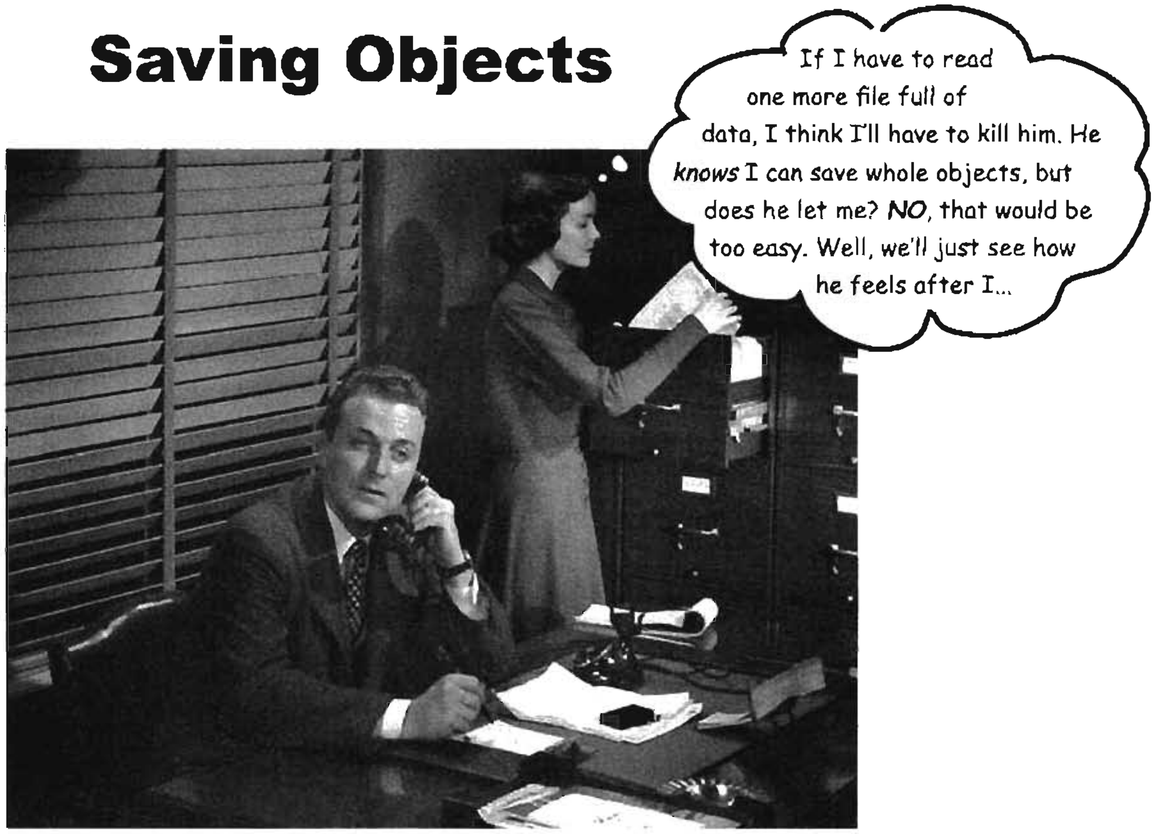
14 serialization and file I/O

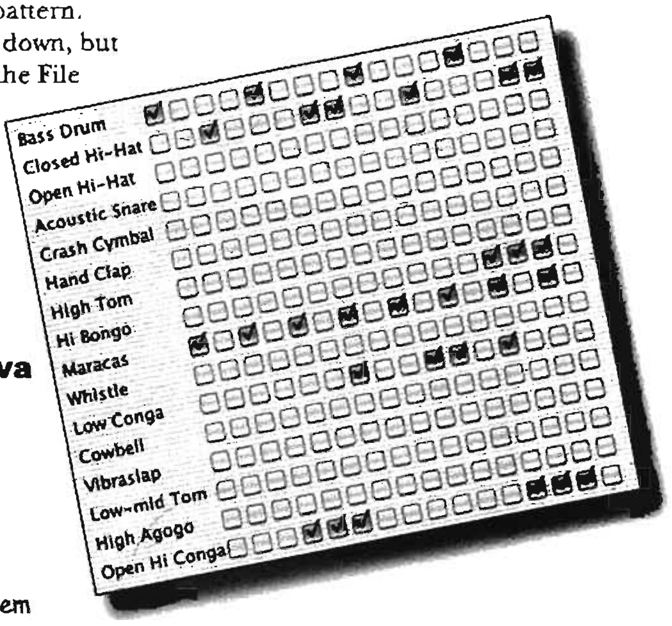
Saving Objects


If I have to read one more file full of data, I think I'll have to kill him. He *knows* I can save whole objects, but does he let me? *NO*, that would be too easy. Well, we'll just see how he feels after I...

Objects can be flattened and inflated. Objects have state and behavior. *Behavior* lives in the *class*, but *state* lives within each individual *object*. So what happens when it's time to *save* the state of an object? If you're writing a game, you're gonna need a Save/ Restore Game feature. If you're writing an app that creates charts, you're gonna need a Save/ Open File feature. If your program needs to save state, *you can do it the hard way*, interrogating each object, then painstakingly writing the value of each instance variable to a file, in a format you create. Or, **you can do it the easy OO way**—you simply freeze-dry/flatten/persist/ dehydrate the object itself, and reconstitute/inflate/restore/rehydrate it to get it back. But you'll still have to do it the hard way *sometimes*, especially when the file your app saves has to be read by some other non-Java application, so we'll look at both in this chapter.

Capture the Beat

You've *made* the perfect pattern. You want to *save* the pattern. You could grab a piece of paper and start scribbling it down, but instead you hit the *Save* button (or choose Save from the File menu). Then you give it a name, pick a directory, and exhale knowing that your masterpiece won't go out the window with the blue screen of death.



You have lots of options for how to save the state of your Java program, and what you choose will probably depend on how you plan to *use* the saved state. Here are the options we'll be looking at in this chapter.

If your data will be used by only the Java program that generated it:

① Use serialization

Write a file that holds flattened (serialized) objects. Then have your program read the serialized objects from the file and inflate them back into living, breathing, heap-inhabiting objects.

If your data will be used by *other* programs:

② Write a plain text file

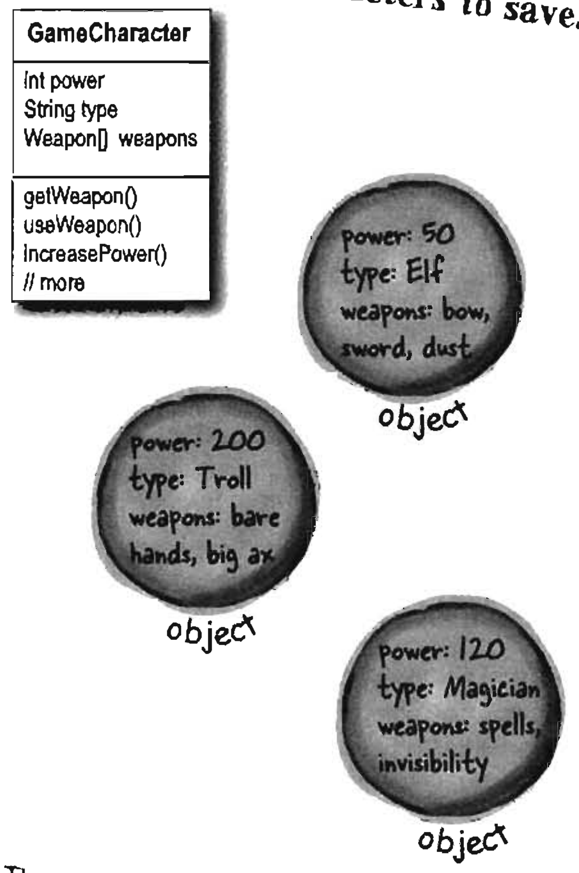
Write a file, with delimiters that other programs can parse. For example, a tab-delimited file that a spreadsheet or database application can use.

These aren't the only options, of course. You can save data in any format you choose. Instead of writing characters, for example, you can write your data as bytes. Or you can write out any kind of Java primitive *as* a Java primitive—there are methods to write ints, longs, booleans, etc. But regardless of the method you use, the fundamental I/O techniques are pretty much the same: write some data to *something*, and usually that something is either a file on disk or a stream coming from a network connection. Reading the data is the same process in reverse: read some data from either a file on disk or a network connection. And of course everything we talk about in this part is for times when you aren't using an actual database.

Saving State

Imagine you have a program, say, a fantasy adventure game, that takes more than one session to complete. As the game progresses, characters in the game become stronger, weaker, smarter, etc., and gather and use (and lose) weapons. You don't want to start from scratch each time you launch the game—it took you forever to get your characters in top shape for a spectacular battle. So, you need a way to save the state of the characters, and a way to restore the state when you resume the game. And since you're also the game programmer, you want the whole save and restore thing to be as easy (and foolproof) as possible.

Imagine you have three game characters to save...



● Option one

Write the three serialized character objects to a file

Create a file and write three serialized character objects. The file won't make sense if you try to read it as text:

```
¯ÌsrGameCharacter
¯%g8 8MÛIpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp8tlfur[Ljava.lang.String;»“VÅ
È{Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

● Option two

Write a plain text file

Create a file and write three lines of text, one per character, separating the pieces of state with commas:

```
50,Elf,bow, sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

The serialized file is much harder for humans to read, but it's much easier (and safer) for your program to restore the three objects from serialization than from reading in the object's variable values that were saved to a text file. For example, imagine all the ways in which you could accidentally read back the values in the wrong order! The type might become "dust" instead of "Elf", while the Elf becomes a weapon...

Writing a serialized object to a file

Here are the steps for serializing (saving) an object. Don't bother memorizing all this; we'll go into more detail later in this chapter.

If the file "MyGame.ser" doesn't exist, it will be created automatically.

1 Make a FileOutputStream

```
FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
```

Make a FileOutputStream object. FileOutputStream knows how to connect to (and create) a file.

2 Make an ObjectOutputStream

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

ObjectOutputStream lets you write objects, but it can't directly connect to a file. It needs to be fed a 'helper'. This is actually called 'chaining' one stream to another.

3 Write the object

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

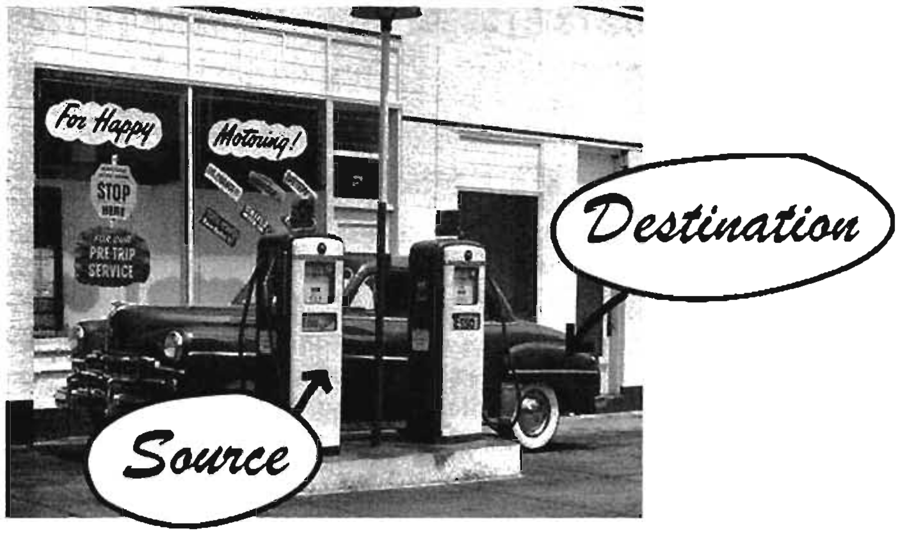
serializes the objects referenced by characterOne, characterTwo, and characterThree, and writes them to the file "MyGame.ser".

