Initializing a static variable

Static variables are initialized when a *class is loaded*. A class is loaded because the JVM decides it's time to load it. Typically, the JVM loads a class because somebody's trying to make a new instance of the class, for the first time, or use a static method or variable of the class. As a programmer, you also have the option of telling the JVM to load a class, but you're not likely to need to do that. In nearly all cases, you're better off letting the JVM decide when to *load* the class.

All static variables in a class are initialized *before* any object of that class can be created.

And there are two guarantees about static initialization:

Static variables in a class are initialized before any *object* of that class can be created.

Static variables in a class are initialized before any *static method* of the class runs.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Tiger Woods");
        System.out.println(Player.playerCount);
    }
}
```

The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Default values for declared but uninitialized static and instance variables are the same:

primitive integers (long, short, etc.): 0

primitive floating points (float, double): 0.0

boolean: false

object references: null

Access a static variable just like a static method—with the class name.

Static variables are initialized when the class is loaded. If you don't explicitly initialize a static variable (by assigning it a value at the time you declare it), it gets a default value, so int variables are initialized to zero, which means we didn't need to explicitly say "playerCount = 0". Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.

```
File Edit Window Help What?
% java PlayerTestDrive
0   <- before any instances are made
1   <- after an object is created
```

static final variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up Math.PI in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

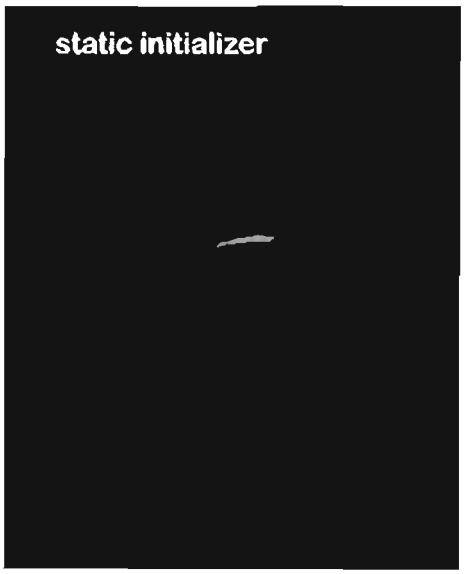
The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one.

Constant variable names should be in all caps!



Initialize a *final* static variable:

● **At the time you declare it:**

```
public class Foo {
   public static final int FOO_X = 25;
}
```

notice the naming convention -- static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

● **In a static initializer:**

```
public class Bar {
   public static final double BAR_SIGN;

   static {
      BAR_SIGN = (double) Math.random();
   }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class Bar {
   public static final double BAR_SIGN;
}
```

no initialization!

The compiler will catch it:

```
File Edit Window Help Jack-In
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

final isn't just for static variables...

You can use the keyword **final** to modify non-static variables too, including instance variables, local variables, and even method parameters. In each case, it means the same thing: the value can't be changed. But you can also use final to stop someone from overriding a method or making a subclass.

non-static final variables

```
class Foof {
    final int size = 3; <- now you can't change size
    final int whuffie;

    Foof() {
       whuffie = 42; <- now you can't change whuffie
    }

    void doStuff(final int x) {
       // you can't change x
    }

    void doMore() {
       final int z = 7;
       // you can't change z
    }
}
```

final method

```
class Poof {
    final void calcWhuffie() {
       // important things
       // that must never be overridden
    }
}
```

final class

```
final class MyMostPerfectClass {
   // cannot be extended
}
```

A final variable means you can't change its value.

A final method means you can't override the method.

A final class means you can't extend the class (i.e. you can't make a subclass).

there are no Dumb Questions

Q: A static method can't access a non-static variable. But can a non-static method access a static variable?

A: Of course. A non-static method in a class can always call a static method in the class or access a static variable of the class.

Q: Why would I want to make a class final? Doesn't that defeat the whole purpose of OO?

A: Yes and no. A typical reason for making a class final is for security. You can't, for example, make a subclass of the String class. Imagine the havoc if someone extended the String class and substituted their own String subclass objects, polymorphically, where String objects are expected. If you need to count on a particular implementation of the methods in a class, make the class final.

Q: Isn't it redundant to have to mark the methods final if the class is final?

A: If the class is final, you don't need to mark the methods final. Think about it—if a class is final it can never be subclassed, so none of the methods can ever be overridden.

On the other hand, if you *do* want to allow others to extend your class, and you want them to be able to override some, but not all, of the methods, then don't mark the class final but go in and selectively mark specific methods as final. A final method means that a subclass can't override that particular method.

BULLET POINTS

- A ***static method*** should be called using the class name rather than an object reference variable: `Math.random()` vs. `myFoo.go()`
- A static method can be invoked without any instances of the method's class on the heap.
- A static method is good for a utility method that does not (and will never) depend on a particular instance variable value.
- A static method is not associated with a particular instance—only the class—so it cannot access any instance variable values of its class. It wouldn't know *which* instance's values to use.
- A static method cannot access a non-static method, since non-static methods are usually associated with instance variable state.
- If you have a class with only static methods, and you do not want the class to be instantiated, you can mark the constructor private.
- A ***static variable*** is a variable shared by all members of a given class. There is only one copy of a static variable in a class, rather than one copy per each individual instance for instance variables.
- A static method can access a static variable.
- To make a constant in Java, mark a variable as both static and final.
- A final static variable must be assigned a value either at the time it is declared, or in a static initializer.

```
static {
    DOG_CODE = 420;
}
```

- The naming convention for constants (final static variables) is to make the name all uppercase.
- A final variable value cannot be changed once it has been assigned.
- Assigning a value to a final *instance* variable must be either at the time it is declared, or in the constructor.
- A final method cannot be overridden.
- A final class cannot be extended (subclassed).

Sharpen your pencil

What's Legal?

Given everything you've just learned about static and final, which of these would compile?

A
```
public class Foo {
   static int x;

   public void go() {
      System.out.println(x);
   }
}
```

B
```
public class Foo2 {
  int x;

  public static void go() {
      System.out.println(x);
  }
}
```

C
```
public class Foo3 {
  final int x;

  public void go() {
      System.out.println(x);
  }
}
```

D
```
public class Foo4 {
  static final int x = 12;

  public void go() {
      System.out.println(x);
  }
}
```

E
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
      System.out.println(x);
  }
}
```

F
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
      System.out.println(x);
  }
}
```

Math methods

Now that we know how static methods work, let's look at some static methods in class Math. This isn't all of them, just the highlights. Check your API for the rest including sqrt(), tan(), ceil(), floor(), and asin().

Math.random()

Returns a double between 0.0 through (but not including) 1.0.

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int it returns an int. Pass it a double it returns a double.

```
int x = Math.abs(-240);  // returns 240
double d = Math.abs(240.45);  // returns  240.45
```

Math.round()

Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.

```
int x = Math.round(-24.8f);  // returns -25
int y = Math.round(24.45f);  // returns 24
```

Remember, floating point literals are assumed to be doubles unless you add the 'f'.

Math.min()

Returns a value that is the minimum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.min(24,240);  // returns 24
double y = Math.min(90876.5, 90876.49);  // returns 90876.49
```

Math.max()

Returns a value that is the maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.max(24,240);  // returns 240
double y = Math.max(90876.5, 90876.49);  // returns 90876.5
```

Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, in all versions of Java prior to 5.0, you cannot put a primitive directly into a collection like ArrayList or HashMap:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

This won't work unless you're using Java 5.0 or greater!! There's no add(int) method in ArrayList that takes an int! (ArrayList only has add() methods that take object references, not primitives.)

There's a wrapper class for every primitive type, and since the wrapper classes are in the java.lang package, you don't need to import them. You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.

Oh yeah, for reasons absolutely nobody on the planet is certain of, the API designers decided not to map the names *exactly* from primitive type to class type. You'll see what we mean:

Boolean
Character
Byte
Short
Integer
Long
Float
Double

Watch out! The names aren't mapped exactly to the primitive types. The class names are fully spelled out.

wrapping a value

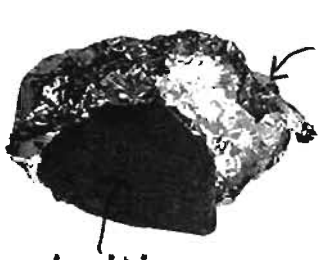
Give the primitive to the wrapper constructor. That's it.

```
int i = 288;
Integer iWrap = new Integer(i);
```

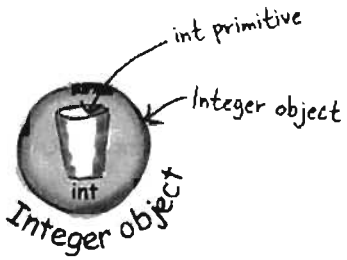
*un*wrapping a value

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

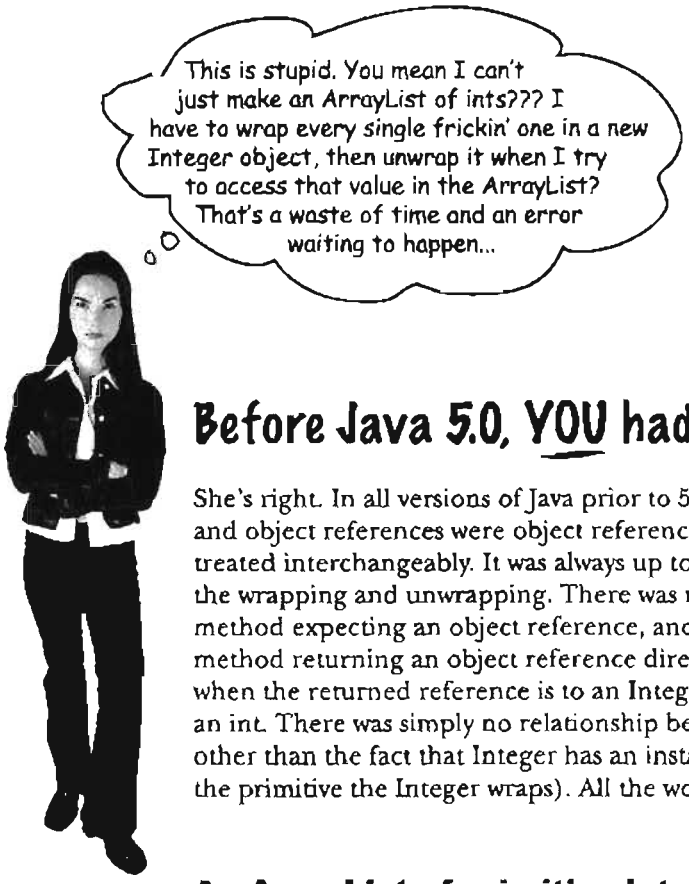
```
int unWrapped = iWrap.intValue();
```



When you need to treat a primitive like an object, wrap it. If you're using any version of Java before 5.0, you'll do this when you need to store a primitive value inside a collection like ArrayList or HashMap.



Note: the picture at the top is a chocolate in a foil wrapper. Get it? Wrapper? Some people think it looks like a baked potato, but that works too.



Before Java 5.0, YOU had to do the work...

She's right. In all versions of Java prior to 5.0, primitives were primitives and object references were object references, and they were NEVER treated interchangeably. It was always up to you, the programmer, to do the wrapping and unwrapping. There was no way to pass a primitive to a method expecting an object reference, and no way to assign the result of a method returning an object reference directly to a primitive variable—even when the returned reference is to an Integer and the primitive variable is an int. There was simply no relationship between an Integer and an int, other than the fact that Integer has an instance variable of type int (to hold the primitive the Integer wraps). All the work was up to you.

An ArrayList of primitive ints

Without autoboxing (Java versions *before* 5.0)

```
public void doNumsOldWay() {
    ArrayList listOfNumbers = new ArrayList();
    listOfNumbers.add(new Integer(3));
    Integer one = (Integer) listOfNumbers.get(0);
    int intOne = one.intValue();
}
```

Make an ArrayList (Remember, before 5.0 you could not specify the TYPE, so all ArrayLists were lists of Objects.)

You can't add the primitive '3' to the list, so you have to wrap it in an Integer first.

It comes out as type Object, but you can cast the Object to an Integer.

Finally you can get the primitive out of the Integer.

Autoboxing: blurring the line between primitive and object

The autoboxing feature added to Java 5.0 does the conversion from primitive to wrapper object *automatically!*

Let's see what happens when we want to make an ArrayList to hold ints.

An ArrayList of primitive ints

With autoboxing (Java versions 5.0 or greater)

```
public void doNumsNewWay() {

    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();

    listOfNumbers.add(3);

    int num = listOfNumbers.get(0);
}
```

Make an ArrayList of type Integer.

Just add it!

Although there is NOT a method in ArrayList for add(int), the compiler does all the wrapping (boxing) for you. In other words, there really IS an Integer object stored in the ArrayList, but you get to "pretend" that the ArrayList takes ints. (You can add both ints and Integers to an ArrayList<Integer>.)

And the compiler automatically unwraps (unboxes) the Integer object so you can assign the int value directly to a primitive without having to call the intValue() method on the Integer object.

Q: Why not declare an ArrayList<int> if you want to hold ints?

A: Because... *you can't*. Remember, the rule for generic types is that you can specify only class or interface types, *not primitives*. So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to *prevent* you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, if you're using a Java 5.0-compliant compiler, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

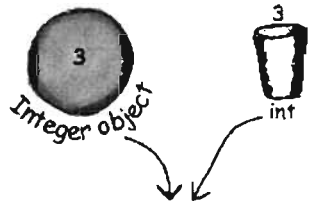
Autoboxing works almost everywhere

Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection... it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

Fun with autoboxing

Method arguments

If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.

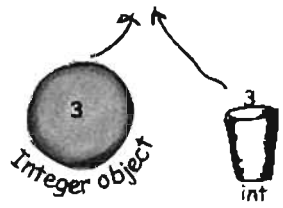



```
void takeNumber(Integer i) { }
```

Return values

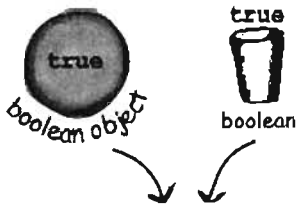
If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.

```
int giveNumber() {
  return x;
}
```



Boolean expressions

Any place a boolean value is expected, you can use either an expression that evaluates to a boolean (4 > 2), or a primitive boolean, or a reference to a Boolean wrapper.




```
if (bool) {
    System.out.println("true");
}
```

Operations on numbers

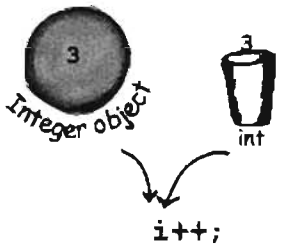
This is probably the strangest one—yes, you can now use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!

But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation. It sure looks weird, though.

```
Integer i = new Integer(42);
i++;
```

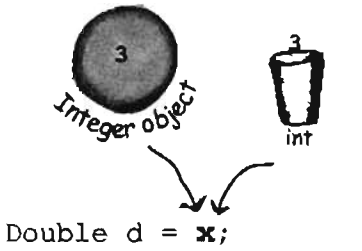
And that means you can also do things like:

```
Integer j = new Integer(5);
Integer k = j + 3;
```



Assignments

You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice-versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.



Sharpen your pencil

Will this code compile? Will it run? If it runs, what will it do?

Take your time and think about this one; it brings up an implication of autoboxing that we didn't talk about.

You'll have to go to your compiler to find the answers. (Yes, we're forcing you to experiment, for your own good of course.)

```
public class TestBox {

   Integer i;
   int j;

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

   public void go() {
      j=i;
      System.out.println(j);
      System.out.println(i);
   }
}
```

But wait! There's more! Wrappers have static utility methods too!

Besides acting like a normal class, the wrappers have a bunch of really useful static methods. We've used one in this book before—Integer.parseInt().

The parse methods take a String and give you back a primitive value.

Converting a String to a primitive value is easy:

No problem to parse "2" into 2

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

You'd think there would be a Boolean.parseBoolean() wouldn't you? But there isn't. Fortunately there's a Boolean constructor that takes (and parses) a String, and then you just get the primitive value by unwrapping it.

But if you try to do this:

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but at runtime it blows up. Anything that can't be parsed as a number will cause a NumberFormatException

You'll get a runtime exception:

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Every method or constructor that parses a String can throw a NumberFormatException. It's a runtime exception, so you don't have to handle or declare it. But you might want to.

(We'll talk about Exceptions in the next chapter.)

And now in reverse... turning a primitive number into a String

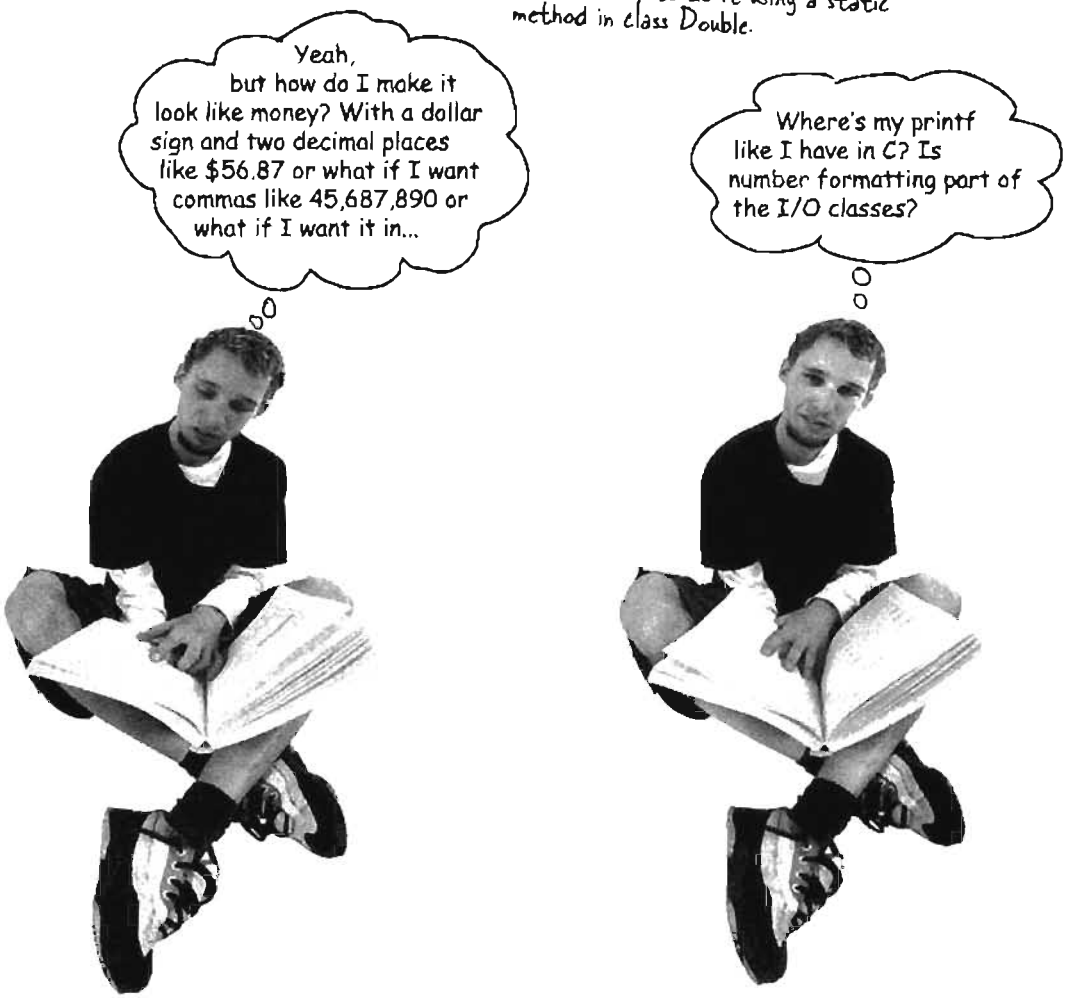
There are several ways to turn a number into a String. The easiest is to simply concatenate the number to an existing String.

```
double d = 42.5;
String doubleString = "" + d;
```

Remember the '+' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.



Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI. Before Java 5.0, most formatting was handled through classes in the java.text package that we won't even look at in this version of the book, now that things have changed.

In Java 5.0, the Java team added more powerful and flexible formatting through a Formatter class in java.util. But you don't need to create and call methods on the Formatter class yourself, because Java 5.0 added convenience methods to some of the I/O classes (including printf()) and the String class. So it's a simple matter of calling a static String.format() method and passing it the thing you want formatted along with formatting instructions.

Of course, you do have to know how to supply the formatting instructions, and that takes a little effort unless you're familiar with the ***printf()*** function in C/C++. Fortunately, even if you *don't* know printf() you can simply follow recipes for the most basic things (that we're showing in this chapter). But you *will* want to learn how to format if you want to mix and match to get *anything* you want.

We'll start here with a basic example, then look at how it works. (Note: we'll revisit formatting again in the I/O chapter.)

Formatting a number to use commas

```
public class TestFormats {

   public static void main (String[] args) {

      String s = String.format("%, d", 1000000000);
      System.out.println(s);
   }
}
```

The number to format (we want it to have commas).

The formatting instructions for how to format the second argument (which in this case is an int value). Remember, there are only two arguments to this method here—the first comma is *INSIDE* the String literal, so it isn't separating arguments to the format method.

```
1,000,000,000
```

Now we get commas inserted into the number.

Formatting deconstructed...

At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):

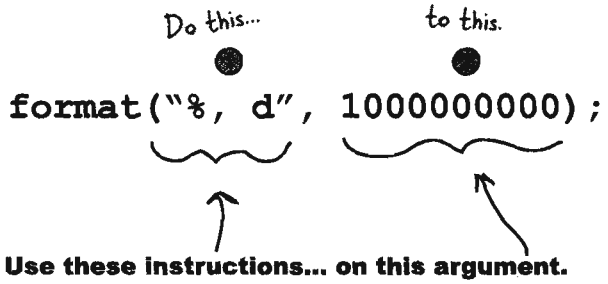
1. **Formatting Instructions**

   You use special format specifiers that describe how the argument should be formatted.

2. **The argument to be formatted.**

   Although there can be more than one argument, we'll start with just one. The argument type can't be just *anything*... it has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a *floating point number*, you can't pass in a Dog or even a String that looks like a floating point number.

Note: if you already know printf() from c/C++, you can probably just skim the next few pages. Otherwise, read carefully!



What do these instructions actually say?

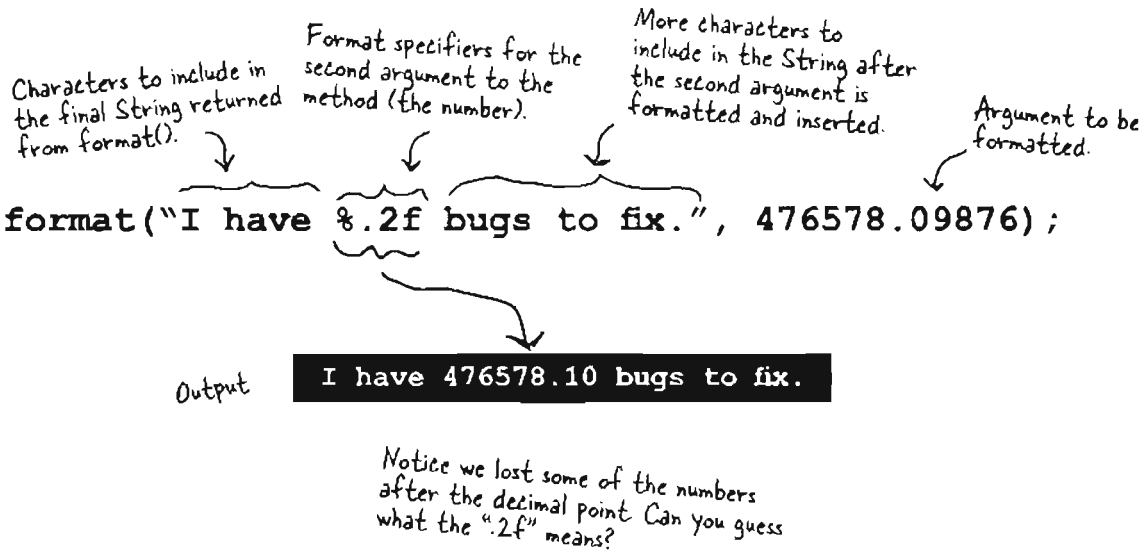
"Take the second argument to this method, and format it as a **d**ecimal integer and insert **commas**."

How do they say that?

On the next page we'll look in more detail at what the syntax "%, d" actually means, but for starters, any time you see the percent sign (%) in a format String (which is always the first argument to a format() method), think of it as representing a variable, and the variable is the other argument to the method. The rest of the characters after the percent sign describe the formatting instructions for the argument.

The percent (%) says, "insert argument here" (and format it using these instructions)

The first argument to a format() method is called the format String, and it can actually include characters that you just want printed as-is, without extra formatting. When you see the % sign, though, think of the percent sign as a variable that represents the other argument to the method.



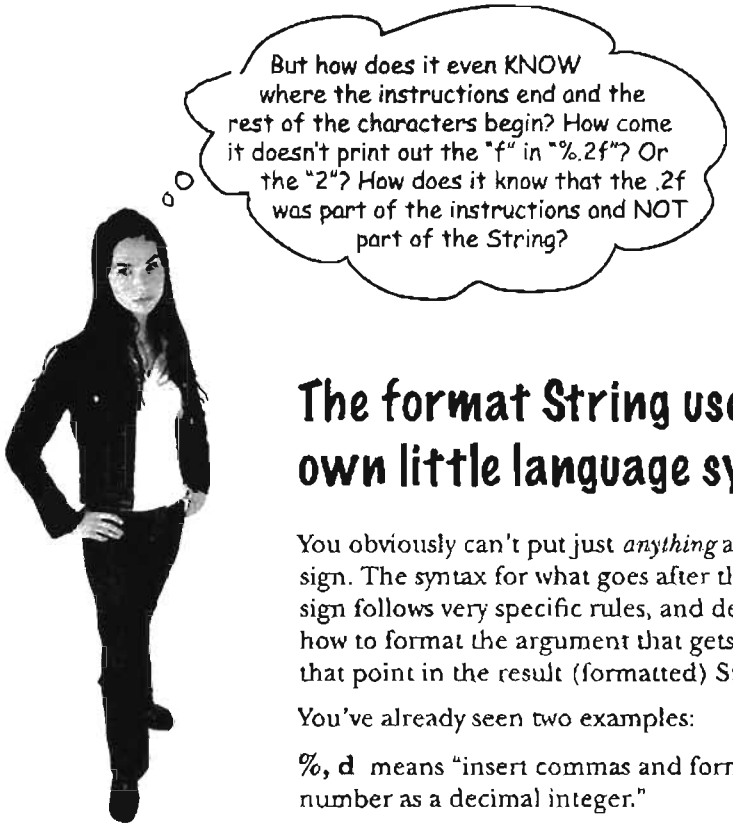
The "%" sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the ".2f" characters after the percent sign. Then the rest of the format String, "bugs to fix", is added to the final output.

Adding a comma

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

By changing the format instructions from "%.2f" to "%,.2f", we got a comma in the formatted number.



The format String uses its own little language syntax

You obviously can't put just *anything* after the "%" sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.

You've already seen two examples:

%, d means "insert commas and format the number as a decimal integer."

and

%.2f means "format the number as a floating point with a precision of two decimal places."

and

%,.2f means "insert commas and format the number as a floating point with a precision of two decimal places."

The real question is really, "How do I know what to put after the percent sign to get it to do what I want?" And that includes knowing the symbols (like "d" for decimal and "f" for floating point) as well as the order in which the instructions must be placed following the percent sign. For example, if you put the comma after the "d" like this: "%d," instead of "%,d" it won't work!

