COMP 14, Fall 2000

Prasun Dewan[1]

16. Files

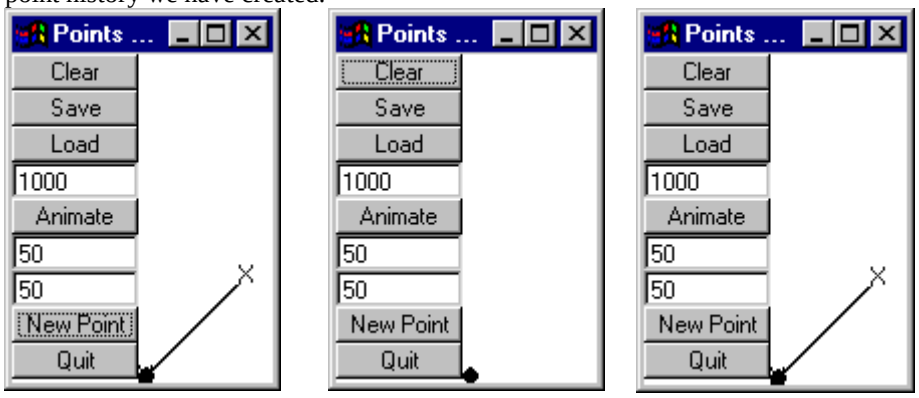
As a user of a computer, you are already familiar with files. When you write a document using an editor, you save it to a file, and when you want to modify the document, you load the document into the editor's memory, make modifications, and then save it back to a file.

We will now look at a file from the programmer's point of view, that is, how we can write programs that can save data to files or load data from files. We will study two kinds of files – binary files and text files.

In theory, files are complex, but in practice, they are easy to use. Like the use of toolkits, the use of files follows a well-defined pattern. Once we have seen the pattern in a file-based program, programming with files, specially binary files, is straightforward. The point history example will be used to illustrate this pattern.

Loading and Saving

Let us add to the point history example three new commands: clear, load, and save. The save command writes the list of points to a new file specified by the user; the load command replaces the current list of points with the one saved in a file, and the clear command empties the list of points. To illustrate how they work, consider Figure 1(a), which shows a point history we have created.



(a) Saving Point History (b) Clearing Point History (c) Loading Saved PointHistory

Figure 1 Persistent Point History

We can preserve our work by executing the save command, specifying a file to hold the data. Later, we can execute the clear command to empty the point history (Figure 1(b)) and perhaps create another point history and save it. Now if we decide we wish to view the original list of points, we can execute the load command, specifying the file in which it was saved (Figure 1(c)).

To write programs such as these that manipulate files, we must understand what a file means from a programs point of view.

File Properties

To a program, a file is yet another type of data structure that is manipulated as an object. However, it has several properties that distinguish it from other data structures such as arrays and vectors:

- *Persistent*: A file does not get destroyed when we quit some program or shut the machine - it continues to persist. Other data structures we have seen so far do not have this property. For instance, the vector created to store a point history gets destroyed when we quit the program or if the machine crashes while the program is running. The reason it can persist is that, unlike all other data structures we have seen so far, it resides on a persistent store such as a diskette or hard disk rather than memory.
- *Large*: Since persistent store can hold much more data than memory, files can be much larger than an in-memory data structure such as an array or a vector.
- *Dynamic*: Unlike an array but like a vector, a file may have a variable number of elements.
- *Shareable*: A file can be accessed by different invocations of the same or different program. For instance, we can start an editor, use it to create a document, write the document to ta file, and then terminate the program. When run the editor again, the new execution of the editor can access the file written by the previous invocation of it. Moreover, a file written by an editor can later be read by a compiler. Other data structures we have seen so far cannot be shared among different (invocations of) programs. The reason a file can be shared is that the data on disk is sharable while the memory of one program cannot (usually) be shared with another program. Not all files are shared -- it is possible to create private or internal files, which are useful if a program needs to create a large data structure that does not fit in its memory
- *External Name*: A shared or external file has a unique external name. The name is external in that, unlike a variable name, it is not private to any program and is thus shared by all programs. This name is typically given by an end-user. A program typically shows creates file browsers to allow he user to conveniently give the name.
- *Sequential Access*: Like a data input stream and an enumeration, we have seen, a "regular'' file is accessed sequentially, starting at the first element and ending at the last element. In the case of special files, called random access files, we can "seek" to a particular position in the file and then read sequentially from there.
- *OS managed*: Unlike other data structures we have seen so far, a file is managed by the operating system rather than the programming language. The operating system stores the file as a sequence of bytes that must be converted to and from programming language data structures.

Java provides a class called, `java.io.File`, for performing the operations on a file that are defined by the operating system. These operations tend to be complex, so it provides
two simpler classes, `java.io.FileInputStream` and `java.io.FileOutputStream`, for reading and writing files.

Reading Bytes

An instance of `FileInputStream` can be used to read bytes sequentially from an existing file. Its constructor takes thename of the file as an argument:

```
FileInputStream fileIn = new FileInputStream(fileName)
```

Figure 2 illustrates the nature and use of a file input stream.

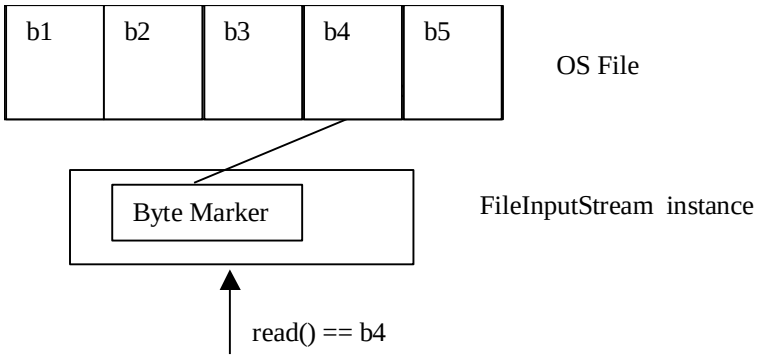


Figure 2 Invoking read on FileInputStream Instance

It has instance variables to keep track of the position of the next byte to read from the file. The first call to `read()` returns the first byte in the file, and successive calls to it return successive bytes in the file. Thus, a file input stream essentially "enumerates" all the bytes stored in an existing file.