4 Close the ObjectOutputStream

```
os.close();
```

Closing the stream at the top closes the ones underneath, so the FileOutputStream (and the file) will close automatically.

Data moves in streams from one place to another.



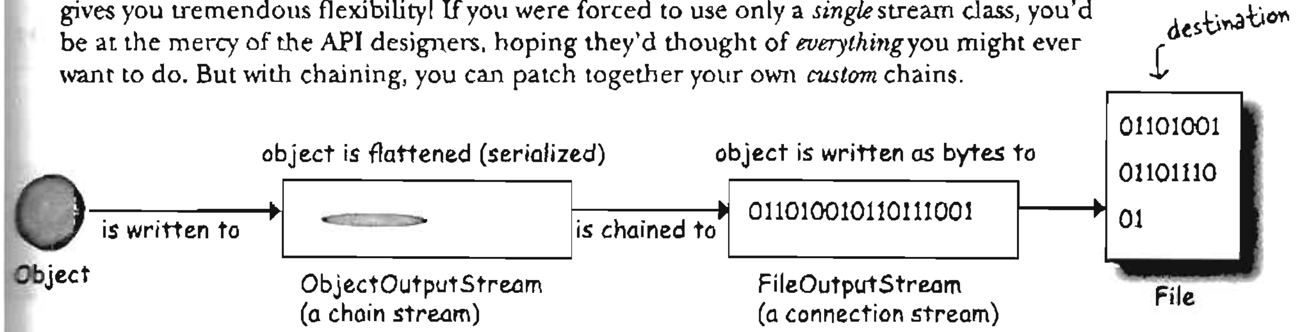
Connection streams represent a connection to a source or destination (file, socket, etc.) while chain streams can't connect on their own and must be chained to a connection stream.

The Java I/O API has ***connection*** streams, that represent connections to destinations and sources such as files or network sockets, and ***chain*** streams that work only if chained to other streams.

Often, it takes at least two streams hooked together to do something useful—*one* to represent the connection and *another* to call methods on. Why two? Because *connection* streams are usually too low-level. FileOutputStream (a connection stream), for example, has methods for writing *bytes*. But we don't want to write *bytes*! We want to write *objects*, so we need a higher-level *chain* stream.

OK, then why not have just a single stream that does *exactly* what you want? One that lets you write objects but underneath converts them to bytes? Think good OO. Each class does *one* thing well. FileOutputStreams write bytes to a file. ObjectOutputStreams turn objects into data that can be written to a stream. So we make a FileOutputStream that lets us write to a file, and we hook an ObjectOutputStream (a chain stream) on the end of it. When we call writeObject() on the ObjectOutputStream, the object gets pumped into the stream and then moves to the FileOutputStream where it ultimately gets written as bytes to a file.

The ability to mix and match different combinations of connection and chain streams gives you tremendous flexibility! If you were forced to use only a *single* stream class, you'd be at the mercy of the API designers, hoping they'd thought of *everything* you might ever want to do. But with chaining, you can patch together your own *custom* chains.


destination
object is flattened (serialized)
object is written as bytes to
is written to
is chained to
011010010110111001
01101001
01101110
01
Object
ObjectOutputStream (a chain stream)
FileOutputStream (a connection stream)
File

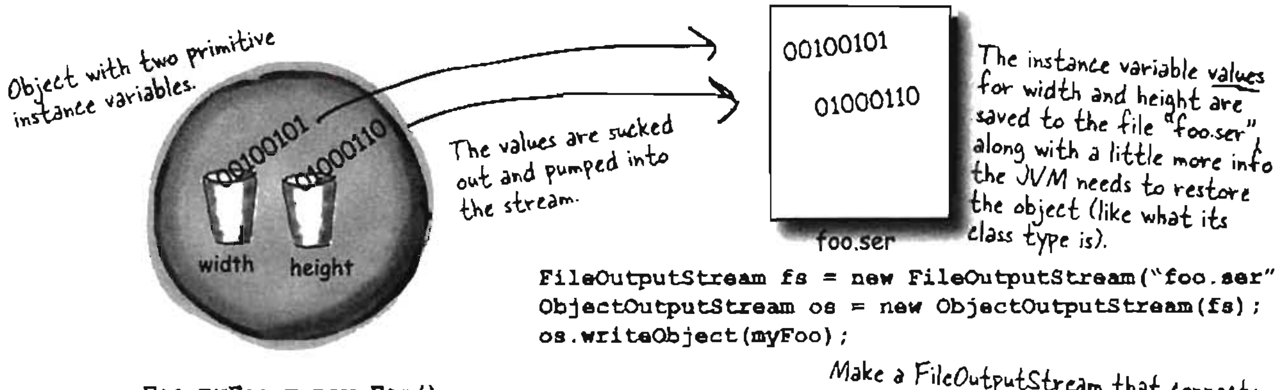
What really happens to an object when it's serialized?

1 Object on the heap

Objects on the heap have state—the value of the object's instance variables. These values make one instance of a class different from another instance of the same class.

2 Object serialized

Serialized objects **save the values of the instance variables**, so that an identical instance (object) can be brought back to life on the heap.



Object with two primitive instance variables.
00100101
01000110
width
height
The values are sucked out and pumped into the stream.
00100101
01000110
foo.ser
The instance variable values for width and height are saved to the file "foo.ser", along with a little more info the JVM needs to restore the object (like what its class type is).

```
FileOutputStream fs = new FileOutputStream("foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

Make a FileOutputStream that connects to the file "foo.ser", then chain an ObjectOutputStream to it, and tell the ObjectOutputStream to write the object.

```
Foo myFoo = new Foo();
myFoo.setWidth(37);
myFoo.setHeight(70);
```

But what exactly IS an object's state? What needs to be saved?

Now it starts to get interesting. Easy enough to save the *primitive* values 37 and 70. But what if an object has an instance variable that's an object *reference*? What about an object that has five instance variables that are object references? What if those object instance variables themselves have instance variables?

Think about it. What part of an object is potentially unique? Imagine what needs to be restored in order to get an object that's identical to the one that was saved. It will have a different memory location, of course, but we don't care about that. All we care about is that out there on the heap, we'll get an object that has the same state the object had when it was saved.

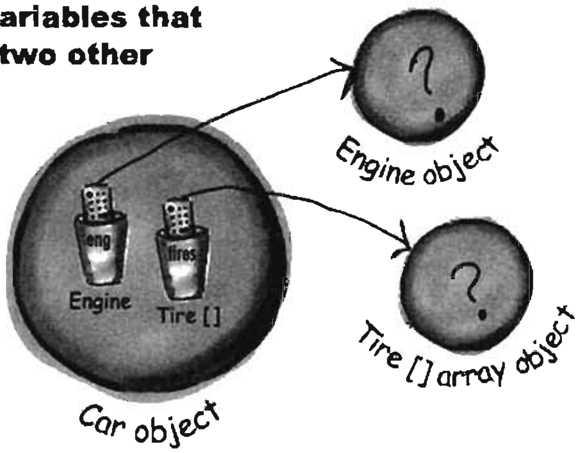
Brain Barbell

What has to happen for the Car object to be saved in such a way that it can be restored back to its original state?

Think of what—and how—you might need to save the Car.

And what happens if an Engine object has a reference to a Carburator? And what's inside the Tire [] array object?

The Car object has two instance variables that reference two other objects.



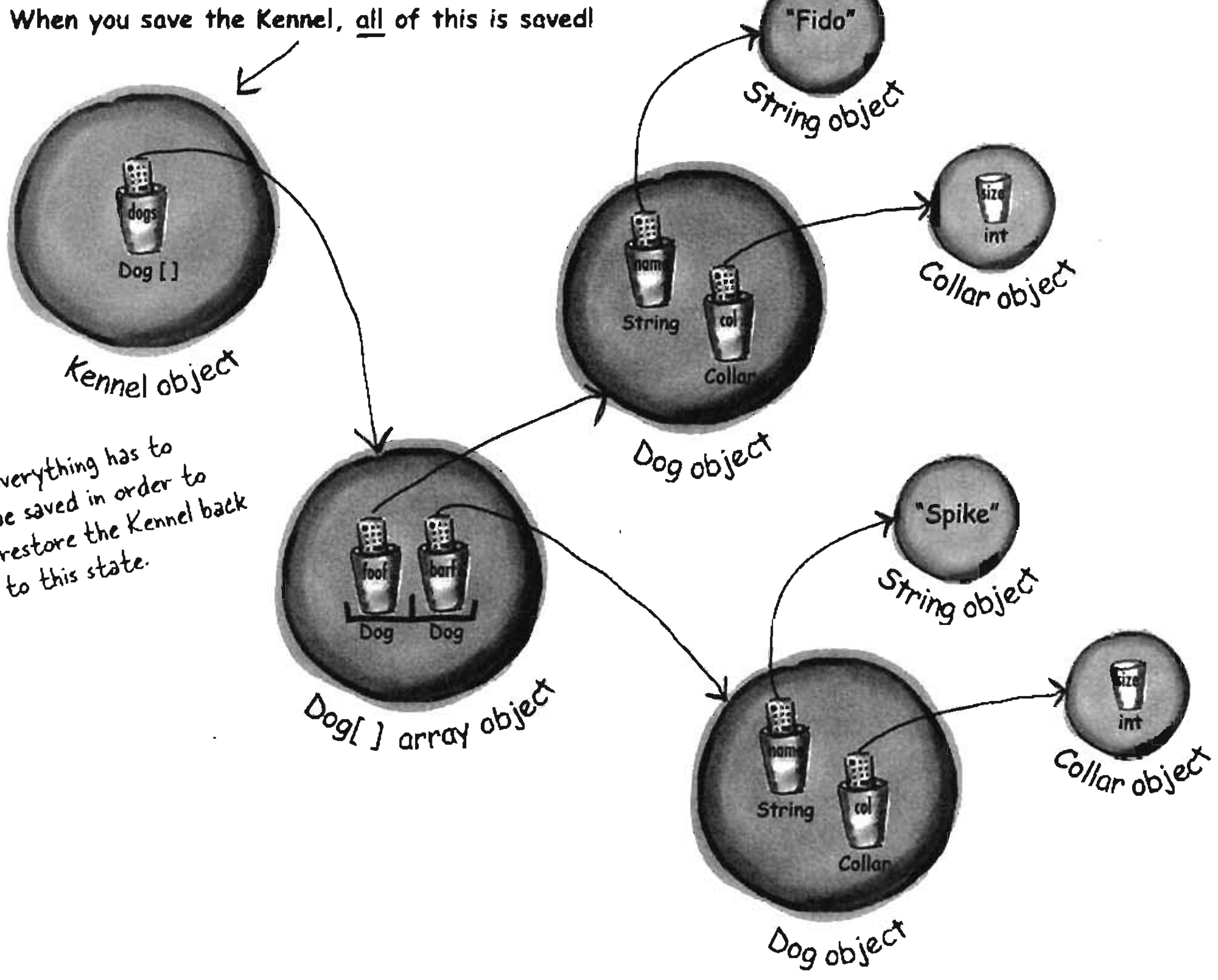
What does it take to save a Car object?

When an object is serialized, all the objects it refers to from instance variables are *also* serialized. And all the objects *those* objects refer to are serialized. And all the objects *those* objects refer to are serialized... and the best part is, it happens automatically!

Serialization saves the entire object graph. All objects referenced by instance variables, starting with the object being serialized.

This Kennel object has a reference to a Dog [] array object. The Dog [] holds references to two Dog objects. Each Dog object holds a reference to a String and a Collar object. The String objects have a collection of characters and the Collar objects have an int.

When you save the Kennel, all of this is saved!



If you want your class to be serializable, implement Serializable

The Serializable interface is known as a *marker* or *tag* interface, because the interface doesn't have any methods to implement. Its sole purpose is to announce that the class implementing it is, well, *serializable.* In other words, objects of that type are saveable through the serialization mechanism. If any superclass of a class is serializable, the subclass is automatically serializable even if the subclass doesn't explicitly declare *implements Serializable.* (This is how interfaces always *work.* If your superclass "IS-A" Serializable, you are too).