Or will it? What do you think this will do:

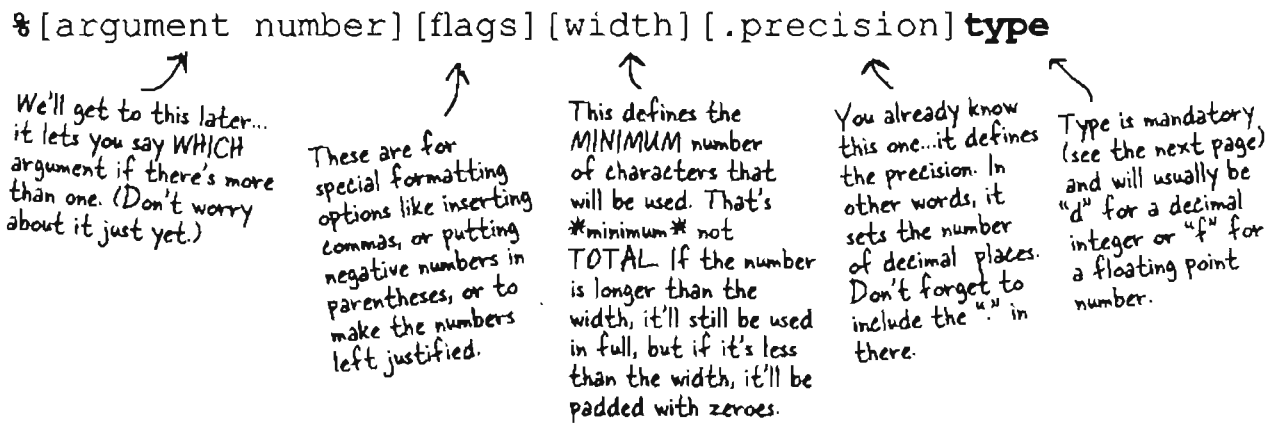
```
String.format("I  have %.2f, bugs to fix.", 476578.09876);
```

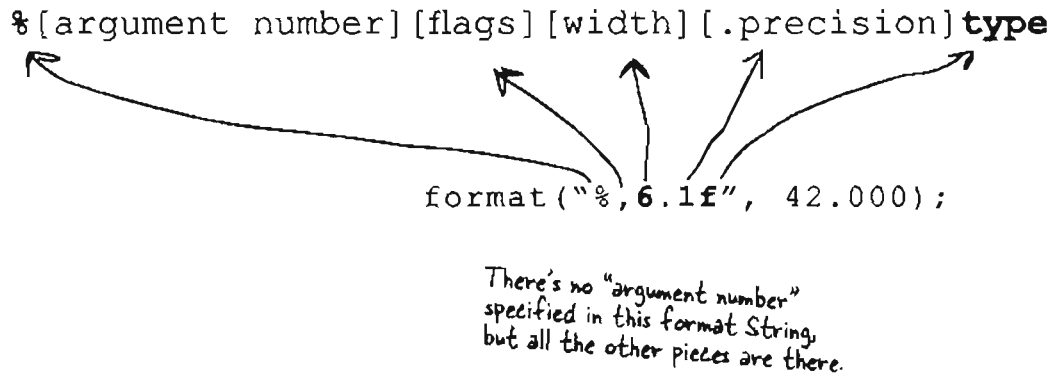
(We'll answer that on the next page.)

The format specifier

Everything after the percent sign up to and including the type indicator (like "d" or "f") are part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters are meant to be part of the output String, until or unless it hits another percent (%) sign. Hmmmm... is that even possible? Can you have more than one formatted argument variable? Put that thought on hold for right now; we'll come back to it in a few minutes. For now, let's look at the syntax for the format specifiers—the things that go after the percent (%) sign and describe how the argument should be formatted.

A format specifier can have up to five different parts (not including the "%"). Everything in brackets [ ] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.





The only required specifier is for TYPE

Although type is the only required specifier, remember that if you *do* put in anything else, type must always come last! There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you'll probably use %d (decimal) or %f (floating point). And typically you'll combine %f with a precision indicator to set the number of decimal places you want in your output.

The TYPE is mandatory, everything else is optional.

%d ***decimal***

```
format("%d", 42);
```

```
42
```

A 42.25 would not work! It would be the same as trying to directly assign a double to an int variable.

The argument must be compatible with an int, so that means only byte, short, int, and char (or their wrapper types).

%f ***floating point***

```
format("%.3f", 42.000000);
```

```
42.000
```

Here we combined the "f" with a precision indicator ".3" so we ended up with three zeroes.

The argument must be of a floating point type, so that means only a float or double (primitive or wrapper) as well as something called BigDecimal (which we don't look at in this book).

%x ***hexadecimal***

```
format("%x", 42);
```

```
2a
```

The argument must be a byte, short, int, long (including both primitive and wrapper types), and BigInteger.

%c ***character***

```
format("%c", 42);
```

```
*
```

The number 42 represents the char "*".

The argument must be a byte, short, char, or int (including both primitive and wrapper types).

You must include a type in your format instructions, and if you specify things besides type, the type must always come last. Most of the time, you'll probably format numbers using either "d" for decimal or "f" for floating point.

What happens if I have more than one argument?

Imagine you want a String that looks like this:

"The rank is *20,456,654* out of *100,567,890.24*."

But the numbers are coming from variables. What do you do? You simply add *two* arguments after the format String (first argument), so that means your call to format() will have three arguments instead of two. And inside that first argument (the format String), you'll have two different format specifiers (two things that start with "%"). The first format specifier will insert the second argument to the method, and the second format specifier will insert the third argument to the method. In other words, the variable insertions in the format String use the order in which the other arguments are passed into the format() method.

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format("The rank is %,d out of %,.2f", one, two);
```

```
The rank is 20,456,654 out of 100,567,890.25
```

When you have more than one argument, they're inserted using the order in which you pass them to the format() method.

We added commas to both variables, and restricted the floating point number (the second variable) to two decimal places.

As you'll see when we get to date formatting, you might actually want to apply different formatting specifiers to the same argument. That's probably hard to imagine until you see how *date* formatting (as opposed to the *number* formating we've been doing) works. Just know that in a minute, you'll see how to be more specific about which format specifiers are applied to which arguments.

Q: Um, there's something REALLY strange going on here. Just how many arguments *can* I pass? I mean, how many overloaded format() methods are IN the String class? So, what happens if I want to pass, say, ten different arguments to be formatted for a single output String?

A: Good catch. Yes, there *is* something strange (or at least new and different) going on, and no there are *not* a bunch of overloaded format() methods to take a different number of possible arguments. In order to support this new formatting (printf-like) API in Java, the language needed another new feature—*variable argument lists* (called *varargs* for short). We'll talk about varargs only in the appendix because outside of formatting, you probably won't use them much in a well-designed system.

So much for numbers, what about dates?

Imagine you want a String that looks like this: "Sunday, Nov 28 2004"

Nothing special there, you say? Well, imagine that all you have to start with is a variable of type Date—A Java class that can represent a timestamp, and now you want to take that object (as opposed to a number) and send it through the formatter.

The main difference between number and date formatting is that date formats use a two-character type that starts with "t" (as opposed to the single character "f" or "d", for example). The examples below should give you a good idea of how it works:

The complete date and time **%tc**

```
String.format("%tc", new Date());
```

```
Sun Nov 28 14:52:41 MST 2004
```

Just the time **%tr**

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

Day of the week, month and day **%tA %tB %td**

There isn't a single format specifier that will do exactly what we want, so we have to combine three of them for day of the week (%tA), month (%tB), and day of the month (%td).

```
Date today = new Date();
String.format("%tA, %tB %td",today,today,today)
```

The comma is not part of the formatting... it's just the character we want printed after the first inserted formatted argument.

But that means we have to pass the Date object in three times, one for each part of the format that we want. In other words, the %tA will give us just the day of the week, but then we have to do it again to get just the month and again for the day of the month.

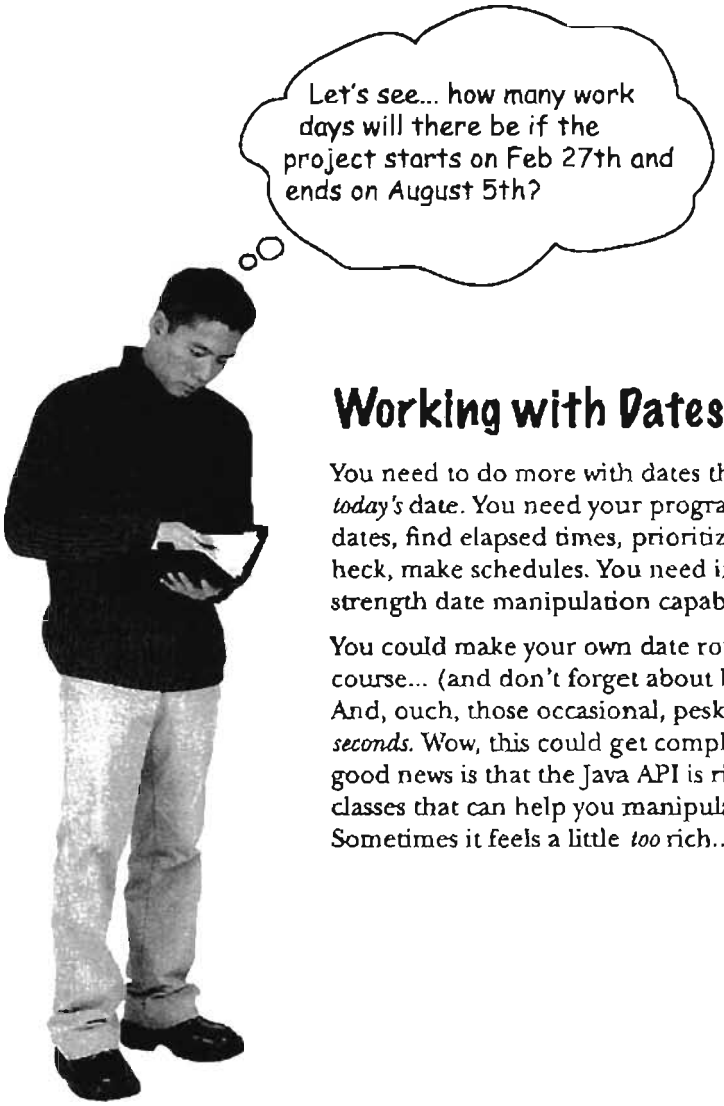
```
Sunday, November 28
```

Same as above, but *without duplicating the arguments* **%tA %tB %td**

```
Date today = new Date();
String.format("%tA, %<tB %<td",today);
```

You can think of this as kind of like calling three different getter methods on the Date object, to get three different pieces of data from it.

The angle-bracket "<" is just another flag in the specifier that tells the formatter to "use the previous argument again." So it saves you from repeating the arguments, and instead you format the same argument three different ways.



Working with Dates

You need to do more with dates than just get *today's* date. You need your programs to adjust dates, find elapsed times, prioritize schedules, heck, make schedules. You need industrial strength date manipulation capabilities.

You could make your own date routines of course... (and don't forget about leap years!) And, ouch, those occasional, pesky leap-*seconds*. Wow, this could get complicated. The good news is that the Java API is rich with classes that can help you manipulate dates. Sometimes it feels a little *too* rich...

Moving backward and forward in time

Let's say your company's work schedule is Monday through Friday. You've been assigned the task of figuring out the last work day in each calendar month this year...

It seems that java.util.Date is actually... out of date

Earlier we used java.util.Date to find today's date, so it seems logical that this class would be a good place to start looking for some handy date manipulation capabilities, but when you check out the API you'll find that most of Date's methods have been deprecated!

The Date class is still great for getting a "time stamp"—an object that represents the current date and time, so use it when you want to say, "give me NOW".

The good news is that the API recommends **java.util.Calendar** instead, so let's take a look:

For a time-stamp of "now", use Date. But for everything else, use Calendar.

Use java.util.Calendar for your date *manipulation*

The designers of the Calendar API wanted to think globally, literally. The basic idea is that when you want to work with dates, you ask for a Calendar (through a static method of the Calendar class that you'll see on the next page), and the JVM hands you back an instance of a concrete subclass of Calendar. (Calendar is actually an abstract class, so you're always working with a concrete subclass.)

More interesting, though, is that the *kind* of calendar you get back will be *appropriate for your locale.* Much of the world uses the Gregorian calendar, but if you're in an area that doesn't use a Gregorian calendar you can get Java libraries to handle other calendars such as Buddhist, or Islamic or Japanese.

The standard Java API ships with **java.util.GregorianCalendar**, so that's what we'll be using here. For the most part, though, you don't even have to think about the kind of Calendar subclass you're using, and instead focus only on the methods of the Calendar class.

Getting an object that extends Calendar

How in the world do you get an "instance" of an abstract class?
Well you don't of course, this won't work:

This WON'T work:

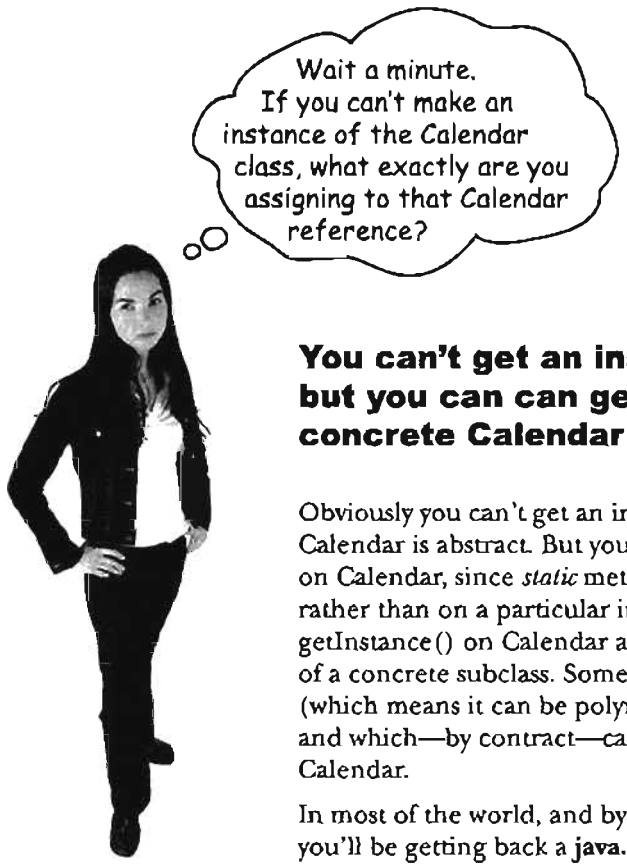
```
Calendar cal = new Calendar();
```

The compiler won't allow this!

Instead, use the static "getInstance()" method:

```
Calendar cal = Calendar.getInstance();
```

This syntax should look familiar at this point – we're invoking a static method.



You can't get an instance of Calendar, but you can can get an instance of a concrete Calendar subclass.

Obviously you can't get an instance of Calendar, because Calendar is abstract. But you're still free to call static methods on Calendar, since *static* methods are called on the *class*, rather than on a particular instance. So you call the static getInstance() on Calendar and it gives you back... an instance of a concrete subclass. Something that extends Calendar (which means it can be polymorphically assigned to Calendar) and which—by contract—can respond to the methods of class Calendar.

In most of the world, and by default for most versions of Java, you'll be getting back a **java.util.GregorianCalendar** instance.

Working with Calendar objects

There are several key concepts you'll need to understand in order to work with Calendar objects:

- **Fields hold *state*** - A Calendar object has many fields that are used to represent aspects of its ultimate state, its date and time. For instance, you can get and set a Calendar's *year* or *month*.
- **Dates and Times can be *incremented*** - The Calendar class has methods that allow you to add and subtract values from various fields, for example "add one to the month", or "subtract three years".
- **Dates and Times can be represented in *milliseconds*** - The Calendar class lets you convert your dates into and out of a millisecond representation. (Specifically, the number of milliseconds that have occured since January 1st, 1970.) This allows you to perform precise calculations such as "elapsed time between two times" or "add 63 hours and 23 minutes and 12 seconds to this time".

An example of working with a Calendar object:

```
Calendar c = Calendar.getInstance();
c.set(2004,0,7,15,40);
long day1 = c.getTimeInMillis();
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);
System.out.println("new hour " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);
System.out.println("add 35 days " + c.getTime());
c.roll(c.DATE, 35);
System.out.println("roll 35 days " + c.getTime());
c.set(c.DATE, 1);
System.out.println("set to 1 " + c.getTime());
```

Set time to Jan. 7, 2004 at 15:40. (Notice the month is zero-based.)

Convert this to a big ol' amount of milliseconds.

Add an hour's worth of millis, then update the time. (Notice the "+=", it's like day1 = day1 + ...).

Add 35 days to the date, which should move us into February.

"Roll" 35 days onto this date. This "rolls" the date ahead 35 days, but DOES NOT change the month!

We're not incrementing here, just doing a "set" of the date.

```
File Edit Window Help Time-Flies
new hour 16
add 35 days Wed Feb 11 16:40:41 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004
```

This output confirms how millis, add, roll, and set work.

Highlights of the Calendar API

We just worked through using a few of the fields and methods in the Calendar class. This is a big API, so we're showing only a few of the most common fields and methods that you'll use. Once you get a few of these it should be pretty easy to bend the rest of the this API to your will.

Key Calendar Methods

add(int field, int amount)
Adds or subtracts time from the calendar's field.

get(int field)
Returns the value of the given calendar field.

getInstance()
Returns a Calendar, you can specify a locale.

getTimeInMillis()
Returns this Calendar's time in millis, as a long.

roll(int field, boolean up)
Adds or subtracts time without changing larger fields.

set(int field, int value)
Sets the value of a given Calendar field.

set(year, month, day, hour, minute) (all ints)
A common variety of set to set a complete time.

setTimeInMillis(long millis)
Sets a Calendar's time based on a long milli-time.

// more...

Key Calendar Fields

DATE / DAY_OF_MONTH
Get / set the day of month

HOUR / HOUR_OF_DAY
Get / set the 12 hour or 24 hour value.

MILLISECOND
Get / set the milliseconds.

MINUTE
Get / set the minute.

MONTH
Get / set the month.

YEAR
Get / set the year.

ZONE_OFFSET
Get / set raw offset of GMT in millis.

// more...

Even more Statics!... static imports

New to Java 5.0... a real mixed blessing. Some people love this idea, some people hate it. Static imports exist only to save you some typing. If you hate to type, you might just like this feature. The downside to static imports is that - if you're not careful - using them can make your code a lot harder to read.

The basic idea is that whenever you're using a static class, a static variable, or an enum (more on those later), you can import them, and save yourself some typing.

Use Carefully:

static imports can make your code confusing to read

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

The syntax to use when declaring static imports.

Same code, with static imports:

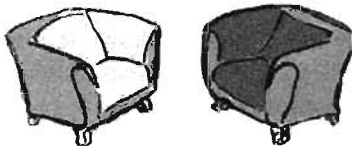
```
import static java.lang.System.out;

import static java.lang.Math.*;

class WithStatic {

  public static void main(String [] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

Static imports in action.

- Caveats & Gotchas

- If you're only going to use a static member a few times, we think you should avoid static imports, to help keep the code more readable.
- If you're going to use a static member a lot, (like doing lots of Math calculations), then it's probably OK to use the static import.
- Notice that you can use wildcards (.*), in your static import declaration.
- A big issue with static imports is that it's not too hard to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use?

Fireside Chats



Tonight's Talk: **An instance variable takes cheap shots at a static variable**

Instance Variable

I don't even know why we're doing this. Everyone knows static variables are just used for constants. And how many of those are there? I think the whole API must have, what, four? And it's not like anybody ever uses them.

Static Variable

You really should check your facts. When was the last time you looked at the API? It's frickin' loaded with statics! It even has entire classes dedicated to holding constant values. There's a class called SwingConstants, for example, that's just full of them.

Full of it. Yeah, you can say that again. OK, so there are a few in the Swing library, but everybody knows Swing is just a special case.

It might be a special case, but it's a really important one! And what about the Color class? What a pain if you had to remember the RGB values to make the standard colors? But the color class already has constants defined for blue, purple, white, red, etc. Very handy.

Ok, but besides a few GUI things, give me an example of just one static variable that anyone would actually use. In the real world.

How's System.out for starters? The out in System.out is a static variable of the System class. You personally don't make a new instance of the System, you just ask the System class for its out variable.

Well, that's another special case. And nobody uses that except for debugging anyway.

Oh, like debugging isn't important?

And here's something that probably never crossed your narrow mind—let's face it, static variables are more efficient. One per class instead of one per instance. The memory savings might be huge!

Instance Variable

Um, aren't you forgetting something?

Static variables are about as un-OO as it gets!! Gee why not just go take a giant backwards step and do some procedural programming while we're at it.

You're like a global variable, and any programmer worth his PDA knows that's usually a Bad Thing.

Yeah you live in a class, but they don't call it *Class*-Oriented programming. That's just stupid. You're a relic. Something to help the old-timers make the leap to java.

Well, OK, every once in a while sure, it makes sense to use a static, but let me tell you, abuse of static variables (and methods) is the mark of an immature OO programmer. A designer should be thinking about *object* state, not *class* state.

Static methods are the worst things of all, because it usually means the programmer is thinking procedurally instead of about objects doing things based on their unique object state.

Riiiiight. Whatever you need to tell yourself...

Static Variable

What?

What do you mean *un*-OO?

I am NOT a global variable. There's no such thing. I live in a class! That's pretty OO you know, a CLASS. I'm not just sitting out there in space somewhere; I'm a natural part of the state of an object; the only difference is that I'm shared by all instances of a class. Very efficient.

Alright just stop right there. THAT is definitely not true. Some static variables are absolutely crucial to a system. And even the ones that aren't crucial sure are handy.

Why do you say that? And what's wrong with static methods?

Sure, I know that objects should be the focus of an OO design, but just because there are some clueless programmers out there... don't throw the baby out with the bytecode. There's a time and place for statics, and when you need one, nothing else beats it.

BE the compiler

The Java file on this page represents a complete program. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it, and if it does compile, what would be its output?

```
class StaticSuper{

  static {
    System.out.println("super static block");
  }

  StaticSuper{
    System.out.println(
      "super constructor");
  }
}

public class StaticTests extends StaticSuper {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("static block " + rand);
   }

   StaticTests() {
     System.out.println("constructor");
   }

  public static void main(String [] args) {
    System.out.println("in main");
    StaticTests st = new StaticTests();
  }
}
```

If it compiles, which of these is the output?

Possible Output

```
File  Edit  Window  Help  Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

Possible Output

```
File  Edit  Window  Help  Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

This chapter explored the wonderful, static, world of Java. Your job is to decide whether each of the following statements is true or false.

True or False

1. To use the Math class, the first step is to make an instance of it.
2. You can mark a constructor with the `static` keyword.
3. Static methods don't have access to instance variable state of the 'this' object.
4. It is good practice to call a static method using a reference variable.
5. Static variables could be used to count the instances of a class.
6. Constructors are called before static variables are initialized.
7. MAX_SIZE would be a good name for a static final variable.
8. A static initializer block runs before a class's constructor runs.
9. If a class is marked final, all of its methods must be marked final.
10. A final method can only be overridden if its class is extended.
11. There is no wrapper class for boolean primitives.
12. A wrapper is used when you want to treat a primitive like an object.
13. The parseXxx methods always return a String.
14. Formatting classes (which are decoupled from I/O), are in the java.format package.

Lunar Code Magnets

This one might actually be useful! In addition to what you've learned in the last few pages about manipulating dates, you'll need a little more information... First, full moons happen every 29.52 days or so. Second, there was a full moon on Jan. 7th, 2004. Your job is to reconstruct the code snippets to make a working Java program that produces the output listed below (plus more full moon dates). (You might not need all of the magnets, and add all the curly braces you need.) Oh, by the way, your output will be different if you don't live in the mountain time zone.

```
File Edit Window Help Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

```
long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("full moon on %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {
public static void main(String [] args) {
day1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
import java.io.*;
("full moon on %t", c));
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(day1);
(String.format
Calendar c = Calendar.getInstance();
```

Exercise Solutions

BE the compiler

```
StaticSuper( ) {
  System.out.println(
    "super constructor");
}
```

StaticSuper is a constructor, and must have ( ) in its signature. Notice that as the output below demonstrates, the static blocks for both classes run before either of the constructors run.

Possible Output

```
File Edit Window Help Cling
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

True or False

1. To use the Math class, the first step is to make an instance of it. **False**
2. You can mark a constructor with the keyword 'static'. **False**
3. Static methods don't have access to an object's instance variables. **True**
4. It is good practice to call a static method using a reference variable. **False**
5. Static variables could be used to count the instances of a class. **True**
6. Constructors are called before static variables are initialized. **False**
7. MAX_SIZE would be a good name for a static final variable. **True**
8. A static initializer block runs before a class's constructor runs. **True**
9. If a class is marked final, all of its methods must be marked final. **False**
10. A final method can only be overridden if its class is extended. **False**
11. There is no wrapper class for boolean primitives. **False**
12. A wrapper is used when you want to treat a primitive like an object. **True**
13. The parseXxx methods always return a String. **False**
14. Formatting classes (which are decoupled from I/O), are in the java.format package. **False**

code magnets solution

Exercise Solutions

```
import java.util.*;
import static java.lang.System.out;
class FullMoons {
  static int DAY_IM = 1000 * 60 * 60 * 24;
  public static void main(String [] args) {
    Calendar c = Calendar.getInstance();
    c.set(2004,0,7,15,40);
    long day1 = c.getTimeInMillis();
    for (int x = 0; x < 60; x++) {
       day1 += (DAY_IM * 29.52)
       c.setTimeInMillis(day1);
       out.println(String.format("full moon on %tc", c));
    }
  }
}
```

Notes on the Lunar Code Magnet:

You might discover that a few of the dates produced by this program are off by a day. This astronomical stuff is a little tricky, and if we made it perfect, it would be too complex to make an exercise here.