Unlike a true enumeration, but like a data input stream, a Java input stream does not have a method to indicate if it has more elements. It throws an `IOException` if we read beyond the end of file.

Writing Bytes

Java provides the class, `FileOutputStream` , for writing a sequence of bytes to a new file. Like the constructor of `FileInputStream`, the constructor of this class takes the name of the file as an argument.

FileOutputStream fileOut = new FileOutputStream(fileName)

However, instead of accessing an existing file, it creates a new file of the specified name.

Figure 3 illustrates the nature of a file output stream

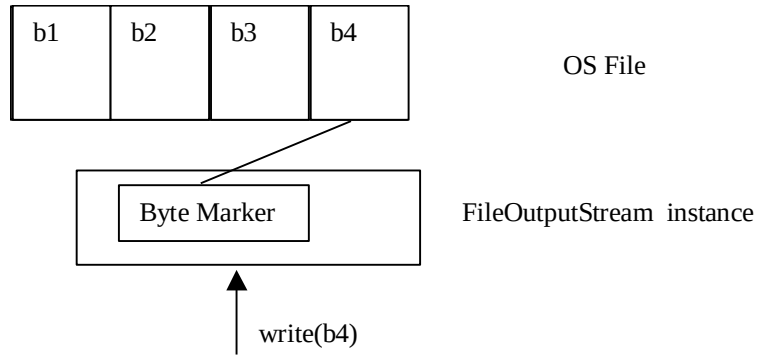


Figure 3 Invoking Write() on FileOutputStream Instance

Instead of generating a stream of bytes from its user, it receives a sequence of bytes, and writes them to the new file. The user invokes the method, `write()`, to supply it with the bytes to be written. This method appends its argument to the file. Thus, it is like `println()`, which appends its arguments to the console output stream.

We have seen above how to read an existing a file and write to a new file. What if we wanted to change an existing file? We must read its contents into memory, change them as appropriate, and then write the new contents into a new file that has the same name as the original file.

The two classes above are usually too low-level for our purposes. They require us to read and write bytes rather than a complete object such as a `PointHistory` instance. There are ways to convert between an object and a sequence of bytes, but these are tedious to use.

Therefore Java provides two classes, `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`, for reading and writing complete objects. These classes are built on top of the `FileInputStream` and `FileOutputStream` classes, respectively, and automatically do the conversion to and from bytes for us.

Reading Objects

Let us first consider an object input stream. It can be used to convert the sequence of bytes enumerated by a file input stream into a sequence of objects represented by the bytes. The file input stream is passed to the constructor used to create the object input stream.

```
ObjectInputStream objectIn = new ObjectInputStream(new FileInputStream (fileName));
```

The following figure illustrates the nature of an object input stream:

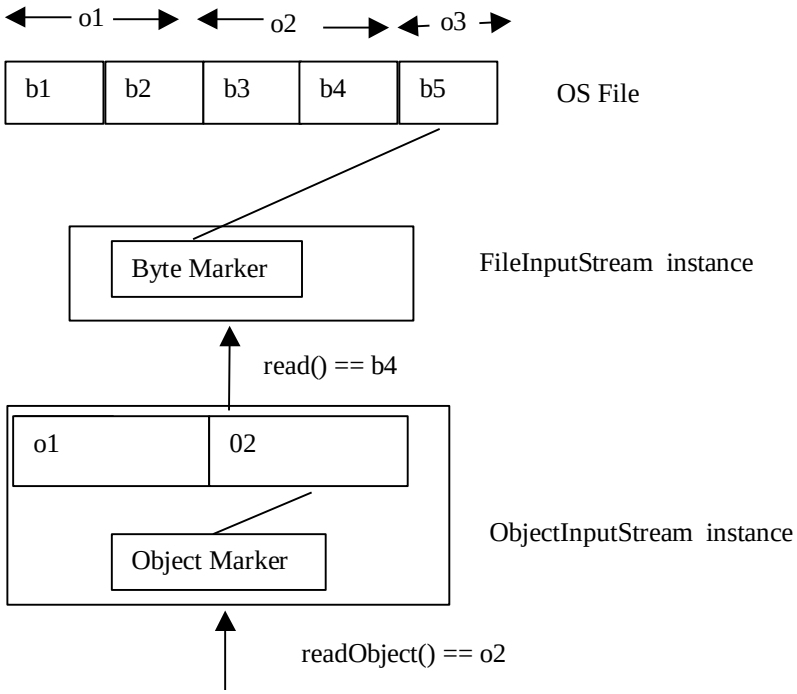


Figure 4 Object Input Stream

The object refers to its file input stream, which is used to enumerate the bytes in the file. The user of the object input stream calls `readObject()` to read the objects enumerated by it. The method uses the file input stream object to read all bytes of the next object and returns this object to the user. This method can return arbitrary objects - hence its return type is `Object`. When we need to treat it as an instance of a more specialized class, we must cast it to the appropriate type:

```
pointHisory.set ((PointHistory) fileStream.readObject(););
```

Java will give a ClassCastException if the type of the read object does not correspond to what we expected. This can happen if we accidentally read a file into which the expected object was not saved.

Even if we do not cast the object, we may get an exception. A program may try to read an object whose type is not defined in it – the object was written by some other program, which defined the type. In this case, Java will give the reading program a `NoClassDefFoundException`.