```
objectOutputStream.writeObject(myBox);
```

Whatever goes here MUST implement Serializable or it will fail at runtime.

```
import java.io.*;

public class Box implements Serializable {

    private int width;
    private int height;

    public void setWidth(int w) {
       width = w;
    }

    public void setHeight(int h) {
       height = h;
    }

public static void main (String[] args) {

      Box myBox = new Box();
      myBox.setWidth(50);
      myBox.setHeight(20);

      try {
         FileOutputStream fs = new FileOutputStream("foo.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(myBox);
         os.close();
      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Serializable is in the java.io package, so you need the import.

No methods to implement, but when you say "implements Serializable", it says to the JVM, "it's OK to serialize objects of this type."

these two values will be saved

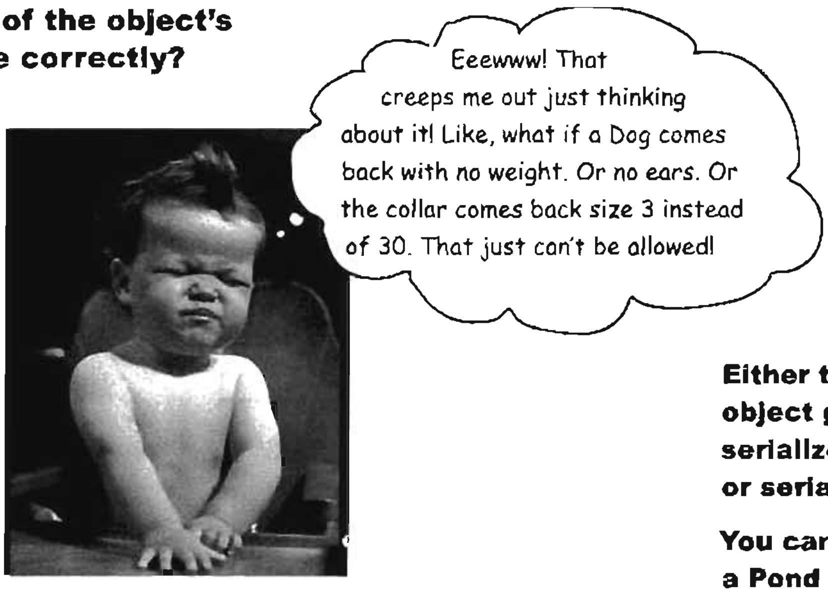
I/O operations can throw exceptions.

Connect to a file named "foo.ser" if it exists. If it doesn't, make a new file named "foo.ser".

Make an ObjectOutputStream chained to the connection stream. Tell it to write the object.

Serialization is all or nothing.

Can you imagine what would happen if some of the object's state didn't save correctly?


Eeewww! That creeps me out just thinking about it! Like, what if a Dog comes back with no weight. Or no ears. Or the collar comes back size 3 instead of 30. That just can't be allowed!

Either the entire object graph is serialized correctly or serialization fails.

You can't serialize a Pond object if its Duck instance variable refuses to be serialized (by not implementing Serializable).

```
import java.io.*;

public class Pond implements Serializable {

    private Duck duck = new Duck();

    public static void main (String[] args) {
       Pond myPond = new Pond();
       try {
         FileOutputStream fs = new FileOutputStream("Pond.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);

         os.writeObject(myPond);
         os.close();

      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Pond objects can be serialized.

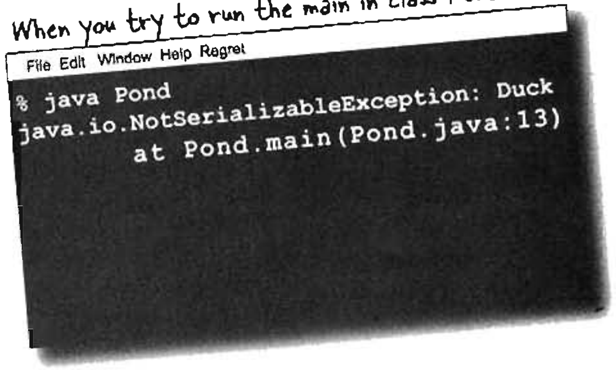
Class Pond has one instance variable, a Duck.

When you serialize myPond (a Pond object), its Duck instance variable automatically gets serialized.

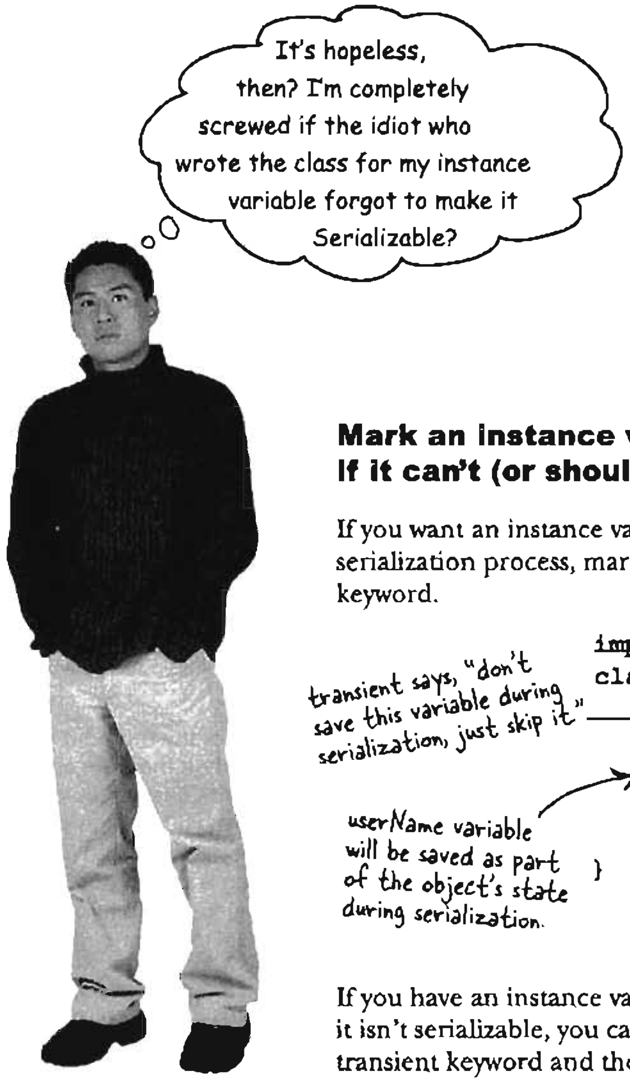
```
public class Duck {
   // duck code here
}
```

Yikes!! Duck is not serializable! It doesn't implement Serializable, so when you try to serialize a Pond object, it fails because the Pond's Duck instance variable can't be saved.

When you try to run the main in class Pond:



```
% java Pond
java.io.NotSerializableException: Duck
        at Pond.main(Pond.java:13)
```


It's hopeless, then? I'm completely screwed if the idiot who wrote the class for my instance variable forgot to make it Serializable?

Mark an instance variable as transient if it can't (or shouldn't) be saved.

If you want an instance variable to be skipped by the serialization process, mark the variable with the **`transient`** keyword.

```
import java.net.*;
class Chat implements Serializable {
    transient String currentID;

    String userName;