Hint: one problem you might try to solve is based on differences in time zones. Can you spot the issue?

```
File Edit Window Help How
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

Risky Behavior

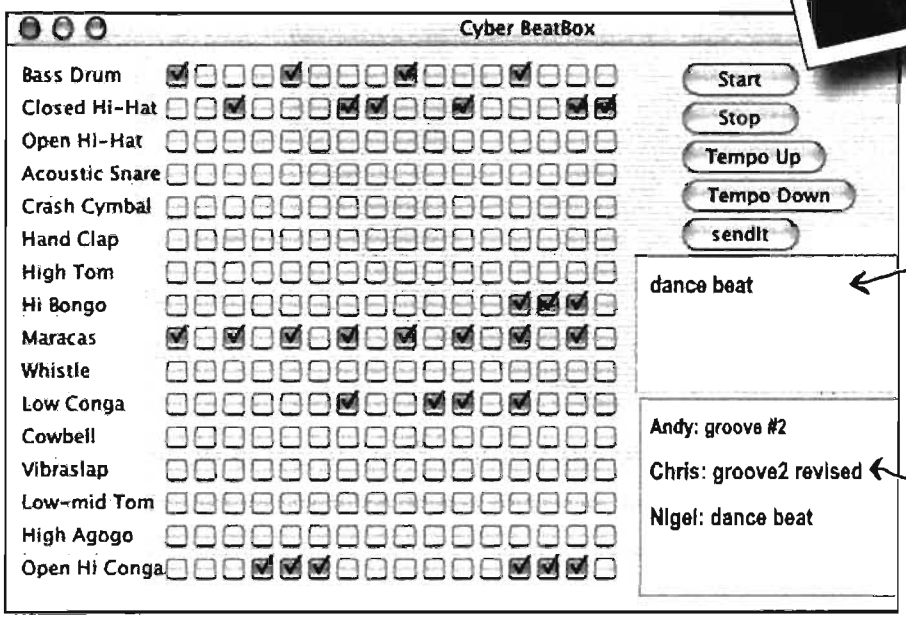
Stuff happens. The file isn't there. The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very* wrong. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? And where do you put the code to *handle* the ***exceptional*** situation? So far in this book, we haven't *really* taken any risks. We've certainly had things go wrong at runtime, but the problems were mostly flaws in our own code. Bugs. And those we should fix at development time. No, the problem-handling code we're talking about here is for code that you *can't* guaranatee will work at runtime. Code that expects the file to be in the right directory, the server to be running, or the Thread to stay asleep. And we have to do this *now*. Because in *this* chapter, we're going to build something that uses the risky JavaSound API. We're going to build a MIDI Music Player.

Let's make a Music Machine

Over the next three chapters, we'll build a few different sound applications, including a BeatBox Drum Machine. In fact, before the book is done, we'll have a multi-player version so you can send your drum loops to another player, kind of like a chat room. You're going to write the whole thing, although you can choose to use Ready-bake code for the GUI parts. OK, so not every IT department is looking for a new BeatBox server, but we're doing this to *learn* more about *Java*. Building a BeatBox is just a way to have fun *while* we're learning Java.

The finished BeatBox looks something like this:

You make a beatbox loop (a 16-beat drum pattern) by putting checkmarks in the boxes.



your message, that gets sent to the other players, along with your current beat pattern, when you hit "SendIt"

incoming messages from other players. Click one to load the pattern that goes with it, and then click 'Start' to play it

Put checkmarks in the boxes for each of the 16 'beats'. For example, on beat 1 (of 16) the Bass drum and the Maracas will play, on beat 2 nothing, and on beat 3 the Maracas and Closed Hi-Hat... you get the idea. When you hit 'Start', it plays your pattern in a loop until you hit 'Stop'. At any time, you can "capture" one of your own patterns by sending it to the BeatBox server (which means any other players can listen to it). You can also load any of the incoming patterns by clicking on the message that goes with it.

We'll start with the basics

Obviously we've got a few things to learn before the whole program is finished, including how to build a Swing GUI, how to *connect* to another machine via networking, and a little I/O so we can *send* something to the other machine.

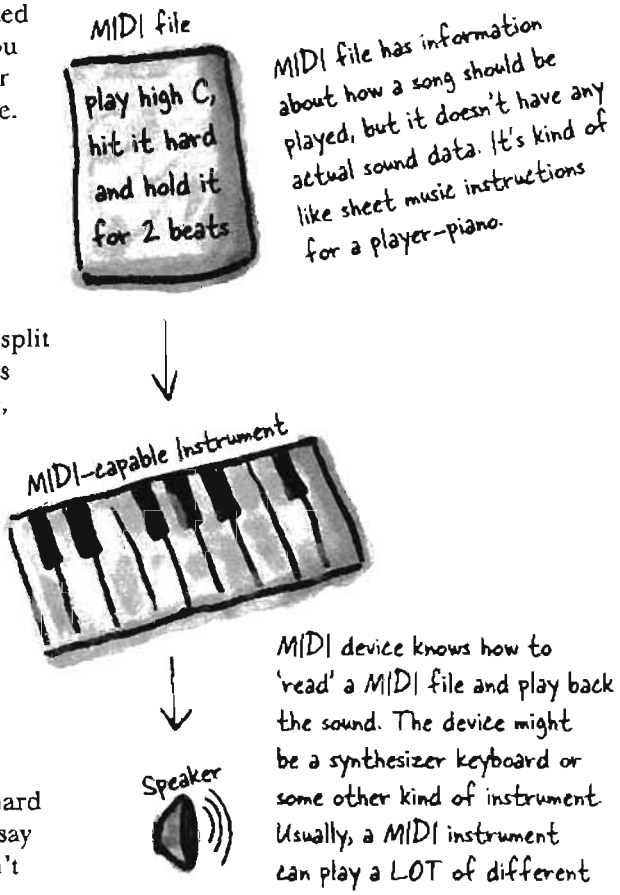
Oh yeah, and the JavaSound API. *That's* where we'll start in this chapter. For now, you can forget the GUI, forget the networking and the I/O, and focus only on getting some MIDI-generated sound to come out of your computer. And don't worry if you don't know a thing about MIDI, or a thing about reading or making music. Everything you need to learn is covered here. You can almost smell the record deal.

The JavaSound API

JavaSound is a collection of classes and interfaces added to Java starting with version 1.3. These aren't special add-ons; they're part of the standard J2SE class library. JavaSound is split into two parts: MIDI and Sampled. We use only MIDI in this book. MIDI stands for Musical Instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate. But for our BeatBox app, you can think of MIDI as *a kind of sheet music* that you feed into some device you can think of like a high-tech 'player piano'. In other words, MIDI data doesn't actually include any *sound*, but it does include the *instructions* that a MIDI-reading instrument can play back. Or for another analogy, you can think of a MIDI file like an HTML document, and the instrument that renders the MIDI file (i.e. *plays* it) is like the Web browser.

MIDI data says *what* to do (play middle C, and here's how hard to hit it, and here's how long to hold it, etc.) but it doesn't say anything at all about the actual *sound* you hear. MIDI doesn't know how to make a flute, piano, or Jimmy Hendrix guitar sound. For the actual sound, we need an instrument (a MIDI device) that can read and play a MIDI file. But the device is usually more like an *entire band or orchestra* of instruments. And that instrument might be a physical device, like the electronic keyboard synthesizers the rock musicians play, or it could even be an instrument built entirely in software, living in your computer.

For our BeatBox, we use only the built-in, software-only instrument that you get with Java. It's called a *synthesizer* (some folks refer to it as a *software synth*) because it *creates* sound. Sound that you *hear*.



First we need a Sequencer

Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback device. Like a CD-player on your stereo, but with a few added features. The Sequencer class is in the javax.sound.midi package (part of the standard Java library as of version 1.3). So let's start by making sure we can make (or get) a Sequencer object.

```
import javax.sound.midi.*;

public class MusicTest1 {

    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();

        System.out.println("We got a sequencer");
    } // close play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // close main
} // close class
```

import the javax.sound.midi package

We need a Sequencer object. It's the main part of the MIDI device/instrument we're using. It's the thing that, well, sequences all the MIDI information into a 'song'. But we don't make a brand new one ourselves -- we have to ask the MidiSystem to give us one.

Something's wrong!

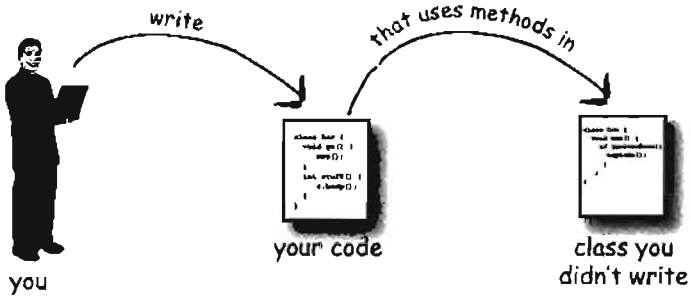
This code won't compile! The compiler says there's an 'unreported exception' that must be caught or declared.

File Edit Window Help SayWhat?

```
% javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown
    Sequencer sequencer = MidiSystem.getSequencer();
                                     ^
1 errors
```

What happens when a method you want to call (probably in a class you didn't write) is risky?

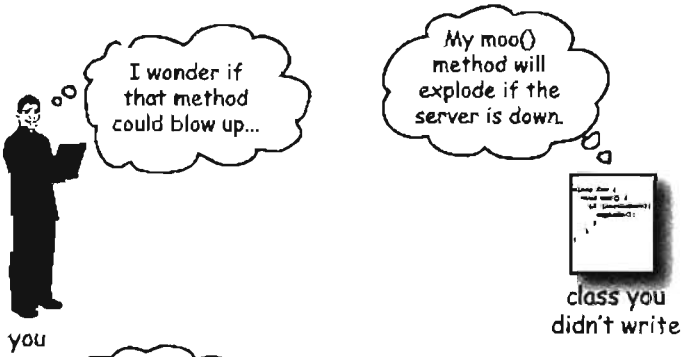
Let's say you want to call a method in a class that you didn't write.



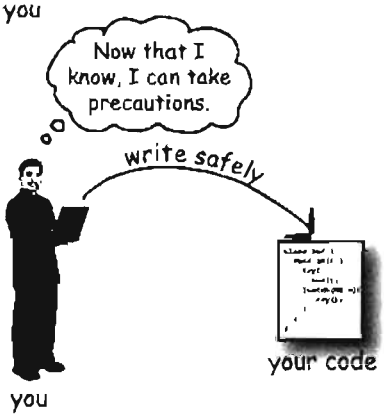
That method does something risky, something that might not work at runtime.


```
void moo() {
   if (serverDown) {
    explode();
   }
}
```

You need to *know* that the method you're calling is risky.



You then write code that can handle the failure if it *does* happen. You need to be prepared, just in case.

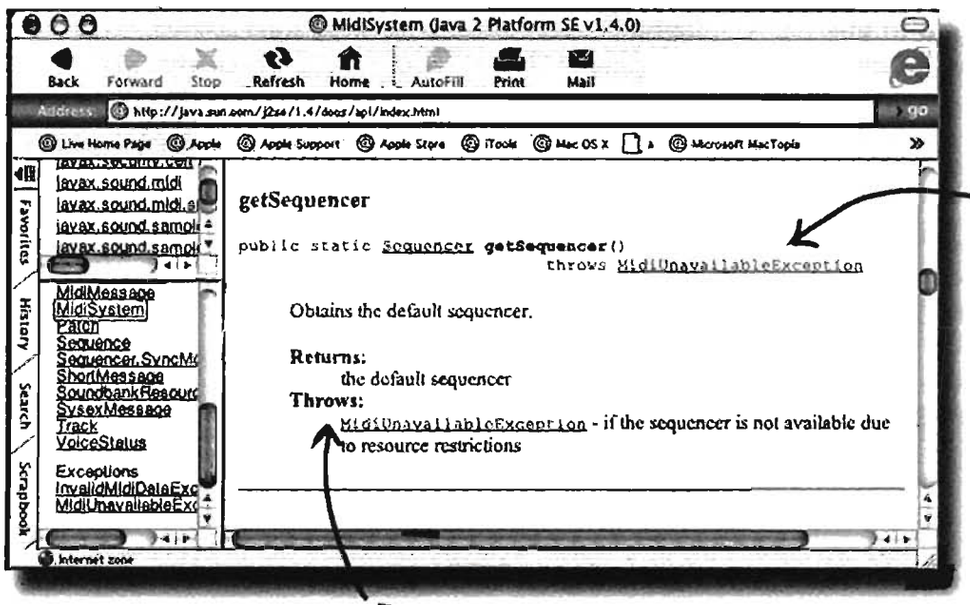


Methods in Java use *exceptions* to tell the calling code, "Something Bad Happened. I failed."

Java's exception-handling mechanism is a clean, well-lighted way to handle "exceptional situations" that pop up at runtime; it lets you put all your error-handling code in one easy-to-read place. It's based on you *knowing* that the method you're calling is risky (i.e. that the method *might* generate an exception), so that you can write code to deal with that possibility. If you *know* you might get an exception when you call a particular method, you can be *prepared* for—possibly even *recover* from—the problem that caused the exception.

So, how *do* you know if a method throws an exception? You find a `throws` clause in the risky method's declaration.

The `getSequencer()` method takes a risk. It can fail at runtime. So it must 'declare' the risk *you* take when you call it.



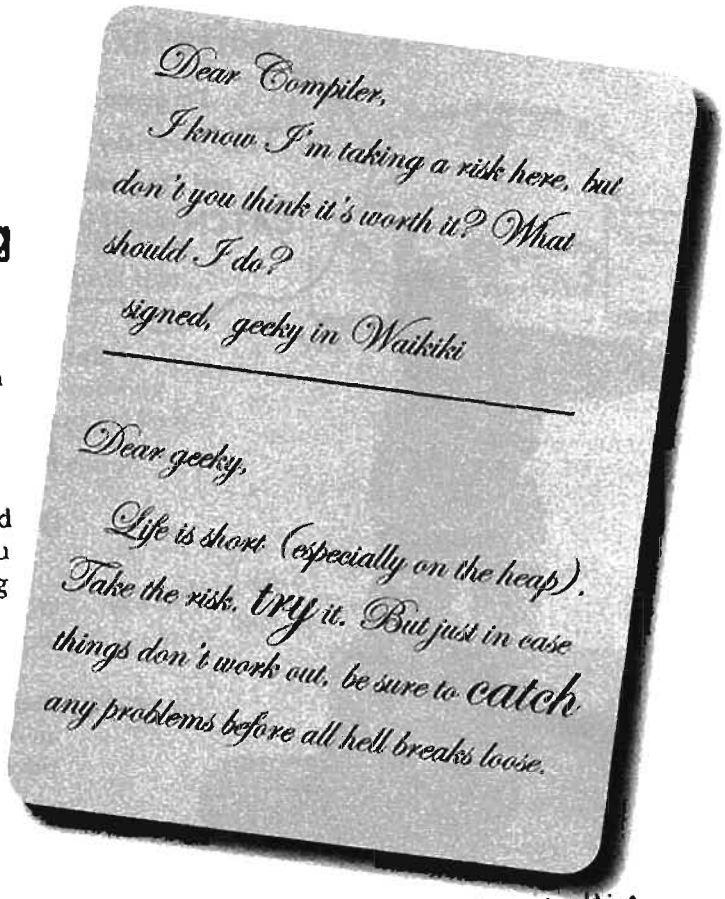
The API docs tell you that getSequencer() can throw an exception: MidiUnavailableException. A method has to declare the exceptions it might throw.

This part tells you WHEN you might get that exception — in this case, because of resource restrictions (which could just means the sequencer is already being used).

The compiler needs to know that YOU know you're calling a risky method.

If you wrap the risky code in something called a **try/catch**, the compiler will relax.

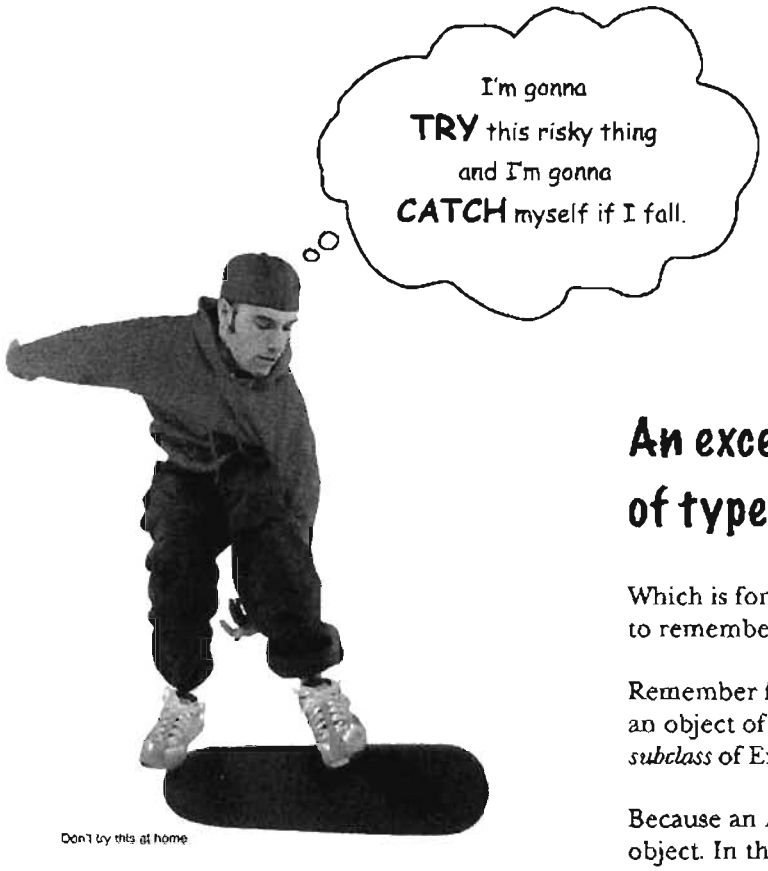
A try/catch block tells the compiler that you *know* an exceptional thing could happen in the method you're calling, and that you're prepared to handle it. That compiler doesn't care *how* you handle it; it cares only that you say you're taking care of it.




```
import javax.sound.midi.*;

public class MusicTest1 {
    public void play() {

        try {
            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Successfully got a sequencer");
        } catch(MidiUnavailableException ex) {
              System.out.println("Bummer");
        }
    } // close play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // close main
} // close class
```

put the risky thing in a 'try' block.

make a 'catch' block for what to do if the exceptional situation happens — in other words, a MidiUnavailableException is thrown by the call to getSequencer().



An exception is an object... of type Exception.

Which is fortunate, because it would be much harder to remember if exceptions were of type Broccoli.

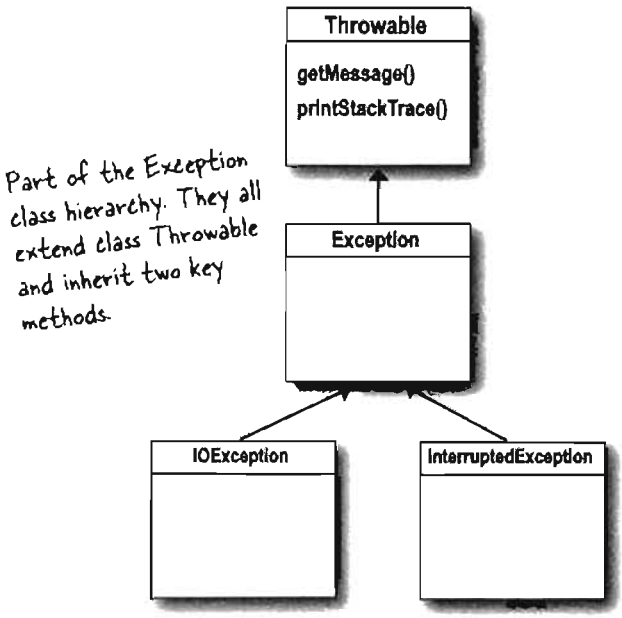
Remember from your polymorphism chapters that an object of type Exception *can* be an instance of any *subclass* of Exception.

Because an *Exception* is an object, what you *catch* is an object. In the following code, the `catch` argument is declared as type Exception, and the parameter reference variable is *ex*.

```
try {
    // do risky thing
} catch(Exception ex) {
    // try to recover
}
```

it's just like declaring a method argument

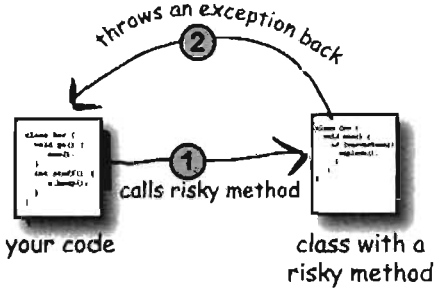
This code only runs if an Exception is thrown.



Part of the Exception class hierarchy. They all extend class Throwable and inherit two key methods.

What you write in a catch block depends on the exception that was thrown. For example, if a server is down you might use the catch block to try another server. If the file isn't there, you might ask the user for help finding it.

If it's *your* code that catches the exception, then whose code throws it?



You'll spend much more of your Java coding time *handling* exceptions than you'll spend *creating* and *throwing* them yourself. For now, just know that when your code *calls* a risky method—a method that declares an exception—it's the risky method that *throws* the exception back to *you*, the caller.

In reality, it might be you who wrote both classes. It really doesn't matter who writes the code... what matters is knowing which method *throws* the exception and which method *catches* it.

When somebody writes code that could throw an exception, they must *declare* the exception.

Risky, exception-throwing code:

this method MUST tell the world (by declaring) that it throws a BadException

```
public void takeRisk() throws BadException {
   if (abandonAllHope) {
     throw new BadException();
   }
}
```

create a new Exception object and throw it

One method will catch what another method throws. An exception is always thrown back to the caller.

The method that throws has to declare that it might throw the exception.

Your code that *calls* the risky method:

```
public void crossFingers() {
   try {
     anObject.takeRisk();
   } catch (BadException ex) {
     System.out.println("Aaargh!");
     ex.printStackTrace();
}
```

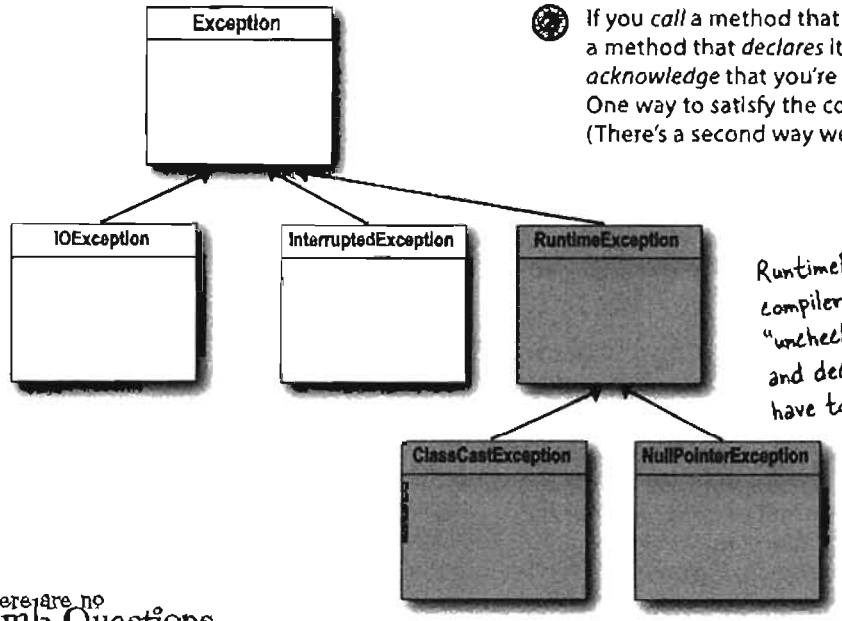
If you can't recover from the exception, at LEAST get a stack trace using the printStackTrace() method that all exceptions inherit.

The compiler checks for everything except RuntimeExceptions.

The compiler guarantees:

- If you *throw* an exception in your code you *must* declare it using the *throws* keyword in your method declaration.
- If you *call* a method that throws an exception (in other words, a method that *declares* it throws an exception), you must *acknowledge* that you're aware of the exception possibility. One way to satisfy the compiler is to wrap the call in a try/catch. (There's a second way we'll look at a little later in this chapter.)

Exceptions that are NOT subclasses of RuntimeException are checked for by the compiler. They're called "checked exceptions"



RuntimeExceptions are NOT checked by the compiler. They're known as (big surprise here) "unchecked exceptions". You can throw, catch, and declare RuntimeExceptions, but you don't have to, and the compiler won't check.

there are no Dumb Questions

Q: Wait just a minute! How come this is the FIRST time we've had to try/catch an Exception? What about the exceptions I've already gotten like NullPointerException and the exception for DivideByZero. I even got a NumberFormatException from the Integer.parseInt() method. How come we didn't have to catch those?

A: The compiler cares about all subclasses of Exception, *unless* they are a special type, RuntimeException. Any exception class that extends RuntimeException gets a free pass. RuntimeExceptions can be thrown anywhere, with or without throws declarations or try/catch blocks. The compiler doesn't bother checking whether a method declares that it throws a RuntimeException, or whether the caller acknowledges that they might get that exception at runtime.

Q: I'll bite. WHY doesn't the compiler care about those runtime exceptions? Aren't they just as likely to bring the whole show to a stop?

A: Most RuntimeExceptions come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent. You *cannot* guarantee the file is there. You *cannot* guarantee the server is up. But you *can* make sure your code doesn't index off the end of an array (that's what the .length attribute is for).

You WANT RuntimeExceptions to happen at development and testing time. You don't want to code in a try/catch, for example, and have the overhead that goes with it, to catch something that shouldn't happen in the first place.

A try/catch is for handling exceptional situations, not flaws in your code. *Use your catch blocks to try to recover from* situations you can't guarantee will succeed. Or at the very least, print out a message to the user and a stack trace, so somebody can figure out what happened.

BULLET POINTS

- A method can throw an exception when something fails at runtime.
- An exception is always an object of type Exception. (Which, as you remember from the polymorphism chapters means the object is from a class that has Exception somewhere up its inheritance tree.)
- The compiler does NOT pay attention to exceptions that are of type **RuntimeException**. A RuntimeException does not have to be declared or wrapped in a try/catch (although you're free to do either or both of those things)
- All Exceptions the compiler cares about are called 'checked exceptions' which really means *compiler*-checked exceptions. Only RuntimeExceptions are excluded from compiler checking. All other exceptions must be acknowledged in your code, according to the rules.
- A method throws an exception with the keyword `throw`, followed by a new exception object:

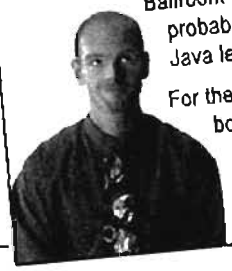
```
throw new NoCaffeineException();
```

- Methods that *might* throw a checked exception ***must*** announce it with a `throws Exception` declaration.
- If your code calls a checked-exception-throwing method, it must reassure the compiler that precautions have been taken.
- If you're prepared to handle the exception, wrap the call in a try/catch, and put your exception handling/recovery code in the catch block.
- If you're not prepared to handle the exception, you can still make the compiler happy by officially 'ducking' the exception. We'll talk about ducking a little later in this chapter.

Metacognitive tip

If you're trying to learn something new, make that the *last* thing you try to learn before going to sleep. So, once you put this book down (assuming you can tear yourself away from it) don't read anything else more challenging than the back of a Cheerios™ box. Your brain needs time to process what you've read and learned. That could take a few *hours*. If you try to shove something new in right on top of your Java, some of the Java might not 'stick.'

Of course, this doesn't rule out learning a physical skill. Working on your latest Ballroom KickBoxing routine probably won't affect your Java learning.



For the best results, read this book (or at least look at the pictures) right before going to sleep.

Sharpen your pencil

Which of these do you think might throw an exception that the compiler would care about? We're only looking for the things that you can't control in your code. We did the first one.

(Because it was the easiest.)

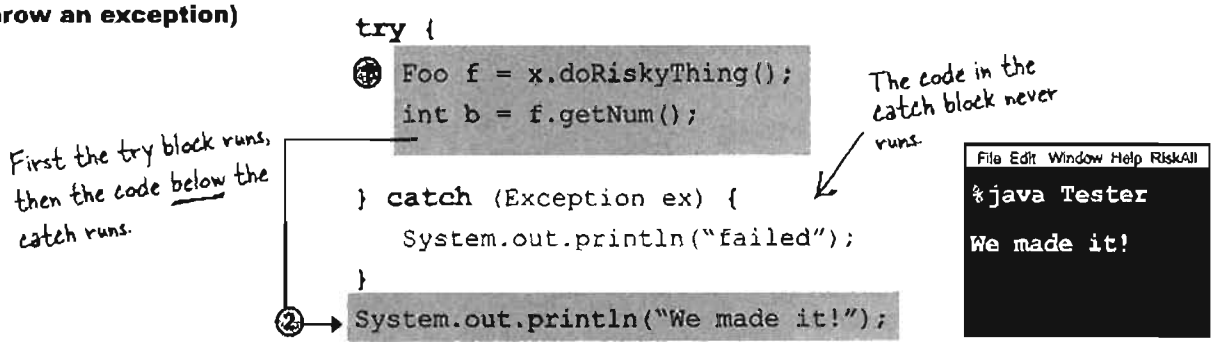
Things you want to do	What might go wrong
✓ connect to a remote server	the server is down
__ access an array beyond its length	
__ display a window on the screen	
__ retrieve data from a database	
__ see if a text file is where you *think* it is	
__ create a new file	
__ read a character from the command-line	

Flow control in try/catch blocks

When you call a risky method, one of two things can happen. The risky method either succeeds, and the try block completes, or the risky method throws an exception back to your calling method.

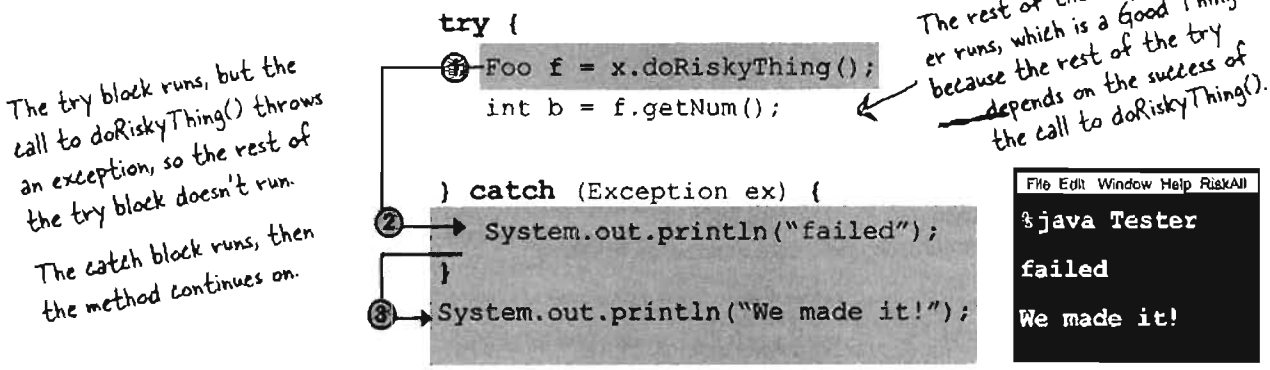
If the try succeeds

(doRiskyThing() does *not* throw an exception)



If the try fails

(because doRiskyThing() *does* throw an exception)



Finally: for the things you want to do *no matter what.*

If you try to cook something, you start by turning on the oven.

If the thing you try is a complete **failure**,
you have to turn off the oven.

If the thing you try **succeeds**,
you have to turn off the oven.

You have to turn off the oven no matter what!

A finally block is where you put code that must run *regardless* of an exception.

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException ex) {
    ex.printStackTrace();
} finally {
    turnOvenOff();
}
```

Without finally, you have to put the turnOvenOff() in *both* the try and the catch because *you have to turn off the oven no matter what.* A finally block lets you put all your important cleanup code in *one* place instead of duplicating it like this:

```
try {
  turnOvenOn();
  x.bake();
  turnOvenOff();
} catch (BakingException ex) {
    ex.printStackTrace();
    turnOvenOff();
}
```

If the try block fails (an exception), flow control immediately moves to the catch block. When the catch block completes, the finally block runs. When the finally block completes, the rest of the method continues on.

