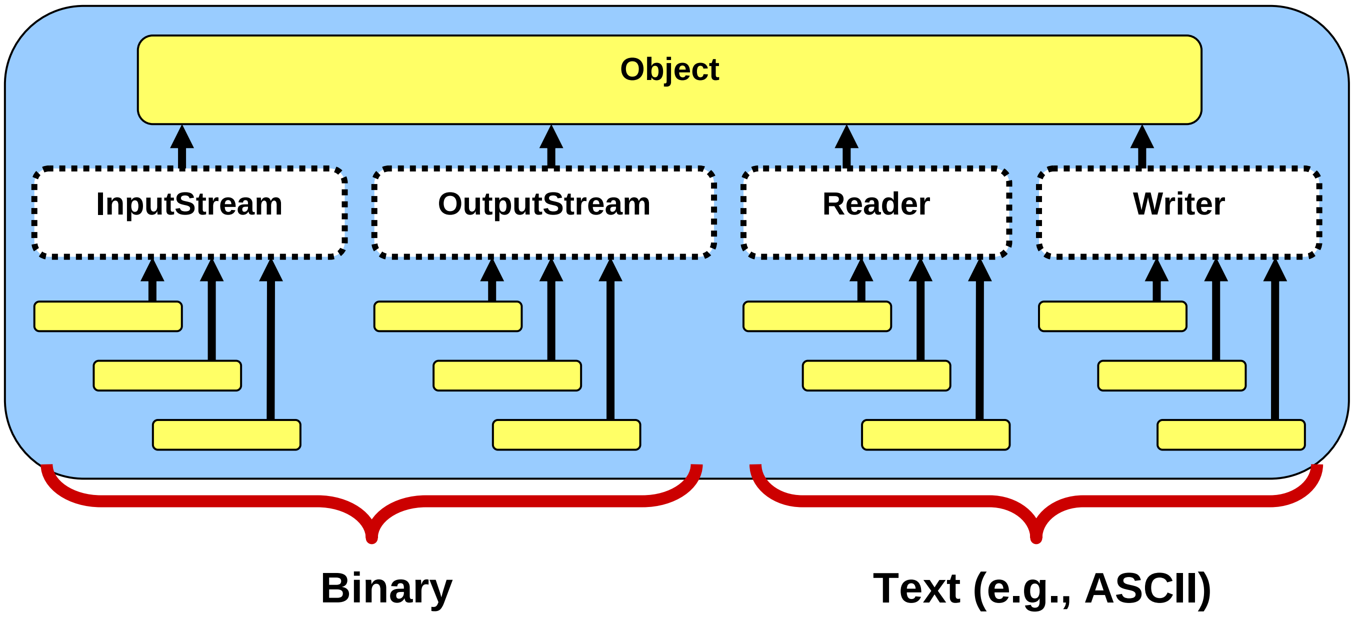
Streams are actually very general in that they provide a way to send or receive information to and from:

- files
- networks
- different programs
- any I/O devices (e.g., console and keyboard)

When we first start executing a JAVA program, 3 streams are automatically created:

```
•  System.in     // for inputting data from the keyboard
•  System.out    // for outputting data to the screen
•  System.err    // for outputting error messages to the screen
```

In fact, there are many stream-related classes in JAVA. We will look at a few and how they are used to do file I/O. The various **Streams** differ in the way that data is "entered into" and "extracted from" the stream. As with **Exceptions**, **Streams** are organized into different hierarchies. JAVA contains four main stream-related hierarchies for transferring data as *binary* bytes or as *text* bytes:



It is interesting to note that there is no common **Stream** class from which these main classes inherit. Instead, these 4 **abstract** classes are the root of more specific subclass hierarchies. A rather large number of classes are provided by JAVA to construct streams with the desired properties. We will examine just a few of the common ones here.

Typically I/O (i.e., input/output) is a bottleneck in many applications. That is, it is very time consuming to do I/O operations when compared to internal operations. For this reason, ***buffers*** are used. Buffered output allows data to be collected for output before it is actually sent to the output device. Only when the buffer gets full does the actual data get sent. This reduces the amount of actual output operations, but each output operation would usually send more data.

(Note: The **flush()** command can be sent to buffered streams in order to empty the buffer and cause the data to be sent "immediately" to the output device. Input data can also be buffered.)

By the way, what is **System.in** and **System.out** exactly ? We can determine their respective classes with the following code:

```
System.out.print("System.in is an instance of ");
System.out.println(System.in.getClass());
System.out.print("System.out is an instance of ");
System.out.println(System.out.getClass());
```

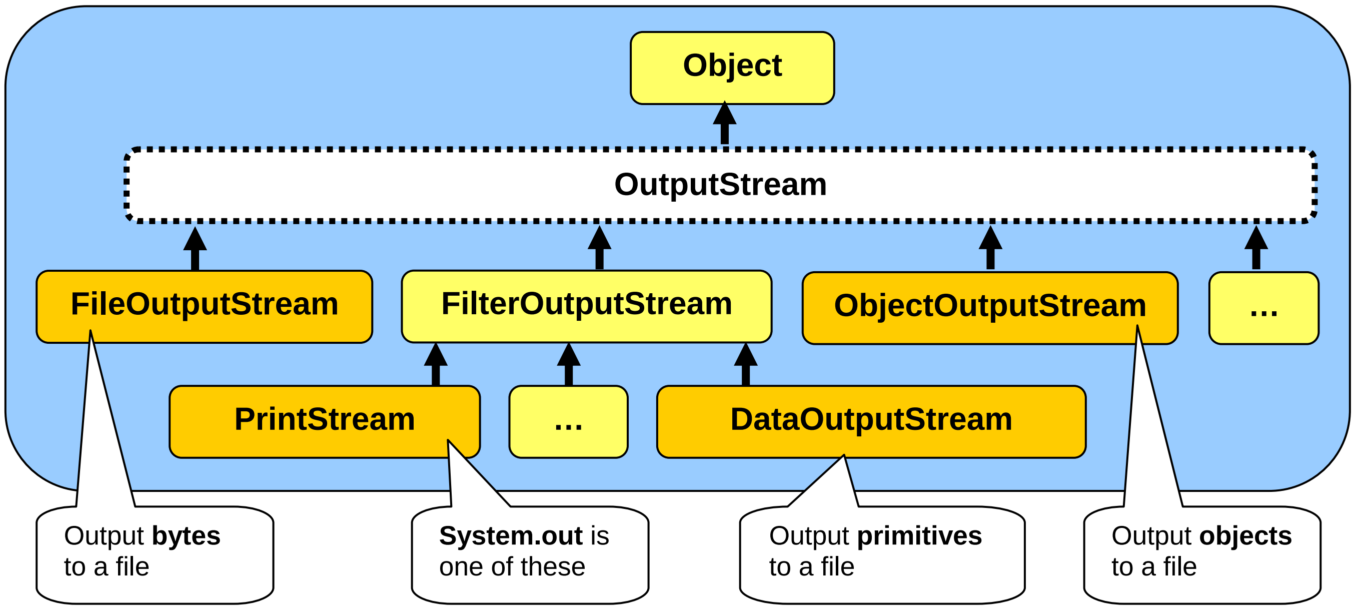
This code produces the following output:

```
System.in is an instance of class java.io.BufferedInputStream
System.out is an instance of class java.io.PrintStream
```