    // more code
}
```

transient says, "don't save this variable during serialization, just skip it."

userName variable will be saved as part of the object's state during serialization.

If you have an instance variable that can't be saved because it isn't serializable, you can mark that variable with the transient keyword and the serialization process will skip right over it.

So why would a variable not be serializable? It could be that the class designer simply *forgot* to make the class implement Serializable. Or it might be because the object relies on runtime-specific information that simply can't be saved. Although most things in the Java class libraries are serializable, you can't save things like network connections, threads, or file objects. They're all dependent on (and specific to) a particular runtime 'experience'. In other words, they're instantiated in a way that's unique to a particular run of your program, on a particular platform, in a particular JVM. Once the program shuts down, there's no way to bring those things back to life in any meaningful way; they have to be created from scratch each time.

there are no Dumb Questions

Q: If serialization is so important, why isn't it the default for all classes? Why doesn't class Object implement Serializable, and then all subclasses will be automatically Serializable.

A: Even though most classes will, and should, implement Serializable, you always have a choice. And you must make a conscious decision on a class-by-class basis, for each class you design, to 'enable' serialization by implementing Serializable.
First of all, if serialization were the default, how would you turn it off? Interfaces indicate functionality, not a *lack* of functionality, so the model of polymorphism wouldn't work correctly if you had to say, "implements NonSerializable" to tell the world that you cannot be saved.

Q: Why would I ever write a class that *wasn't* serializable?

A: There are very few reasons, but you might, for example, have a security issue where you don't want a password object stored. Or you might have an object that makes no sense to save, because its key instance variables are themselves not serializable, so there's no useful way for you to make *your* class serializable.

Q: If a class I'm using isn't serializable, but there's no good reason (except that the designer just forgot or was stupid), can I subclass the 'bad' class and make the *subclass* serializable?

A: Yes! If the class itself is extendable (i.e. not final), you can make a serializable subclass, and just substitute the subclass everywhere your code is expecting the superclass type. (Remember, polymorphism allows this.) Which brings up another interesting issue: what does it *mean* if the superclass is not serializable?

Q: You brought it up: what *does* it mean to have a serializable subclass of a non-serializable superclass?

A: First we have to look at what happens when a class is deserialized, (we'll talk about that on the next few pages). In a nutshell, when an object is deserialized and its superclass is *not* serializable, the superclass constructor will run just as though a new object of that type were being created. If there's no decent reason for a class to not be serializable, making a serializable subclass might be a good solution.

Q: Whoa! I just realized something big... if you make a variable 'transient', this means the variable's value is skipped over during serialization. Then what happens to it? We solve the problem of having a non-serializable instance variable by making the instance variable transient, but don't we NEED that variable when the object is brought back to life? In other words, isn't the whole point of serialization to preserve an object's state?

A: Yes, this is an issue, but fortunately there's a solution. If you serialize an object, a transient reference instance variable will be brought back as *null*, regardless of the value it had at the time it was saved. That means the entire object graph connected to that particular instance variable won't be saved. This could be bad, obviously, because you probably need a non-null value for that variable.

You have two options:

1) When the object is brought back, reinitialize that null instance variable back to some default state. This works if your deserialized object isn't dependent on a particular value for that transient variable. In other words, it might be important that the Dog have a Collar, but perhaps all Collar objects are the same so it doesn't matter if you give the resurrected Dog a brand new Collar; nobody will know the difference.

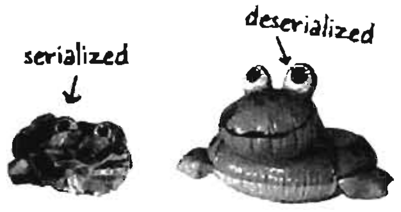
2) If the value of the transient variable *does* matter (say, if the color and design of the transient Collar are unique for each Dog) then you need to save the key values of the Collar and use them when the Dog is brought back to essentially re-create a brand new Collar that's identical to the original.

Q: What happens if two objects in the object graph are the same object? Like, if you have two different Cat objects in the Kennel, but both Cats have a reference to the same Owner object. Does the Owner get saved twice? I'm hoping not.

A: Excellent question! Serialization is smart enough to know when two objects in the graph are the same. In that case, only *one* of the objects is saved, and during deserialization, any references to that single object are restored.

Deserialization: restoring an object

The whole point of serializing an object is so that you can restore it back to its original state at some later date, in a different 'run' of the JVM (which might not even be the same JVM that was running at the time the object was serialized). Deserialization is a lot like serialization in reverse.



1 Make a FileInputStream

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

If the file "MyGame.ser" doesn't exist, you'll get an exception.

Make a FileInputStream object. The FileInputStream knows how to connect to an existing file.

2 Make an ObjectInputStream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

ObjectInputStream lets you read objects, but it can't directly connect to a file. It needs to be chained to a connection stream, in this case a FileInputStream.

3 read the objects

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Each time you say readObject(), you get the next object in the stream. So you'll read them back in the same order in which they were written. You'll get a big fat exception if you try to read more objects than you wrote.

4 Cast the objects

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

The return value of readObject() is type Object (just like with ArrayList), so you have to cast it back to the type you know it really is.

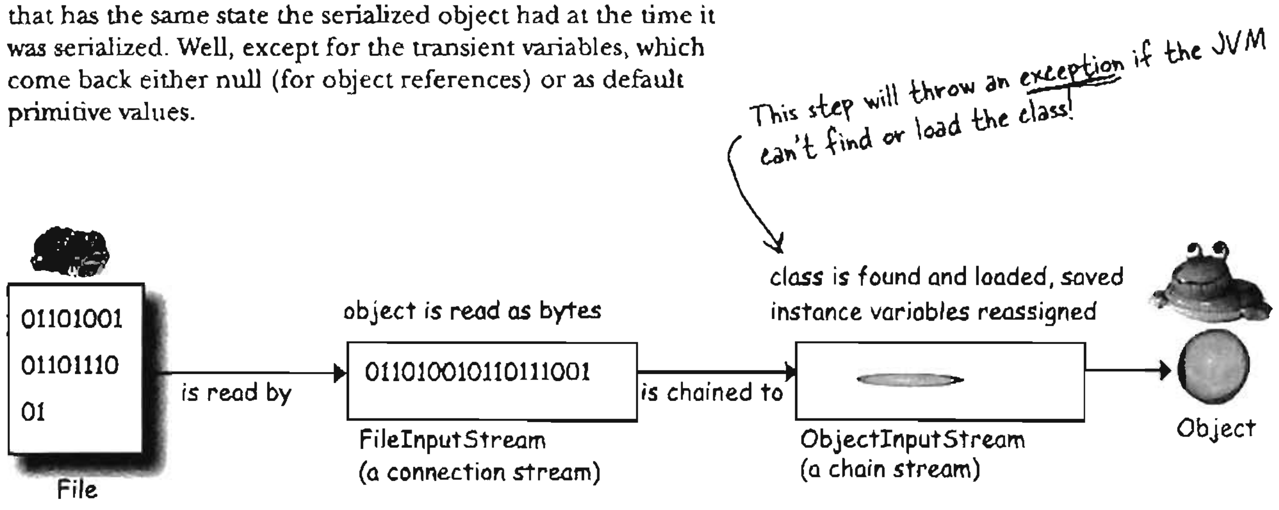
5 Close the ObjectInputStream

```
os.close();
```

Closing the stream at the top closes the ones underneath, so the FileInputStream (and the file) will close automatically.

What happens during deserialization?

When an object is deserialized, the JVM attempts to bring the object back to life by making a new object on the heap that has the same state the serialized object had at the time it was serialized. Well, except for the transient variables, which come back either null (for object references) or as default primitive values.



1. The object is **read** from the stream.

2. The JVM determines (through info stored with the serialized object) the object's **class type**.

3. The JVM attempts to **find and load** the object's **class**. If the JVM can't find and/or load the class, the JVM throws an exception and the deserialization fails.

4. A new object is given space on the heap, but the **serialized object's constructor does NOT run**! Obviously, if the constructor ran, it would restore the state of the object back to its original 'new' state, and that's not what we want. We want the object to be restored to the state it had *when it was serialized*, not when it was first created.

5 If the object has a non-serializable class somewhere up its inheritance tree, the **constructor for that non-serializable class will run** along with any constructors above that (even if they're serializable). Once the constructor chaining begins, you can't stop it, which means all superclasses, beginning with the first non-serializable one, will reinitialize their state.

6 The object's **instance variables are given the values from the serialized state**. Transient variables are given a value of null for object references and defaults (0, false, etc.) for primitives.

there are no Dumb Questions

Q: Why doesn't the class get saved as part of the object? That way you don't have the problem with whether the class can be found.

A: Sure, they could have made serialization work that way. But what a tremendous waste and overhead. And while it might not be such a hardship when you're using serialization to write objects to a file on a local hard drive, serialization is also used to send objects over a network connection. If a class was bundled with each serialized (shippable) object, bandwidth would become a much larger problem than it already is.

For objects serialized to ship over a network, though, there actually *is* a mechanism where the serialized object can be 'stamped' with a URL for where its class can be found. This is used in Java's Remote Method Invocation (RMI) so that you can send a serialized object as part of, say, a method argument, and if the JVM receiving the call doesn't have the class, it can use the URL to fetch the class from the network and load it, all automatically. (We'll talk about RMI in chapter 17.)

Q: What about static variables? Are they serialized?

A: Nope. Remember, static means "one per class" not "one per object". Static variables are not saved, and when an object is deserialized, it will have whatever static variable its class *currently* has. The moral: don't make serializable objects dependent on a dynamically-changing static variable! It might not be the same when the object comes back.

Saving and restoring the game characters

Make some characters...

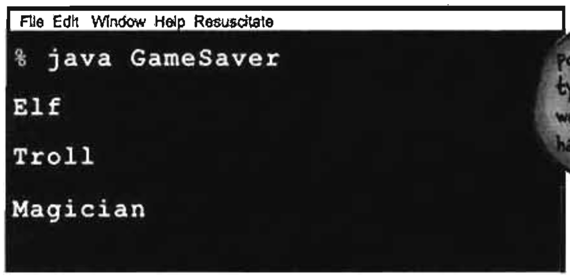
```
import java.io.*;

public class GameSaverTest {
   public static void main(String[] args) {
     GameCharacter one = new GameCharacter(50, "Elf", new String[] {"bow", "sword", "dust"});
     GameCharacter two = new GameCharacter(200, "Troll", new String[] {"bare hands", "big ax"});
     GameCharacter three = new GameCharacter(120, "Magician", new String[] {"spells", "invisibility"});

     // imagine code that does things with the characters that might change their state values

     try {
        ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Game.ser"));
        os.writeObject(one);
        os.writeObject(two);
        os.writeObject(three);
        os.close();
    } catch(IOException ex) {
      ex.printStackTrace();
    }
    one = null;
    two = null;
    three = null;
```

← We set them to null so we can't access the objects on the heap.

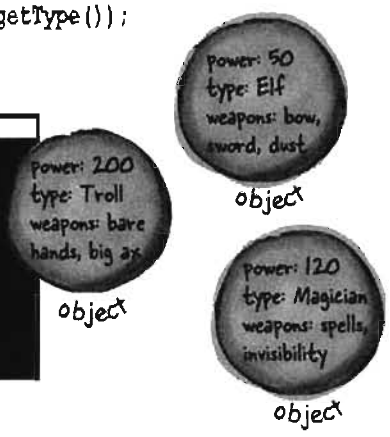
Now read them back in from the file...

```
    try {
      ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
      GameCharacter oneRestore = (GameCharacter) is.readObject();
      GameCharacter twoRestore = (GameCharacter) is.readObject();
      GameCharacter threeRestore = (GameCharacter) is.readObject();

      System.out.println("One's type: " + oneRestore.getType());
      System.out.println("Two's type: " + twoRestore.getType());
      System.out.println("Three's type: " + threeRestore.getType());
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
```

← Check to see if it worked.



```
% java GameSaver
Elf
Troll
Magician
```


power: 50
type: Elf
weapons: bow, sword, dust
object
power: 200
type: Troll
weapons: bare hands, big ax
object
power: 120
type: Magician
weapons: spells, invisibility
object

The GameCharacter class

```
import java.io.*;

public class GameCharacter implements Serializable {
   int power;
   String type;
   String[] weapons;

   public GameCharacter(int p, String t, String[] w) {
       power = p;
       type = t;
       weapons = w;
   }

   public int getPower() {
     return power;
   }

   public String getType() {
      return type;
   }

   public String getWeapons() {
       String weaponList = "";

       for (int i = 0; i < weapons.length; i++) {
          weaponList += weapons[i] + " ";
       }
       return weaponList;
   }
```

This is a basic class just for testing Serialization, and we don't have an actual game, but we'll leave that to you to experiment.

Object Serialization

BULLET POINTS

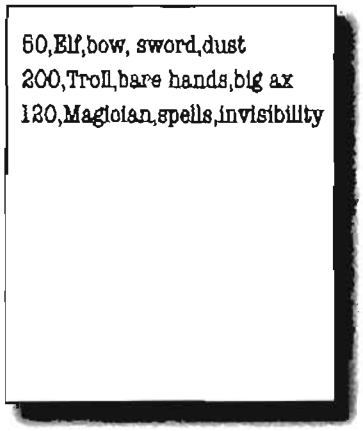
- You can save an object's state by serializing the object.
- To serialize an object, you need an ObjectOutputStream (from the java.io package)
- Streams are either connection streams or chain streams
- Connection streams can represent a connection to a source or destination, typically a file, network socket connection, or the console.
- Chain streams cannot connect to a source or destination and must be chained to a connection (or other) stream.
- To serialize an object to a file, make a FileOuputStream and chain it into an ObjectOutputStream.
- To serialize an object, call *writeObject(theObject)* on the ObjectOutputStream. You do not need to call methods on the FileOutputStream.
- To be serialized, an object must implement the Serializable interface. If a superclass of the class implements Serializable, the subclass will automatically be serializable even if it does not specifically declare *implements Serializable.*
- When an object is serialized, its entire object graph is serialized. That means any objects referenced by the serialized object's instance variables are serialized, and any objects referenced by those objects...and so on.
- If any object in the graph is not serializable, an exception will be thrown at runtime, unless the instance variable referring to the object is skipped.
- Mark an instance variable with the *transient* keyword if you want serialization to skip that variable. The variable will be restored as null (for object references) or default values (for primitives).
- During deserialization, the class of all objects in the graph must be available to the JVM.
- You read objects in (using readObject()) in the order in which they were originally written.
- The return type of readObject() is type Object, so deserialized objects must be cast to their real type.
- Static variables are not serialized! It doesn't make sense to save a static variable value as part of a specific object's state, since all objects of that type share only a single value—the one in the class.

Writing a String to a Text File

Saving objects, through serialization, is the easiest way to save and restore data between runnings of a Java program. But sometimes you need to save data to a plain old text file. Imagine your Java program has to write data to a simple text file that some other (perhaps non-Java) program needs to read. You might, for example, have a servlet (Java code running within your web server) that takes form data the user typed into a browser, and writes it to a text file that somebody else loads into a spreadsheet for analysis.

Writing text data (a String, actually) is similar to writing an object, except you write a String instead of an object, and you use a FileWriter instead of a FileOutputStream (and you don't chain it to an ObjectOutputStream).

What the game character data might look like if you wrote it out as a human-readable text file.