If the try block succeeds (*no* exception), flow control skips over the catch block and moves to the finally block. When the finally block completes, the rest of the method continues on.

If the try or catch block has a return statement, finally will still run! Flow jumps to the finally, then back to the return.

Sharpen your pencil

Flow Control

Look at the code to the left. What do you think the output of this program would be? What do you think it would be if the third line of the program were changed to: `String test = "yes";` ? Assume ScaryException extends Exception.

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = "no";
    try {
       System.out.println("start try");
       doRisky(test);
       System.out.println("end try");
    } catch ( ScaryException se) {
        System.out.println("scary exception");
    } finally {
        System.out.println("finally");
    }
    System.out.println("end of main");
  }

  static void doRisky(String test) throws ScaryException {
    System.out.println("start risky");
    if ("yes".equals(test)) {
      throw new ScaryException();
    }
    System.out.println("end risky");
    return;
  }
}
```

Output when `test = "no"`

Output when `test = "yes"`

When test = "no": start try - start risky - end risky - end try - finally - end of main
When test = "yes": start try - start risky - scary exception - finally - end of main

Did we mention that a method can throw more than one exception?

A method can throw multiple exceptions if it darn well needs to. But a method's declaration must declare *all* the checked exceptions it can throw (although if two or more exceptions have a common superclass, the method can declare just the superclass.)

Catching multiple exceptions

The compiler will make sure that you've handled *all* the checked exceptions thrown by the method you're calling. Stack the *catch* blocks under the *try*, one after the other. Sometimes the order in which you stack the catch blocks matters, but we'll get to that a little later.

```
public class Laundry {

    public void doLaundry() throws PantsException, LingerieException {
       // code that could throw either exception
    }
}
```

This method declares two, count 'em, TWO exceptions.

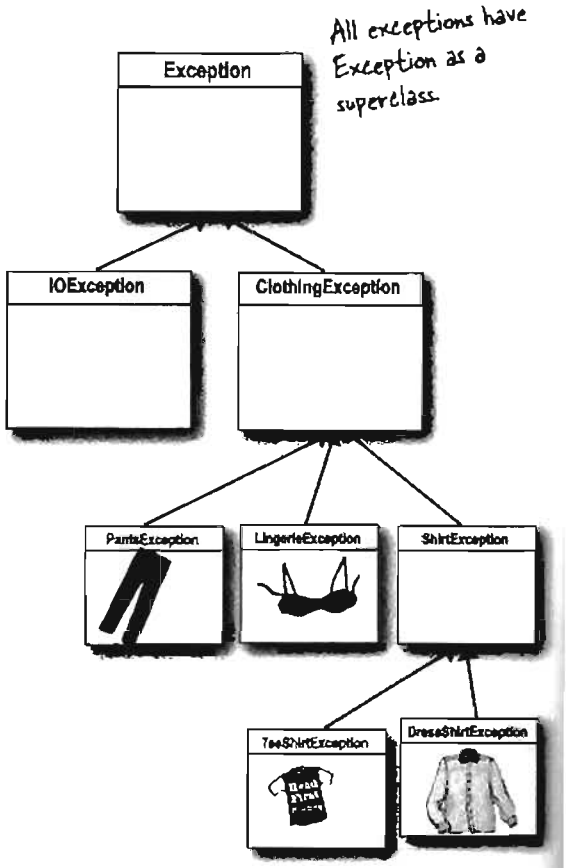
```
public class Foo {

    public void go() {
       Laundry laundry = new Laundry();
       try {
            laundry.doLaundry();
        } catch(PantsException pex) {
            // recovery code

        } catch(LingerieException lex) {
            // recovery code
        }
    }
}
```

if doLaundry() throws a PantsException, it lands in the PantsException catch block.

if doLaundry() throws a LingerieException, it lands in the LingerieException catch block.

Exceptions are polymorphic

Exceptions are objects, remember. There's nothing all that special about one, except that it is *a thing that can be thrown*. So like all good objects, Exceptions can be referred to polymorphically. A LingerieException *object*, for example, could be assigned to a ClothingException *reference*. A PantsException could be assigned to an Exception reference. You get the idea. The benefit for exceptions is that a method doesn't have to explicitly declare every possible exception it might throw; it can declare a superclass of the exceptions. Same thing with catch blocks—you don't have to write a catch for each possible exception as long as the catch (or catches) you have can handle any exception thrown.



① You can DECLARE exceptions using a supertype of the exceptions you throw.

```
public void doLaundry() throws ClothingException {
```

Declaring a ClothingException lets you throw any subclass of ClothingException. That means doLaundry() can throw a PantsException, LingerieException, TeeShirtException, and DressShirtException without explicitly declaring them individually.

② You can CATCH exceptions using a supertype of the exception thrown.

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
    // recovery code
}
```

can catch any ClothingException subclass

```
try {
    laundry.doLaundry();
} catch(ShirtException sex) {
    // recovery code
}
```

can catch only TeeShirtException and DressShirtException

Just because you CAN catch everything with one big super polymorphic catch, doesn't always mean you SHOULD.

You *could* write your exception-handling code so that you specify only *one* catch block, using the supertype Exception in the catch clause, so that you'll be able to catch *any* exception that might be thrown.

```
try {
   laundry.doLaundry();
} catch(Exception ex) {
     // recovery code...
}
```

Recovery from WHAT? This catch block will catch ANY and all exceptions, so you won't automatically know what went wrong.

Write a different catch block for each exception that you need to handle uniquely.

For example, if your code deals with (or recovers from) a TeeShirtException differently than it handles a LingerieException, write a catch block for each. But if you treat all other types of ClothingException in the same way, then add a ClothingException catch to handle the rest.

```
try {
   laundry.doLaundry();
} catch(TeeShirtException tex) {
    // recovery from TeeShirtException
} catch(LingerieException lex) {
    // recovery from LingerieException
} catch(ClothingException cex) {
    // recovery from all others
}
```

TeeShirtExceptions and LingerieExceptions need different recovery code, so you should use different catch blocks.

All other ClothingExceptions are caught here.

Multiple catch blocks must be ordered from smallest to biggest

TeeShirtExceptions are caught here, but no other exceptions will fit.

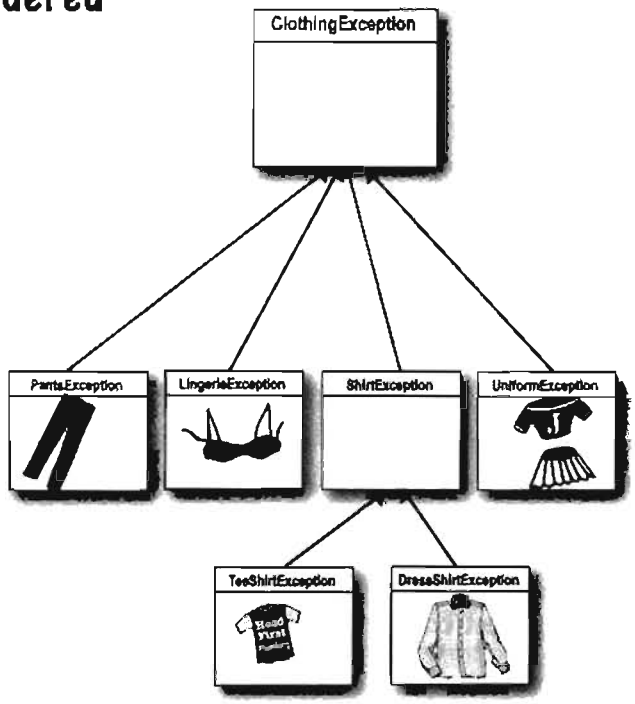
```
catch(TeeShirtException tex)
```

TeeShirtExceptions will never get here, but all other ShirtException subclasses are caught here.

```
catch(ShirtException sex)
```

All ClothingExceptions are caught here, although TeeShirtException and ShirtException will never get this far.

```
catch(ClothingException cex)
```



The higher up the inheritance tree, the bigger the catch 'basket'. As you move down the inheritance tree, toward more and more specialized Exception classes, the catch 'basket' is smaller. It's just plain old polymorphism.

A ShirtException catch is big enough to take a TeeShirtException or a DressShirtException (and any future subclass of anything that extends ShirtException). A ClothingException is even bigger (i.e. there are more things that can be referenced using a ClothingException type). It can take an exception of type ClothingException (duh), and any ClothingException subclasses: PantsException, UniformException, LingerieException, and ShirtException. The mother of all catch arguments is type **Exception**; it will catch *any* exception, including runtime (unchecked) exceptions, so you probably won't use it outside of testing.

You can't put bigger baskets above smaller baskets.

Well, you *can* but it won't compile. Catch blocks are not like overloaded methods where the best match is picked. With catch blocks, the JVM simply starts at the first one and works its way down until it finds a catch that's broad enough (in other words, high enough on the inheritance tree) to handle the exception. If your first catch block is `catch(Exception ex)`, the compiler knows there's no point in adding any others—they'll never be reached.

Don't do this!


```
try {
  laundry.doLaundry();
} catch(ClothingException cex) {
    // recovery from ClothingException
} catch(LingerieException lex) {
   // recovery from LingerieException
} catch(ShirtException sex) {
    // recovery from ShirtException
}
```

Size matters when you have multiple catch blocks. The one with the biggest basket has to be on the bottom. Otherwise, the ones with smaller baskets are useless.

Siblings can be in any order, because they can't catch one another's exceptions.

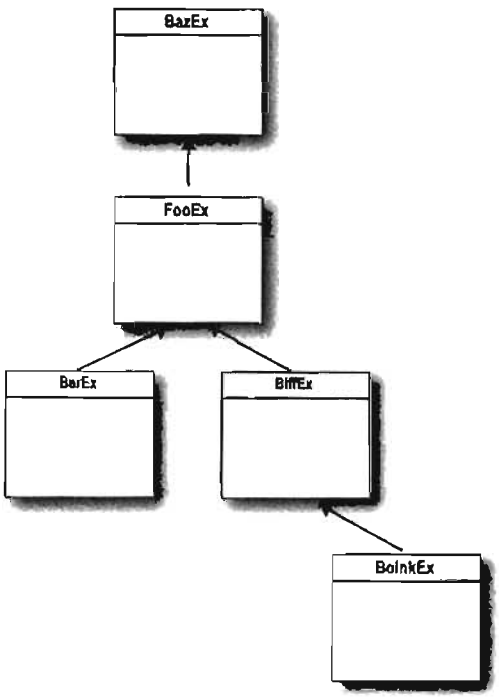
You could put ShirtException above LingerieException and nobody would mind. Because even though ShirtException is a bigger (broader) type because it can catch other classes (its own subclasses), ShirtException can't catch a LingerieException so there's no problem.

polymorphic puzzle

Assume the try/catch block here is legally coded. Your task is to draw two different class diagrams that can accurately reflect the Exception classes. In other words, what class inheritance structures would make the try/catch blocks in the sample code legal?

```
try {
  x.doRisky();
}  catch(AlphaEx a) {
    // recovery from AlphaEx
}  catch(BetaEx b) {
   // recovery from BetaEx
}  catch(GammaEx c) {
    // recovery from GammaEx
}  catch(DeltaEx d) {
    // recovery from DeltaEx
}
```

Your task is to create two different *legal* try / catch structures (similar to the one above left), to accurately represent the class diagram shown on the left. Assume ALL of these exceptions might be thrown by the method with the try block.



When you don't want to handle an exception...

just duck it

If you don't want to handle an exception, you can duck it by declaring it.

When you call a risky method, the compiler needs you to acknowledge it. Most of the time, that means wrapping the risky call in a try/catch. But you have another alternative, simply *duck* it and let the method that called *you* catch the exception.

It's easy—all you have to do is *declare* that *you* throw the exceptions. Even though, technically, *you* aren't the one doing the throwing, it doesn't matter. You're still the one letting the exception whiz right on by.

But if you duck an exception, then you don't have a try/catch, so what happens when the risky method (doLaundry()) *does* throw the exception?

When a method throws an exception, that method is popped off the stack immediately, and the exception is thrown to the next method down the stack—the *caller*. But if the *caller* is a *ducker*, then there's no catch for it so the *caller* pops off the stack immediately, and the exception is thrown to the next method and so on... where does it end? You'll see a little later.


```
public void foo() throws ReallyBadException {
    // call risky method without a try/catch
    laundry.doLaundry();
}
```

You don't REALLY throw it, but since you don't have a try/catch for the risky method you call, YOU are now the "risky method". Because now, whoever calls YOU has to deal with the exception.

Ducking (by declaring) only delays the inevitable

Sooner or later, *somebody* has to deal with it. But what if *main()* ducks the exception?

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

    public static void main (String[] args) throws ClothingException {
        Washer a = new Washer();
        a.foo();
    }
}
```

Both methods duck the exception (by declaring it) so there's nobody to handle it! This compiles just fine.

1 doLaundry() throws a ClothingException

main() calls foo()

foo() calls doLaundry()

doLaundry() is running and throws a ClothingException

2 foo() ducks the exception

doLaundry() pops off the stack immediately and the exception is thrown back to foo().

But foo() doesn't have a try/catch, so...

3 main() ducks the exception

foo() pops off the stack immediately and the exception is thrown back to... who? What? There's nobody left but the JVM, and it's thinking, "Don't expect ME to get you out of this."

4 The JVM shuts down

We're using the tee-shirt to represent a Clothing Exception. We know, we know... you would have preferred the blue jeans.

Handle or Declare. It's the law.

So now we've seen both ways to satisfy the compiler when you call a risky (exception-throwing) method.

HANDLE

Wrap the risky call in a try/catch

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
      // recovery code
}
```

This had better be a big enough catch to handle all exceptions that doLaundry() might throw. Or else the compiler will still complain that you're not catching all of the exceptions.

DECLARE (duck it)

Declare that YOUR method throws the same exceptions as the risky method you're calling.

```
void foo() throws ClothingException {
    laundry.doLaundry();
}
```

The doLaundry() method throws a ClothingException, but by declaring the exception, the foo() method gets to duck the exception. No try/catch.

But now this means that whoever calls the foo() method has to follow the Handle or Declare law. If foo() ducks the exception (by declaring it), and main() calls foo(), then main() has to deal with the exception.

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

   public static void main (String[] args) {
        Washer a = new Washer();
        a.foo();
    }
}
```

TROUBLE!!

Now main() won't compile, and we get an "unreported exception" error. As far as the compiler's concerned, the foo() method throws an exception.

Because the foo() method ducks the ClothingException thrown by doLaundry(), main() has to wrap a.foo() in a try/catch, or main() has to declare that it, too, throws ClothingException!

Getting back to our music code...

Now that you've completely forgotten, we started this chapter with a first look at some JavaSound code. We created a Sequencer object but it wouldn't compile because the method Midi.getSequencer() declares a checked exception (MidiUnavailableException). But we can fix that now by wrapping the call in a try/catch.

```
public void play() {
        try {

            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Successfully got a sequencer");

        } catch(MidiUnavailableException ex) {
            System.out.println("Bummer");
        }
    } // close play
```

No problem calling getSequencer(), now that we've wrapped it in a try/catch block.

The catch parameter has to be the 'right' exception. If we said 'catch(FileNotFoundException f), the code would not compile, because polymorphically a MidiUnavilableException won't fit into a FileNotFoundException.

Remember it's not enough to have a catch block... you have to catch the thing being thrown!

Exception Rules

1 You cannot have a catch or finally without a try

```
void go() {
   Foo f = new Foo();
   f.foof();
   catch(FooException ex) { }
}
```

NOT LEGAL! Where's the try?

2 You cannot put code between the try and the catch

```
try {
   x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

NOT LEGAL! You can't put code between the try and the catch.

3 A try MUST be followed by either a catch or a finally

```
try {
   x.doStuff();
} finally {
    // cleanup
}
```

LEGAL because you have a finally, even though there's no catch. But you cannot have a try by itself.

4 A try with only a finally (no catch) must still declare the exception.

```
void go() throws FooException {
    try {
       x.doStuff();
    } finally { }
}
```

A try without a catch doesn't satisfy the handle or declare law

Code Kitchen

You don't have to do it yourself, but it's a lot more fun if you do.

The rest of this chapter is optional: you can use Ready-bake code for all the music apps.

But if you want to learn more about JavaSound, turn the page.

Making actual sound

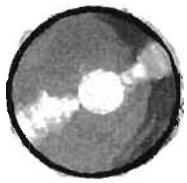
Remember near the beginning of the chapter, we looked at how MIDI data holds the instructions for *what* should be played (and *how* it should be played) and we also said that MIDI data doesn't actually *create any sound that you hear.* For sound to come out of the speakers, the MIDI data has to be sent through some kind of MIDI device that takes the MIDI instructions and renders them in sound, either by triggering a hardware instrument or a 'virtual' instrument (software synthesizer). In this book, we're using only software devices, so here's how it works in JavaSound:

You need FOUR things:

1. **The thing that plays the music**
2. **The music to be played...a song.**
3. **The part of the Sequence that holds the actual information**
4. **The actual music information: notes to play, how long, etc.**

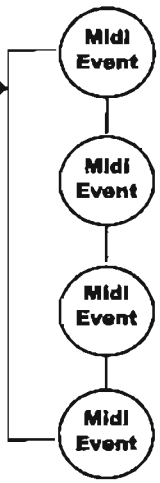
Sequencer plays → **Sequence** has a → **Track** holds → **MidiEvent**

The Sequencer is the thing that actually causes a song to be played. Think of it like a **music CD player.**



The Sequence is the song, the musical piece that the Sequencer will play. For this book, think of the Sequence as a music CD, but **the whole CD plays just one song.**

For this book, we only need one Track, so just imagine a a music CD with only one song. A single Track. This Track is where all the song data (MIDI information) lives.



A MIDI event is a message that the Sequencer can understand. A MIDI event might say (if it spoke English), "At this moment in time, play middle C, play it this fast and this hard, and hold it for this long."

A MIDI event might also say something like, "Change the current instrument to Flute."

For this book, think of the Sequence as a single-song CD (has only one Track). The information about how to play the song lives on the Track, and the Track is part of the Sequence.

And you need FIVE steps:

1. Get a **Sequencer** and open it

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

2. Make a new **Sequence**

```
Sequence seq = new Sequence(timing, 4);
```

3. Get a new **Track** from the Sequence

```
Track t = seq.createTrack();
```

4. Fill the Track with **MidiEvents** and give the Sequence to the Sequencer

```
t.add(myMidiEvent1);
player.setSequence(seq);
```

Your very first sound player app

Type it in and run it. You'll hear the sound of someone playing a single note on a piano! (OK, maybe not some*one*, but some*thing*.)

```
import javax.sound.midi.*;

public class MiniMiniMusicApp {

     public static void main(String[] args) {
        MiniMiniMusicApp mini = new MiniMiniMusicApp();
        mini.play();
     } // close main

    public void play() {

      try {

        Sequencer player = MidiSystem.getSequencer();
        player.open();

        Sequence seq = new Sequence(Sequence.PPQ, 4);

        Track track = seq.createTrack();

        ShortMessage a = new ShortMessage();
        a.setMessage(144, 1, 44, 100);
        MidiEvent noteOn = new MidiEvent(a, 1);
        track.add(noteOn);

        ShortMessage b = new ShortMessage();
        b.setMessage(128, 1, 44, 100);
        MidiEvent noteOff = new MidiEvent(b, 16);
        track.add(noteOff);

        player.setSequence(seq);

        player.start();

      } catch (Exception ex) {
          ex.printStackTrace();
      }
  } // close play
} // close class
```

Don't forget to import the midi package

get a Sequencer and open it (so we can use it... a Sequencer doesn't come already open)

Don't worry about the arguments to the Sequence constructor. Just copy these (think of 'em as Ready-bake arguments).

Ask the Sequence for a Track. Remember, the Track lives in the Sequence, and the MIDI data lives in the Track.

Put some MidiEvents into the Track. This part is mostly Ready-bake code. The only thing you'll have to care about are the arguments to the setMessage() method, and the arguments to the MidiEvent constructor. We'll look at those arguments on the next page.

Give the Sequence to the Sequencer (like putting the CD in the CD player)

Start() the Sequencer (like pushing PLAY)

Making a MidiEvent (song data)

A MidiEvent is an instruction for part of a song. A series of MidiEvents is kind of like sheet music, or a player piano roll. Most of the MidiEvents we care about describe ***a thing to do*** and the ***moment in time to do it.*** The moment in time part matters, since timing is everything in music. This note follows this note and so on. And because MidiEvents are so detailed, you have to say at what moment to *start* playing the note (a NOTE ON event) and at what moment to *stop* playing the notes (NOTE OFF event). So you can imagine that firing the "stop playing note G" (NOTE OFF message) *before* the "start playing Note G" (NOTE ON) message wouldn't work.

The MIDI instruction actually goes into a Message object; the MidiEvent is a combination of the Message plus the moment in time when that message should 'fire'. In other words, the Message might say, "Start playing Middle C" while the MidiEvent would say, "Trigger this message at beat 4".

So we always need a Message and a MidiEvent.

The Message says *what* to do, and the MidiEvent says *when* to do it.

A MidiEvent says what to do and when to do it.

Every instruction must include the timing for that instruction.

In other words, at which beat that thing should happen.

1. Make a **Message**

```
ShortMessage a = new ShortMessage();
```

2. Put the **Instruction** in the Message

```
a.setMessage(144, 1, 44, 100);
```

This message says, "start playing note 44" (we'll look at the other numbers on the next page)

3. Make a new **MidiEvent** using the Message

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

The instructions are in the message, but the MidiEvent adds the moment in time when the instruction should be triggered. This MidiEvent says to trigger message 'a' at the first beat (beat 1).

4. Add the MidiEvent to the **Track**

```
track.add(noteOn);
```

A Track holds all the MidiEvent objects. The Sequence organizes them according to when each event is supposed to happen, and then the Sequencer plays them back in that order. You can have lots of events happening at the exact same moment in time. For example, you might want two notes played simultaneously, or even different instruments playing different sounds at the same time.

MIDI message: the heart of a MidiEvent

A MIDI message holds the part of the event that says *what* to do. The actual instruction you want the sequencer to execute. The first argument of an instruction is always the type of the message.The values you pass to the other three arguments depend on the type of message. For example, a message of type 144 means "NOTE ON". But in order to carry out a NOTE ON, the sequencer needs to know a few things. Imagine the sequencer saying, "OK, I'll play a note, but *which channel*? In other words, do you want me to play a Drum note or a Piano note? And *which note*? Middle-C? D Sharp? And while we're at it, at *which velocity* should I play the note?

To make a MIDI message, make a ShortMessage instance and invoke setMessage(), passing in the four arguments for the message. But remember, the message says only *what* to do, so you still need to stuff the message into an event that adds *when* that message should 'fire'.

Anatomy of a message

The *first* argument to setMessage() always represents the message 'type', while the *other* three arguments represent different things depending on the message type.

```
a.setMessage(144,  1,  44,  100);
```

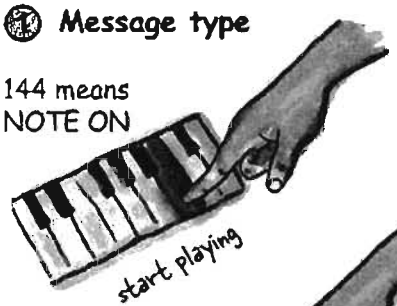
message type, channel, note to play, velocity

The last 3 args vary depending on the message type. This is a NOTE ON message, so the other args are for things the Sequencer needs to know in order to play a note.

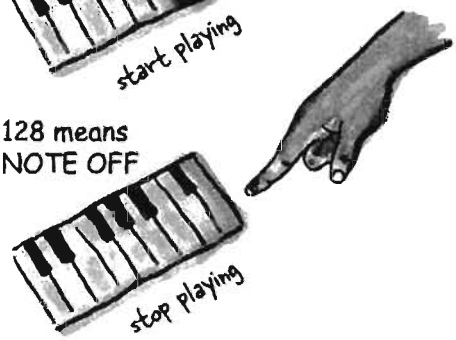
The *Message* says what to do, the *MidiEvent* says when to do it.

Message type

144 means NOTE ON



128 means NOTE OFF

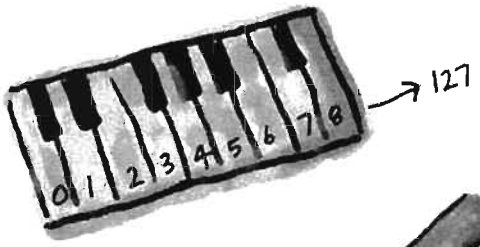


Channel

Think of a channel like a musician in a band. Channel 1 is musician 1 (the keyboard player), channel 9 is the drummer, etc.

Note to play

A number from 0 to 127, going from low to high notes.



Velocity

How fast and hard did you press the key? 0 is so soft you probably won't hear anything, but 100 is a good default.

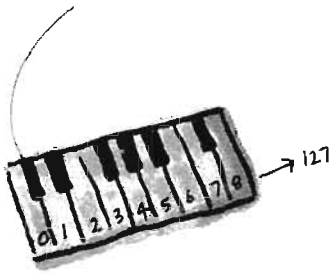
Change a message

Now that you know what's in a Midi message, you can start experimenting. You can change the note that's played, how long the note is held, add more notes, and even change the instrument.

Change the note

Try a number between 0 and 127 in the note on and note off messages.

```
a.setMessage(144, 1, 20, 100);
```



Change the duration of the note

Change the note off event (not the *message*) so that it happens at an earlier or later beat.