Writing Objects

Analagously, an object output stream writes complete objects to a new file, using a file output stream to write the individual bytes of the object. The latter is passed as an argument to the constructor of the former:

ObjectOutputStream objectOut = new ObjectOutputStream(new FileOutputStream (fileName));

The following figure shows how this class works:

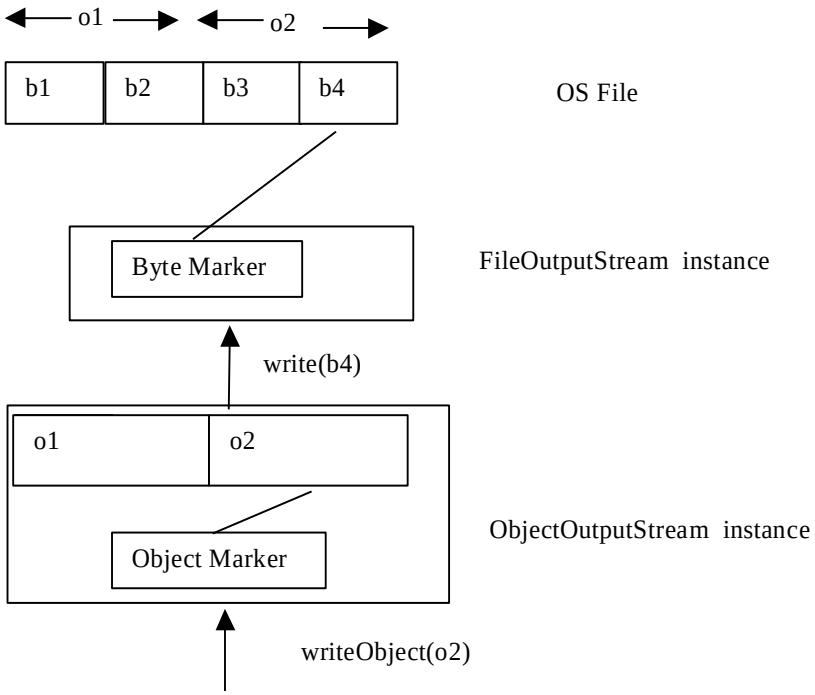


Figure 5 Object Output Stream

The user of an object output stream calls `writeObject()` to write complete objects to the file. The method converts its argument to a series of bytes representing the object, and uses its file output stream to write the bytes to the file. The method can write arbitrary objects; therefore, its argument type is `Object`.

Serializing/Deserializing Objects

Not all objects can be written and read by the object output and input streams: only those that are *serializable*. The term, “serializable”, indicates that the hierarchical physical structure of an instance of the type must be serialzed to/deserialized from a sequence of bytes. For instance, assuming a point history model with two points, the following structure must be serialized/deserialized when the object is saved/loaded:

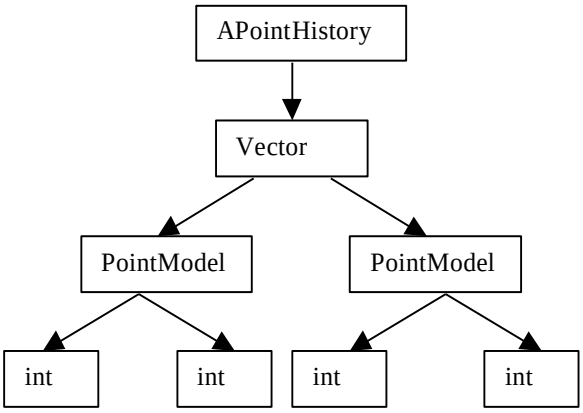


Figure 6 Point History Instance to be Deserialized/Serialized

Here, we assume that a point model defines two integer variables to store the coordinates of the point it represents. In fact, it defines several other variables, which are not shown here to simplify the picture. For the same reason, the animating point and animation delay of a point history are not shown.

How do we make an object serializable? Java provides a predefined interface called `java.io.Serializable` to identify serializable objects. An object is serializable if its class implements this interface. To make a point history model serializable, we can change the header of its class, as shown below:

public class APointHistory **implements** PointHistory, Serializable

The serializable "interface" does not declare any methods. Thus, an implementation of this interface does not require the definition of any additional code. Its purpose is allow Java to distinguish between serializable and non-serializable types, simply serving as a tag.

We could have also made point histories serializable by changing `PointHistory`:

public interface PointHistory **extends** Serializable

Since `APointHistory` implements `PointHistory`, by making the change above, we have ensured that it also now implements `Serializable`. Changing the interface is preferable to changing the class because we automatically ensure that all implementations of it alos implement the serializable interface.

We will refer to a class that implements the serializable interface a serializable class, and an interface that extends this interface as a serializable interface.

Java serializes an object by recursively serializing the contents of (public and non-public) variables of an object. It requires the types of all components in the physical structure of the object to be serializable. Thus, taking the point history example above, it requires AP`ointHistory`, `Vector`, `PointModel`, and `int` to be serializable. All of these types except `APointHistory` are predefined types and have been created as serializable. Thus, our job was to make AP`ointHistory` serializable. If any of the types encountered during serialization is not in fact serializable, Java throws the exception, `NotSerializableException`.

Serialization and Subclassing

Consider now AP`ointHistoryModel`. Now that we have ensured that its superclass, `APointHistory`, is serializable, do we have to do anything more to ensure that it too is serializable?

Recall that an object is serializable as long as its class implements the serializable interface. Moreover, a subclass implements all the interfaces of its (direct and indirect) superclasses. Thus, `APointHistoryModel` automatically

became serializable once we made its superclass, `APointHistory`, serializable. This means that when an instance of APointHistoryModel is serialized, not only is the vector:

Vector contents = new Vector();

declared in `APointHistory` serialized, but also the vector:

Vector listeners = new Vector();

declared in `APointHistoryModel` . In general, when a class is serializable, variables declared in it and all of its subclasses are serialized.

However, the variables declared in the superclass of a serializable class are not serialized unless the superclass is also serializable. Suppose we made `APointHistoryModel` serializable but not its superclass, `APointHistory`. When an instance of this class is serialized, the listener vector is serialized, because it is defined in a serializable class, but not the contents vector, because it is declared in a non-serializable class. Therefore, if we wish certain variables of an object to be serialized, we must select the "super most" class (that is the topmost class in the class hierarchy) among the classes that declare these variables, and make it serializable. Once we do so, all variables declared in other classes also become serializable, because they are declared in the subclasses of this class. In our example, if we wish both vectors to be serialized, we must make `APointHistory` as serializable.

Transient Variables

Should we make both vectors of a point history model serializable? We certainly need the contents vector to be serialized, because it holds the points we wish to restore with the load command (Figure 1 (c)). What about the listener vector? Recall that this vector holds the views of the model. They define the user-interface of the object, which we do not wish to load to/restore from persistent store. We wish to manipulate the model using the user-interface from which the load command was executed (which may be the same as the one from which the save command was issued.) Therefore, we do not wish to serialize the listeners vector.