So we have been using these streams for displaying information and getting information from the user through the keyboard. We will now look at how the classes are arranged in the different stream sub-hierarchies.

11.2 Reading and Writing Binary Data

First, let us examine a portion of JAVA's **OutputStream** sub-hierarchy:



The streams in this sub-hierarchy are responsible for outputting binary data. That is, data which is in the form of bytes or data types. **OutputStreams** have a **write()** method that allows us to output a single byte of data at a time.

To open a file for binary writing, we can create an instance of **FileOutputStream** using one of the following constructors:

```
new FileOutputStream(String fileName);
new FileOutputStream(String fileName, boolean append);
```

The first constructor opens a new file output stream with that name. If one exists already with that name, it is overwritten (i.e., erased). The second constructor allows you to determine whether you want an existing file to be overwritten or appended to. If the file does not exist, a new one with the given name is created. If the file already exists prior to opening then the following rules apply:

- if append = **false** the existing file's contents is discarded and the file will be overwritten.
- if append = **true** the new data to be written to the file is appended to the end of the file.

We can output simple bytes to a **FileOutputStream** by using the **write()** method, which takes a single byte (i.e., a number from 0 to 255) as follows:

```
FileOutputStream out;

out = new FileOutputStream("myFile.dat");
out.write('H');
out.write(69);
out.write(76);
out.write('L');
out.write('O');
out.write('!');
out.close();
```

This code outputs the characters **HELLO!** to a file called **"myFile.dat"**. The file will be created (if not existing already) in the ***current*** directory/folder (i.e., the directory/folder that your JAVA program was run from). Alternatively, you can specify where to create the file by specifying the whole path name instead of just the file name as follows:

```
FileOutputStream out;
out = new FileOutputStream("F:\\My Documents\\myFile.dat");
```

Notice the use of "two" backslash characters within the String constant (because the backslash character is a special character which requires it to be preceded by a backslash ... just like **\n** is used to create a new line).

Using this strategy, we can output either characters or positive **integers** in the range from 0 to 255. Notice in the code that we closed the file stream when done. This is important to ensure that the operating system (e.g., Windows 7) releases the file handles correctly).

When working with files in this way, two exceptions may occur:

- opening a file for reading or writing may generate a **java.io.FileNotFoundException**
- reading or writing to/from a file may generate a **java.io.IOException**

You should handle these exceptions with appropriate **try**/**catch** blocks:

```
import java.io.*;

public class FileOutputStreamTestProgram {
    public static void main(String args[]) {
        try {
            FileOutputStream out;
            out = new FileOutputStream("myFile.dat");
            out.write('H');  out.write(69);
            out.write(76);   out.write('L');
            out.write('O');  out.write('!');
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

Since all streams are a part of the **java.io** package we need to import them.

The code above allows us to output any data as long as it is in **byte** format. This can be tedious. For example, if we have the integer 7293901 and we want to output it, we have a few choices:

- break up the integer into its 7 digits and output these digits one at a time (very tedious)
- output the 4 bytes corresponding to the integer itself (recall that an **int** is stored as 4 bytes)

Either way, these are not fun. Fortunately, JAVA provides a **DataOutputStream** class which allows us to output whole primitives (e.g., **ints**, **floats**, **doubles**) as well as whole **Strings** to a file! Here is an example of how to use it to output information from a **BankAccount** object …

```
import java.io.*;

public class DataOutputStreamTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount        aBankAccount;
            DataOutputStream   out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new DataOutputStream(new FileOutputStream("myAcc.dat"));
            out.writeUTF(aBankAccount.getOwner());
            out.writeInt(aBankAccount.getAccountNumber());
            out.writeFloat(aBankAccount.getBalance());
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

The **DataOutputStream** acts as a “wrapper” class around the **FileOutputStream**. It takes care of breaking our primitive data types and Strings into separate bytes to be sent to the **FileOutputStream**.