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```

Change the instrument

Add a new message, BEFORE the note-playing message, that sets the instrument in channel 1 to something other than the default piano. The change-instrument message is '192', and the third argument represents the actual instrument (try a number between 0 and 127)

```
first.setMessage(192, 1, 102, 0);
```

change-instrument message in channel 1 (musician 1) to instrument 102

change the instrument **and** note

Version 2: Using command-line args to experiment with sounds

This version still plays just a single note, but you get to use command-line arguments to change the instrument and note. Experiment by passing in two int values from 0 to 127. The first int sets the instrument, the second int sets the note to play.

```
import javax.sound.midi.*;

public class MiniMusicCmdLine {    // this is the first one

   public static void main(String[] args) {
        MiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
            System.out.println("Don't forget the instrument and note args");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
   } // close main

   public void play(int instrument, int note) {

      try {

         Sequencer player = MidiSystem.getSequencer();
         player.open();
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         MidiEvent event = null;

         ShortMessage first = new ShortMessage();
         first.setMessage(192, 1, instrument, 0);
         MidiEvent changeInstrument = new MidiEvent(first, 1);
         track.add(changeInstrument);

         ShortMessage a = new ShortMessage();
         a.setMessage(144, 1, note, 100);
         MidiEvent noteOn = new MidiEvent(a, 1);
         track.add(noteOn);

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, note, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);
         player.setSequence(seq);
         player.start();

      } catch (Exception ex) {ex.printStackTrace();}
   } // close play
} // close class
```

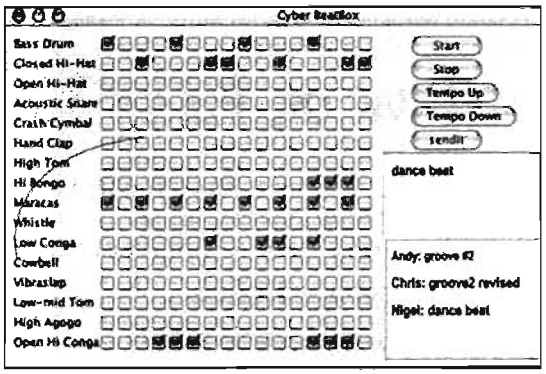
Run it with two int args from 0 to 127. Try these for starters:

```
File Edit Window Help Attenuate
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```

Where we're headed with the rest of the CodeKitchens

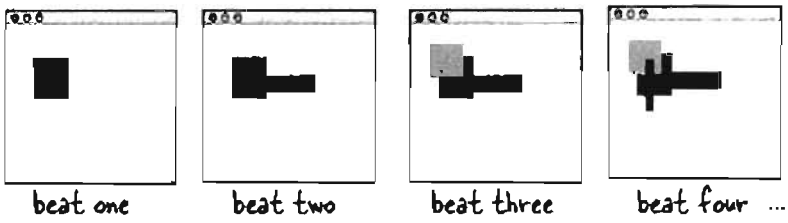
Chapter 15: the goal

When we're done, we'll have a working BeatBox that's also a Drum Chat Client. We'll need to learn about GUIs (including event handling), I/O, networking, and threads. The next three chapters (12, 13, and 14) will get us there.



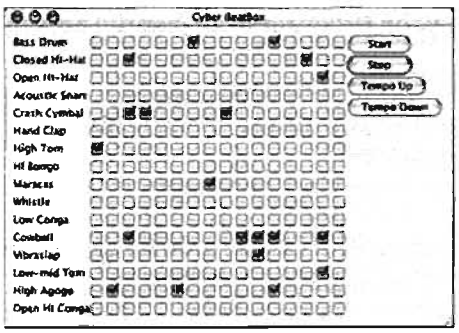
Chapter 12: MIDI events

This CodeKitchen lets us build a little "music video" (bit of a stretch to call it that...) that draws random rectangles to the beat of the MIDI music. We'll learn how to construct and play a lot of MIDI events (instead of just a couple, as we do in the current chapter).



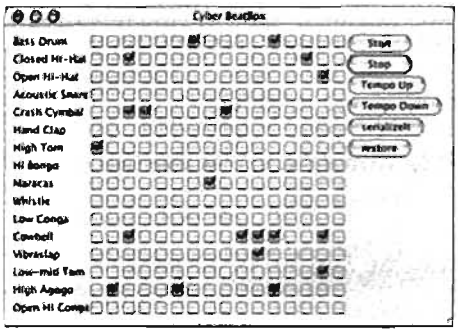
Chapter 13: Stand-alone BeatBox

Now we'll actually build the real BeatBox, GUI and all. But it's limited—as soon as you change a pattern, the previous one is lost. There's no Save and Restore feature, and it doesn't communicate with the network. (But you can still use it to work on your drum pattern skills.)



Chapter 14: Save and Restore

You've made the perfect pattern, and now you can save it to a file, and reload it when you want to play it again. This gets us ready for the final version (chapter 15), where instead of writing the pattern to a file, we send it over a network to the chat server.



exercise: True or False

This chapter explored the wonderful world of exceptions. Your job is to decide whether each of the following exception-related statements is true or false.

TRUE OR FALSE

1. A try block must be followed by a catch *and* a finally block.

2. If you write a method that might cause a compiler-checked exception, you *must* wrap that risky code in a try / catch block.

3. Catch blocks can be polymorphic.

4. Only 'compiler checked' exceptions can be caught.

5. If you define a try / catch block, a matching finally block is optional.

6. If you define a try block, you can pair it with a matching catch or finally block, or both.

7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try / catch block.

8. The main( ) method in your program must handle all unhandled exceptions thrown to it.

9. A single try block can have many different catch blocks.

10. A method can only throw one kind of exception.

11. A finally block will run regardless of whether an exception is thrown.

12. A finally block can exist without a try block.

13. A try block can exist by itself, without a catch block or a finally block.

14. Handling an exception is sometimes referred to as 'ducking'.

15. The order of catch blocks never matters.

16. A method with a try block and a finally block, can optionally declare the exception.

17. Runtime exceptions must be *handled* or *declared*.

Code Magnets

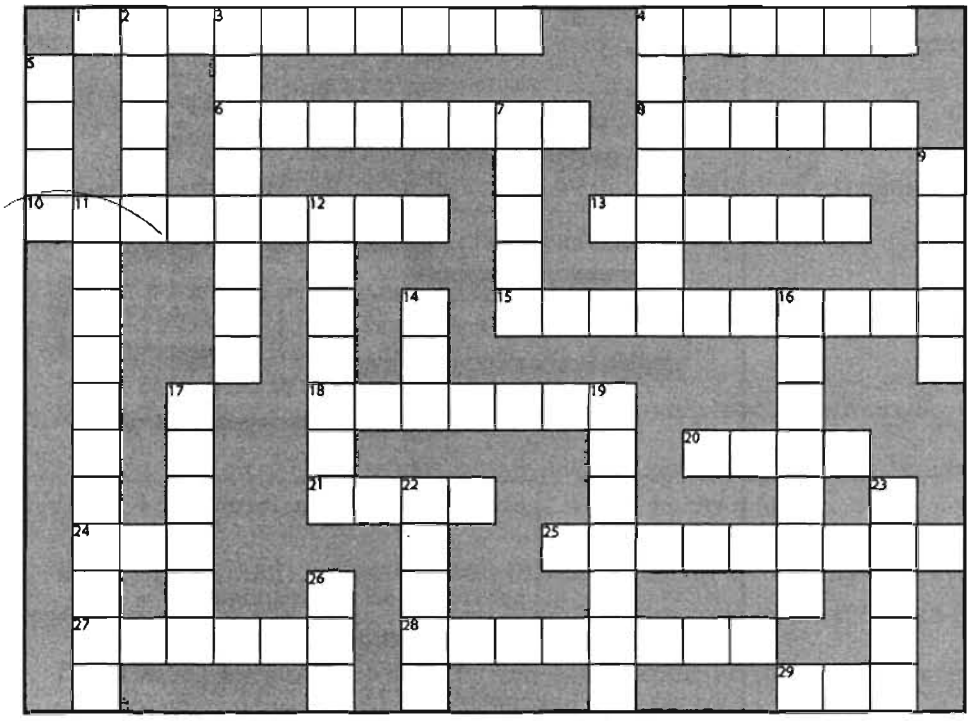
A working Java program is scrambled up on the fridge. Can you reconstruct all the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
File Edit Window Help ThrowUp
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

```
System.out.print("r");
try {
doRisky(test);
System.out.print("t");
System.out.println("s");
} finally {
System.out.print("o");
class MyEx extends Exception { }
public class ExTestDrive {
System.out.print("w");
if ("yes".equals(t)) {
System.out.print("a");
throw new MyEx();
} catch (MyEx e) {
static void doRisky(String t) throws MyEx {
System.out.print("h");
public static void main(String [] args) {
String test = args[0];
```

puzzle: crossword

JavaCross 7.0



You know what to do!

Across

1. To give value
4. Flew off the top
6. All this and more!
8. Start
10. The family tree
13. No ducking
15. Problem objects
18. One of Java's '49'
20. Class hierarchy
21. Too hot to handle
24. Common primitive
25. Code recipe
27. Unruly method action
28. No Picasso here
29. Start a chain of events

Down

2. Currently usable
3. Template's creation
4. Don't show the kids
5. Mostly static API class
7. Not about behavior
9. The template
11. Roll another one off the line
12. Javac saw it coming
14. Attempt risk
16. Automatic acquisition
17. Changing method
19. Announce a duck
22. Deal with it
23. Create bad news
26. One of my roles

More Hints:

Across

20. Also a type of collection
21. Quack
27. Starts a problem
28. Not Abstract
6. A Java child
8. Start a method
13. Instead of declare

Down

2. Or a mouthwash
3. For ____ (not example)
5. Numbers...
9. Only public or default
16. ____ the family fortune
17. Not a 'getter'

Exercise Solutions

True or False

1. False, either or both.
2. False, you can declare the exception.
3. True.
4. False, runtime exception can be caught.
5. True.
6. True, both are acceptable.
7. False, the declaration is sufficient.
8. False, but if it doesn't the JVM may shut down.
9. True.
10. False.
11. True. It's often used to clean-up partially completed tasks.
12. False.
13. False.
14. False, ducking is synonomous with declaring.
15. False, broadest exceptions must be caught by the last catch blocks.
16. False, if you don't have a catch block, you *must* declare.
17. False.

Code Magnets

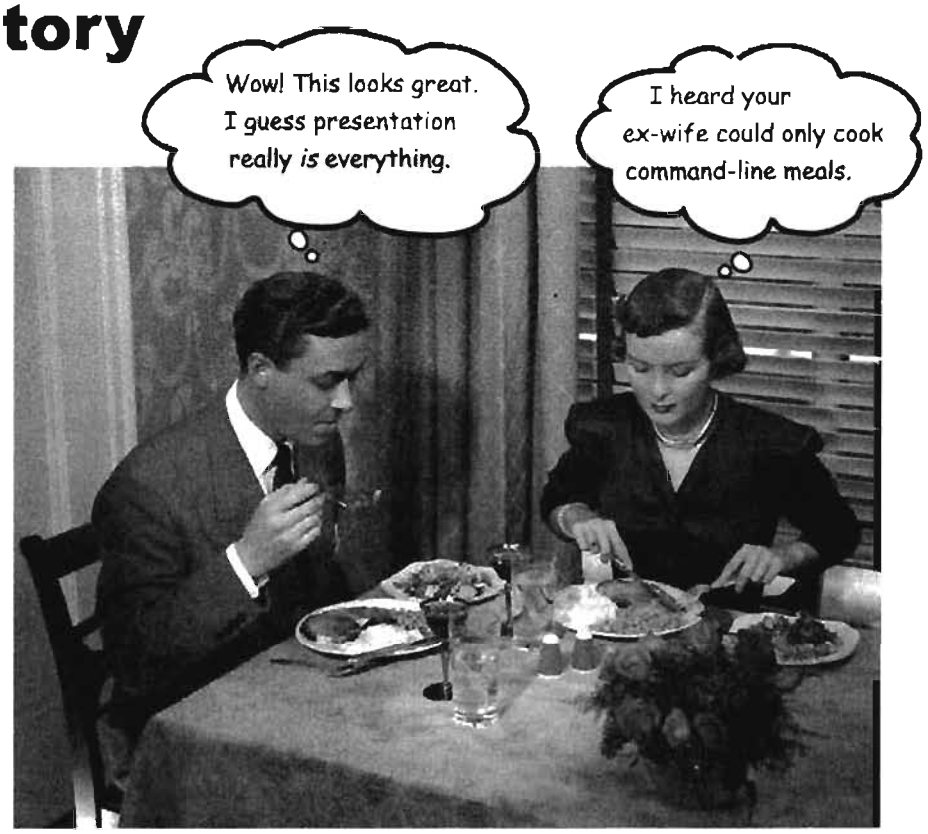
```
class MyEx extends Exception { }

public class ExTestDrive {

  public static void main(String [] args) {
    String test = args[0];
    try {

       System.out.print("t");

       doRisky(test);

       System.out.print("o");

    } catch ( MyEx e) {

        System.out.print("a");

    } finally {

        System.out.print("w");
    }
    System.out.println("s");
  }

  static void doRisky(String t) throws MyEx {
    System.out.print("h");

    if ("yes".equals(t)) {

      throw new MyEx();
    }

    System.out.print("r");

  }
}
```

```
File Edit Window Help Chill
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

JavaCross Answers

	1 A	2 S	S	3 I	G	N	M	E	N	T			4 P	O	P	P	E	D				
5 M		C		N									R									
A		O		6 S	U	B	C	L	A	7 S	S		8 I	N	V	O	K	E				
T		P		T						T			V						9 C			
10 H	11 I	E	R	A	R	12 C	H	Y		A		13 H	A	N	D	L	E		L			
	N			N		H				T			T						A			
	S			C		E		14 T		15 E	X	C	E	P	T	16 I	O	N	S			
	T			E		C		R								N			S			
	A		17 S			18 K	E	Y	W	O	R	19 D				H						
	N		E			E						E		20 T	R	E	E					
	T		T			21 D	U	22 C	K			C				R		23 T				
	24 I	N	T					A			25 A	L	G	O	R	I	T	H	M			
	A		E			26 I		T				A				T		R				
	27 T	H	R	O	W	S		28 C	O	N	C	R	E	T	E			O				
	E					A		H				E				29 N	E	W				

A Very Graphic Story



Face it, you need to make GUIs. If you're building applications that other people are going to use, you *need* a graphical interface. If you're building programs for yourself, you *want* a graphical interface. Even if you believe that the rest of your natural life will be spent writing server-side code, where the client user interface is a web page, sooner or later you'll need to write tools, and you'll want a graphical interface. Sure, command-line apps are retro, but not in a good way. They're weak, inflexible, and unfriendly. We'll spend two chapters working on GUIs, and learn key Java language features along the way including **Event Handling** and **Inner Classes**. In this chapter, we'll put a button on the screen, and make it do something when you click it. We'll paint on the screen, we'll display a jpeg image, and we'll even do some animation.

It all starts with a window

A JFrame is the object that represents a window on the screen. It's where you put all the interface things like buttons, checkboxes, text fields, and so on. It can have an honest-to-goodness menu bar with menu items. And it has all the little windowing icons for whatever platform you're on, for minimizing, maximizing, and closing the window.

The JFrame looks different depending on the platform you're on. This is a JFrame on Mac OS X:

a JFrame with a menu bar and two 'widgets' (a button and a radio button)

"If I see one more command-line app, you're fired."

Put widgets in the window

Once you have a JFrame, you can put things ('widgets') in it by adding them to the JFrame. There are a ton of Swing components you can add; look for them in the javax.swing package. The most common include JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField, and JTable. Most are really simple to use, but some (like JTable) can be a bit more complicated.

Making a GUI is easy:

1. Make a frame (a JFrame)

```
JFrame frame = new JFrame();
```

2. Make a widget (button, text field, etc.)

```
JButton button = new JButton("click me");
```

3. Add the widget to the frame

```
frame.getContentPane().add(button);
```

You don't add things to the frame directly. Think of the frame as the trim around the window, and you add things to the window pane.

4. Display it (give it a size and make it visible)

```
frame.setSize(300,300);
frame.setVisible(true);
```

Your first GUI: a button on a frame

```
import javax.swing.*;

public class SimpleGui1 {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click me");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(button);

        frame.setSize(300,300);

        frame.setVisible(true);
    }
}
```

don't forget to import this swing package

make a frame and a button

(you can pass the button constructor the text you want on the button)

this line makes the program quit as soon as you close the window (if you leave this out it will just sit there on the screen forever)

add the button to the frame's content pane

give the frame a size, in pixels

finally, make it visible!! (if you forget this step, you won't see anything when you run this code)

Let's see what happens when we run it:

```
%java SimpleGui1
```

Whoa! That's a Really Big Button.

The button fills all the available space in the frame. Later we'll learn to control where (and how big) the button is on the frame.

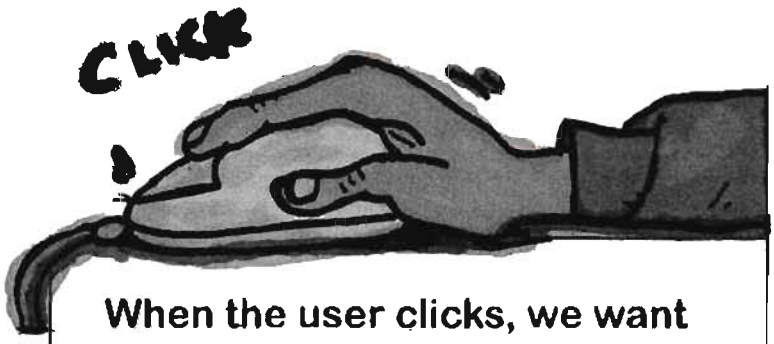
But nothing happens when I click it...

That's not exactly true. When you press the button it shows that 'pressed' or 'pushed in' look (which changes depending on the platform look and feel, but it always does *something* to show when it's being pressed).

The real question is, "How do I get the button to do something specific when the user clicks it?"

We need two things:

1. A ***method*** to be called when the user clicks (the thing you want to happen as a result of the button click).
2. A way to ***know*** when to trigger that method. In other words, a way to know when the user clicks the button!



When the user clicks, we want to know.

We're interested in the user-takes-action-on-a-button event.

there are no Dumb Questions

Q: **Will a button look like a Windows button when you run on Windows?**

A: If you want it to. You can choose from a few "look and feels"—classes in the core library that control what the interface looks like. In most cases you can choose between at least two different looks: the standard Java look and feel, also known as ***Metal***, and the native look and feel for your platform. The Mac OS X screens in this book use either the OS X ***Aqua*** look and feel, or the ***Metal*** look and feel.

Q: **Can I make a program look like Aqua all the time? Even when it's running under Windows?**

A: Nope. Not all look and feels are available on every platform. If you want to be safe, you can either explicitly set the look and feel to Metal, so that you know exactly what you get regardless of where the app is running, or don't specify a look and feel and accept the defaults.

Q: **I heard Swing was dog-slow and that nobody uses it.**

A: This was true in the past, but isn't a given anymore. On weak machines, you might feel the pain of Swing. But on the newer desktops, and with Java version 1.3 and beyond, you might not even notice the difference between a Swing GUI and a native GUI. Swing is used heavily today, in all sorts of applications.

Getting a user event

Imagine you want the text on the button to change from *click me* to *I've been clicked* when the user presses the button. First we can write a method that changes the text of the button (a quick look through the API will show you the method):

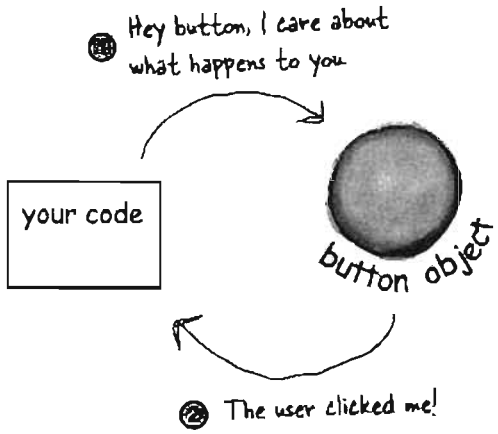
```
public void changeIt() {
     button.setText("I've been clicked!");
}
```

But *now* what? How will we *know* when this method should run? ***How will we know when the button is clicked?***

In Java, the process of getting and handling a user event is called *event-handling*. There are many different event types in Java, although most involve GUI user actions. If the user clicks a button, that's an event. An event that says "The user wants the action of this button to happen." If it's a "Slow Tempo" button, the user wants the slow-tempo action to occur. If it's a Send button on a chat client, the user wants the send-my-message action to happen. So the most straightforward event is when the user clicked the button, indicating they want an action to occur.

With buttons, you usually don't care about any intermediate events like button-is-being-pressed and button-is-being-released. What you want to say to the button is, "I don't care how the user plays with the button, how long they hold the mouse over it, how many times they change their mind and roll off before letting go, etc. ***Just tell me when the user means business!*** In other words, don't call me unless the user clicks in a way that indicates he wants the darn button to do what it says it'll do!"

First, the button needs to know that we care.



Second, the button needs a way to call us back when a button-clicked event occurs.

1) How could you tell a button object that you care about its events? That you're a concerned listener?

2) How will the button call you back? Assume that there's no way for you to tell the button the name of your unique method (changeIt()). So what else can we use to reassure the button that we have a specific method it can call when the event happens? [hint: think Pet]

If you care about the button's events, implement an interface that says, "I'm listening for your events."

A listener interface is the bridge between the listener (you) and event source (the button).

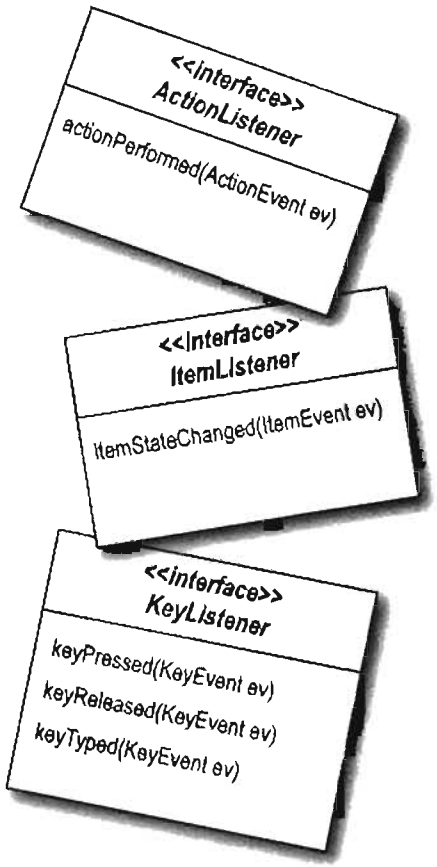
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through the java.awt.event package in the API, you'll see a bunch of event classes (easy to spot—they all have *Event* in the name). You'll find MouseEvent, KeyEvent, WindowEvent, ActionEvent, and several others.

An event *source* (like a button) creates an *event object* when the user does something that matters (like *click* the button). Most of the code you write (and all the code in this book) will *receive* events rather than *create* events. In other words, you'll spend most of your time as an event *listener* rather than an event *source*.

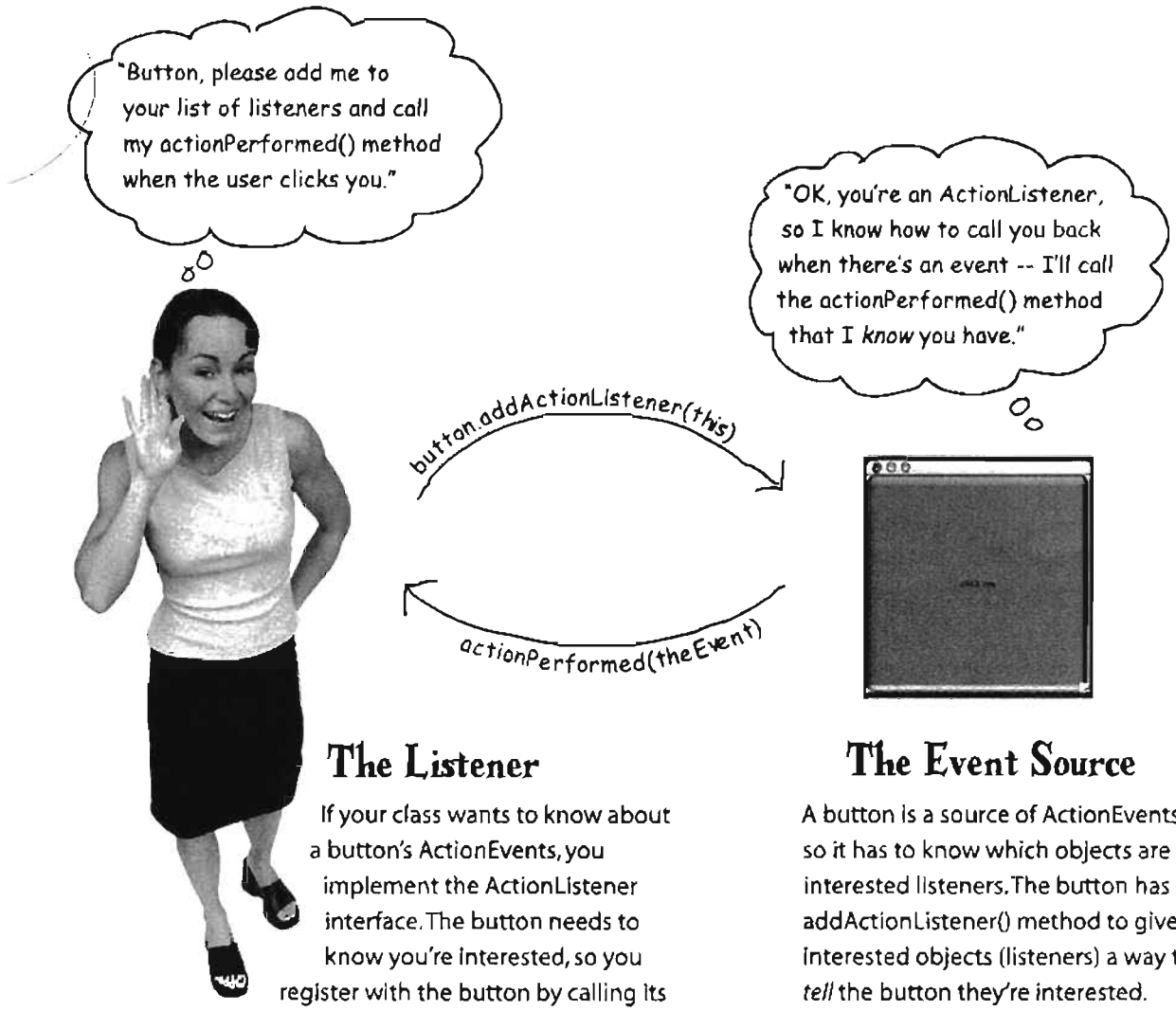
Every event type has a matching listener interface. If you want MouseEvents, implement the MouseListener interface. Want WindowEvents? Implement WindowListener. You get the idea. And remember your interface rules—to implement an interface you *declare* that you implement it (class Dog implements Pet), which means you must *write implementation methods* for every method in the interface.

Some interfaces have more than one method because the event itself comes in different flavors. If you implement MouseListener, for example, you can get events for mousePressed, mouseReleased, mouseMoved, etc. Each of those mouse events has a separate method in the interface, even though they all take a MouseEvent. If you implement MouseListener, the mousePressed() method is called when the user (you guessed it) presses the mouse. And when the user lets go, the mouseReleased() method is called. So for mouse events, there's only one event *object*, MouseEvent, but several different event *methods*, representing the different *types* of mouse events.

When you implement a listener interface, you give the button a way to call you back. The interface is where the call-back method is declared.



How the listener and source communicate:



The Listener

If your class wants to know about a button's ActionEvents, you implement the ActionListener interface. The button needs to know you're interested, so you register with the button by calling its addActionListener(this) and passing an ActionListener reference to it (in this case, *you* are the ActionListener so you pass *this*). The button needs a way to call you back when the event happens, so it calls the method in the listener interface. As an ActionListener, you *must* implement the interface's sole method, actionPerformed(). The compiler guarantees it.

The Event Source

A button is a source of ActionEvents, so it has to know which objects are interested listeners. The button has an addActionListener() method to give interested objects (listeners) a way to *tell* the button they're interested.

When the button's addActionListener() runs (because a potential listener invoked it), the button takes the parameter (a reference to the listener object) and stores it in a list. When the user clicks the button, the button 'fires' the event by calling the actionPerformed() method on each listener in the list.