We cannot make a subclass of a serializable class as non-serializable. Thus, once we make `APointHistory` serializable to ensure that the contents vector is serialized, we cannot make APointHistoryModel as non-serializable to ensure that the listeners vector is not serialized. What we can do, though, is declare a variable of a serializable class as *transient*:

transient Vector listeners = new Vector();

Java serializes only non-transient components of an object.

A File-based Program

We have by now learnt enough about files to start implementing the user-interface of Figure 1. There are a few, minor details left, such as how to create a file dialogue box. These we will learn as we create the new implementation.

Naturally, we will extend our previous implementation of the point history user-interface rather than create a brand new one. In fact, we can use the view without making any change to it. We must also extend the controller to add the three new commands. We will need to also change the model: It is not possible for an object other than the model to erase an element from the point history. Thus, we need to extend the model support the clear command. As we develop our implementation, we will see other reasons for changing the model. Once we decide to create a new model, view, or controller class, we need to also change the composer. Thus, we will extend the composer class to bind the new model to the new controller and previous view.

We have seen how to extend a controller to add the three new buttons. Moreover, the clear listener is straightforward, it simply calls the method in the model to clear the history. The listeners of the other two buttons are more interesting. Consider, first the save listener. We might be tempted to write it like the listeners we have written so far, simply calling a method in the model to process the command:

```
public class ASaveActionListener implements ActionListener {
        PointHistory pointHistory;
        public ASaveActionListener (PersistentPointHistory thePointHistory) {
                pointHistory = thePointHistory;
        }
```

```
		public void actionPerformed(ActionEvent ae) {
			pointHistory.save();
	}
}
```

We can similarly write a load listener that calls a load method in the model to process the load command.Here we are assuming a new model interface, PersistentPointHistory, that provides a method to save itself to and load itself from a file.

External File-Interface Code

Requiring a model to manipulate a file is not a good idea, just as requiring it to display itself is a bad idea. Manipulating a file is similar to interacting with a user in that there are many ways to do so. For instance, we might want to prompt the user for a file name in a console window or in a file dialog box. Moreover, as we will see below, we may use a text file or a binary file Therefore, like the code to implement the user-interface of an object, the code to implement the file interface of an object (that is, load and save the object) should be outside the type of the object. In our example, a good place to put it is in the save and load listeners, which do not do much else.

Saving Point History

Figure 7 shows the modified save listener.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Frame;
import java.awt.FileDialog;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ASaveActionListener implements ActionListener {
	FileDialog fileDialog = new FileDialog(new Frame(), "Save to", FileDialog.SAVE);
	PointHistory pointHistory;

	public ASaveActionListener (PointHistory thePointHistory) {
		pointHistory = thePointHistory;
		fileDialog.setFile("pointHistory.obj");
	}

	public void actionPerformed(ActionEvent ae) {
		try {
		ObjectOutputStream fileStream = new  ObjectOutputStream(new FileOutputStream
(getFileName(fileDialog)));
		fileStream.writeObject(pointHistory);
		fileStream.close();
		} catch (Exception e) {
		System.err.println(e);
		}
	}
	public static String getFileName(FileDialog fileDialog)  {
		fileDialog.setVisible(true);
		return  fileDialog.getDirectory() +  fileDialog.getFile();
	}
}
```

Figure 7 Saving a File

The constructor of this version of the listener, unlike the one of the previous version, takes an argument of type `PointHistory`, rather than then new model type, `PersistentPointHistory`. This means the save listener does not need any new method in the model, doing its job completely without involving the latter.

The point history is accessed by the action performed method, which is responsible for saving it in a file. This task can be broken down into three simple sub-tasks:

1. As the user for the name of the file in which the point history is to be saved.
2. Create a file output stream from this file name to write bytes to the file.
3. Create an object output stream from the file output stream, and write the point hsitory to the this stream.

File-Access Exceptions

The statements of the action performed method are enclosed in a try-catch block. The reason is that several exceptions can arise when reading and writing a file. The file to be read may not exist. It may not be possible to create a new file. The object to be read/written may have a non-transient component whose type is not serializable. Therefore file reading and writing code must catch these exceptions. It is simplest to define a single try and catch block of the form we have used for input from the console window:

```
try {
   file accessing code
} catch (Exception e) {
   System.err.println(e);
}
```

The catch block is used to catch an exception of any type, and, as before, the `println` displays the kind of exception. This is exactly what the action performed method does.

File Dialog

The action performed method must allow the user to specify the name of the file in which the point history should be written. The name is simply a string, so this task seems trivial – the method can simply read this value from the console window. In fact, this was the way traditional file-based programs input file names. However, this is not the kind of user-interface we expect today to specify a file name. We expect to interact, not with a console window, but a visually pleasing file dialog box in which a default file name is filled. We also expect capabilities to browse the directory structure to identify the directory of the file. We expect a warning if we try to overwrite an existing file. Java provides the predefined class, `FileDialog`, for automating these tasks. Each instance of it created a separate file dialog box to prompt a user for a file name, allowing the user to browse the directory structure to identify the name (Figure 8). It works in two modes. In the load mode it prompts for the name of an existing file in the directory, and in the save mode it prompts for the name of a new file, giving a warning if an existing file has the same name.

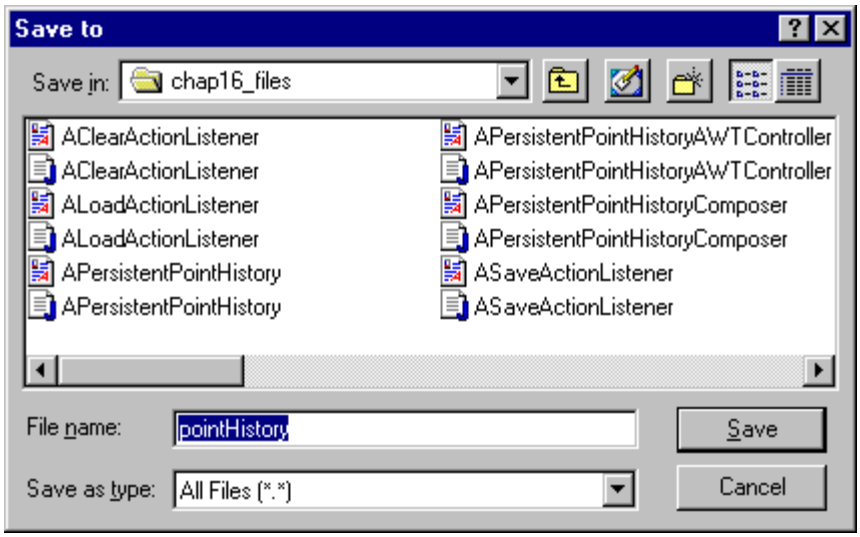


Figure 8 Save Dialogue Box

The declaration:

FileDialog fileDialog = new FileDialog(new Frame(), "Save to", FileDialog.SAVE);

creates a new file dialogue box in the save mode. The statement:

fileDialog.setFile("pointHistory.obj");

sets a default name file name (Figure 8), which the user can change. When a file dialog is created, it is not displayed to the user. Therefore the call:

fileDialog.setVisible();

is made to display it to the user. This statement blocks the program until the user closes the displayed file dialog (by pressing the save or cancel dialogue box button), when the dialogue box automatically becomes invisible again. The file name input by the user is computed by the expression:

fileDialog.getDirectory() + fileDialog.getFile();

`getDirectory()` returns the directory of the file and `getFile()` returns the local name of the file within the directory. The concatenation of these two strings is the full name of the file, which is what is expected by the constructors of the `FileInputStream` and `FileOutputStream`.

Serializing PointHistory

The statement:

ObjectOutputStream fileStream = new ObjectOutputStream(new FileOutputStream (getFileName(fileDialog)));

first creates a file output stream from the full file name, and the creates an object output stream from the file output stream. Once we have the object output stream, we can simply call the `writeObject()` method it to write the point history to it:

fileStream.writeObject(pointHistory);

We do not need to implement any special method in the point history to write it to the file. We do need to make the interface, `PointHistory`, serializable::

public interface PointHistory **extends** Serializable

We also need to decide which the variables of a point history model are transient:

transient Vector listeners = new Vector();

As we see here, unfortunately, it is not possible to localize all changes required to make a point history persistent in the new interface, `PersistentPointHistory`, and its implementation, `APersistentPointHistory`. We must go back and change existing interfaces and classes

Opening and Closing a File

When Java creates a stream for a file, it *opens* the file, that is, builds an internal link to the file. Once the action performed method has finished processing the file, it *closes* the file, that is, remove the link by invoking the `close()` method on the object output stream:

```
fileStream.close();
```

All of the streams we have seen such as the file and data streams implement this method.

Loading Point History

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Frame;
import java.awt.FileDialog;
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class ALoadActionListener implements ActionListener {
        FileDialog fileDialog = new FileDialog(new Frame(), "Load from", FileDialog.LOAD);
        PersistentPointHistory pointHistory;

        public ALoadActionListener (PersistentPointHistory thePointHistory) {
                pointHistory = thePointHistory;
                fileDialog.setFile("pointHistory.obj");
        }

        public void actionPerformed(ActionEvent ae) {
                PointHistory savedPointHistory = null;
                 try {
                ObjectInputStream fileStream = new ObjectInputStream(new FileInputStream
(ASaveActionListener.getFileName(fileDialog)));
                        savedPointHistory = (PointHistory) fileStream.readObject();
                pointHistory.set (savedPointHistory);
                fileStream.close();
                } catch (Exception e) {
                        System.err.println(e);
                }
        }
}
```

Figure 9 Load Listener

The load listener is similar to the save listener (Figure 9). Instead of :

- (file and object) output streams, it uses the corresponding input streams.
- creating the file dialog box in the save mode, it creates it in the load mode.
- writing to the object output stream, it reads from the object input stream.

The `read()` method invoked on the object input stream retrieves whatever object was saved in the file, returning a value of type `Object`. Expecting the user to input the name of a file in which a previous point history was saved, the program casts the return value to a `PointHistory`. A `ClassCastException` would be raised if some other kind of object was saved in the file.

At this point we have two models: the new one retrieved from the file, `savedPointHistory`, and the old one to which the views and controllers of the user-interface are attached, `pointHistory`. The user wishes to see the point

history of the new model in the user-interface of the old model, that is, see the new point history in the existing view panel(s) and manipulate it using the existing control panel(s) (Figure 1). We could disconnect the view(s) and controller(s) of the old model and attach them to the new one, but is it is simpler to make the point history of the connected model a copy of the point history of the loaded model. The load listener expects the new type, `PersistentPointHistory`, to set its contents to the contents of another point history. Therefore, it simply invokes this operation, passing as an argument to it the saved object.

Persistent Point History

We are finally ready to see the interface, PersistentPointHistory, and its implementation APersistentPointHistory:

```
public interface PersistentPointHistory extends PointHistoryModel {
	public void clear();
	public void set (PointHistory otherPointHistory);
}

import java.util.Vector;
public class APersistentPointHistory extends APointHistoryModel implements PersistentPointHistory {
	public void clear() {
		contents = new Vector();
		notifyListeners();
	}

	public void set (PointHistory otherPointHistory) {
		clear();
		for (int index = 0; index < otherPointHistory.size(); index++)
			contents.addElement(otherPointHistory.elementAt(index));
		notifyListeners();
	}
}
```

The clear method does not need to delete each element from the point history. It simply assigns a new vector to the inherited variable, `contents`, which stores the point history. The set method clears the contents of the target point history and then adds to this vector each element of the point history to be copied. Since both the clear and set methods change the model, they notify all listeners of the model about the change.

By implementing the set method to support the load command, we do not violate the principle of keeping file manipulation code outside a model, because it is does not access files. In fact, the load listener could have itself implemented the set method using the clear and add element methods. However, this method provides a general purpose facility for copying a point history which is useful in situations other than loading the object from a file. Hence it has been added to the point history.

Binary Files

We can try to determine how the point history is serialized by using a text editor to open the file in which it is saved.