```
50,Elf,bow, sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

To write a serialized object:

```
objectOutputStream.writeObject(someObject);
```

To write a String:

```
fileWriter.write("My first String to save");
```

```
import java.io.*;

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

        try {
            FileWriter writer = new FileWriter("Foo.txt");

            writer.write("hello foo!");

            writer.close();

        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

We need the java.io package for FileWriter

If the file "Foo.txt" does not exist, FileWriter will create it

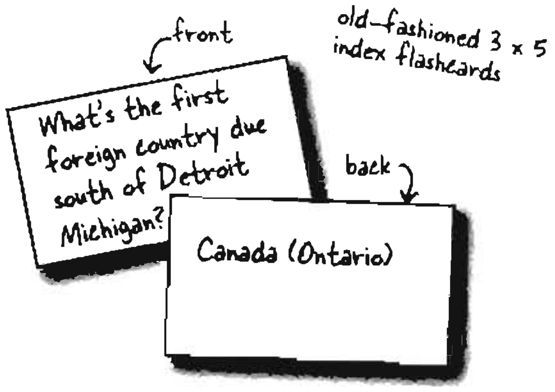
ALL the I/O stuff must be in a try/catch. Everything can throw an IOException!!

The write() method takes a String

Close it when you're done!

Text File Example: e-Flashcards

Remember those flashcards you used in school? Where you had a question on one side and the answer on the back? They aren't much help when you're trying to understand something, but nothing beats 'em for raw drill-and-practice and rote memorization. *When you have to burn in a fact.* And they're also great for trivia games.

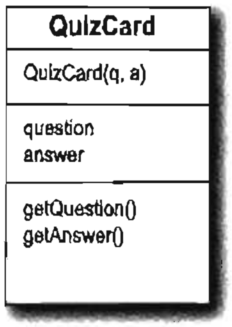

front
old-fashioned 3 x 5 index flashcards
What's the first foreign country due south of Detroit Michigan?
back
Canada (Ontario)

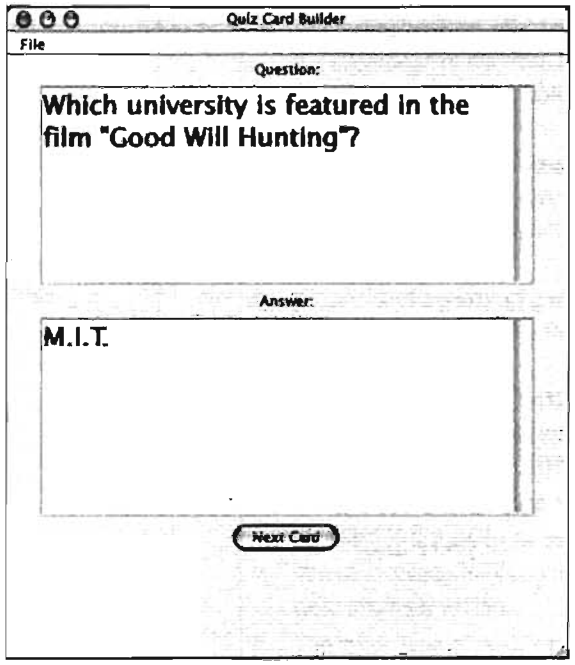
We're going to make an electronic version that has three classes:

1) ***QuizCardBuilder***, a simple authoring tool for creating and saving a set of e-Flashcards.

2) ***QuizCardPlayer***, a playback engine that can load a flashcard set and play it for the user.

3) ***QuizCard***, a simple class representing card data. We'll walk through the code for the builder and the player, and have you make the QuizCard class yourself, using this



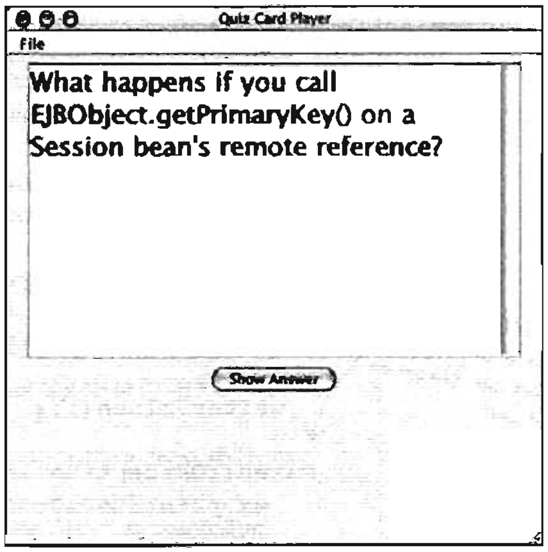


Which university is featured in the film "Good Will Hunting"?

M.I.T.

QuizCardBuilder

Has a File menu with a "Save" option for saving the current set of cards to a text file.



What happens if you call EJBObject.getPrimaryKey() on a Session bean's remote reference?

QuizCardPlayer

Has a File menu with a "Load" option for loading a set of cards from a text file.

Quiz Card Builder (code outline)

```
public class QuizCardBuilder {

   public void go() {
      // build and display gui
   }

   private class NextCardListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // add the current card to the list and clear the text areas
      }
   }

   private class SaveMenuListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // bring up a file dialog box
         // let the user name and save the set
      }
   }

   private class NewMenuListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // clear out the card list, and clear out the text areas
      }
   }

   private void saveFile(File file) {
      // iterate through the list of cards, and write each one out to a text file
      // in a parseable way (in other words, with clear separations between parts)
   }
}
```

Builds and displays the GUI, including making and registering event listeners.

Inner class

Triggered when user hits 'Next Card' button; means the user wants to store that card in the list and start a new card.

Inner class

Triggered when use chooses 'Save' from the File menu; means the user wants to save all the cards in the current list as a 'set' (like, Quantum Mechanics Set, Hollywood Trivia, Java Rules, etc.).

Inner class

Triggered by choosing 'New' from the File menu; means the user wants to start a brand new set (so we clear out the card list and the text areas).

Called by the SaveMenuListener; does the actual file writing.

Quiz Card Builder code

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardBuilder {

    private JTextArea question;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private JFrame frame;


    public static void main (String[] args) {
       QuizCardBuilder builder = new QuizCardBuilder();
       builder.go();
    }


    public void go() {
        // build gui

        frame = new JFrame("Quiz Card Builder");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);
        question = new JTextArea(6,20);
        question.setLineWrap(true);
        question.setWrapStyleWord(true);
        question.setFont(bigFont);

        JScrollPane qScroller = new JScrollPane(question);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        answer = new JTextArea(6,20);
        answer.setLineWrap(true);
        answer.setWrapStyleWord(true);
        answer.setFont(bigFont);

        JScrollPane aScroller = new JScrollPane(answer);
        aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);


        JButton nextButton = new JButton("Next Card");

        cardList = new ArrayList<QuizCard>();

        JLabel qLabel = new JLabel("Question:");
        JLabel aLabel = new JLabel("Answer:");

        mainPanel.add(qLabel);
        mainPanel.add(qScroller);
        mainPanel.add(aLabel);
        mainPanel.add(aScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem newMenuItem = new JMenuItem("New");
```

This is all GUI code here. Nothing special, although you might want to look at the MenuBar, Menu, and MenuItems code.

```
        JMenuItem saveMenuItem = new JMenuItem("Save");
        newMenuItem.addActionListener(new NewMenuListener());

        saveMenuItem.addActionListener(new SaveMenuListener());
        fileMenu.add(newMenuItem);
        fileMenu.add(saveMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(500,600);
        frame.setVisible(true);
    }
```

We make a menu bar, make a File menu, then put 'new' and 'save' menu items into the File menu. We add the menu to the menu bar, then tell the frame to use this menu bar. Menu items can fire an ActionEvent

```
    public class NextCardListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {

          QuizCard card = new QuizCard(question.getText(), answer.getText());
          cardList.add(card);
          clearCard();
       }
     }

     public class SaveMenuListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
           QuizCard card = new QuizCard(question.getText(), answer.getText());
           cardList.add(card);

           JFileChooser fileSave = new JFileChooser();
           fileSave.showSaveDialog(frame);
           saveFile(fileSave.getSelectedFile());
        }
     }
```

Brings up a file dialog box and waits on this line until the user chooses 'Save' from the dialog box. All the file dialog navigation and selecting a file, etc. is done for you by the JFileChooser! It really is this easy.

```
    public class NewMenuListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
          cardList.clear();
          clearCard();
       }
    }

    private void clearCard() {
       question.setText("");
       answer.setText("");
       question.requestFocus();
    }
```

The method that does the actual file writing (called by the SaveMenuListener's event handler). The argument is the 'File' object the user is saving. We'll look at the File class on the next page.

```
    private void saveFile(File file) {
      try {
         BufferedWriter writer = new BufferedWriter(new FileWriter(file));

         for(QuizCard card:cardList) {
            writer.write(card.getQuestion() + "/");
            writer.write(card.getAnswer() + "\n");
         }
        writer.close();

      } catch(IOException ex) {
         System.out.println("couldn't write the cardList out");
         ex.printStackTrace();
      }
    }
}
```

We chain a BufferedWriter on to a new FileWriter to make writing more efficient (We'll talk about that in a few pages).

Walk through the ArrayList of cards and write them out, one card per line, with the question and answer separated by a "/", and then add a newline character ("\n")

The java.io.File class

The java.io.File class *represents* a file on disk, but doesn't actually represent the *contents* of the file. What? Think of a File object as something more like a *pathname* of a file (or even a *directory*) rather than The Actual File Itself. The File class does not, for example, have methods for reading and writing. One VERY useful thing about a File object is that it offers a much safer way to represent a file than just using a String file name. For example, most classes that take a String file name in their constructor (like FileWriter or FileInputStream) can take a File object instead. You can construct a File object, verify that you've got a valid path, etc. and then give that File object to the FileWriter or FileInputStream.

A File object represents the name and path of a file or directory on disk, for example:

/Users/Kathy/Data/GameFile.txt

But it does NOT represent, or give you access to, the data *in* the file!