Getting a button's ActionEvent

1. Implement the ActionListener interface
2. Register with the button (tell it you want to listen for events)
3. Define the event-handling method (implement the actionPerformed() method from the ActionListener interrface)

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

public class SimpleGui1B implements ActionListener {
    JButton button;

    public static void main (String[] args) {
       SimpleGui1B gui = new SimpleGui1B();
       gui.go();
     }

    public void go() {
       JFrame frame = new JFrame();
       button = new JButton("click me");

       button.addActionListener(this);

       frame.getContentPane().add(button);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
       button.setText("I've been clicked!");
    }
}
```

a new import statement for the package that ActionListener and ActionEvent are in.

Implement the interface. This says, "an instance of SimpleGui1B IS-A ActionListener".

(The button will give events only to ActionListener implementers)

register your interest with the button. This says to the button, "Add me to your list of listeners". The argument you pass MUST be an object from a class that implements ActionListener!!

Implement the ActionListener interface's actionPerformed() method.. This is the actual event-handling method!

The button calls this method to let you know an event happened. It sends you an ActionEvent object as the argument, but we don't need it. Knowing the event happened is enough info for us.

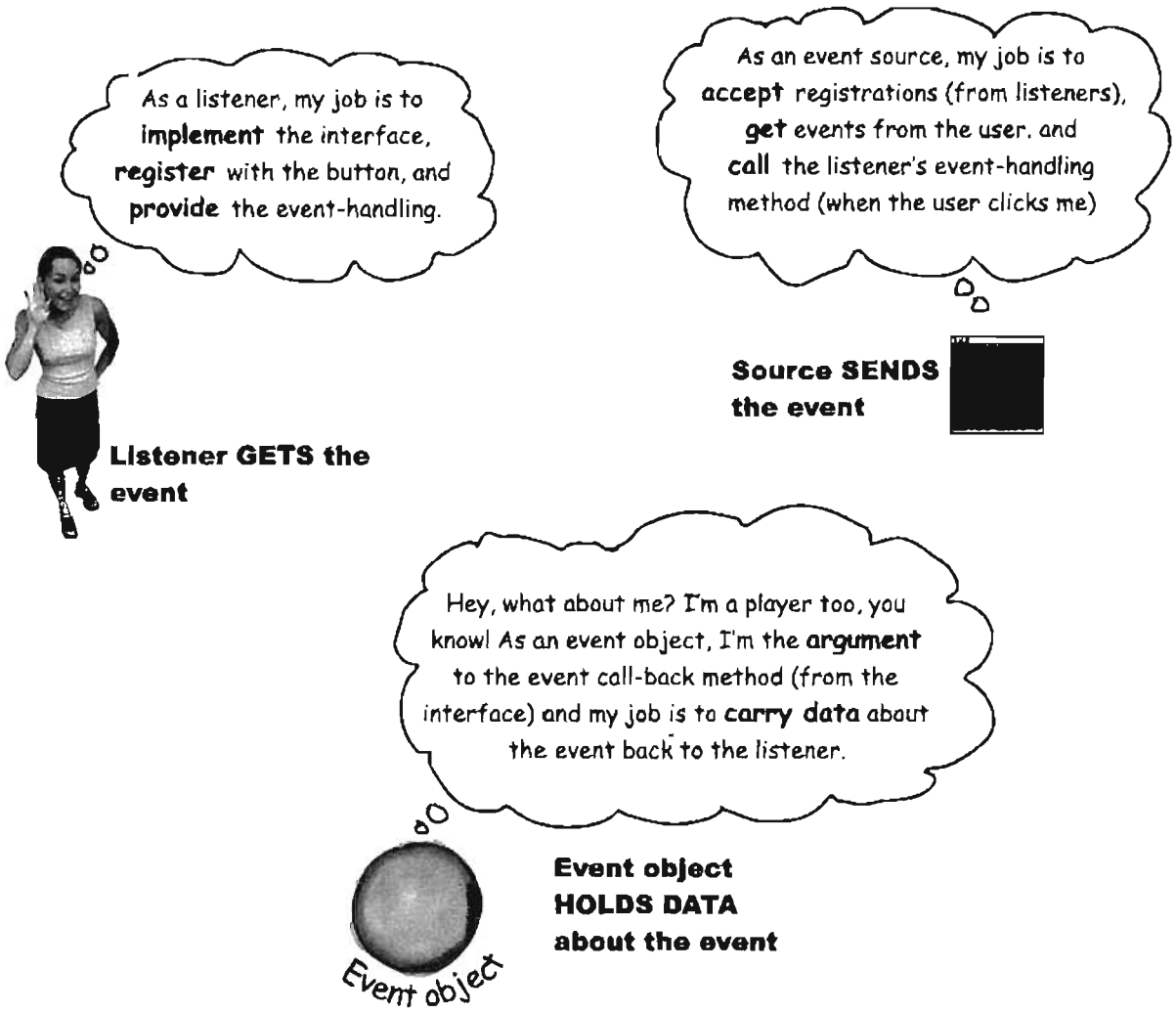
Listeners, Sources, and Events

For most of your stellar Java career, *you* will not be the *source* of events.

(No matter how much you fancy yourself the center of your social universe.)

Get used to it. ***Your job is to be a good listener.***

(Which, if you do it sincerely, *can* improve your social life.)



there are no Dumb Questions

Q: **Why can't I be a source of events?**

A: You CAN. We just said that *most* of the time you'll be the receiver and not the originator of the event (at least in the *early* days of your brilliant Java career). Most of the events you might care about are 'fired' by classes in the Java API, and all you have to do is be a listener for them. You might, however, design a program where you need a custom event, say, StockMarketEvent thrown when your stock market watcher app finds something it deems important. In that case, you'd make the StockWatcher object be an event source, and you'd do the same things a button (or any other source) does—make a listener interface for your custom event, provide a registration method (addStockListener()), and when somebody calls it, add the caller (a listener) to the list of listeners. Then, when a stock event happens, instantiate a StockEvent object (another class you'll write) and send it to the listeners in your list by calling their stockChanged(StockEvent ev) method. And don't forget that for every *event type* there must be a *matching listener interface* (so you'll create a StockListener interface with a stockChanged() method).

Q: **I don't see the importance of the event object that's passed to the event call-back methods. If somebody calls my mousePressed method, what other info would I need?**

A: A lot of the time, for most designs, you don't need the event object. It's nothing more than a little data carrier, to send along more info about the event. But sometimes you might need to query the event for specific details about the event. For example, if your mousePressed() method is called, you know the mouse was pressed. But what if you want to know exactly where the mouse was pressed? In other words, what if you want to know the X and Y screen coordinates for where the mouse was pressed?

Or sometimes you might want to register the *same* listener with *multiple* objects. An onscreen calculator, for example, has 10 numeric keys and since they all do the same thing, you might not want to make a separate listener for every single key. Instead, you might register a single listener with each of the 10 keys, and when you get an event (because your event call-back method is called) you can call a method on the event object to find out *who* the real event source was. In other words, *which key sent this event*.

Sharpen your pencil

Each of these widgets (user interface objects) are the source of one or more events. Match the widgets with the events they might cause. Some widgets might be a source of more than one event, and some events can be generated by more than one widget.

Widgets	Event methods
check box	windowClosing()
text field	actionPerformed()
scrolling list	itemStateChanged()
button	mousePressed()
dialog box	keyTyped()
radio button	mouseExited()
menu item	focusGained()

How do you KNOW if an object is an event source?

Look in the API.

OK. Look for what?

A method that starts with 'add', ends with 'Listener', and takes a listener interface argument. If you see:

`addKeyListener(KeyListener k)`

you know that a class with this method is a source of KeyEvents. There's a naming pattern.

Getting back to graphics...

Now that we know a little about how events work (we'll learn more later), let's get back to putting stuff on the screen. We'll spend a few minutes playing with some fun ways to get graphic, before returning to event handling.

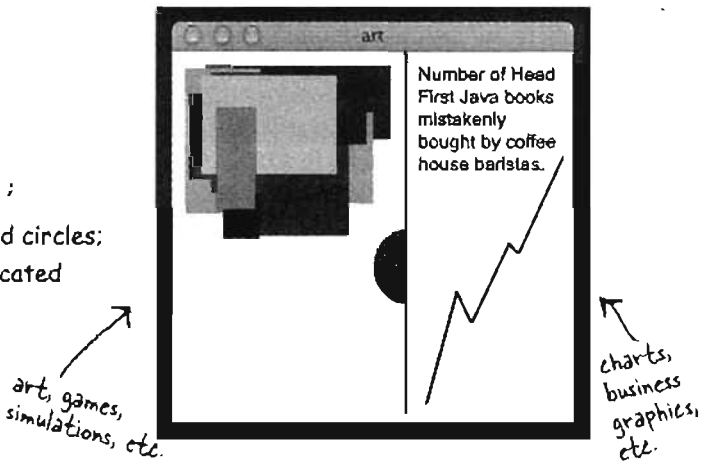
Three ways to put things on your GUI:

Put widgets on a frame

Add buttons, menus, radio buttons, etc.

```
frame.getContentPane().add(myButton);
```

The javax.swing package has more than a dozen widget types.



Draw 2D graphics on a widget

Use a graphics object to paint shapes.

```
graphics.fillOval(70,70,100,100);
```

You can paint a lot more than boxes and circles; the Java2D API is full of fun, sophisticated graphics methods.

Put a JPEG on a widget

You can put your own images on a widget.

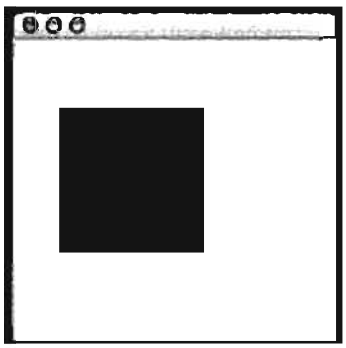
```
graphics.drawImage(myPic,10,10,this);
```

Make your own drawing widget

If you want to put your own graphics on the screen, your best bet is to make your own paintable widget. You plop that widget on the frame, just like a button or any other widget, but when it shows up it will have your images on it. You can even make those images move, in an animation, or make the colors on the screen change every time you click a button.

It's a piece of cake.

Make a subclass of JPanel and override one method, paintComponent().



All of your graphics code goes inside the paintComponent() method. Think of the paintComponent() method as the method called by the system to say, "Hey widget, time to paint yourself." If you want to draw a circle, the paintComponent() method will have code for drawing a circle. When the frame holding your drawing panel is displayed, paintComponent() is called and your circle appears. If the user iconifies/minimizes the window, the JVM knows the frame needs "repair" when it gets de-iconified, so it calls paintComponent() again. Anytime the JVM thinks the display needs refreshing, your paintComponent() method will be called.

One more thing, ***you never call this method yourself!*** The argument to this method (a Graphics object) is the actual drawing canvas that gets slapped onto the *real* display. You can't get this by yourself; it must be handed to you by the system. You'll see later, however, that you *can* ask the system to refresh the display (repaint()), which ultimately leads to paintComponent() being called.

```
import java.awt.*;
import javax.swing.*;

class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {

        g.setColor(Color.orange);

        g.fillRect(20,50,100,100);
    }
}
```

you need both of these

Make a subclass of JPanel, a widget that you can add to a frame just like anything else. Except this one is your own customized widget.

This is the Big Important Graphics method. You will NEVER call this yourself. The system calls it and says, "Here's a nice fresh drawing surface, of type Graphics, that you may paint on now."

Imagine that 'g' is a painting machine. You're telling it what color to paint with and then what shape to paint (with coordinates for where it goes and how big it is)

Fun things to do in paintComponent()

Let's look at a few more things you can do in paintComponent(). The most fun, though, is when you start experimenting yourself. Try playing with the numbers, and check the API for class Graphics (later we'll see that there's even *more* you can do besides what's in the Graphics class).

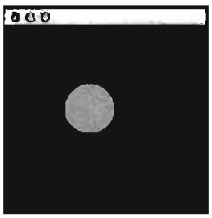
Display a JPEG

```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

your file name goes here

The x,y coordinates for where the picture's top left corner should go. This says "3 pixels from the left edge of the panel and 4 pixels from the top edge of the panel". These numbers are always relative to the widget (in this case your JPanel subclass), not the entire frame.

Paint a randomly-colored circle on a black background



```
public void paintComponent(Graphics g) {

    g.fillRect(0,0,this.getWidth(), this.getHeight());

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);

    Color randomColor = new Color(red, green, blue);
    g.setColor(randomColor);
    g.fillOval(70,70,100,100);
}
```

fill the entire panel with black (the default color)

The first two args define the (x,y) upper left corner, relative to the panel, for where drawing starts, so 0, 0 means "start 0 pixels from the left edge and 0 pixels from the top edge." The other two args say, "Make the width of this rectangle as wide as the panel (this.width()), and make the height as tall as the panel (this.height)"

You can make a color by passing in 3 ints to represent the RGB values.

start 70 pixels from the left, 70 from the top, make it 100 pixels wide, and 100 pixels tall.

Behind every good Graphics reference is a Graphics2D object.

The argument to paintComponent() is declared as type Graphics (java.awt.Graphics).

```
public void paintComponent(Graphics g) { }
```

So the parameter 'g' IS-A Graphics object. Which means it *could* be a *subclass* of Graphics (because of polymorphism). And in fact, it *is*.

The object referenced by the 'g' parameter is actually an instance of the* Graphics2D *class.

Why do you care? Because there are things you can do with a Graphics2D reference that you can't do with a Graphics reference. A Graphics2D object can do more than a Graphics object, and it really is a Graphics2D object lurking behind the Graphics reference.

Remember your polymorphism. The compiler decides which methods you can call based on the reference type, not the object type. If you have a Dog object referenced by an Animal reference variable:

```
Animal a = new Dog();
```

You can NOT say:

```
a.bark();
```

Even though you know it's really a Dog back there. The compiler looks at 'a', sees that it's of type Animal, and finds that there's no remote control button for bark() in the Animal class. But you can still get the object back to the Dog it really *is* by saying:

```
Dog d = (Dog) a;
d.bark();
```

So the bottom line with the Graphics object is this:

If you need to use a method from the Graphics2D class, you can't *use* the the paintComponent parameter ('g') straight from the method. But you can *cast* it with a new Graphics2D variable.

```
Graphics2D g2d = (Graphics2D) g;
```

Methods you can call on a Graphics reference:

- drawImage()
- drawLine()
- drawPolygon
- drawRect()
- drawOval()
- fillRect()
- fillRoundRect()
- setColor()

To cast the Graphics2D *object* to a Graphics2D *reference*:

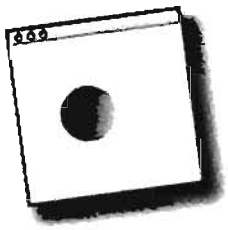
```
Graphics2D g2d = (Graphics2D) g;
```

Methods you can call on a Graphics2D reference:

- fill3DRect()
- draw3DRect()
- rotate()
- scale()
- shear()
- transform()
- setRenderingHints()

(these are not complete method lists, check the API for more)

Because life's too short to paint the circle a solid color when there's a gradient blend waiting for you.



```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);

    g2d.setPaint(gradient);

    g2d.fillOval(70,70,100,100);
```

it's really a Graphics2D object masquerading as a mere Graphics object

cast it so we can call something that Graphics2D has but Graphics doesn't

starting point

starting color

ending point

ending color

this sets the virtual paint brush to a gradient instead of a solid color

the fillOval() method really means "fill the oval with whatever is loaded on your paintbrush (i.e. the gradient)"

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 255);
    green = (int) (Math.random() * 255);
    blue = (int) (Math.random() * 255);
    Color endColor = new Color(red, green, blue);

    GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
```

this is just like the one above, except it makes random colors for the start and stop colors of the gradient. Try it!

BULLET POINTS

EVENTS

- To make a GUI, start with a window, usually a JFrame
  `JFrame frame = new JFrame();`
- You can add widgets (buttons, text fields, etc.) to the JFrame using:
  `frame.getContentPane().add(button);`
- Unlike most other components, the JFrame doesn't let you add to it directly, so you must add to the JFrame's content pane.
- To make the window (JFrame) display, you must give it a size and tell it be visible:
  `frame.setSize(300,300);`
  `frame.setVisible(true);`
- To know when the user clicks a button (or takes some other action on the user interface) you need to listen for a GUI event.
- To listen for an event, you must register your interest with an event source. An event source is the thing (button, checkbox, etc.) that 'fires' an event based on user interaction.
- The listener interface gives the event source a way to call you back, because the interface defines the method(s) the event source will call when an event happens.
- To register for events with a source, call the source's registration method. Registration methods always take the form of: ***add<EventType>Listener***. To register for a button's ActionEvents, for example, call:
  `button.addActionListener(this);`
- Implement the listener interface by implementing all of the interface's event-handling methods. Put your event-handling code in the listener call-back method. For ActionEvents, the method is:

```
public void actionPerformed(ActionEvent
                            event) {
    button.setText("you clicked!");
}
```

- The event object passed into the event-handler method carries information about the event, including the source of the event.

GRAPHICS

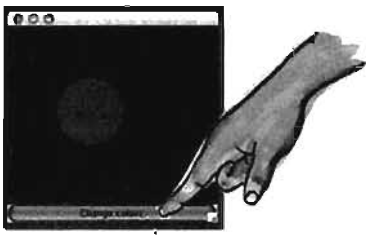
- You can draw 2D graphics directly on to a widget.
- You can draw a .gif or .jpeg directly on to a widget.
- To draw your own graphics (including a .gif or .jpeg), make a subclass of JPanel and override the paintComponent() method.
- The paintComponent() method is called by the GUI system. YOU NEVER CALL IT YOURSELF. The argument to paintComponent() is a Graphics object that gives you a surface to draw on, which will end up on the screen. You cannot construct that object yourself.
- Typical methods to call on a Graphics object (the paintComponent parameter) are:
  `graphics.setColor(Color.blue);`
  `g.fillRect(20,50,100,120);`
- To draw a .jpg, construct an Image using:
  `Image image = new ImageIcon("catzilla.jpg").getImage();`
  and draw the imagine using:
  `g.drawImage(image,3,4,this);`
- The object referenced by the Graphics parameter to paintComponent() is actually an instance of the Graphics2D class. The Graphics 2D class has a variety of methods including:
  fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()
- To invoke the Graphics2D methods, you must cast the parameter from a Graphics object to a Graphics2D object:
  `Graphics2D g2d = (Graphics2D) g;`

We can get an event.
We can paint graphics.
But can we paint graphics *when* we get an event?

Let's hook up an event to a change in our drawing panel. We'll make the circle change colors each time you click the button. Here's how the program flows:

Start the app

1. The frame is built with the two widgets (your drawing panel and a button). A listener is created and registered with the button. Then the frame is displayed and it just waits for the user to click.



2. The user clicks the button and the button creates an event object and calls the listener's event handler.

3. The event handler calls repaint() on the frame. The system calls paintComponent() on the drawing panel.

4. Voila! A new color is painted because paintComponent() runs again, filling the circle with a random color.

GUI layouts: putting more than one widget on a frame

We cover GUI layouts in the *next* chapter, but we'll do a quickie lesson here to get you going. By default, a frame has five regions you can add to. You can add only *one* thing to each region of a frame, but don't panic! That one thing might be a panel that holds three other things including a panel that holds two more things and... you get the idea. In fact, we were 'cheating' when we added a button to the frame using:

```
frame.getContentPane().add(button);
```

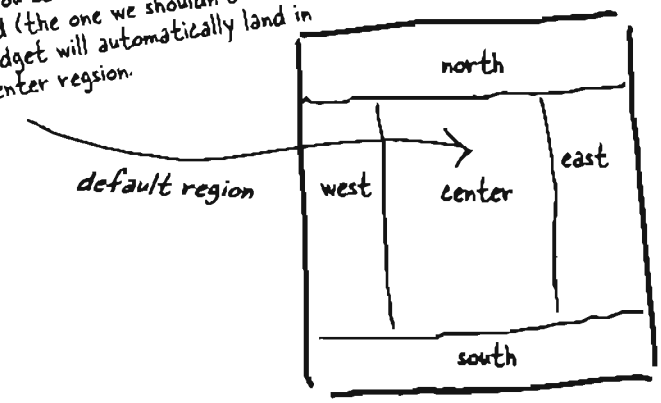
This isn't really the way you're supposed to do it (the one-arg add method).

This is the better (and usually mandatory) way to add to a frame's default content pane. Always specify WHERE (which region) you want the widget to go.

When you call the single-arg add method (the one we shouldn't use) the widget will automatically land in the center region.

```
frame.getContentPane().add(BorderLayout.CENTER, button);
```

we call the two-argument add method, that takes a region (using a constant) and the widget to add to that region.



Sharpen your pencil

Given the pictures on page 351, write the code that adds the button and the panel to the frame.

The circle changes color each time you click the button.

The custom drawing panel (instance of MyDrawPanel) is in the CENTER region of the frame.

Button is in the SOUTH region of the frame

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

public class SimpleGui3C implements ActionListener {

    JFrame frame;

    public static void main (String[] args) {
       SimpleGui3C gui = new SimpleGui3C();
       gui.go();
    }

    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton button = new JButton("Change colors");
       button.addActionListener(this);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(BorderLayout.SOUTH, button);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Add the listener (this) to the button.

Add the two widgets (button and drawing panel) to the two regions of the frame.

When the user clicks, tell the frame to repaint() itself. That means paintComponent() is called on every widget in the frame!

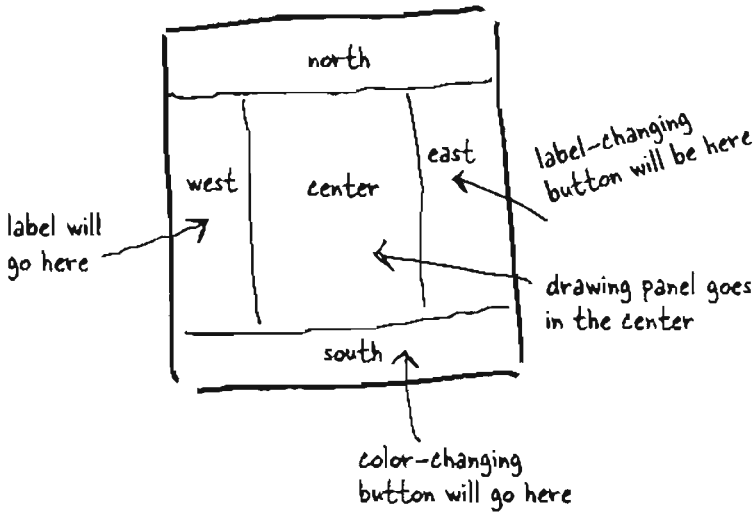
```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
     // Code to fill the oval with a random color
     // See page 347 for the code
   }

}
```

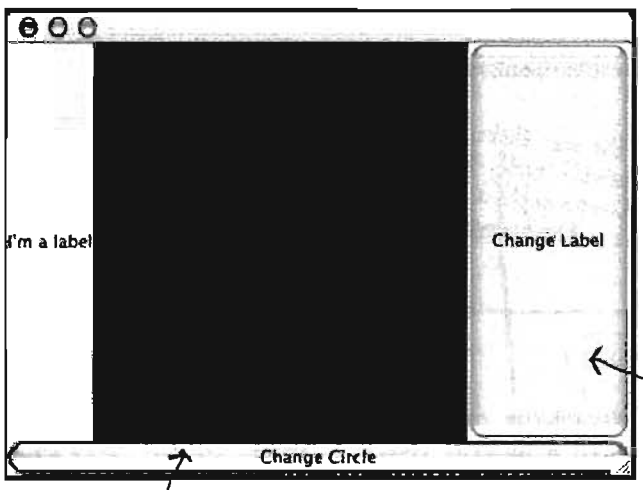
The drawing panel's paintComponent() method is called every time the user clicks.

Let's try it with TWO buttons

The south button will act as it does now, simply calling repaint on the frame. The second button (which we'll stick in the east region) will change the text on a label. (A label is just text on the screen.)

So now we need FOUR widgets





And we need to get TWO events

Uh-oh.

Is that even possible? How do you get *two* events when you have only *one* actionPerformed() method?

How do you get action events for two different buttons, when each button needs to do something different?

option one
Implement two actionPerformed() methods

```
class MyGui implements ActionListener {
    // lots of code here and then:

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }

    public void actionPerformed(ActionEvent event) {
        label.setText("That hurt!");
    }
}
```

But this is impossible!

Flaw: You can't! You can't implement the same method twice in a Java class. It won't compile. And even if you *could*, how would the event source know *which* of the two methods to call?

option two
Register the same listener with both buttons.

```
class MyGui implements ActionListener {
    // declare a bunch of instance variables here

    public void go() {
        // build gui
        colorButton = new JButton();
        labelButton = new JButton();
        colorButton.addActionListener(this);
        labelButton.addActionListener(this);
        // more gui code here ...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint();
        } else {
            label.setText("That hurt!");
        }
    }
}
```

Register the same listener with both buttons

Query the event object to find out which button actually fired it, and use that to decide what to do.

Flaw: this does work, but in most cases it's not very OO. One event handler doing many different things means that you have a single method doing many different things. If you need to change how *one* source is handled, you have to mess with *everybody's* event handler. Sometimes it *is* a good solution, but usually it hurts maintainability and extensibility.

How do you get action events for two different buttons, when each button needs to do something different?

option three
Create two separate ActionListener classes

```
class MyGui  {
   JFrame frame;
   JLabel label;
   void gui() {
      // code to instantiate the two listeners and register one
      // with the color button and the other with the label button
   }
} // close class
```

```
class ColorButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Won't work! This class doesn't have a reference to the 'frame' variable of the MyGui class

---

```
class LabelButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("That hurt!");
    }
}
```

Problem! This class has no reference to the variable 'label'

Flaw: these classes won't have access to the variables they need to act on, 'frame' and 'label'. You could fix it, but you'd have to give each of the listener classes a reference to the main GUI class, so that inside the actionPerformed() methods the listener could use the GUI class reference to access the variables of the GUI class. But that's breaking encapsulation, so we'd probably need to make getter methods for the gui widgets (getFrame(), getLabel(), etc.). And you'd probably need to add a constructor to the listener class so that you can pass the GUI reference to the listener at the time the listener is instantiated. And, well, it gets messier and more complicated.

There has got to be a better way!

Wouldn't it be wonderful if you could have two different listener classes, but the listener classes could access the instance variables of the main GUI class, almost as if the listener classes *belonged* to the other class. Then you'd have the best of both worlds. Yeah, that would be dreamy.

But it's just a fantasy...

Inner class to the rescue!

You *can* have one class nested inside another. It's easy. Just make sure that the definition for the inner class is *inside* the curly braces of the outer class.

Simple inner class:

```
class MyOuterClass {

    class MyInnerClass {
        void go() {
        }
    }

}
```

Inner class is fully enclosed by outer class

An inner class can use all the methods and variables of the outer class, even the private ones.

The inner class gets to use those variables and methods just as if the methods and variables were declared within the inner class.

An inner class gets a special pass to use the outer class's stuff. *Even the private stuff.* And the inner class can use those private variables and methods of the outer class as if the variables and members were defined in the inner class. That's what's so handy about inner classes—they have most of the benefits of a normal class, but with special access rights.

Inner class using an outer class variable

```
class MyOuterClass {

    private int x;

    class MyInnerClass {
        void go() {
            x = 42;
        }
    } // close inner class

} // close outer class
```

use 'x' as if it were a variable of the inner class!

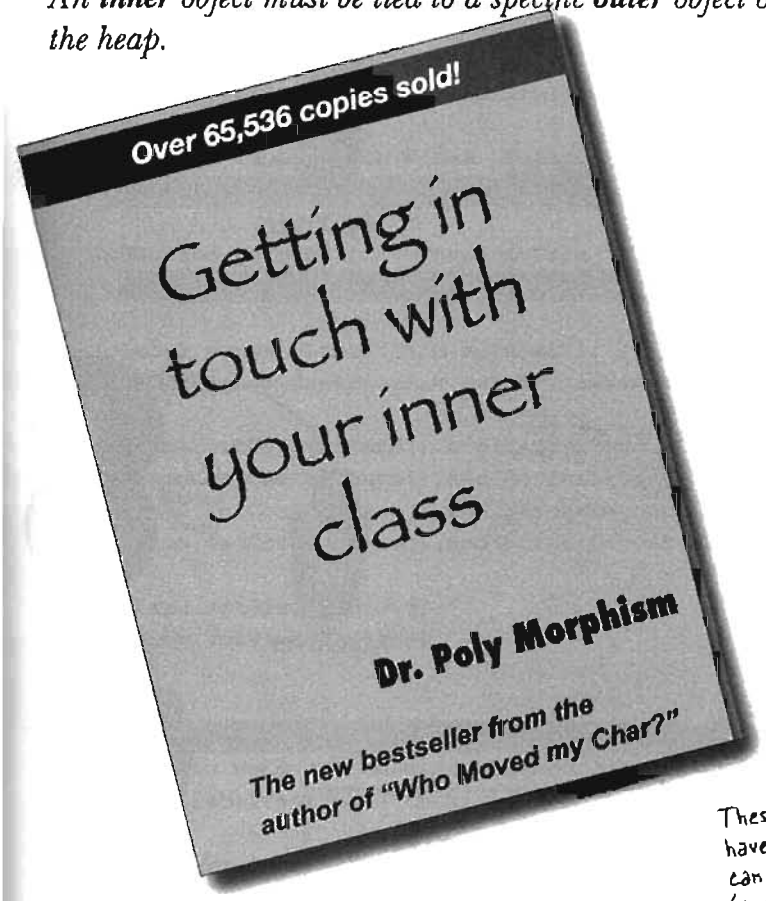
An inner class instance must be tied to an outer class instance*.

Remember, when we talk about an inner *class* accessing something in the outer class, we're really talking about an *instance* of the inner class accessing something in an *instance* of the outer class. But *which* instance?

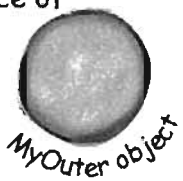
Can *any* arbitrary instance of the inner class access the methods and variables of *any* instance of the outer class? **No!**

*An **inner** object must be tied to a specific **outer** object on the heap.*

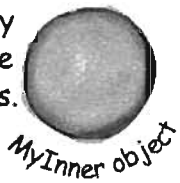
An inner object shares a special bond with an outer object.