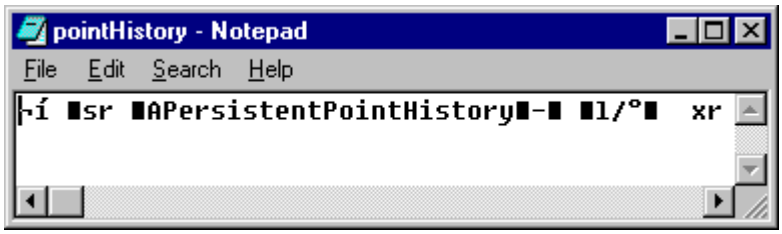


Figure 10 Viewing a Binary File Using a Text Editor

Figure 10 shows what we would see. Java writes down the class of the object in the file so that it knows how to deserialize it. Besides the class name, there is no other intelligent information displayed by the text editor. The reason is that when Java serializes an object, it writes the in-memory, binary image of the object in the file. The image can consist of text (printable characters) and other kinds of values such as integers and reals. When we ask the text editor to display this image, it assumes that the image represents a sequence of printable characters. The parts of the image for which this assumption is correct are displayed as expected, others are displayed as "garbage".

A file such as this that contains the binary image of on object is called a *binary file*. A file consisting only of text is called a *text file*.

Text Files

We have seen how Java allows us to directly read/write bytes or complete objects from/to a binary file. It also gives us the option of reading/writing text from/to a text file exactly the way we read/write from/to the console. In fact, the same code is used to do both tasks, as we will see.

A data input stream can be used not only to read text from `System.in`, the console input, but also an arbitrary file. Instead of passing the console input to the constructor used to create a data input stream, we can pass a file input stream.

dataIn = new DataInputStream (new FileInputStream(fileName)).

Now, when `readLine()` is invoked on the data input stream:

String nextLine = dataIn.readLine()

instead of reading from the console it reads from the file . The job of a data input stream is to convert a sequence of bytes (received from console or a file) into higher-level text units such as a line.

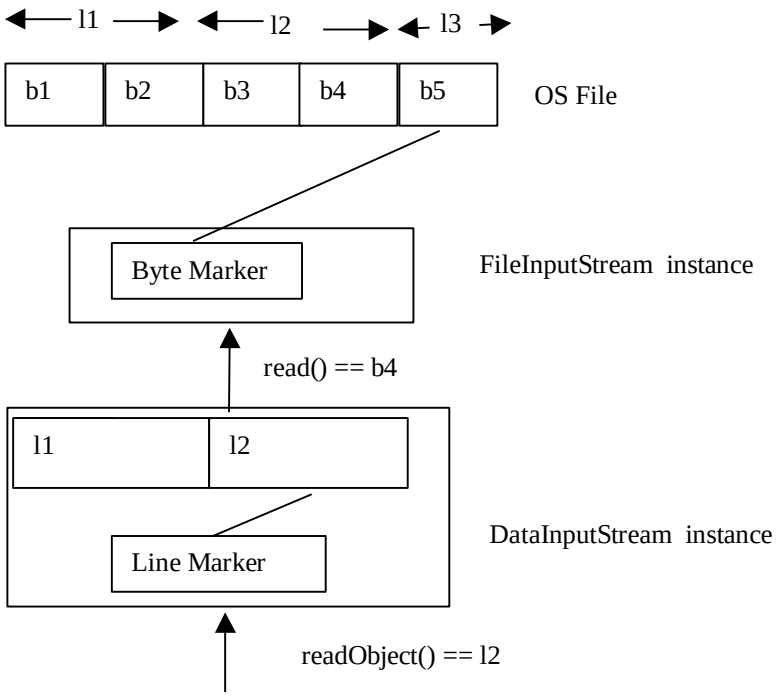


Figure 11 DataInput Stream

Analogously, the class `PrintStream` can be used to print text to a file just as we output text to the console. We must pass to its constructor a file output stream we create for the file:

dataOut = **new** PrintStream(**new** FileOutputStream(fileName))

Now we can invoke on the print stream all of the operations we have used on `System.out`, the console output.

dataOut.println(point.getX());

The object converts the text to be printed to a sequence of bytes, which is written by the file output stream to the file, as shown in Figure 12

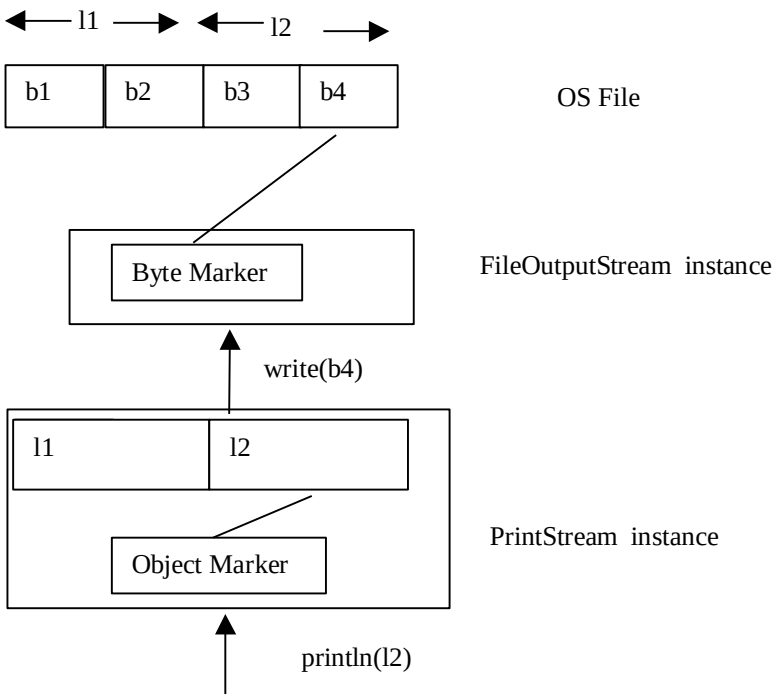


Figure 12 PrintStream

A Text-File based Program

To better understand how text files work, let us see how we can implement the load and save commands using text files. This means we must change the save and load listeners. The save listener is given in Figure 13.