There are methods to write each of the primitives as well as **Strings**:

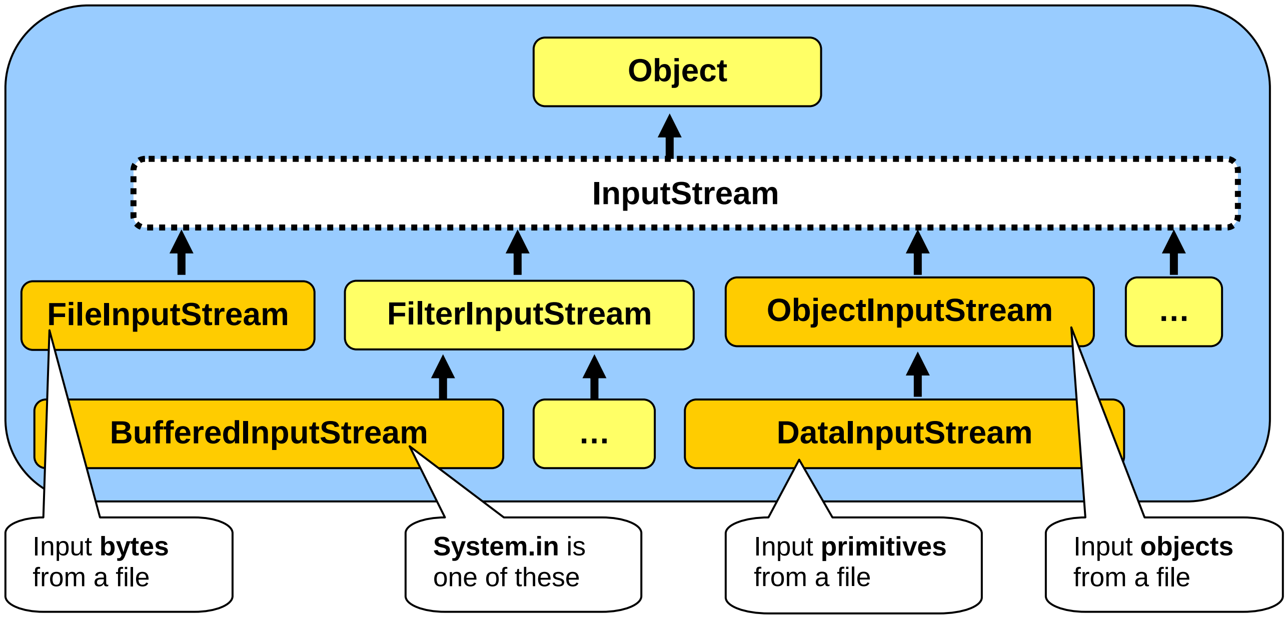
```
writeUTF(String aString)              writeShort(short aShort)
writeInt(int anInt)                   writeBoolean(boolean aBool)
writeFloat(float aFloat)              writeByte(int aByte)
writeLong(long aLong)                 writeChar(char aChar)
writeDouble(double aDouble)
```

The output from a **DataOutputStream** is not very nice to look at (i.e., it is in binary format). The **myAcc.dat** file would display as follows if we tried to view it with Windows Notepad:

```
Rob Banks   † BÈ
```

This is the binary representation of the data, which usually takes up less space than text files. The disadvantage of course, is that we cannot make sense of the data if we try to read it with our eyes as text. However, rest assured that the data is saved properly.

Let us now examine how we could read that information back in from the file with a different program. To start, we need to take a look at the **InputStream** sub-hierarchy as follows …



Notice that it is quite similar to the **OutputStream** hierarchy. In fact, its usage is also very similar. We can read back in the byte data from our file by using **FileInputStream** now as follows:

```
import java.io.*;

public class FileInputStreamTestProgram {
    public static void main(String args[]) {
        try {
            FileInputStream in = new FileInputStream("myFile.dat");
            while(in.available() > 0)
                System.out.print(in.read() + " ");
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (EOFException e) {
            System.out.println("Error: EOF encountered, file may be corrupt");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Notice that we now use **read()** to read in a single byte from the file. Notice as well that we can use the **available()** method which returns the number of bytes available to be read in from the file (i.e., the file size minus the number of byte as already read in).

Also, notice now that we are forced to handle an **EOFException** which can occur if the file is corrupted and the end of the file character is reached before the number of available bytes has been read in.

Recall that the order in which you catch your exceptions is important! Catch the most specific ones first. Since **IOException** is a superclass of **EOFException**, it must go afterwards.

The code reads the data back in from our file (i.e., the characters HELLO!) and outputs their ASCII (i.e., byte) values to the console:

```
72 69 76 76 79 33
```

Try changing **in.read()** to **(char)in.read()** (i.e., type-cast the byte to a char) and see what happens...

That was fairly simple ... but what about getting back those primitives ? You guessed it! We will use **DataInputStream**:

```java
import java.io.*;

public class DataInputStreamTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount       aBankAccount;
            DataInputStream   in;

            in = new DataInputStream(new FileInputStream("myAccount.dat"));

            String name = in.readUTF();
            int      acc = in.readInt();
            float    bal = in.readFloat();

            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (EOFException e) {
            System.out.println("Error: EOF encountered, file may be corrupt");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Notice that we re-create a new **BankAccount** object and "fill-it-in" with the incoming file data. Note, that in order for the above code to compile, we would need to write a public constructor for the **BankAccount** class that takes an owner name, balance and account number (i.e., previously, in our **BankAccount** class, we had no way of specifying the **accountNumber** since it was set automatically) …

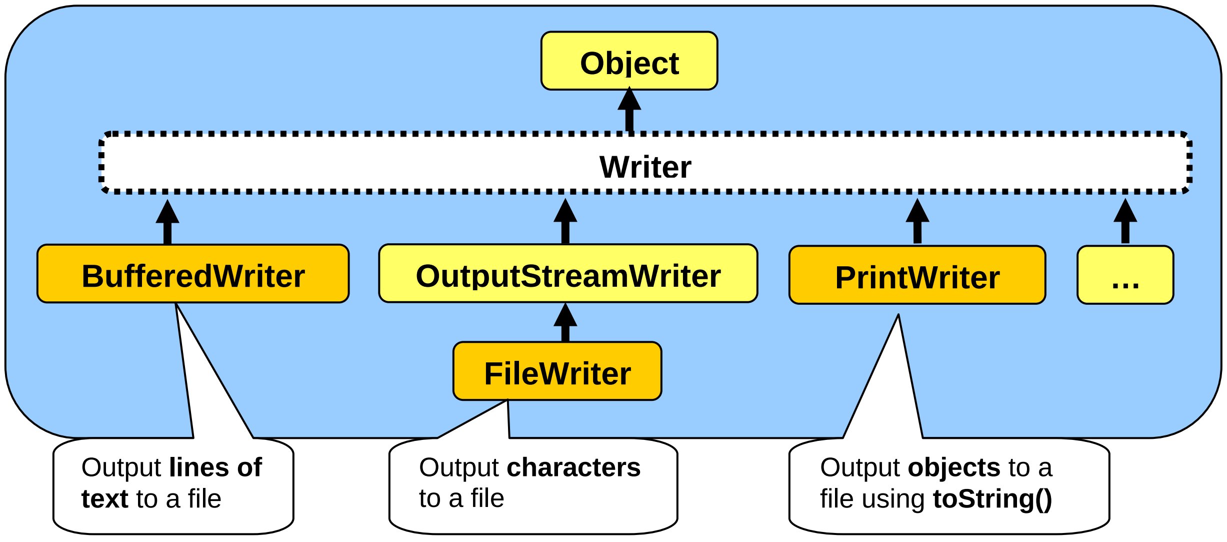
```
BankAccount(String initName, float initBal, int num) {
    ownerName = initName;
    accountNumber = num;
    balance = initBal;
}
```