Some things you can do with a File object:

1. **Make a File object representing an existing file**

```
File f = new File("MyCode.txt");
```

2. **Make a new directory**

```
File dir = new File("Chapter7");
dir.mkdir();
```

3. **List the contents of a directory**

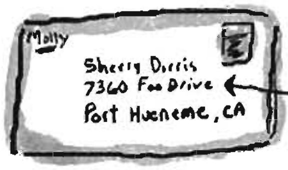
```
if (dir.isDirectory()) {
      String[] dirContents = dir.list();
      for (int i = 0; i < dirContents.length; i++) {
          System.out.println(dirContents[i]);
      }
}
```

4. **Get the absolute path of a file or directory**

```
System.out.println(dir.getAbsolutePath());
```

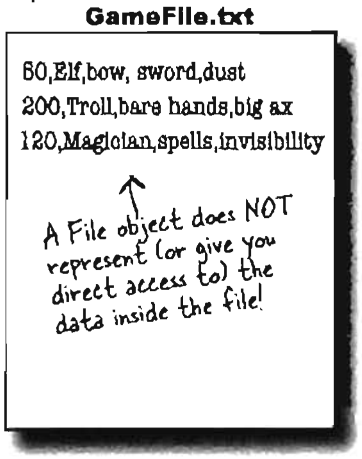
5. **Delete a file or directory (returns true if successful)**

```
boolean isDeleted = f.delete();
```



An address is NOT the same as the actual house! A File object is like a street address... it ~~represents~~ the name and location of a particular file, but it isn't the file itself.

A File object represents the filename "GameFile.txt"

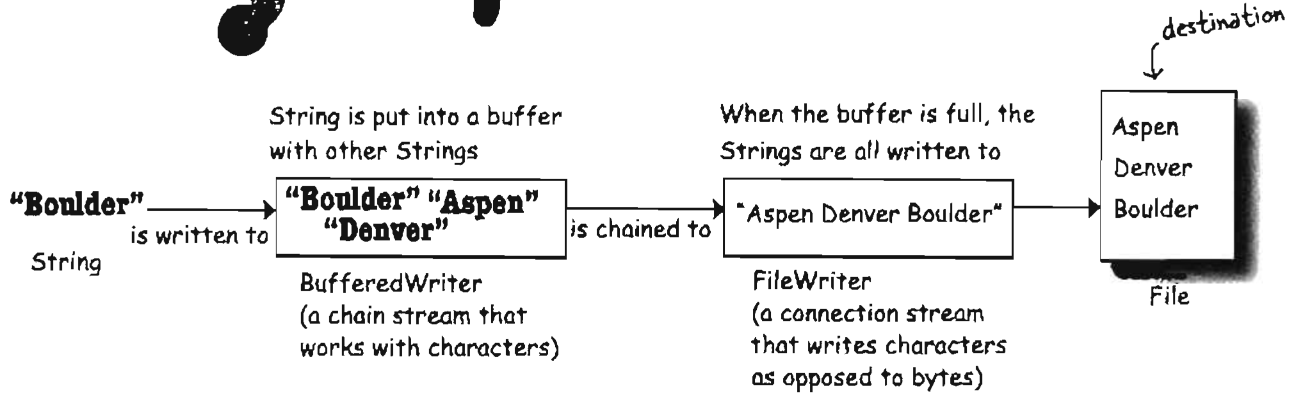


A File object does NOT represent (or give you direct access to) the data inside the file!

The beauty of buffers

If there were no buffers, it would be like shopping without a cart. You'd have to carry each thing out to your car, one soup can or toilet paper roll at a time.

buffers give you a temporary holding place to group things until the holder (like the cart) is full. You get to make far fewer trips when you use a buffer.



destination
String is put into a buffer with other Strings
When the buffer is full, the Strings are all written to
"Boulder"
is written to
String
"Boulder" "Aspen" "Denver"
is chained to
"Aspen Denver Boulder"
Aspen
Denver
Boulder
BufferedWriter (a chain stream that works with characters)
FileWriter (a connection stream that writes characters as opposed to bytes)
File

```
BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));
```

Notice that we don't even need to keep a reference to the FileWriter object. The only thing we care about is the BufferedWriter, because that's the object we'll call methods on, and when we close the BufferedWriter, it will take care of the rest of the chain.

The cool thing about buffers is that they're *much* more efficient than working without them. You can write to a file using FileWriter alone, by calling write(someString), but FileWriter writes each and every thing you pass to the file each and every time. That's overhead you don't want or need, since every trip to the disk is a Big Deal compared to manipulating data in memory. By chaining a BufferedWriter onto a FileWriter, the BufferedWriter will hold all the stuff you write to it until it's full. *Only when the buffer is full will the FileWriter actually be told to write to the file on disk.*

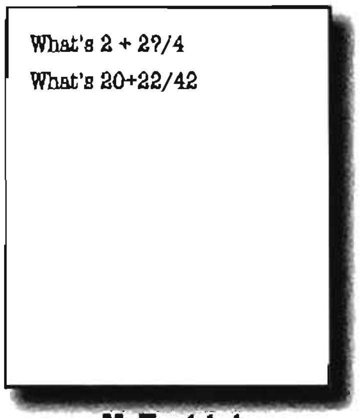
If you do want to send data *before* the buffer is full, you do have control. ***Just Flush It.*** Calls to writer.flush() say, "send whatever's in the buffer, ***now!***"

Reading from a Text File

Reading text from a file is simple, but this time we'll use a File object to represent the file, a FileReader to do the actual reading, and a BufferedReader to make the reading more efficient.

The read happens by reading lines in a *while* loop, ending the loop when the result of a readLine() is null. That's the most common style for reading data (pretty much anything that's not a Serialized object): read stuff in a while loop (actually a while loop *test*), terminating when there's nothing left to read (which we know because the result of whatever read method we're using is null).

A file with two lines of text



What's 2 + 2?/4
What's 20+22/42

MyText.txt

```
import java.io.*;
```

Don't forget the import

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

       try {
          File myFile = new File("MyText.txt");
          FileReader fileReader = new FileReader(myFile);
```

A FileReader is a connection stream for characters, that connects to a text file

```
          BufferedReader reader = new BufferedReader(fileReader);
```

Chain the FileReader to a BufferedReader for more efficient reading. It'll go back to the file to read only when the buffer is empty (because the program has read everything in it).

Make a String variable to hold each line as the line is read

```
          String line = null;

          while ((line = reader.readLine()) != null) {
             System.out.println(line);
          }
          reader.close();

      } catch(Exception ex) {
           ex.printStackTrace();
      }
   }
}
```

This says, "Read a line of text, and assign it to the String variable 'line'. While that variable is not null (because there WAS something to read) print out the line that was just read."

Or another way of saying it, "While there are still lines to read, read them and print them."

Quiz Card Player (code outline)

```
public class QuizCardPlayer {

  public void go() {
    // build and display gui
  }

  class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // if this is a question, show the answer, otherwise show next question
      // set a flag for whether we're viewing a question or answer
    }
  }

  class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // bring up a file dialog box
      // let the user navigate to and choose a card set to open
    }
  }

  private void loadFile(File file) {
    // must build an ArrayList of cards, by reading them from a text file
    // called from the OpenMenuListener event handler, reads the file one line at a time
    // and tells the makeCard() method to make a new card out of the line
    // (one line in the file holds both the question and answer, separated by a "/")
  }


  private void makeCard(String lineToParse) {
    // called by the loadFile method, takes a line from the text file
    // and parses into two pieces—question and answer—and creates a new QuizCard
    // and adds it to the ArrayList called CardList
  }
}
```

Quiz Card Player code

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardPlayer {

    private JTextArea display;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private QuizCard currentCard;
    private int currentCardIndex;
    private JFrame frame;
    private JButton nextButton;
    private boolean isShowAnswer;


    public static void main (String[] args) {
       QuizCardPlayer reader = new QuizCardPlayer();
       reader.go();
    }

    public void go() {

        // build gui

        frame = new JFrame("Quiz Card Player");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);

        display = new JTextArea(10,20);
        display.setFont(bigFont);

        display.setLineWrap(true);
        display.setEditable(false);

        JScrollPane qScroller = new JScrollPane(display);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        nextButton = new JButton("Show Question");
        mainPanel.add(qScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());

        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem loadMenuItem = new JMenuItem("Load card set");
        loadMenuItem.addActionListener(new OpenMenuListener());
        fileMenu.add(loadMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(640,500);
        frame.setVisible(true);

    } // close go
```

Just GUI code on this page; nothing special

```
public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
       if (isShowAnswer) {
          // show the answer because they've seen the question
          display.setText(currentCard.getAnswer());
          nextButton.setText("Next Card");
          isShowAnswer = false;
       } else {
          // show the next question
          if (currentCardIndex < cardList.size()) {

             showNextCard();

          } else {
             // there are no more cards!
             display.setText("That was last card");
             nextButton.setEnabled(false);
          }
       }
    }
 }
```

Check the isShowAnswer boolean flag to see if they're currently viewing a question or an answer, and do the appropriate thing depending on the answer.

```
public class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        JFileChooser fileOpen = new JFileChooser();
        fileOpen.showOpenDialog(frame);
        loadFile(fileOpen.getSelectedFile());
    }
 }
```

Bring up the file dialog box and let them navigate to and choose the file to open.

```
private void loadFile(File file) {

   cardList = new ArrayList<QuizCard>();
   try {
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line = reader.readLine()) != null) {
         makeCard(line);
      }
      reader.close();

   } catch(Exception ex) {
      System.out.println("couldn't read the card file");
      ex.printStackTrace();
   }

  // now time to start by showing the first card
  showNextCard();
}
```

Make a BufferedReader chained to a new FileReader, giving the FileReader the File object the user chose from the open file dialog.

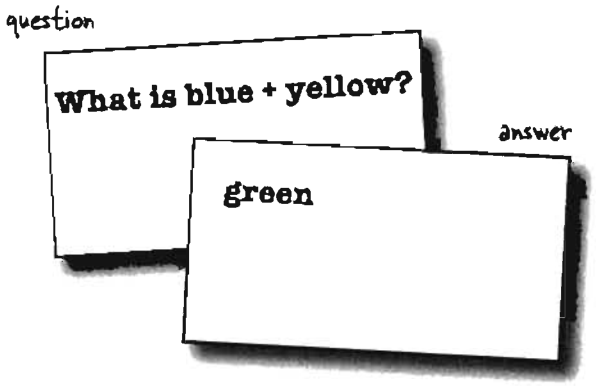
Read a line at a time, passing the line to the makeCard() method that parses it and turns it into a real QuizCard and adds it to the ArrayList.

```
private void makeCard(String lineToParse) {
   String[] result = lineToParse.split("/");
   QuizCard card = new QuizCard(result[0], result[1]);
   cardList.add(card);
   System.out.println("made a card");
}

private void showNextCard() {
     currentCard = cardList.get(currentCardIndex);
     currentCardIndex++;
     display.setText(currentCard.getQuestion());
     nextButton.setText("Show Answer");
     isShowAnswer = true;
}
// close class
```

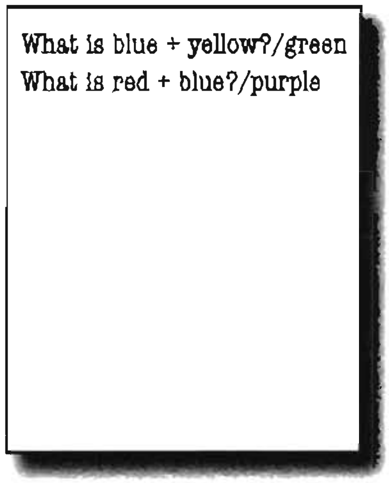
Each line of text corresponds to a single flashcard, but we have to parse out the question and answer as separate pieces. We use the String split() method to break the line into two tokens (one for the question and one for the answer). We'll look at the split() method on the next page.