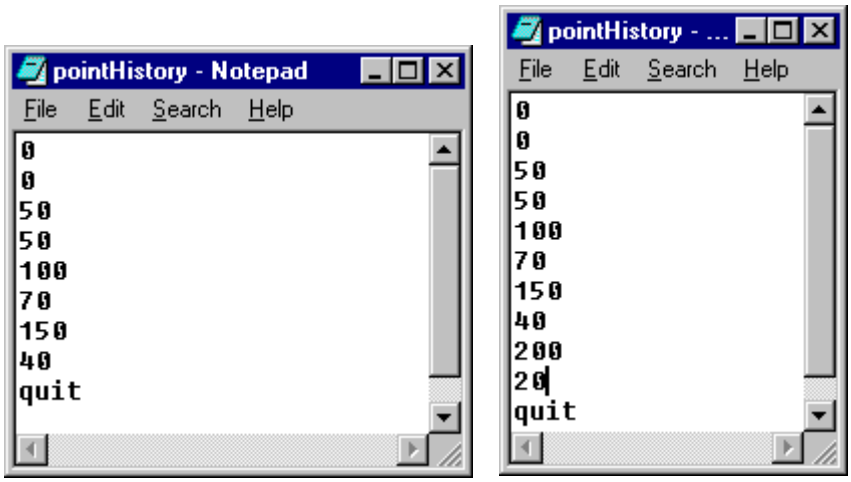
```
import java.awt.Frame;
import java.awt.FileDialog;
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.IOException;
import shapes.PointModel;
public class AnotherSaveActionListener implements ActionListener {
        FileDialog fileDialog = new FileDialog(new Frame(), "Save to", FileDialog.SAVE);
        PointHistory pointHistory;

        public AnotherSaveActionListener (PointHistory thePointHistory) {
                pointHistory = thePointHistory;
                fileDialog.setFile("pointHistory.txt");
        }
        public void actionPerformed(ActionEvent ae) {
                try {
                PrintStream dataOut = new  PrintStream(new FileOutputStream
(ASaveActionListener.getFileName(fileDialog)));
                writePoints(pointHistory, dataOut);
                dataOut.close();
                } catch (Exception e) {
                System.err.println(e);
```

```
                }
        }
        public static void  writePoints (PointHistory pointHistory, PrintStream dataOut) {
                for (int index = 0; index < pointHistory.size(); index++) {
                        PointModel point = pointHistory.elementAt(index);
                        dataOut.println(point.getX());
                        dataOut.println(point.getY());
                }
                dataOut.println("quit");
        }
}
```

Figure 13 Text Save Listener

Instead of creating an object output stream, it creates a print stream. It then writes the elements of the point history to this stream. It writes each coordinate of each point in a separate line, and then writes the string, "quit", to make the end of the history.

Since the file written by the save listener is a text file, we can examine it using a text editor to see what was actually written, as shown in Figure 14. This is useful for debugging the program. Moreover, if users of the program understand the format of the file, they can even modify the file using the text editor, as shown in Figure 14(b).



(a) (b)

Figure 14 Examining and Modifying a Saved Text File

The corresponding load listener is shown in Figure 14. Instead of creating an object input stream, as the previous load listener did, it creates a data input stream. It first clears the current point history. It then reads point coordinates from the text file until it encounters the "quit" string.. It collects each pair of x and y coordinates and adds the point to the current point history.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Frame;
import java.awt.FileDialog;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class AnotherLoadActionListener implements ActionListener {
        FileDialog fileDialog = new FileDialog(new Frame(), "Load from", FileDialog.LOAD);
        PersistentPointHistory pointHistory;
```

```
	public AnotherLoadActionListener (PersistentPointHistory thePointHistory) {
		pointHistory = thePointHistory;
		fileDialog.setFile("pointHistory.txt");
	}
	public void actionPerformed(ActionEvent ae) {
		try {
		DataInputStream dataIn = new DataInputStream(new FileInputStream
(ASaveActionListener.getFileName(fileDialog)));
		pointHistory.clear();
			readPoints(pointHistory, dataIn);
		dataIn.close();
		} catch (Exception e) {
			System.err.println(e);
		}
	}
	public static void  readPoints (PointHistory pointHistory, DataInputStream dataIn) {
		try {
			while (true) {
				String input = dataIn.readLine();
				if (input.equals("quit")) break;
				int x = Integer.parseInt(input);
				input = dataIn.readLine();
				int y = Integer.parseInt(input);
				pointHistory.addElement (x,y);
			}
		} catch (IOException e) {
			System.err.println(e);
		}
	}
}
```

Figure 15 Text Load Listener

Sharing Console and File I/O Code

The two listeners have been written to allow their code to be shared by classes that wish to point history from the console or write it to the console. Figure 16 shows a console controller using code of the load action listener.

```
import java.io.DataInputStream;
public class AnotherPointHistoryConsoleController implements PointHistoryConsoleController {
	PointHistory pointHistory ;

	public AnotherPointHistoryConsoleController (PointHistory thePointHistory) {
		pointHistory = thePointHistory;
	}
	public void processCommands() {
		System.out.println("Please enter the points followed by quit:");
		AnotherLoadActionListener.readPoints(pointHistory, new DataInputStream(System.in));
	}
}
```

Figure 16 Console Controller Constructed from Load Listener Code

The controller calls the `readPoint()` method of the load listener, passing it a data input stream created from the standard input rather than a file input stream. As a result, the method reads from the console, as shown in Figure 17.

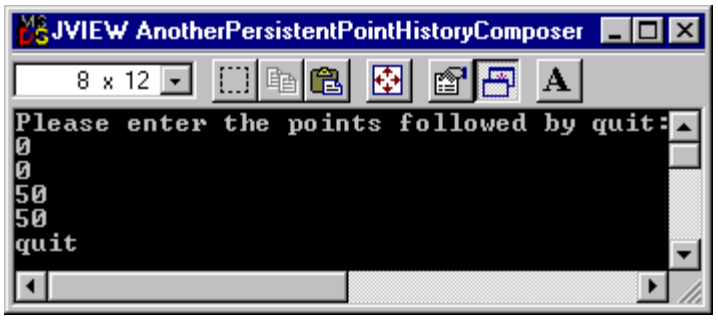


Figure 17 Console Controller User-Interface

Similarly, Figure 18 shows a console view using code of the save action listener.