As with the output streams, there are methods to read in the other primitives:

```
String readUTF()                          short readShort()
int readInt()                             boolean readBoolean()
float readFloat()                         int readByte()
long readLong()                           char readChar()
double readDouble()
```

11.3 Reading and Writing Text Data

Here is the **Writer** class sub-hierarchy which is used for writing **text** data to a stream:


Object
Writer
BufferedWriter
OutputStreamWriter
PrintWriter
…
FileWriter
Output lines of text to a file
Output characters to a file
Output objects to a file using toString()

Notice that there are 3 main classes we will use for writing **characters**, **lines of characters** and general **objects** to a text file. When objects are written to the text file, the **toString()** method for the object is called and the resulting **String** is saved to the file.

We can output (in text format) to a file using simply the **print()** or **println()** methods with the **PrintWriter** class as follows …

```
import java.io.*;

public class PrintWriterTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount  aBankAccount;
            PrintWriter  out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new PrintWriter(new FileWriter("myAccount2.dat"));

            out.println(aBankAccount.getOwner());
            out.println(aBankAccount.getAccountNumber());
            out.println(aBankAccount.getBalance());
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

Wow! Outputting text to a file is as easy as outputting it to the console window ! But what does it look like ? If we opened the file with Windows Notepad, we would notice that the result is a “pleasant looking” text format:

```
Rob Banks
100000
100.0
```

In fact, we can actually write any object using the **println()** method. JAVA will use that object's **toString()** method. So if we replaced this code:

```
    out.println(aBankAccount.getOwner());
    out.println(aBankAccount.getAccountNumber());
    out.println(aBankAccount.getBalance());
```

with this code:

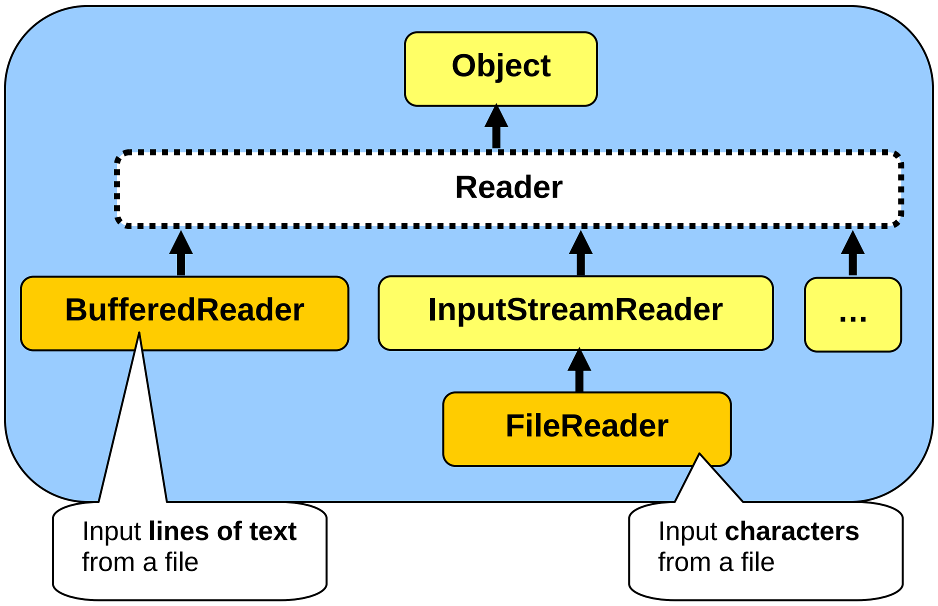
```
    out.println(aBankAccount);
```

we would end up with the following saved to the file:

```
Account #100000 with $100.0
```

So it actually does behave just like the **System.out** console. We would need to be careful though, because you will notice that the BankAccount's **toString()** method in the example above did not display the owner's name. So the file does not record that owner's name and therefore we could never read that name back in again … it would be lost forever. Notice as well how the **PrintWriter** wraps the **FileWriter** class just as the **DataOutputStream** wrapped the **FileOutputStream**.

It is also easy to read back in the information that was saved to a text file. Here is the hierarchy of classes for reading text files →


Object
Reader
BufferedReader
InputStreamReader
…
FileReader
Input **lines of text** from a file
Input **characters** from a file

Notice that we can only read in *characters* and *lines of characters* from the text file, but NOT general objects. We will see later how we can re-build read-in objects.

Most of the time, we will make use of the **BufferedReader** class by using the **readLine()** method as follows:

```
import java.io.*;

public class BufferedReaderTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount      aBankAccount;
            BufferedReader   in;

            in  = new BufferedReader(new FileReader("myAccount2.dat"));
            String name = in.readLine();
            int     acc = Integer.parseInt(in.readLine());
            float   bal = Float.parseFloat(in.readLine());

            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (EOFException e) {
            System.out.println("Error: EOF encountered, file may be corrupt");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Note the use of "primitive data type" wrapper classes to read data types. We could have used the **Scanner** class here to simplify the code:

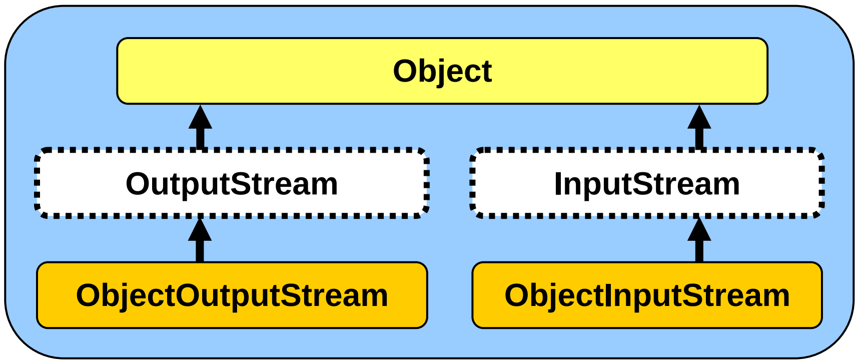
```
import java.io.*;
import java.util.*;   // Needed for use of Scanner and NoSuchElementException

public class BufferedReaderTestProgram2 {
    public static void main(String args[]) {
        try {
            BankAccount  aBankAccount;

            Scanner in = new Scanner(new FileReader("myAccount2.dat"));
            String name = in.nextLine();
            int     acc = in.nextInt();
            float   bal = in.nextFloat();
            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (NoSuchElementException e) {
            System.out.println("Error: EOF encountered, file may be corrupt");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Notice here that we now catch a **NoSuchElementException** instead of an **EOFException**. This is how the **Scanner** detects the end of the file. The main advantage of using this **Scanner** class is that we do not have to use any wrapper classes to convert the input strings to primitives.

11.4 Reading and Writing Whole Objects

So far, we have seen ways of saving and loading bytes and characters to a file. Also, we have seen how **DataOutputStream**/**DataInputStream** and **PrintWriter**/**BufferedReader** classes can make our life simpler since they deal with larger (more manageable) chunks of data such as primitives and Strings. We also looked at how we can save a whole object (i.e., a **BankAccount**) to a file by extracting its attributes and saving them individually.

Now we will look at an even simpler way to save/load a whole object to/from a file using the **ObjectInputStream** and **ObjectOutputStream** classes:



These classes allow us to save or load entire JAVA objects with one method call, instead of having to break apart the object into its attributes. Here is how we do it:

```
import java.io.*;

public class ObjectOutputStreamTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount          aBankAccount;
            ObjectOutputStream   out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new ObjectOutputStream(new FileOutputStream("myAcc.dat"));
            out.writeObject(aBankAccount);
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

Wow! It is VERY easy to write out an object. We simply supply the object that we want to save to the file as a parameter to the **writeObject()** method. Notice that the **ObjectOutputStream** class is a wrapper around the **FileOutputStream**. That is because ultimately, the object is reduced to a set of bytes by the **writeObject()** method, which are then saved to the file. The process of breaking down the object into bytes is called ***serialization***. Thus, when an object is saved to a file, it is automatically de-constructed into bytes, these bytes are then saved to a file, and then the bytes are read back in later and the object is re-constructed again. This is all done automatically by JAVA, so we don't have to be too concerned about it.

In order to be able to save an object to a file using the **ObjectOutputStream**, the object must be ***serializable*** (i.e., able to be serialized…or reduced to a set of bytes). To do this, we need to inform JAVA that our object implements the **java.io.Serializable** interface as follows …

```
public class BankAccount implements java.io.Serializable {
     ...
}
```

This particular interface does not actually have any methods within it that we need to implement. Instead, it merely acts as a “flag” that indicates your permission for this object to be serialized. It allows a measure of security for our objects (i.e., only **serializable** objects are able to be broken down into bytes and sent to files or over the network).

Most standard JAVA classes are **serializable** by default and so they can be saved/loaded to/from a file in this manner. When allowing our own objects to be serialized, we must make sure that all of the “pieces” of the object are also **serializable**. For example, assume that our **BankAccount** is defined as follows:

```
public class BankAccount implements java.io.Serializable {
     Customer   owner;
     float      balance;
     int        accountNumber;
     ...
}
```

In this case, since owner is not a **String** but a **Customer** object, then we must make sure that **Customer** is also **Serializable**:

```
public class Customer implements java.io.Serializable
{
    ...
}
```

We would need to then check whether **Customer** itself uses other objects and ensure that they too are **serializable** … and so on. To understand this, just think of a meat grinder. If some hard marbles were placed within out meat, we cannot expect it to come out through the grinder since they cannot be reduced to a smaller form. Similarly, if we have any non-**serializable** objects in our original object, we cannot properly serialize the object.

So what does a serialized object look like anyway ? Here is what the file would look like from our previous example if opened in Windows Notepad:

```
¬í ⏐sr BankAccount“ÈSòñ┤úä┐ └I accountNumberF •balanceL ⏐ownert
↕Ljava/lang/String;xp  † BÈ  t     Rob Banks
```

Weird … it seems to be a mix of binary and text. As it turns out, JAVA saves all the attribute information for the object, including their types and values, as well as some other information. It does this in order to be able to re-create the object when it is read back in.

The object can be read back in by using the **readObject()** method in the **ObjectInputStream** class as follows:

```
import java.io.*;

public class ObjectInputStreamTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount          aBankAccount;
            ObjectInputStream    in;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            in = new ObjectInputStream(new FileInputStream("myAcc.dat"));
            aBankAccount = (BankAccount)in.readObject();

            System.out.println(aBankAccount);
            in.close();

        } catch (ClassNotFoundException e) {
            System.out.println("Error: Object'c class does not match");
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Note, that the **ObjectInputStream** wraps the **FileInputStream**. Also, notice that once read in, the object must be *type-casted* to the appropriate type (in this case **BankAccount**). Also, if there is any problem trying to re-create the object according to the type of object that we are loading, then a **ClassNotFoundException** may be generated, so we have to handle it. Finally, in order for this to work, you must also make sure that your object (i.e., **BankAccount**) has a zero-parameter constructor, otherwise an **IOException** will occur when JAVA tries to rebuild the object. Although not shown in our example above, you may also make use of the **available()** method to determine whether or not the end of the file has been reached.

Although this method is extremely easy to use, there is a potentially disastrous disadvantage. The object that is saved to the file using this strategy is actually saved in binary format which depends on the class name, the object's attribute types and names as well as the method signatures and their names. So if you change the class definition after it has been saved to the file, it may not be able to be read back in again !!! Some changes to the class do not cause problems such as adding an attribute or changing its access modifiers.

So as a warning, when saving objects to a file using this strategy, you should always keep a backed-up version of all of your code so that you will be able to read these files with this backed-up code in the future.

Supplemental Information (Disguising Serialized Data)

You can actually write your own methods for serializing your objects. One reason for doing this may be to encrypt some information beforehand (such as a password). You can decide which parts of the object will be serialized and which parts will not. You can declare any object attribute as being ***transient*** (which means that it will not be serialized) as follows:

```
    transient String password;
```

This will tell JAVA that you do not want the password saved automatically upon serialization. That way you can write your own method to encrypt it before it is serialized.

To do this, you would need to write two methods called **writeObject(ObjectOutputStream)** and **readObject(ObjectInputStream)**. These methods will automatically be called by JAVA upon serialization and they override the default writing behavior. In fact, there are **defaultWriteObject()** and **defaultReadObject()** methods which do the default serialization behavior (i.e., the serializing before you decided to do your own). Here are examples of what you can do:

```
    void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        // ... do extra stuff here to append to end of file
        out.writeObject(myField.encrypt());
    }
    void readObject(ObjectInputStream in) throws IOException,
                                             ClassNotFoundException {
        in.defaultReadObject();
        // ... do extra stuff here to read from end of file
        myField = ((myFieldType)in.readObject()).decrypt();
    }
```

11.5 Saving and Loading the Autoshow

Let us now consider a real example that shows how to save and load information from the **Autoshow** example that we implemented earlier in the course. We will save the autoshow's information in a text file so that we can print it out or read it easily. So, we will be using the **PrintWriter** and **BufferedReader** classes. We need to decide how to format the text in the file.

The **Autoshow** class had these attributes:

```
public class Autoshow {
    String            name;
    ArrayList<Car>    cars;
    ...
}
```

where as the **Car** object had these attributes:

```
public class Car {
    String      make;
    String      model;
    String      color;
    int         topSpeed;
    boolean     has4Doors;
      ...
}
```

So, to save the autoshow, we need to save the name of the autoshow as well as the individual cars in the show. For each car, the file should show the **make**, **model**, **color**, **topSpeed** and **has4Doors**. Perhaps we will separate the cars by a blank line to indicate that one car's data ends and another's begins as follows:

```
AutoRama 2009

Porsche
959
Red
240
false

Pontiac
Grand-Am
White
160
true

Ford
Mustang
White
230
false

…
```

This seems like a reasonable way to save the autoshow so that the data is readable in a text program. You will notice that each **Car** object is saved the same way. Hence, it would be good to start by writing some methods that can save/load **Car** objects.

We can write the following method in the **Car** class to begin…

```
public void saveTo(PrintWriter aFile) {
    ...
}
```

Notice that the method will take a single parameter which is a **PrintWriter** object to represent the file that we are saving to. Where does this file come from ? It actually does not matter. When writing this method, we should just assume that someone opened a file and handed it to us and now it is our job to write the **Car** information to the file specified through this incoming parameter. Note as well, that since we did not open the file (i.e., the **PrintWriter** was handed to us), we should also not close the file. It is the opener's responsibility to close it.

So, now how do we write the **Car** information to the file ? We simply do it as if we were writing to the **System** console:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(make);
    aFile.println(model);
    aFile.println(color);
    aFile.println(topSpeed);
    aFile.println(has4Doors);
}
```

That was easy. Remember though, that in order for JAVA to recognize the **PrintWriter** object, we will need to import **java.io.PrintWriter** at the top of our **Car** class. In fact, as you will see soon, we will need more classes from the **java.io** package, so it would be best to simply **import java.io.*;**

The method for loading a **Car** back in from the file is also quite easy. Again, it should read from a file (i.e., a **BufferedReader** object) that is passed in as a parameter, not a file that ***we*** open or close. Then all we need to do is to read the information from the file. But what do we "do" with the information once it has been read in ? Probably, we return it from the method so that whoever called this "load" method can decide what to do with the loaded car information. So, the method should probably return a **Car** object. Here is the method that we will write:

```
public static Car loadFrom(BufferedReader aFile) {
     ...
}
```

Notice that the method is **static**. This is not required, but it allows us to call the method as follows:

```
     Car c = Car.loadFrom(aFile);
```

instead of doing this:

```
     Car c = new Car().loadFrom(aFile);
```

That is the only difference. The **static** version is more logical. So … now … what goes into the method ? Well, we need to at least create and return a new **Car** object, so we can start with that:

```
public static Car loadFrom(BufferedReader aFile) {
    Car  loadedCar = new Car();
    // ...
    return loadedCar;
}
```

To read in the car, we should recall that we use **readLine()** to read a single line of text from a **BufferedReader** file. We need to read the lines of text in the same order that they were outputted (i.e., **make**, **model**, **color**, **topSpeed** and then **has4Doors**). Then we can set the attributes of the **Car** to the data that was read in. Here is the code:

```
public static Car loadFrom(BufferedReader aFile) throws IOException {
    Car  loadedCar = new Car();

    loadedCar.make = aFile.readLine();
    loadedCar.model = aFile.readLine();
    loadedCar.color = aFile.readLine();
    loadedCar.topSpeed = Integer.parseInt(aFile.readLine());
    loadedCar.has4Doors = Boolean.parseBoolean(aFile.readLine());

    return loadedCar;
}
```

Notice that the method may throw an **IOException** (due to the fact that JAVA's **readLine()** method declares that it throws an **IOException**). We could have caught the exception here and handled it. However, since this method is just a helper method in a larger application, we are unsure what to do here if an error occurs. Therefore, by declaring that this method throws an **IOException**, we will be forced to handle that exception from the place where we call this **loadFrom()** method. Also notice that we are calling the zero-parameter constructor for the Car here … we would need to make sure that such a constructor is available.