Parsing with String split()

Imagine you have a flashcard like this:



Saved in a question file like this:



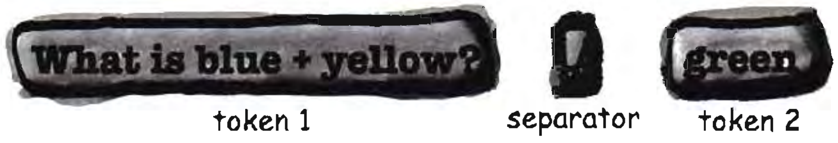
What is blue + yellow?/green
What is red + blue?/purple

How do you separate the question and answer?

When you read the file, the question and answer are smooshed together in one line, separated by a forward slash "/" (because that's how we wrote the file in the QuizCardBuilder code).

String split() lets you break a String into pieces.

The split() method says, "give me a separator, and I'll break out all the pieces of this String for you and put them in a String array."



In the QuizCardPlayer app, this is what a single line looks like when it's read in from the file.

```
String toTest = "What is blue + yellow?/green";
String[] result = toTest.split("/");
for (String token:result) {
   System.out.println(token);
}
```

The split() method takes the "/" and uses it to break apart the String into (in this case) two pieces. (Note: split() is FAR more powerful than what we're using it for here. It can do extremely complex parsing with filters, wildcards, etc.)

Loop through the array and print each token (piece). In this example, there are only two tokens: "What is blue + yellow?" and "green".

there are no Dumb Questions

Q: OK, I look in the API and there are about five million classes in the java.io package. How the heck do you know which ones to use?

A: The I/O API uses the modular 'chaining' concept so that you can hook together connection streams and chain streams (also called 'filter' streams) in a wide range of combinations to get just about anything you could want.

The chains don't have to stop at two levels; you can hook multiple chain streams to one another to get just the right amount of processing you need.

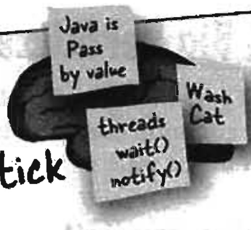
Most of the time, though, you'll use the same small handful of classes. If you're writing text files, BufferedReader and BufferedWriter (chained to FileReader and FileWriter) are probably all you need. If you're writing serialized objects, you can use ObjectOutputStream and ObjectInputStream (chained to FileInputStream and FileOutputStream).

In other words, 90% of what you might typically do with Java I/O can use what we've already covered.

Q: What about the new I/O nio classes added in 1.4?

A: The java.nio classes bring a big performance improvement and take greater advantage of native capabilities of the machine your program is running on. One of the key new features of nio is that you have direct control of buffers. Another new feature is non-blocking I/O, which means your I/O code doesn't just sit there, waiting, if there's nothing to read or write. Some of the existing classes (including FileInputStream and FileOutputStream) take advantage of some of the new features, under the covers. The nio classes are more complicated to use, however, so unless you *really* need the new features, you might want to stick with the simpler versions we've used here. Plus, if you're not careful, nio can lead to a performance *loss*. Non-nio I/O is probably right for 90% of what you'll normally do, especially if you're just getting started in Java.

But you *can* ease your way into the nio classes, by using FileInputStream and accessing its *channel* through the getChannel() method (added to FileInputStream as of version 1.4).



Make it Stick

Roses are first, violets are next.
Readers and Writers are only for text.

BULLET POINTS

- To write a text file, start with a FileWriter connection stream.
- Chain the FileWriter to a BufferedWriter for efficiency.
- A File object represents a file at a particular path, but does not represent the actual contents of the file.
- With a File object you can create, traverse, and delete directories.
- Most streams that can use a String filename can use a File object as well, and a File object can be safer to use.
- To read a text file, start with a FileReader connection stream.
- Chain the FileReader to a BufferedReader for efficiency.
- To parse a text file, you need to be sure the file is written with some way to recognize the different elements. A common approach is to use some kind of character to separate the individual pieces.
- Use the String split() method to split a String up into individual tokens. A String with one separator will have two tokens, one on each side of the separator. *The separator doesn't count as a token.*

Version ID: A Big Serialization Gotcha

Now you've seen that I/O in Java is actually pretty simple, especially if you stick to the most common connection/chain combinations. But there's one issue you might *really* care about.

Version Control is crucial!

If you serialize an object, you must have the class in order to deserialize and use the object. OK, that's obvious. But what might be less obvious is what happens if you ***change the class*** in the meantime? Yikes. Imagine trying to bring back a Dog object when one of its instance variables (non-transient) has changed from a double to a String. That violates Java's type-safe sensibilities in a Big Way. But that's not the only change that might hurt compatibility. Think about the following:

Changes to a class that can hurt deserialization:

Deleting an instance variable

Changing the declared type of an instance variable

Changing a non-transient instance variable to transient

Moving a class up or down the inheritance hierarchy

Changing a class (anywhere in the object graph) from Serializable to not Serializable (by removing 'implements Serializable' from a class declaration)

Changing an instance variable to static

Changes to a class that are usually OK:

Adding new instance variables to the class (existing objects will deserialize with default values for the instance variables they didn't have when they were serialized)

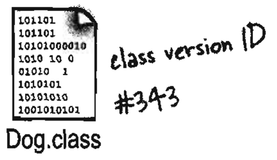
Adding classes to the inheritance tree

Removing classes from the inheritance tree

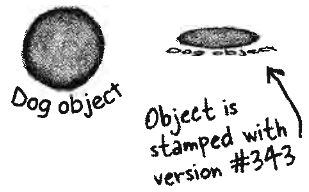
Changing the access level of an instance variable has no affect on the ability of deserialization to assign a value to the variable

Changing an instance variable from transient to non-transient (previously-serialized objects will simply have a default value for the previously-transient variables)

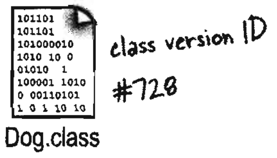
1. You write a Dog class



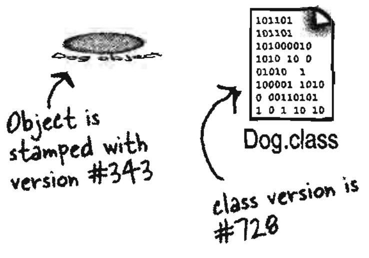
2. You serialize a Dog object using that class



3. You change the Dog class



4. You deserialize a Dog object using the changed class



5. Serailization fails!!

   The JVM says, "you can't teach an old Dog new code".

Using the serialVersionUID

Each time an object is serialized, the object (including every object in its graph) is 'stamped' with a version ID number for the object's class. The ID is called the serialVersionUID, and it's computed based on information about the class structure. As an object is being deserialized, if the class has changed since the object was serialized, the class could have a different serialVersionUID, and deserialization will fail! But you can control this.

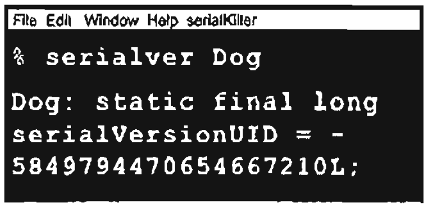
If you think there is ANY possibility that your class might *evolve*, put a serial version ID in your class.

When Java tries to deserialize an object, it compares the serialized object's serialVersionUID with that of the class the JVM is using for deserializing the object. For example, if a Dog instance was serialized with an ID of, say 23 (in reality a serialVersionUID is much longer), when the JVM deserializes the Dog object it will first compare the Dog object serialVersionUID with the Dog class serialVersionUID. If the two numbers don't match, the JVM assumes the class is not compatible with the previously-serialized object, and you'll get an exception during deserialization.

So, the solution is to put a serialVersionUID in your class, and then as the class evolves, the serialVersionUID will remain the same and the JVM will say, "OK, cool, the class is compatible with this serialized object." even though the class has actually changed.

This works *only* if you're careful with your class changes! In other words, *you* are taking responsibility for any issues that come up when an older object is brought back to life with a newer class.

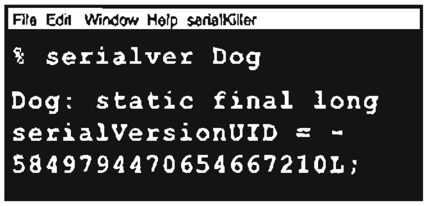
To get a serialVersionUID for a class, use the serialver tool that ships with your Java development kit.


File Edit Window Help serialKiller

```
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

When you think your class might evolve after someone has serialized objects from it...

1. Use the serialver command-line tool to get the version ID for your class


File Edit Window Help serialKiller

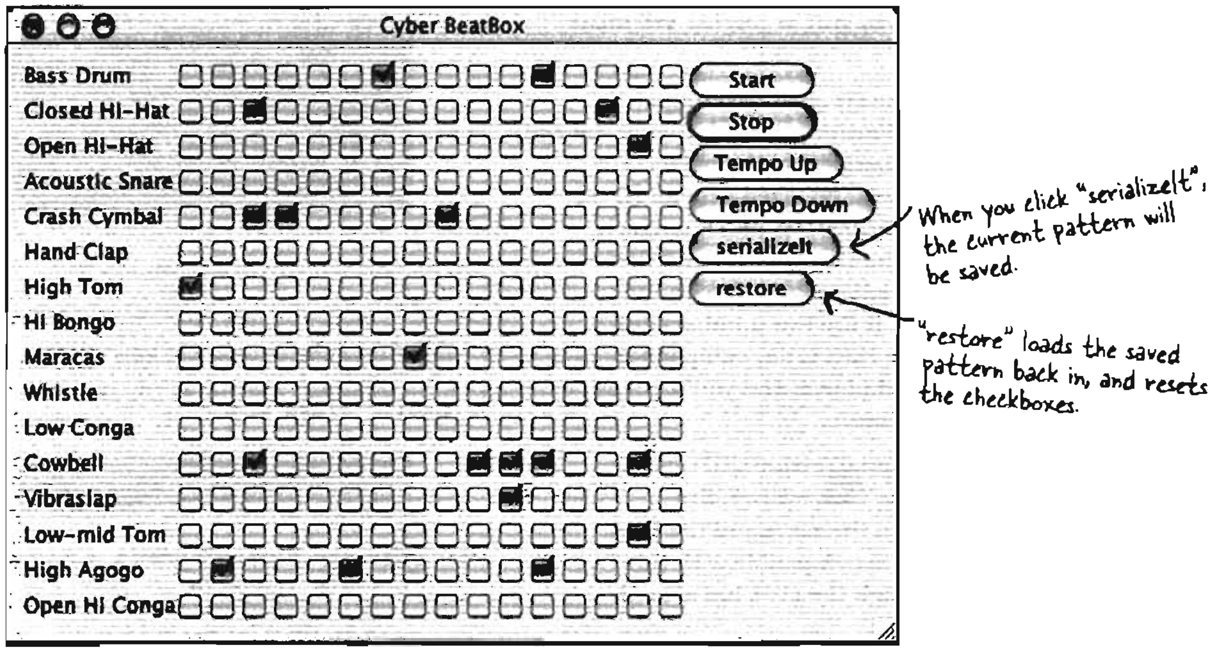
```
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

2. Paste the output into your class

```
public class Dog {

   static final long serialVersionUID =
                     -6849794470754667710L;

   private String name;
   private int size;

   // method code here
}
```

3. Be sure that when you make changes to the class, you take responsibility in your code for the consequences of the changes you made to the class! For example, be sure that your new Dog class can deal with an old Dog being deserialized with default values for instance variables added to the class *after* the Dog was serialized.