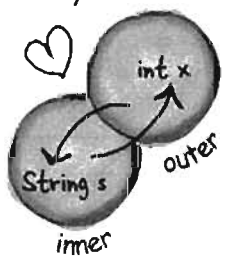
1. Make an instance of the outer class



2. Make an instance of the inner class, by using the instance of the outer class.



3. The outer and inner objects are now intimately linked.



These two objects on the heap have a special bond. The inner can use the outer's variables (and vice-versa).

*There's an exception to this, for a very special case—an inner class defined within a static method. But we're not going there, and you might go your entire Java life without ever encountering one of these.

How to make an instance of an inner class

If you instantiate an inner class from code *within* an outer class, the instance of the outer class is the one that the inner object will 'bond' with. For example, if code within a method instantiates the inner class, the inner object will bond to the instance whose method is running.

Code in an outer class can instantiate one of its own inner classes, in exactly the same way it instantiates any other class... **new MyInner()**

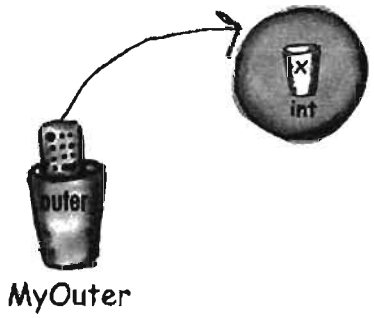
```
class MyOuter  {

   private int x;

   MyInner inner = new MyInner();

   public void doStuff() {
      inner.go();
   }

   class MyInner {
      void go() {
         x = 42;
      }
   } // close inner class

} // close outer class
```

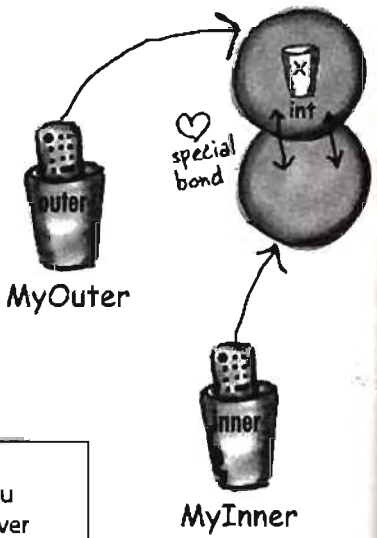
The outer class has a private instance variable 'x'

Make an instance of the inner class

call a method on the inner class

The method in the inner class uses the outer class instance variable 'x', as if 'x' belonged to the inner class.





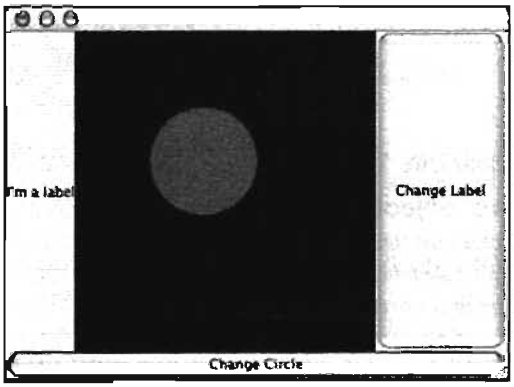
Side bar

You *can* instantiate an inner instance from code running *outside* the outer class, but you have to use a special syntax. Chances are you'll go through your entire Java life and never need to make an inner class from outside, but just in case you're interested...

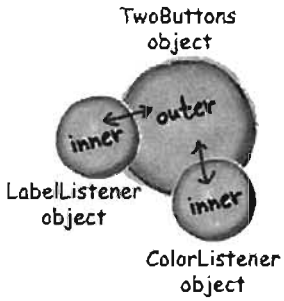
```
class Foo {
   public static void main (String[] args) {
      MyOuter outerObj = new MyOuter();
      MyOuter.MyInner innerObj = outerObj.new MyInner();
   }
}
```

Now we can get the two-button code working

```
public class TwoButtons {                    <- the main GUI class doesn't implement ActionListener now

    JFrame frame;
    JLabel label;

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

    public void go() {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton labelButton = new JButton("Change Label");
        labelButton.addActionListener(new LabelListener());

        JButton colorButton = new JButton("Change Circle");
        colorButton.addActionListener(new ColorListener());

        label = new JLabel("I'm a label");
        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
        frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
        frame.getContentPane().add(BorderLayout.EAST, labelButton);
        frame.getContentPane().add(BorderLayout.WEST, label);

        frame.setSize(300,300);
        frame.setVisible(true);
    }

    class LabelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            label.setText("Ouch!");
        }
    } // close inner class

    class ColorListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            frame.repaint();
        }
    } // close inner class
}
```



instead of passing (this) to the button's listener registration method, pass a new instance of the appropriate listener class.



Now we get to have TWO ActionListeners in a single class!

inner class knows about 'label'

the inner class gets to use the 'frame' instance variable, without having an explicit reference to the outer class object

Java Exposed

This weeks interview:
Instance of an Inner Class

HeadFirst: What makes inner classes important?

Inner object: Where do I start? We give you a chance to implement the same interface more than once in a class. Remember, you can't implement a method more than once in a normal Java class. But using *inner* classes, each inner class can implement the *same* interface, so you can have all these *different* implementations of the very same interface methods.

HeadFirst: Why would you ever *want* to implement the same method twice?

Inner object: Let's revisit GUI event handlers. Think about it... if you want *three* buttons to each have a different event behavior, then use *three* inner classes, all implementing ActionListener—which means each class gets to implement its own actionPerformed method.

HeadFirst: So are event handlers the only reason to use inner classes?

Inner object: Oh, gosh no. Event handlers are just an obvious example. Anytime you need a separate class, but still want that class to behave as if it were part of *another* class, an inner class is the best—and sometimes *only*—way to do it.

HeadFirst: I'm still confused here. If you want the inner class to *behave* like it belongs to the outer class, why have a separate class in the first place? Why wouldn't the inner class code just be *in* the outer class in the first place?

Inner object: I just *gave* you one scenario, where you need more than one implementation of an interface. But even when you're not using interfaces, you might need two different *classes* because those classes represent two different *things*. It's good OO.

HeadFirst: Whoa. Hold on here. I thought a big part of OO design is about reuse and maintenance. You know, the idea that if you have two separate classes, they can each be modified and used independently, as opposed to stuffing it all into one class yada yada yada. But with an *inner* class, you're still just working with one *real* class in the end, right? The enclosing class is the only one that's reusable and separate from everybody else. Inner classes aren't exactly reusable. In fact, I've heard them called "Reuseless—useless over and over again."

Inner object: Yes it's true that the inner class is not *as* reusable, in fact sometimes not reusable at all, because it's intimately tied to the instance variables and methods of the outer class. But it—

HeadFirst: —which only proves my point! If they're not reusable, why bother with a separate class? I mean, other than the interface issue, which sounds like a workaround to me.

Inner object: As I was saying, you need to think about IS-A and polymorphism.

HeadFirst: OK. And I'm thinking about them because...

Inner object: Because the outer and inner classes might need to pass *different* IS-A tests! Let's start with the polymorphic GUI listener example. What's the declared argument type for the button's listener registration method? In other words, if you go to the API and check, what kind of *thing* (class or interface type) do you have to pass to the addActionListener() method?

HeadFirst: You have to pass a listener. Something that implements a particular listener interface, in this case ActionListener. Yeah, we know all this. What's your point?

Inner object: My point is that polymorphically, you have a method that takes only one particular *type*. Something that passes the IS-A test for ActionListener. But—and here's the big thing—what if your class needs to be an IS-A of something that's a *class* type rather than an interface?

HeadFirst: Wouldn't you have your class just *extend* the class you need to be a part of? Isn't that the whole point of how subclassing works? If B is a subclass of A, then anywhere an A is expected a B can be used. The whole pass-a-Dog-where-an-Animal-is-the-declared-type thing.

Inner object: Yes! Bingo! So now what happens if you need to pass the IS-A test for two different classes? Classes that aren't in the same inheritance hierarchy?

HeadFirst: Oh, well you just... hmmm. I think I'm getting it. You can always *implement* more than one interface, but you can *extend* only *one* class. You can only be one kind of IS-A when it comes to *class* types.

Inner object: Well done! Yes, you can't be both a Dog and a Button. But if you're a Dog that needs to sometimes be a Button (in order to pass yourself to methods that take a Button), the Dog class (which extends Animal so it can't extend Button) can have an *inner* class that acts on the Dog's behalf as a Button, by extending Button, and thus wherever a Button is required the Dog can pass his inner Button instead of himself. In other words, instead of saying x.takeButton(this), the Dog object calls x.takeButton(new MyInnerButton()).

HeadFirst: Can I get a clear example?

Inner object: Remember the drawing panel we used, where we made our own subclass of JPanel? Right now, that class is a separate, non-inner, class. And that's fine, because the class doesn't need special access to the instance variables of the main GUI. But what if it did? What if we're doing an animation on that panel, and it's getting its coordinates from the main application (say, based on something the user does elsewhere in the GUI). In that case, if we make the drawing panel an inner class, the drawing panel class gets to be a subclass of JPanel, while the outer class is still free to be a subclass of something else.

HeadFirst: Yes I see! And the drawing panel isn't reusable enough to be a separate class anyway, since what it's actually painting is specific to this one GUI application.

Inner object: Yes! You've got it!

HeadFirst: Good. Then we can move on to the nature of the *relationship* between you and the outer instance.

Inner object: What is it with you people? Not enough sordid gossip in a serious topic like polymorphism?

HeadFirst: Hey, you have no idea how much the public is willing to pay for some good old tabloid dirt. So, someone creates you and becomes instantly bonded to the outer object, is that right?

Inner object: Yes that's right. And yes, some have compared it to an arranged marriage. We don't have a say in which object we're bonded to.

HeadFirst: Alright, I'll go with the marriage analogy. Can you get a *divorce* and remarry something *else*?

Inner object: No, it's for life.

HeadFirst: Whose life? Yours? The outer object? Both?

Inner object: Mine. I can't be tied to any other outer object. My only way out is garbage collection.

HeadFirst: What about the outer object? Can it be associated with any other inner objects?

Inner object: So now we have it. This is what you *really* wanted. Yes, yes. My so-called 'mate' can have as many inner objects as it wants.

HeadFirst: Is that like, serial monogamy? Or can it have them all at the same time?

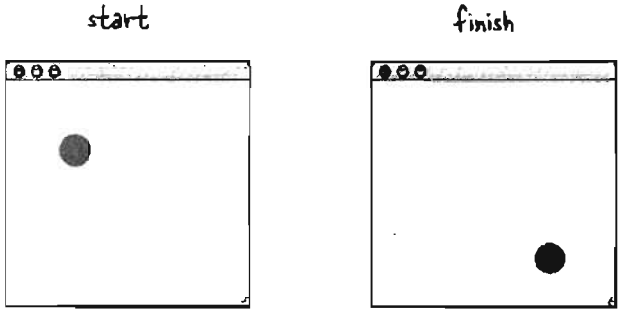
Inner object: All at the same time. There. Satisfied?

HeadFirst: Well, it does make sense. And let's not forget, it was *you* extolling the virtues of "multiple implementations of the same interface". So it makes sense that if the outer class has three buttons, it would need three different inner classes (and thus three different inner class objects) to handle the events. Thanks for everything. Here's a tissue.

Using an inner class for animation

We saw why inner classes are handy for event listeners, because you get to implement the same event-handling method more than once. But now we'll look at how useful an inner class is when used as a subclass of something the outer class doesn't extend. In other words, when the outer class and inner class are in different inheritance trees!

Our goal is to make a simple animation, where the circle moves across the screen from the upper left down to the lower right.



How simple animation works

1. Paint an object at a particular x and y coordinate

```
g.fillOval(20,50,100,100);
```

20 pixels from the left, 50 pixels from the top

2. Repaint the object at a different x and y coordinate

```
g.fillOval(25,55,100,100);
```

25 pixels from the left, 55 pixels from the top

(the object moved a little down and to the right)

3. Repeat the previous step with changing x and y values for as long as the animation is supposed to continue.

there are no Dumb Questions

Q: Why are we learning about animation here? I doubt if I'm going to be making games.

A: You might not be making games, but you might be creating simulations, where things change over time to show the results of a process. Or you might be building a visualization tool that, for example, updates a graphic to show how much memory a program is using, or to show you how much traffic is coming through your load-balancing server. Anything that needs to take a set of continuously-changing numbers and translate them into something useful for getting information out of the numbers.

Doesn't that all sound business-like? That's just the "official justification," of course. The real reason we're covering it here is just because it's a simple way to demonstrate another use of inner classes. (And because we just *like* animation, and our next Head First book is about J2EE and we *know* we can't get animation in that one.)

What we really want is something like...

```
class MyDrawPanel extends JPanel {
      public void paintComponent(Graphics g) {
         g.setColor(Color.orange);
         g.fillOval(x,y,100,100);
      }
}
```

each time paintComponent() is called, the oval gets painted at a different location

Sharpen your pencil

But where do we get the new x and y coordinates?

And who calls repaint()?

See if you can **design a simple solution** to get the ball to animate from the top left of the drawing panel down to the bottom right. Our answer is on the next page, so don't turn this page until you're done!

Big Huge Hint: make the drawing panel an inner class.

Another Hint: don't put any kind of repeat loop in the paintComponent() method.

Write your ideas (or the code) here:

The complete simple animation code

```
import javax.swing.*;
import java.awt.*;

public class SimpleAnimation {

    int x = 70;
    int y = 70;

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

    public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);

       for (int i = 0; i < 130; i++) {

          x++;
          y++;

          drawPanel.repaint();

          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }

    }// close go() method

    class MyDrawPanel extends JPanel {

       public void paintComponent(Graphics g) {
          g.setColor(Color.green);
          g.fillOval(x,y,40,40);

       }
    } // close inner class
} // close outer class
```

make two instance variables in the main GUI class, for the x and y coordinates of the circle.

Nothing new here. Make the widgets and put them in the frame.

This is where the action is!

repeat this 130 times

increment the x and y coordinates

tell the panel to repaint itself (so we can see the circle in the new location)

Slow it down a little (otherwise it will move so quickly you won't SEE it move). Don't worry, you weren't supposed to already know this. We'll get to threads in chapter 15.

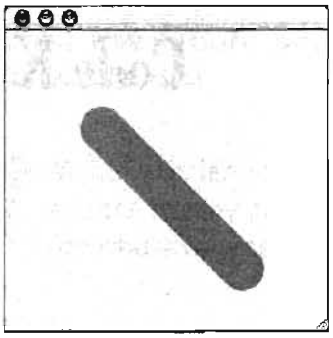
Now it's an inner class.

Use the continually-updated x and y coordinates of the outer class.

Uh-oh. It didn't move... it *smeared*.

What did we do wrong?

There's one little flaw in the paintComponent() method.



Not exactly the look we were going for.

We forgot to erase what was already there! So we got trails.

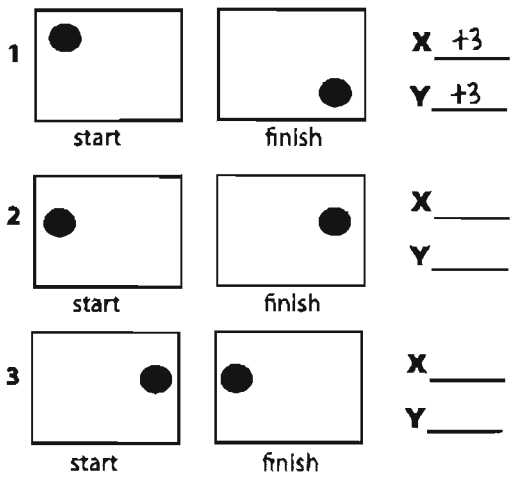
To fix it, all we have to do is fill in the entire panel with the background color, before painting the circle each time. The code below adds two lines at the start of the method: one to set the color to white (the background color of the drawing panel) and the other to fill the entire panel rectangle with that color. In English, the code below says, "Fill a rectangle starting at x and y of 0 (0 pixels from the left and 0 pixels from the top) and make it as wide and as high as the panel is currently.

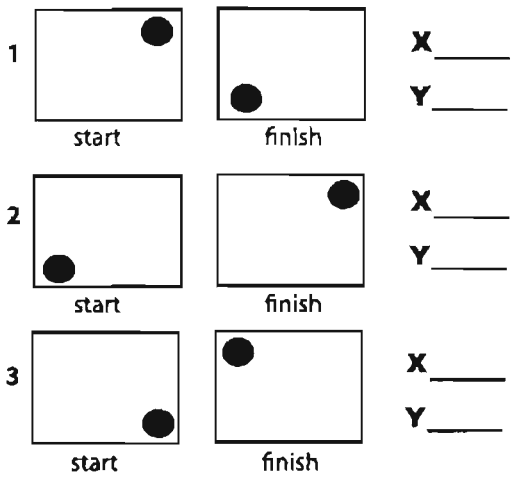
```
public void paintComponent(Graphics g) {
   g.setColor(Color.white);
   g.fillRect(0,0,this.getWidth(), this.getHeight());

   g.setColor(Color.green);
   g.fillOval(x,y,40,40);

}
```

getWidth() and getHeight() are methods inherited from JPanel.

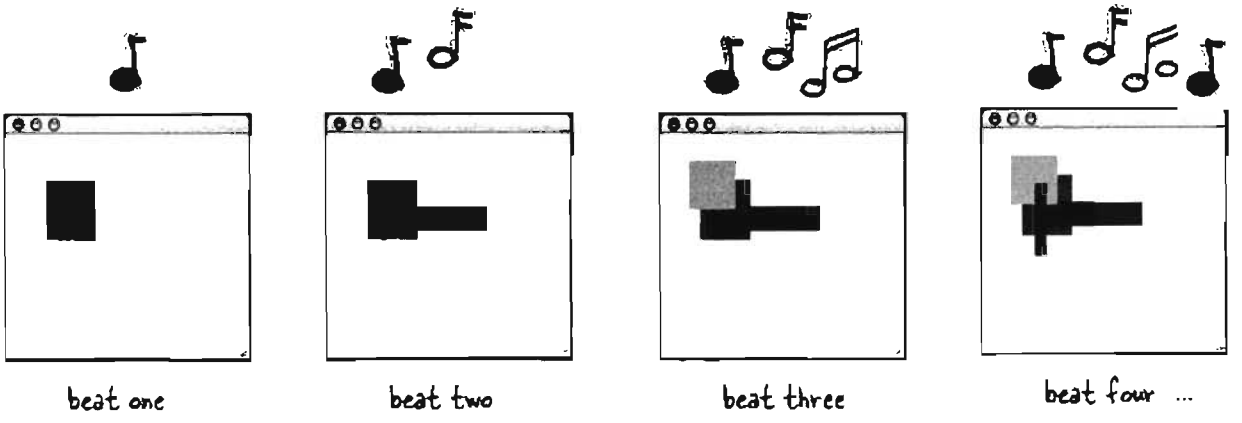
Sharpen your pencil (optional, just for fun)

What changes would you make to the x and y coordinates to produce the animations below? (assume the first one example moves in 3 pixel increments)





Code Kitchen



Let's make a music video. We'll use Java-generated random graphics that keep time with the music beats.

Along the way we'll register (and listen for) a new kind of non-GUI event, triggered by the music itself.

Remember, this part is all optional. But we think it's good for you. And you'll like it. And you can use it to impress people.

(Ok, sure, it might work only on people who are really easy to impress, but still...)

Listening for a non-GUI event

OK, maybe not a music video, but we *will* make a program that draws random graphics on the screen with the beat of the music. In a nutshell, the program listens for the beat of the music and draws a random graphic rectangle with each beat.

That brings up some new issues for us. So far, we've listened for only GUI events, but now we need to listen for a particular kind of MIDI event. Turns out, listening for a non-GUI event is just like listening for GUI events: you implement a listener interface, register the listener with an event source, then sit back and wait for the event source to call your event-handler method (the method defined in the listener interface).

The simplest way to listen for the beat of the music would be to register and listen for the actual MIDI events, so that whenever the sequencer gets the event, our code will get it too and can draw the graphic. But... there's a problem. A bug, actually, that won't let us listen for the MIDI events *we're* making (the ones for NOTE ON).

So we have to do a little work-around. There is another type of MIDI event we can listen for, called a ControllerEvent. Our solution is to register for ControllerEvents, and then make sure that for every NOTE ON event, there's a matching ControllerEvent fired at the same 'beat'. How do we make sure the ControllerEvent is fired at the same time? We add it to the track just like the other events! In other words, our music sequence goes like this:

BEAT 1 - NOTE ON, CONTROLLER EVENT

BEAT 2 - NOTE OFF

BEAT 3 - NOTE ON, CONTROLLER EVENT

BEAT 4 - NOTE OFF

and so on.

Before we dive into the full program, though, let's make it a little easier to make and add MIDI messages/events since in *this* program, we're gonna make a lot of them.

What the music art program needs to do:

- Make a series of MIDI messages/events to play random notes on a piano (or whatever instrument you choose)
- Register a listener for the events
- Start the sequencer playing
- Each time the listener's event handler method is called, draw a random rectangle on the drawing panel, and call repaint.

We'll build it in three iterations:

- Version One: Code that simplifies making and adding MIDI events, since we'll be making a lot of them.
- Version Two: Register and listen for the events, but without graphics. Prints a message at the command-line with each beat.
- Version Three: The real deal. Adds graphics to version two.

utility method for events

An easier way to make messages / events

Right now, making and adding messages and events to a track is tedious. For each message, we have to make the message instance (in this case, ShortMessage), call setMessage(), make a MidiEvent for the message, and add the event to the track. In last chapter's code, we went through each step for every message. That means eight lines of code just to make a note play and then stop playing! Four lines to add a NOTE ON event, and four lines to add a NOTE OFF event.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

Things that have to happen for each event:

1. Make a message instance

```
ShortMessage first = new ShortMessage();
```

2. Call setMessage() with the instructions

```
first.setMessage(192, 1, instrument, 0)
```

3. Make a MidiEvent instance for the message

```
MidiEvent noteOn = new MidiEvent(first, 1);
```

4. Add the event to the track

```
track.add(noteOn);
```

Let's build a static utility method that makes a message and returns a MidiEvent

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
       ShortMessage a = new ShortMessage();
       a.setMessage(comd, chan, one, two);
       event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
}
```

the four arguments for the message

The event 'tick' for WHEN this message should happen

whoo! A method with five parameters.

make the message and the event, using the method parameters

return the event (a MidiEvent all loaded up with the message)

Example: how to use the new static makeEvent() method

There's no event handling or graphics here, just a sequence of 15 notes that go up the scale. The point of this code is simply to learn how to use our new makeEvent() method. The code for the next two versions is much smaller and simpler thanks to this method.

```
import javax.sound.midi.*;
public class MiniMusicPlayer1 {

   public static void main(String[] args) {

      try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();


         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4) {


         track.add(makeEvent(144,1,i,100,i));
         track.add(makeEvent(128,1,i,100,i + 2));

       } // end loop

          sequencer.setSequence(seq);
          sequencer.setTempoInBPM(220);
          sequencer.start();
      } catch (Exception ex) {ex.printStackTrace();}
  } // close main


   public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
          MidiEvent event = null;
          try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

          }catch(Exception e) { }
         return event;
       }
} // close class
```

don't forget the import

make (and open) a sequencer

make a sequence and a track

make a bunch of events to make the notes keep going up (from piano note 5 to piano note 61)

call our new makeEvent() method to make the message and event, then add the result (the MidiEvent returned from makeEvent()) to the track. These are NOTE ON (144) and NOTE OFF (128) pairs

start it running

Version Two: registering and getting ControllerEvents

We need to listen for ControllerEvents, so we implement the listener interface

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

    public static void main(String[] args) {
        MiniMusicPlayer2 mini = new MiniMusicPlayer2();
        mini.go();
    }
  public void go() {

    try {
        Sequencer sequencer = MidiSystem.getSequencer();
        sequencer.open();
```

Register for events with the sequencer. The event registration method takes the listener AND an int array representing the list of ControllerEvents you want. We want only one event, #127.

```
        int[] eventsIWant = {127};
        sequencer.addControllerEventListener(this, eventsIWant);

        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track track = seq.createTrack();

        for (int i = 5; i < 60; i+= 4) {
            track.add(makeEvent(144,1,i,100,i));

            track.add(makeEvent(176,1,127,0,i));

            track.add(makeEvent(128,1,i,100,i + 2));
        } // end loop

        sequencer.setSequence(seq);
        sequencer.setTempoInBPM(220);
        sequencer.start();
    } catch (Exception ex) {ex.printStackTrace();}
  } // close
```

Here's how we pick up the beat -- we insert our OWN ControllerEvent (176 says the event type is ControllerEvent) with an argument for event number #127. This event will do NOTHING! We put it in JUST so that we can get an event each time a note is played. In other words, its sole purpose is so that something will fire that WE can listen for (we can't listen for NOTE ON/OFF events). Note that we're making this event happen at the SAME tick as the NOTE ON. So when the NOTE ON event happens, we'll know about it because OUR event will fire at the same time.

```
  public void controlChange(ShortMessage event) {
        System.out.println("la");
  }
```

The event handler method (from the ControllerEvent listener interface). Each time we get the event, we'll print "la" to the command-line

```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
          ShortMessage a = new ShortMessage();
          a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

        }catch(Exception e) { }
        return event;
      }
} // close class
```

Code that's different from the previous version is highlighted in gray. (and we're not running it all within main() this time)

Version Three: drawing graphics in time with the music

This final version builds on version two by adding the GUI parts. We build a frame, add a drawing panel to it, and each time we get an event, we draw a new rectangle and repaint the screen. The only other change from version two is that the notes play randomly as opposed to simply moving up the scale.

The most important change to the code (besides building a simple GUI) is that we make the drawing panel implement the ControllerEventListener rather than the program itself. So when the drawing panel (an inner class) gets the event, it knows how to take care of itself by drawing the rectangle.

Complete code for this version is on the next page.

The drawing panel inner class:

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public void paintComponent(Graphics g) {
      if (msg) {
        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);
        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);
        g.fillRect(x,y,ht, width);
        msg = false;

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

The drawing panel is a listener

We set a flag to false, and we'll set it to true only when we get an event.

We got an event, so we set the flag to true and call repaint()

We have to use a flag because OTHER things might trigger a repaint(), and we want to paint ONLY when there's a ControllerEvent

The rest is code to generate a random color and paint a semi-random rectangle.

MiniMusicPlayer3 code

This is the complete code listing for Version Three. It builds directoy on Version Two. Try to annotate it yourself, without looking at the previous pages.