Now we will write some test code to see if it works. Notice in the following code how we separated the write and read tests …

```
import java.io.*;

public class CarSaveLoadTestProgram {
    private static void writeTest() throws IOException {
        PrintWriter   file1, file2;
        Car           car1, car2;

        file1 = new PrintWriter(new FileWriter("car1.txt"));
        file2 = new PrintWriter(new FileWriter("car2.txt"));
        car1 = new Car("Pontiac", "Grand-Am", "White", 160, true);
        car2 = new Car("Ford", "Mustang", "White", 230, false);
        car1.saveTo(file1);
        car2.saveTo(file2);
        file1.close();
        file2.close();
    }

    private static void readTest() throws IOException {
        BufferedReader     file1, file2;
        Car                car1, car2;

        file1 = new BufferedReader(new FileReader("car1.txt"));
        file2 = new BufferedReader(new FileReader("car2.txt"));
        car1 = Car.loadFrom(file1);
        car2 = Car.loadFrom(file2);
        System.out.println(car1);
        System.out.println(car2);
        file1.close();
        file2.close();
    }

    public static void main(String args[]) throws IOException {
        writeTest();
        readTest();
    }
}
```

Notice that we simply ignored handling any **IOExceptions** by declaring that the test methods and the **main()** method all throw the **IOException**. If we now look at the "car1.txt" and "car2.txt" files, we see that it seems to have saved properly:

car1.txt looks like this:

```
Pontiac
Grand-Am
White
160
true
```

car2.txt looks like this:

```
Ford
Mustang
White
230
false
```

Now for the fun part. Lets make this work with the **Autoshow**. We will make **saveTo()** and **loadFrom()** methods in the **Autoshow** class as well. This time, we need to save ALL the **Car** objects from the autoshow's **cars** list.

Recall that we need to first save the autoshow's name to the file:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(name);
    ...
}
```

Now we can iterate through the cars and save them one by one, leaving a blank line in between each, to make the file more readable:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(name);
    for (Car c:  cars) {
        aFile.println();    // Leave a blank line before writing the next one
        c.saveTo(aFile);
    }
}
```

Notice that we are making use of the **Car** class's **saveTo()** method. This is important, since it makes good use of pre-existing code and is more modular. Again, we should **import java.io.*** at the top of our **Autoshow** class.

The method for loading an **Autoshow** from the file is also quite easy. It should create and return an **Autoshow** object whose name is the name that is the first line of the file. We will make it a **static** method as well:

```
public static Autoshow loadFrom(BufferedReader aFile) throws IOException {
    Autoshow aShow = new Autoshow(aFile.readLine());
    ...
    return aShow;
}
```

Again, we will need to make sure that a zero-parameter constructor is available in the **Autoshow** class. Now we now need to read in the name at the top of the file and then read in each car individually. How do we know how many cars to read ? Well, we can simply read until there are no more cars left. The **ready()** method in the **BufferedReader** class returns **true** as long as there is another line to be read in the file, otherwise it returns **false**. We can simply keep reading in cars until we get a **!ready()** as follows …

```
public static Autoshow loadFrom(BufferedReader aFile) throws IOException {
    Autoshow aShow = new Autoshow(aFile.readLine());

    while (aFile.ready()) { //read until no more available (i.e., not ready)
        aFile.readLine(); //read the blank line
        aShow.getCars().add(Car.loadFrom(aFile));//read & add the car
    }
    return aShow;
}
```

Notice that each time we load a new car from the file, we must not forget to add it to the new autoshow object's **cars** collection. Here is a method for testing out our saving and loading:

```
import java.io.*;
public class AutoshowSaveLoadTestProgram {
    // This method tests the writing of an autoshow to a file
    private static void writeTest() throws IOException {
        // First make an Autoshow and add lots of cars to the show
        Autoshow  show = new Autoshow("AutoRama 2009");
        show.getCars().add(new Car("Porsche", "959", "Red", 240, false));
        show.getCars().add(new Car("Pontiac", "Grand-Am", "White", 160, true));
        show.getCars().add(new Car("Ford", "Mustang", "White", 230, false));
        show.getCars().add(new Car("Volkswagon", "Beetle", "Blue", 140, false));
        show.getCars().add(new Car("Volkswagon", "Jetta", "Silver", 180, true));
        show.getCars().add(new Car("Geo", "Storm", "Yellow", 110, true));
        show.getCars().add(new Car("Toyota", "MR2", "Black", 220, false));
        show.getCars().add(new Car("Ford", "Escort", "Yellow", 10, true));
        show.getCars().add(new Car("Honda", "Civic", "Black", 220, true));
        show.getCars().add(new Car("Nissan", "Altima", "Silver", 180, true));
        show.getCars().add(new Car("BMW", "5", "Gold", 260, true));
        show.getCars().add(new Car("Prelude", "Honda", "White", 90, false));
        show.getCars().add(new Car("Mazda", "RX7", "Red", 240, false));
        show.getCars().add(new Car("Mazda", "MX6", "Green", 160, true));
        show.getCars().add(new Car("Pontiac", "G6", "Black", 140, false));

        // Now open the file and save the autoshow
        PrintWriter     aFile;
        aFile = new PrintWriter(new FileWriter("autoshow.txt"));
        show.saveTo(aFile);
        aFile.close();
    }

    // This method tests the reading of an autoshow from a file
    private static void readTest() throws IOException {
        BufferedReader  aFile;

        aFile = new BufferedReader(new FileReader("autoshow.txt"));
        Autoshow aShow = Autoshow.loadFrom(aFile);
        aShow.printByMake();
        aFile.close();
    }

    public static void main(String args[]) throws IOException {
        writeTest();    // Write an autoshow to the file
        readTest();     // Read an autoshow from the file
    }
}
```

From running the test, we can see that the **Autoshow** does indeed load properly.