Code Kitchen



Let's make the BeatBox save and restore our favorite pattern

Saving a BeatBox pattern

Remember, in the BeatBox, a drum pattern is nothing more than a bunch of checkboxes. When it's time to play the sequence, the code walks through the checkboxes to figure out which drums sounds are playing at each of the 16 beats. So to save a pattern, all we need to do is save the state of the checkboxes.

We can make a simple boolean array, holding the state of each of the 256 checkboxes. An array object is serializable as long as the things *in* the array are serializable, so we'll have no trouble saving an array of booleans.

To load a pattern back in, we read the single boolean array object (deserialize it), and restore the checkboxes. Most of the code you've already seen, in the Code Kitchen where we built the BeatBox GUI, so in this chapter, we look at only the save and restore code.

This CodeKitchen gets us ready for the next chapter, where instead of writing the pattern to a *file*, we send it over the *network* to the server. And instead of loading a pattern *in* from a file, we get patterns from the *server*, each time a participant sends one to the server.

Serializing a pattern

This is an inner class inside the BeatBox code.

```
public class MySendListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {

        boolean[] checkboxState = new boolean[256];

        for (int i = 0; i < 256; i++) {

            JCheckBox check = (JCheckBox) checkboxList.get(i);
            if (check.isSelected()) {
               checkboxState[i] = true;
            }
        }

        try {
           FileOutputStream fileStream = new FileOutputStream(new File("Checkbox.ser"));
           ObjectOutputStream os = new ObjectOutputStream(fileStream);
           os.writeObject(checkboxState);
        } catch(Exception ex) {
           ex.printStackTrace();
        }

      } // close method
    } // close inner class
```

It all happens when the user clicks the button and the ActionEvent fires.

Make a boolean array to hold the state of each checkbox.

Walk through the checkboxList (ArrayList of checkboxes), and get the state of each one, and add it to the boolean array.

This part's a piece of cake. Just write/serialize the one boolean array!

Restoring a BeatBox pattern

This is pretty much the save in reverse... read the boolean array and use it to restore the state of the GUI checkboxes. It all happens when the user hits the "restore" 'button.

Restoring a pattern

This is another inner class inside the BeatBox class.

```
public class MyReadInListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {
       boolean[] checkboxState = null;
       try {
           FileInputStream fileIn = new FileInputStream(new File("Checkbox.ser"));
           ObjectInputStream is = new ObjectInputStream(fileIn);
           checkboxState = (boolean[]) is.readObject();

       } catch(Exception ex) {ex.printStackTrace();}


       for (int i = 0; i < 256; i++) {
          JCheckBox check = (JCheckBox) checkboxList.get(i);
          if (checkboxState[i]) {
             check.setSelected(true);
          } else {
             check.setSelected(false);
           }
       }

       sequencer.stop();
       buildTrackAndStart();

     } // close method
  } // close inner class
```

Read the single object in the file (the boolean array) and cast it back to a boolean array (remember, readObject() returns a reference of type Object.

Now restore the state of each of the checkboxes in the ArrayList of actual JCheckBox objects (checkboxList).

Now stop whatever is currently playing, and rebuild the sequence using the new state of the checkboxes in the ArrayList.

Sharpen your pencil

This version has a huge limitation! When you hit the "serializeIt" button, it serializes automatically, to a file named "Checkbox.ser" (which gets created if it doesn't exist). But each time you save, you overwrite the previously-saved file.

Improve the save and restore feature, by incorporating a JFileChooser so that you can name and save as many different patterns as you like, and load/restore from *any* of your previously-saved pattern files.

Sharpen your pencil

Can they be saved?

Which of these do you think are, or should be, serializable? If not, why not? Not meaningful? Security risk? Only works for the current execution of the JVM? Make your best guess, without looking it up in the API.

Object type	**Serializable?**	**If not, why not?**
Object	Yes / No	______
String	Yes / No	______
File	Yes / No	______
Date	Yes / No	______
OutputStream	Yes / No	______
JFrame	Yes / No	______
Integer	Yes / No	______
System	Yes / No	______

What's Legal?

Circle the code fragments that would compile (assuming they're within a legal class).

```
FileReader fileReader = new FileReader();
BufferedReader reader = new BufferedReader(fileReader);
```

```
FileOutputStream f = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);
```

```
BufferedReader reader = new BufferedReader(new FileReader(file));
String line = null;
while ((line = reader.readLine()) != null) {
     makeCard(line);
}
```

```
ObjectInputStream is = new ObjectInputStream(new FileOutputStream("Game.ser"));
GameCharacter oneAgain = (GameCharacter) is.readObject();
```

This chapter explored the wonerful world of Java I/O. Your job is to decide whether each of the following I/O-related statements is true or false.

True or False

1. Serialization is appropriate when saving data for non-Java programs to use.

2. Object state can be saved only by using serialization.

3. ObjectOutputStream is a class used to save serialized objects.

4. Chain streams can be used on their own or with connection streams.

5. A single call to writeObject() can cause many objects to be saved.

6. All classes are serializable by default.

7. The transient modifier allows you to make instance variables serializable.

8. If a superclass is not serializable then the subclass can't be serializable.

9. When objects are deserialized, they are read back in last-in, first out sequence.

10. When an object is deserialized, its constructor does not run.

11. Both serialization and saving to a text file can throw exceptions.

12. BufferedWriters can be chained to FileWriters.

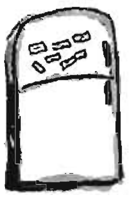
13. File objects represent files, but not directories.

14. You can't force a buffer to send its data before it's full.

15. Both file readers and file writers can be buffered.

16. The String split() method includes separators as tokens in the result array.

17. *Any* change to a class breaks previously serialized objects of that class.



Code Magnets

This one's tricky, so we promoted it from an Exercise to full Puzzle status. Reconstruct the code snippets to make a working Java program that produces the output listed below? (You might not need all of the magnets, and you may reuse a magnet more than once.)

```
class DungeonGame implements Serializable {
```

```
try {
```

```
FileOutputStream fos = new
   FileOutputStream("dg.ser");
```

```
short getZ() {
  return z;
```

```
e.printStackTrace();
```

```
oos.close();
```

```
ObjectInputStream ois = new
  ObjectInputStream(fis);
```

```
int getX() {
  return x;
```

```
System.out.println(d.getX()+d.getY()+d.getZ());
```

```
FileInputStream fis = new
  FileInputStream("dg.ser");
```

```
ublic int x = 3;
ransient long y = 4;
private short z = 5;
```

```
long getY() {
  return y;
```

```
class DungeonTest {
```

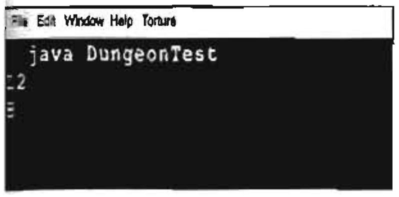
```
ois.close();
```

```
import java.io.*;
```

```
fos.writeObject(d);
```

```
} catch (Exception e) {
```

```
d = (DungeonGame) ois.readObject();
```



```
 java DungeonTest
2
```

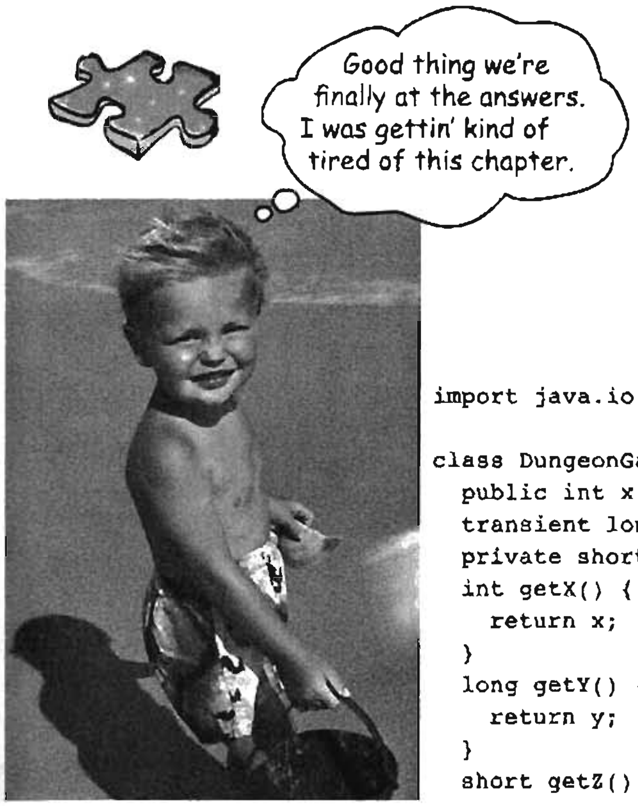
```
ObjectOutputStream oos = new
  ObjectOutputStream(fos);
```

```
oos.writeObject(d);
```

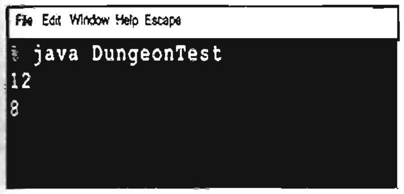
```
public static void main(String [] args) {
  DungeonGame d = new DungeonGame();
```

Exercise Solutions

1. Serialization is appropriate when saving data for non-Java programs to use.	**False**
2. Object state can be saved only by using serialization.	**False**
3. ObjectOutputStream is a class used to save serialized objects.	**True**
4. Chain streams can be usedon their own or with connection streams.	**False**
5. A single call to writeObject() can cause many objects to be saved.	**True**
6. All classes are serializable by default.	**False**
7. The transient modifier allows you to make instance variables serializable.	**False**
8. If a superclass is not serializable then the subclass can't be serializable.	**False**
9. When objects are deserialized they are read back in last-in, first out sequence.	**False**
10. When an object is deserialized, its constructor does not run.	**True**
11. Both serialization and saving to a text file can throw exceptions.	**True**
12. BufferedWriters can be chained to FileWriters.	**True**
13. File objects represent files, but not directories.	**False**
14. You can't force a buffer to send its data before it's full.	**False**
15. Both file readers and file writers can optionally be buffered.	**True**
16. The String split() method includes separators as tokens in the result array.	**False**
17. *Any* change to a class breaks previously serialized objects of that class.	**False**


Good thing we're finally at the answers. I was gettin' kind of tired of this chapter.

```
import java.io.*;

class DungeonGame implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DungeonTest {
  public static void main(String [] args) {
    DungeonGame d = new DungeonGame();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dg.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dg.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (DungeonGame) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```



```
% java DungeonTest
12
8
```