```
public class AnotherPointHistoryConsoleView implements PointHistoryListener {
        public void pointHistoryUpdated(PointHistory pointHistory) {
                System.out.println ("**********");
                AnotherSaveActionListener.writePoints(pointHistory, System.out);
                System.out.println ("**********");
        }
}
```

Figure 18 Console View Constructed from Load Listener Code

The method calles `witePoints(),` passing it the console output rather than a print stream constructed from a file output stream. As a result, the method writes the points to the console, as shown in Figure 19.

Figure 19 Console View

The “quit” at the end the display of points is distracting. Unfortunately, it is a consequence of our decision to use reuse the file manipulation code, which requires the output and input of an object to be identical, since the output written by a save command is read by the load command.

Text Vs Binary File

When we save/load an object, we have seen that we have the option of using a text or binary file. Which approach is better?

The advantages of text files are:

- *Familiar/Reusable interface*: We access them using the same interface we used for accessing a text window. In fact, if we proceduralize our program appropriately, we can write one method, such as `writePoints()` and `readPoints()` to transfer data to and from a text file and a text window.
- *Readable by a text editor*: We can examine and modify the contents of a text file using a text editor, which is useful in debugging a program and provides the end user with another, familiar, user-interface for entering data.

The advantages of binary files are:

- Automatic Serializing/Deserializing: When we use a text file, we are responsible for writing methods such as `writePoints()` and `readPoints()` for serializing/deserializing the object to/from the text file. While this may seem easy for a simple structure as a point history, it is quite tedious for a more complex structures, specially ones that contains multiple instance variables that must be saved/loaded. As we saw above, this step is automated if we use `writeObject()` and `readObject()` to serialize and deserialize an object to and from a binary file.
- Space: Binary files take less space. For instance, a text file would hold the value **true** as four characters (16 bytes) - 't' 'r' 'u' 'e'; whereas a binary file would store it as a single byte – 0 -which is its memory representation.

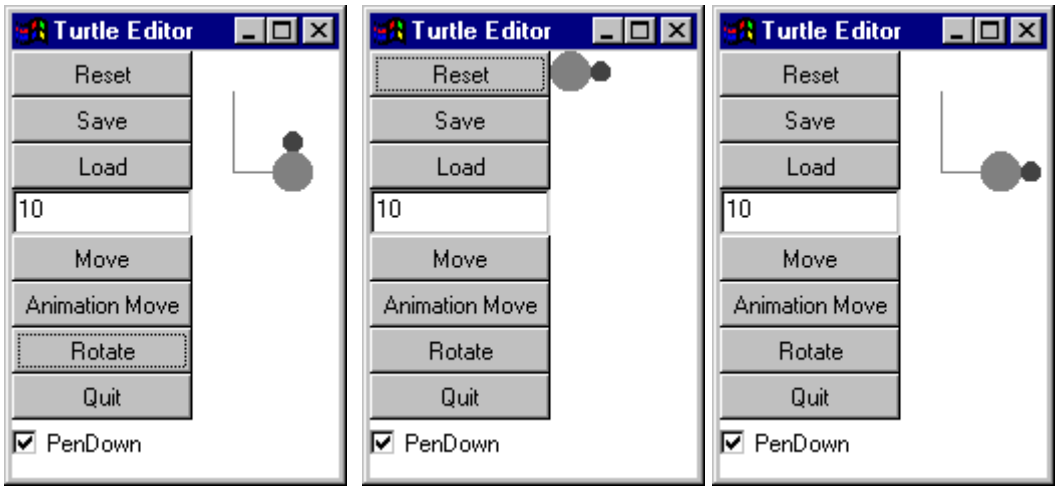
Since few Java programs actually access a text window, the advantage of binary files usually outweigh those of text files.

Summary

- A file is a large, variable-sized, persistent, sequentially-accessed data collecton with an external name that is managed by the operating system and shared among multiple programs.
- Before a program can manipulate a file, it must establish a link between a file-stream object and a file. After manipulating the file, it should close the link.
- It is possible to transfer bytes, text, or entire objects to and from files.
- Text files are manipulated like a console window and can be manipulated using a text editor.
- Binary files are more compact and save the programmer from writing code to serialize and deserialize non-transient variables of objects.
- A class is serializable if it implements the serializable interface, which contains no methods, serving only as a tag to the Java serializer.
- When a class is serializable, variables declared in it and all of its subclasses are serialized, but those declared in its superclass are not serialized unless the superclass is also serializable.
- Code to loads and save an object should be external to the class of the object. In a toolkit-based application, it should be in the listeners of buttons used for invoking these operations.
- A large fraction of the code required to manipulate a binary file is independent of the object being saved and loaded.

Exercises

1. Why should the code for loading and saving an object be external to the class of the object?
2. In the listener that loads the point history from a binary file (Figure ?), why is `pointHistory` declared to be of type `PeristentPointHistory` and `savedPointHistory` of type `PointHistory`?
3. What are the various exceptions that can occur in try block of the load listener?
4. Add reset, save, and load commands to your solution to the turtle problem of the previous chapter.



(a) Saving Turtle (b) Resetting Turtle (c) Loading Turtle

Figure 20 Persistent Turtle

These commands work on the "view state" of the turtle editor, that is, those aspects of the turtle editor that are shown in the view. These include the position and direction of the turtle, and the lines drawn by it. The reset command restores the view-state to its initial configuration. Thus, it moves the turtle to its initial position and direction and erases any lines that have been drawn (Figure 20(a)). The save command saves the view-state to a file. The load command restores this state from a file. Both the save and reset commands refresh the views of the turtle after they have changed its view state.

5. It may be useful to also reset, save, and load the "controller state" of the turtle editor, that is, those aspects of it such as the move distance that are shown in the control panel. However, implementing this feature is bit more difficult. Changing the model in response to these commands is straightforward, but updating the control panel to reflect changes in this state is not. Outline a solution for updating the control in response to changes to the controller state.

, refreshing the view to show this state.