Storing Data on a Single Line:

What if we want to store the **Car** data on a single line as follows:

```
Porsche 959 Red 240 false
Pontiac Grand-Am White 160 true
Ford Mustang White 230 false
```

Well, saving a **Car** is easy:

```java
public void saveTo(PrintWriter aFile) {
    aFile.println(make + " " + model + " " + color + " " +
                  topSpeed + " " + has4Doors);
}
```

For reading however, we will need to alter the load method to use a **StringTokenizer** to extract the pieces:

```java
public static Car loadFrom(BufferedReader aFile) throws IOException {
    Car  loadedCar = new Car();

    StringTokenizer wholeLine = new StringTokenizer(aFile.readLine());
    loadedCar.make = wholeLine.nextToken();
    loadedCar.model = wholeLine.nextToken();
    loadedCar.color = wholeLine.nextToken();
    loadedCar.topSpeed = Integer.parseInt(wholeLine.nextToken());
    loadedCar.has4Doors = Boolean.parseBoolean(wholeLine.nextToken());

    return loadedCar;
}
```

The methods for saving and loading the **Autoshow** would not change much, except that the blank line need not be written nor read in after each **Car**.

But what if the **Car** make has two words like this ?

```
PT Cruiser Chrysler Silver 120 true
```

Now its tougher since the name requires two tokens, not one. We can save and load using commas as our delimiters and then extract the pieces of data one at a time …

```
PT Cruiser,Chrysler,Silver,120,true
```

This solves the problem.

11.6 The File Class

The **File** class allows us to retrieve information about a file or folder on our computer. However, it has nothing to do with reading and writing information to and from the files.

To make a **File** object, there are three commonly used constructors:

```
File file1 = new File("C:\\Data\\myFile.dat");
File file2 = new File("C:\\Data\\", "myFile.dat");
File file3 = new File(new File("."), "myFile.dat");
```

In the first constructor, we supply the entire file name as a string which includes the path to the file. (A ***path*** is a sequence of folders on the computer that lead to the file … starting with the root drive letter).

In the second constructor, we can supply the pathname as a separate string from the file name. The third constructor actually uses another **File** object as a parameter which must represent a folder/directory on the computer. The **"."** as a filename indicates the current directory/folder. The **".."** as a filename indicates the directory/folder above the current directory/folder. Alternatively we could supply any path name here.

Once we create this **File** object, there are a set of methods that we can use to ask questions about this file or folder. Here are just some of the available methods:

`boolean canRead()`
Returns whether or not this file is readable.

`boolean canWrite()`
Returns whether or not this file is writable.

`boolean exists()`
Returns whether or not this file or directory exists in the specified path.

`boolean isFile()`
Returns whether or not this represents a file (as opposed to a directory/folder).

`boolean isDirectory()`
Returns whether or not this represents a directory/folder (as opposed to a file).

`boolean isAbsolute()`
Returns whether or not this represents an absolute path to a file or directory.

Here are other methods for accessing components (i.e., filenames and pathnames) of a file or directory:

`String getAbsolutePath()`
Return a String with the absolute path of the file or directory.

`String getName()`
Return a String with the name of the file or directory.

`String getPath()`
Return a String with the path of the file or directory.

`String getParent()`
Return a String with the parent directory of the file or directory.

Here are some other user useful methods:

`long length()`
Return the length of the file in bytes. If the **File** object is a directory, return 0.

`long lastModified()`
Return a system-dependent representation of the time at which the file or directory was last modified. The value returned is only useful for comparison with other values returned by this method.

`String[] list()`
Return an array of Strings representing the contents of a directory.

For the purpose of demonstration, here is a program that gives a directory listing of the files and folders on the root of your **C:** drive, but it does not go into each folder recursively …

```
import java.io.*;

public class FileClassTestProgram {
    public static void main(String args[]) {
        // The dot means the current directory
        File currDir = new File("C:\\");
        System.out.println("The directory name is: " + currDir.getName());
        System.out.println("The path name is: " + currDir.getPath());
        System.out.println("The actual path name is: " +
                                        currDir.getAbsolutePath());

        System.out.println("Here are the files in the current directory: ");
        String[]  files = currDir.list();
        for (int i=0; i<files.length; i++) {
            if (new File("C:\\", files[i]).isDirectory())
                 System.out.print("*** ");
            System.out.println(files[i]);
        }
    }
}
```

Here is the output (which differs of course depending where you run your code from):

```
The directory name is:
The path name is: C:\
The actual path name is: C:\
Here are the files in the current directory:
*** 1eceb6cd306883b5737c1dbf5404e4
a-1049-1-7C23.zip
AUTOEXEC.BAT
BOOT.BAK
boot.ini
*** Config.Msi
CONFIG.SYS
*** CtDriverInstTemp
*** Documents and Settings
*** Downloaded Videos
*** Downloads
*** drivers
hiberfil.sys
*** i386
INFCACHE.1
IO.SYS
IPH.PH
*** java

… etc …

*** System Volume Information
*** temp
*** TempArchive
*** Users
*** WINDOWS
```

File Separators:

Depending on which type of computer you have, folders are specified in different ways. For example, windows uses a '\' character to separate folders in a pathname, whereas Unix/Linux uses '/' and Classic Mac OS uses ":".

If we were to hard-code out pathnames into our programs, then our code would not be portable to different machines. For example, this pathname:

```
String fileName = "C:\\FunInc\\models\\BankAccount.java";
```

would be ok for a windows-based machine, but for a Unix/Linux machine, the pathname would be invalid. We would need to use something like this for Linux:

```
String fileName = "usr/FunInc/models/BankAccount.java";
```

… and further…something like this for Mac OS:

```
String fileName = "C:FunInc:models:BankAccount.java";
```

In order to make our code portable, JAVA has defined a **static** constant called ***separator*** in the **File** class which will represent the appropriate file separation character depending on the machine that our code is running on. Hence, the following code will be portable for all machines:

```
String fileName = File.separator + "FunInc" + File.separator +
                  "models" + File.separator + "BankAccount.java";
```

If you do this in your programs, your code will always be portable and you will save time when porting your code to other machines.