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

     static JFrame f = new JFrame("My First Music Video");
     static MyDrawPanel ml;

     public static void main(String[] args) {
           MiniMusicPlayer3 mini = new MiniMusicPlayer3();
           mini.go();
     } // close method


     public  void setUpGui() {
        ml = new MyDrawPanel();
        f.setContentPane(ml);
        f.setBounds(30,30, 300,300);
        f.setVisible(true);
    } // close method

    public void go() {
       setUpGui();

       try {

          Sequencer sequencer = MidiSystem.getSequencer();
          sequencer.open();
          sequencer.addControllerEventListener(ml, new int[] {127});
          Sequence seq = new Sequence(Sequence.PPQ, 4);
          Track track = seq.createTrack();

          int r = 0;
          for (int i = 0; i < 60; i+= 4) {

             r = (int) ((Math.random() * 50) + 1);
             track.add(makeEvent(144,1,r,100,i));
             track.add(makeEvent(176,1,127,0,i));
             track.add(makeEvent(128,1,r,100,i + 2));
          } // end loop

          sequencer.setSequence(seq);
          sequencer.start();
          sequencer.setTempoInBPM(120);
       } catch (Exception ex) {ex.printStackTrace();}
    } // close method
```

Who am I?

A bunch of Java hot-shots, in full costume, are playing the party game "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one guy, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

Any of the charming personalities from this chapter just might show up!

I got the whole GUI, in my hands. _____

Every event type has one of these. _____

The listener's key method. _____

This method gives JFrame its size. _____

You add code to this method but never call it. _____

When the user actually does something, it's an _____ . _____

Most of these are event sources. _____

I carry data back to the listener. _____

An addXxxListener( ) method says an object is an _____ . _____

How a listener signs up. _____

The method where all the graphics code goes. _____

I'm typically bound to an instance. _____

The 'g' in (Graphics g), is really of class. _____

The method that gets paintComponent( ) rolling. _____

The package where most of the Swingers reside. _____

BE the compiler

The Java file on this page represents a complete source file. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it, and if it does compile, what would it do?

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

class InnerButton {

  JFrame frame;
  JButton b;

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

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                  JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                  BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

Exercise Solutions

Who am I?

I got the whole GUI, in my hands.	JFrame
Every event type has one of these.	listener interface
The listener's key method.	actionPerformed( )
This method gives JFrame its size.	setSize( )
You add code to this method but never call it.	paintComponent( )
When the user actually does something, it's an ____	event
Most of these are event sources.	swing components
I carry data back to the listener.	event object
An addXxxListener( ) method says an object is an ___	event source
How a listener signs up.	addActionListener( )
The method where all the graphics code goes.	paintComponent( )
I'm typically bound to an instance.	inner class
The 'g' in (Graphics g), is really of this class.	Graphics2d
The method that gets paintComponent( ) rolling.	repaint( )
The package where most of the Swingers reside.	javax.swing

BE the compiler

Once this code is fixed, it will create a GUI with a button that toggles between A and B when you click it.

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

class InnerButton {

  JFrame frame;
  JButton b;

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

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener( new BListener() );

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

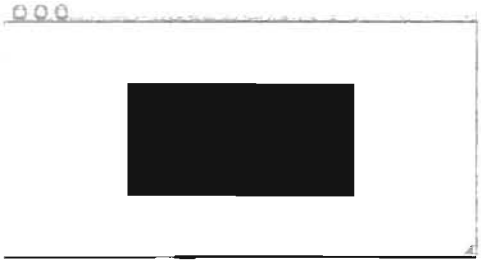
The addActionListener( ) method takes a class that implements the ActionListener interface

ActionListener is an interface, interfaces are implemented, not extended

Pool Puzzle

```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
    }
    public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(
                     JFrame.EXIT_ON_CLOSE);
       MyDrawP drawP = new MyDrawP();
       frame.getContentPane().add(drawP);
       frame.setSize(500,270);
       frame.setVisible(true);
       for (int i = 0; i < 124; i++,x++,y++ ) {
          x++;
          drawP.repaint();
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
    }
    class MyDrawP extends JPanel {
       public void paintComponent(Graphics g  ) {
          g.setColor(Color.white);
          g.fillRect(0,0,500,250);
          g.setColor(Color.blue);
          g.fillRect(x,y,500-x*2,250-y*2);
       }
    }
}
```

The Amazing, Shrinking, Blue Rectangle.



Work on Your Swing

Swing is easy. Unless you actually *care* where things end up on the screen. Swing code *looks* easy, but then you compile it, run it, look at it and think, "hey, *that's* not supposed to go *there*." The thing that makes it *easy* to *code* is the thing that makes it *hard* to *control*—the **Layout Manager**. Layout Manager objects control the size and location of the widgets in a Java GUI. They do a ton of work on your behalf, but you won't always like the results. You want two buttons to be the same size, but they aren't. You want the text field to be three inches long, but it's nine. Or one. And *under* the label instead of *next* to it. But with a little work, you can get layout managers to submit to your will. In this chapter, we'll work on our Swing and in addition to layout managers, we'll learn more about widgets. We'll make them, display them (where *we* choose), and use them in a program. It's not looking too good for Suzy.

Swing components

Component is the more correct term for what we've been calling a *widget*. The *things* you put in a GUI. *The things a user sees and interacts with.* Text fields, buttons, scrollable lists, radio buttons, etc. are all components. In fact, they all extend `javax.swing.JComponent`.

A widget is technically a Swing Component. Almost every thing you can stick in a GUI extends from javax.swing.JComponent.

Components can be nested

In Swing, virtually *all* components are capable of holding other components. In other words, *you can stick just about anything into anything else.* But most of the time, you'll add *user interactive* components such as buttons and lists into *background* components such as frames and panels. Although it's *possible* to put, say, a panel inside a button, that's pretty weird, and won't win you any usability awards.

With the exception of JFrame, though, the distinction between *interactive* components and *background* components is artificial. A JPanel, for example, is usually used as a background for grouping other components, but even a JPanel can be interactive. Just as with other components, you can register for the JPanel's events including mouse clicks and keystrokes.

Four steps to making a GUI (review)

- Make a window (a JFrame)

```
JFrame frame = new JFrame();
```

- Make a component (button, text field, etc.)

```
JButton button = new JButton("click me");
```

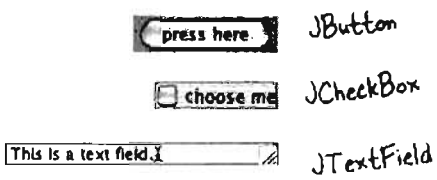
- Add the component to the frame

```
frame.getContentPane().add(BorderLayout.EAST, button);
```

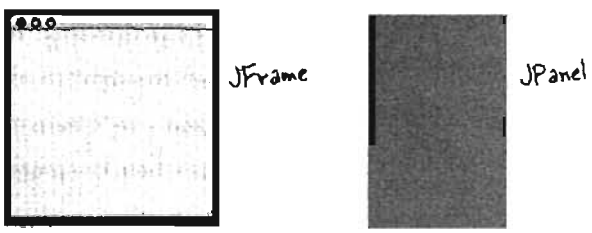
- Display it (give it a size and make it visible)

```
frame.setSize(300,300);
frame.setVisible(true);
```

Put interactive components:

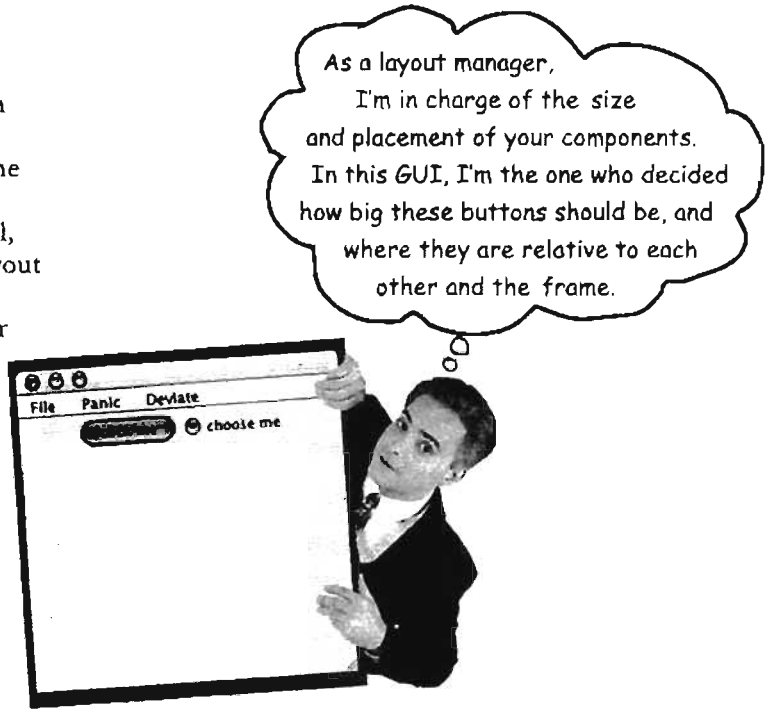


Into background components:



Layout Managers

A layout manager is a Java object associated with a particular component, almost always a *background* component. The layout manager controls the components contained *within* the component the layout manager is associated with. In other words, if a frame holds a panel, and the panel holds a button, the panel's layout manager controls the size and placement of the button, while the frame's layout manager controls the size and placement of the panel. The button, on the other hand, doesn't need a layout manager because the button isn't holding other components.



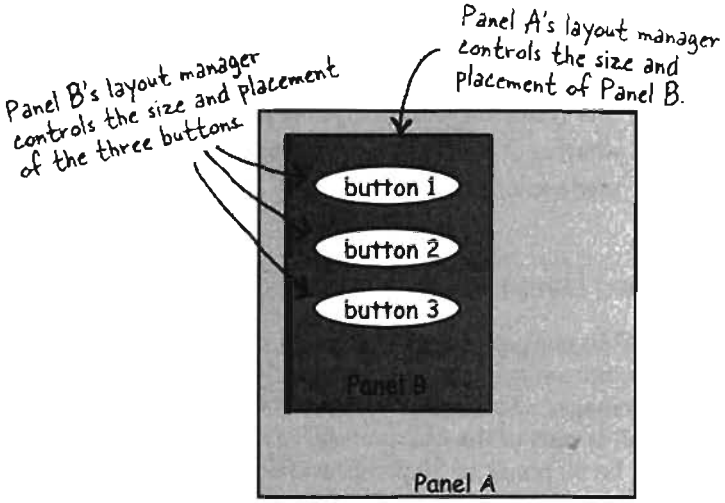
If a panel holds five things, even if those five things each have their own layout managers, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things, then those *other* things are placed according to the layout manager of the thing holding them.

When we say *hold* we really mean *add* as in, a panel *holds* a button because the button was *added* to the panel using something like:

```
myPanel.add(button);
```

Layout managers come in several flavors, and each background component can have its own layout manager. Layout managers have their own policies to follow when building a layout. For example, one layout manager might insist that all components in a panel must be the same size, arranged in a grid, while another layout manager might let each component choose its own size, but stack them vertically. Here's an example of nested layouts:

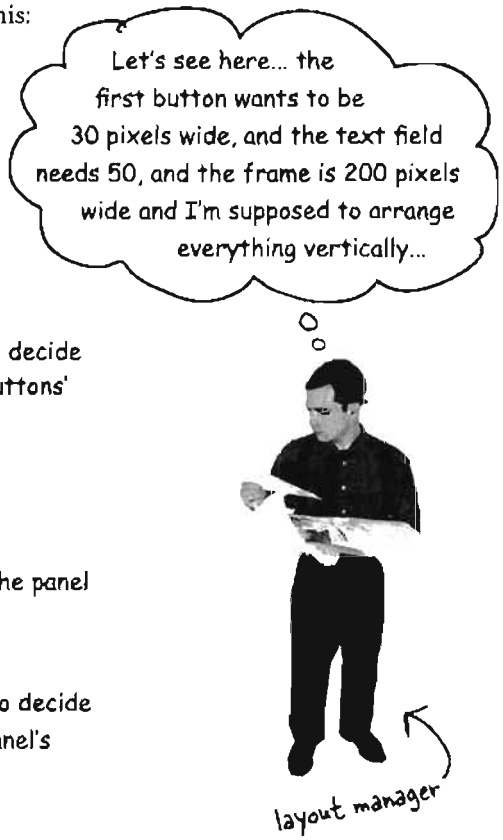
```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```



Panel A's layout manager has NOTHING to say about the three buttons. The hierarchy of control is only one level—Panel A's layout manager controls only the things added directly to Panel A, and does not control anything nested within those added components.

How does the layout manager decide?

Different layout managers have different policies for arranging components (like, arrange in a grid, make them all the same size, stack them vertically, etc.) but the components being layed out do get at least *some* small say in the matter. In general, the process of laying out a background component looks something like this:



A layout scenario:

1. Make a panel and add three buttons to it.
2. The panel's layout manager asks each button how big that button prefers to be.
3. The panel's layout manager uses its layout policies to decide whether it should respect all, part, or none of the buttons' preferences.
4. Add the panel to a frame.
5. The frame's layout manager asks the panel how big the panel prefers to be.
6. The frame's layout manager uses its layout policies to decide whether it should respect all, part, or none of the panel's preferences.

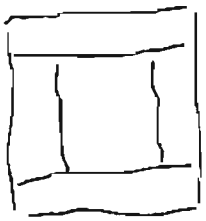
Different layout managers have different policies

Some layout managers respect the size the component wants to be. If the button wants to be 30 pixels by 50 pixels, that's what the layout manager allocates for that button. Other layout managers respect only part of the component's preferred size. If the button wants to be 30 pixels by 50 pixels, it'll be 30 pixels by however wide the button's background *panel* is. Still other layout managers respect the preference of only the *largest* of the components being layed out, and the rest of the components in that panel are all made that same size. In some cases, the work of the layout manager can get very complex, but most of the time you can figure out what the layout manager will probably do, once you get to know that layout manager's policies.

The Big Three layout managers: border, flow, and box.

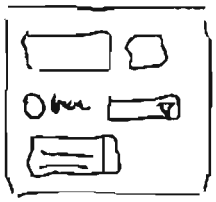
BorderLayout

A BorderLayout manager divides a background component into five regions. You can add only one component per region to a background controlled by a BorderLayout manager. Components laid out by this manager usually don't get to have their preferred size. **BorderLayout is the default layout manager for a frame!**



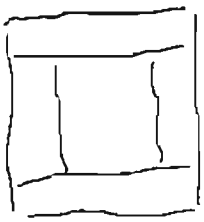
FlowLayout

A FlowLayout manager acts kind of like a word processor, except with components instead of words. Each component is the size it wants to be, and they're laid out left to right in the order that they're added, with "word-wrap" turned on. So when a component won't fit horizontally, it drops to the next "line" in the layout. **FlowLayout is the default layout manager for a panel!**



BoxLayout

A BoxLayout manager is like FlowLayout in that each component gets to have its own size, and the components are placed in the order in which they're added. But, unlike FlowLayout, a BoxLayout manager can stack the components vertically (or horizontally, but usually we're just concerned with vertically). It's like a FlowLayout but instead of having automatic 'component wrapping', you can insert a sort of 'component return key' and **force** the components to start a new line.



BorderLayout cares about five regions: east, west, north, south, and center

Let's add a button to the east region:

```
import javax.swing.*;
import java.awt.*;          <-- BorderLayout is in java.awt package

public class Button1 {

    public static void main (String[] args) {
       Button1 gui = new Button1();
       gui.go();
    }

    public void go() {
       JFrame frame = new JFrame();
       JButton button = new JButton("click me");
       frame.getContentPane().add(BorderLayout.EAST, button);
       frame.setSize(200,200);
       frame.setVisible(true);
    }
}
```

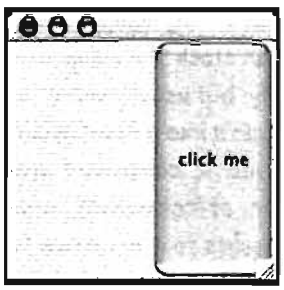
specify the region

Brain Barbell

How did the BorderLayout manager come up with this size for the button?

What are the factors the layout manager has to consider?

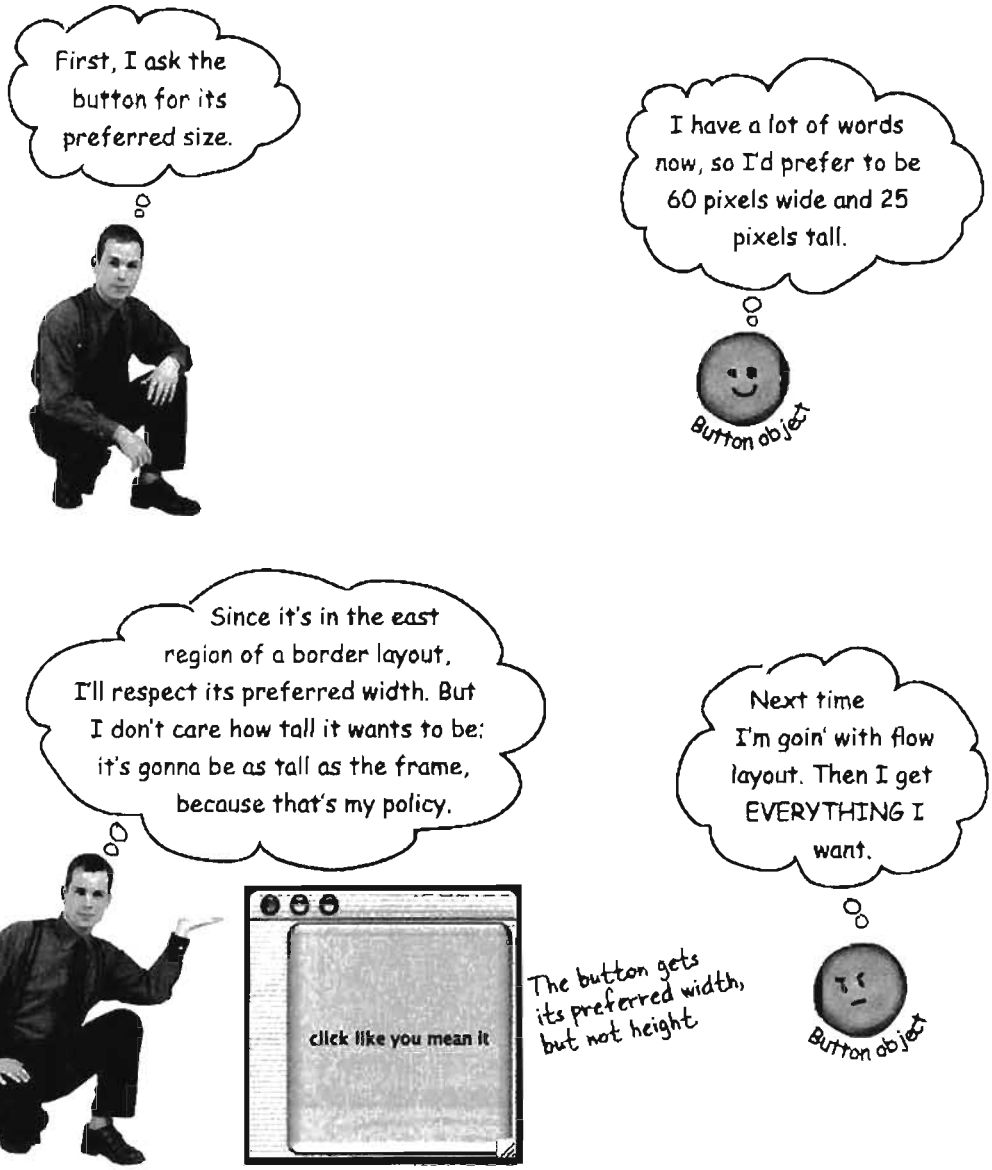
Why isn't it wider or taller?



Watch what happens when we give the button more characters...

We changed only the text on the button

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("click like you mean it");
    frame.getContentPane().add(BorderLayout.EAST, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```



Let's try a button in the NORTH region

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("There is no spoon...");
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

The button is as tall as it wants to be, but as wide as the frame.

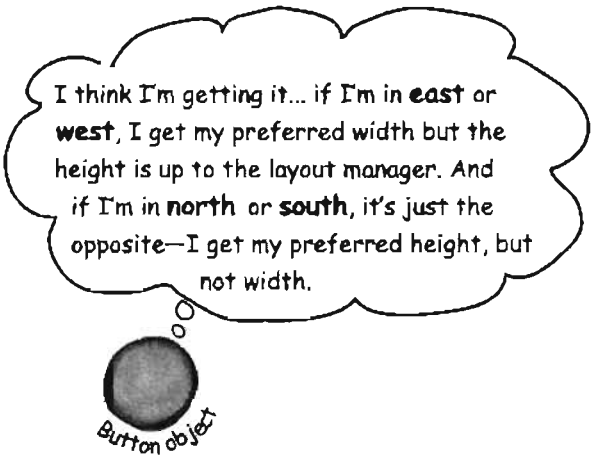
Now let's make the button ask to be taller

How do we do that? The button is already as wide as it can ever be—as wide as the frame. But we can try to make it taller by giving it a bigger font.

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("Click This!");
    Font bigFont = new Font("serif", Font.BOLD, 28);
    button.setFont(bigFont);
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

A bigger font will force the frame to allocate more space for the button's height

The width stays the same, but now the button is taller. The north region stretched to accomodate the button's new preferred height



But what happens in the center region?

The center region gets whatever's left!

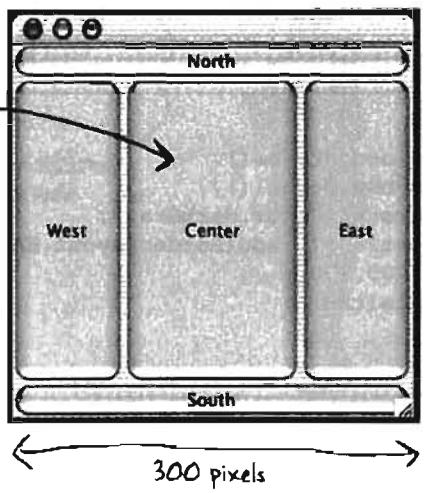
(except in one special case we'll look at later)

```
public void go() {
    JFrame frame = new JFrame();

    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton center = new JButton("Center");

    frame.getContentPane().add(BorderLayout.EAST, east);
    frame.getContentPane().add(BorderLayout.WEST, west);
    frame.getContentPane().add(BorderLayout.NORTH, north);
    frame.getContentPane().add(BorderLayout.SOUTH, south);
    frame.getContentPane().add(BorderLayout.CENTER, center);

    frame.setSize(300,300);
    frame.setVisible(true);
}
```

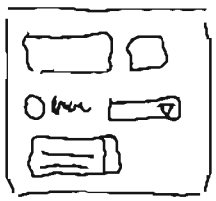
Components in the center get whatever space is left over, based on the frame dimensions (300 x 300 in this code).

Components in the east and west get their preferred width.

Components in the north and south get their preferred height



When you put something in the north or south, it goes all the way across the frame, so the things in the east and west won't be as tall as they would be if the north and south regions were empty.



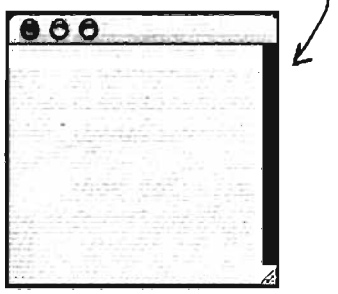
FlowLayout cares about the flow of the components:

left to right, top to bottom, in the order they were added.

Let's add a panel to the east region:

A JPanel's layout manager is FlowLayout, by default. When we add a panel to a frame, the size and placement of the panel is still under the BorderLayout manager's control. But anything *inside* the *panel* (in other words, components added to the panel by calling `panel.add(aComponent)`) are under the panel's FlowLayout manager's control. We'll start by putting an empty panel in the frame's east region, and on the next pages we'll add things to the panel.

The panel doesn't have anything in it, so it doesn't ask for much width in the east region.



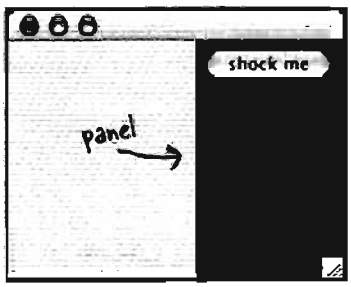
```
import javax.swing.*;
import java.awt.*;

public class Panel1 {

    public static void main (String[] args) {
        Panel1 gui = new Panel1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        panel.setBackground(Color.darkGray);
        frame.getContentPane().add(BorderLayout.EAST, panel);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

Make the panel gray so we can see where it is on the frame.

Let's add a button to the panel

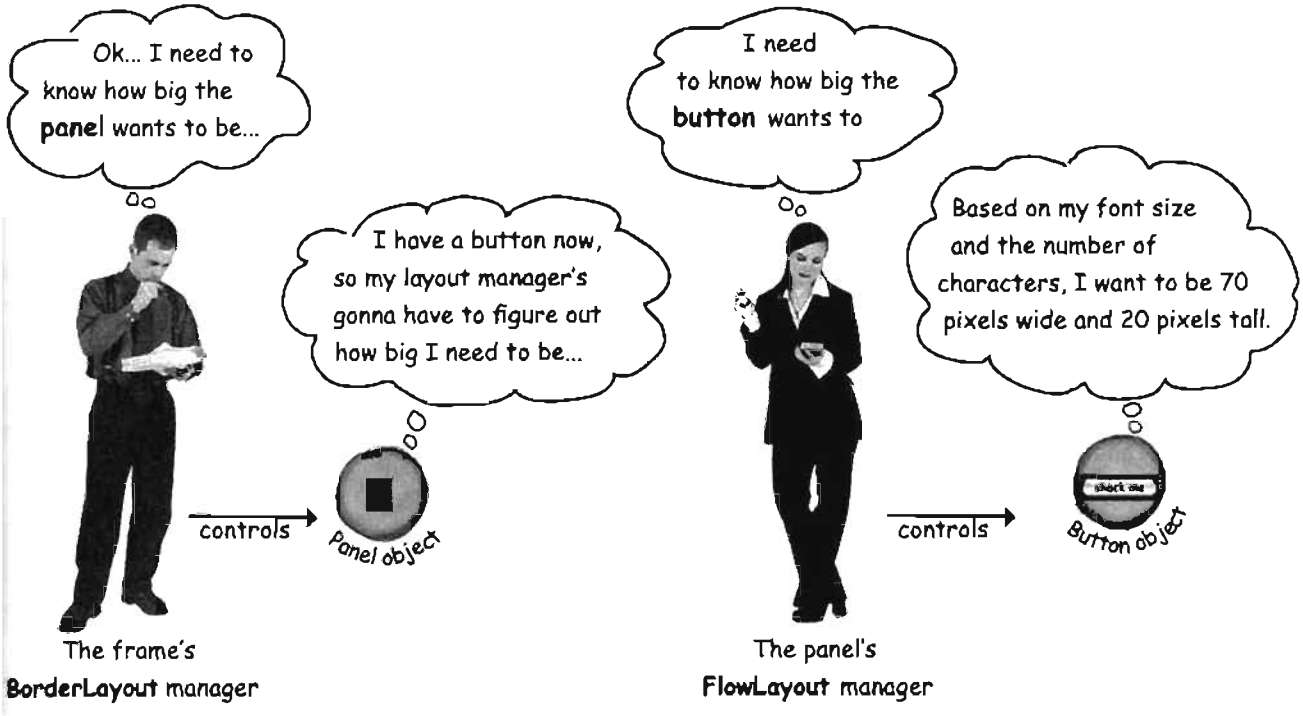
```
public void go() {
     JFrame frame = new JFrame();
     JPanel panel = new JPanel();
     panel.setBackground(Color.darkGray);

     JButton button = new JButton("shock me");

     panel.add(button);
     frame.getContentPane().add(BorderLayout.EAST, panel);

     frame.setSize(250,200);
     frame.setVisible(true);
}
```

Add the button to the panel and add the panel to the frame. The panel's layout manager (flow) controls the button, and the frame's layout manager (border) controls the panel.



The panel expanded!

And the button got its preferred size in both dimensions, because the panel uses flow layout, and the button is part of the panel (not the frame).



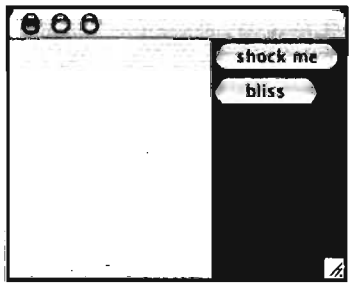
What happens if we add TWO buttons to the panel?

```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");

    panel.add(button);
    panel.add(buttonTwo);

    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

make TWO buttons

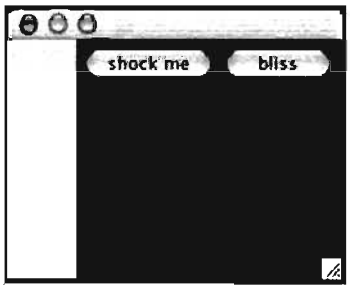
add BOTH to the panel

what we *wanted*:



We want the buttons stacked on top of each other

what we *got*:



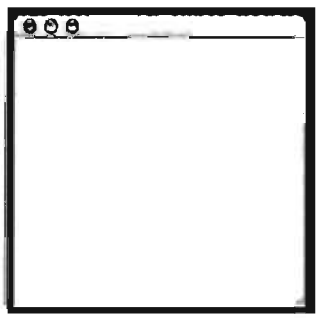
The panel expanded to fit both buttons side by side.

notice that the 'bliss' button is smaller than the 'shock me' button... that's how flow layout works. The button gets just what it needs (and no more).

Sharpen your pencil

If the code above were modified to the code below, what would the GUI look like?

```
JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("bliss");
JButton buttonThree = new JButton("huh?");
panel.add(button);
panel.add(buttonTwo);
panel.add(buttonThree);
```



Draw what you think the GUI would look like if you ran the code to the left.

(Then try it!)

BoxLayout to the rescue!

It keeps components stacked, even if there's room to put them side by side.

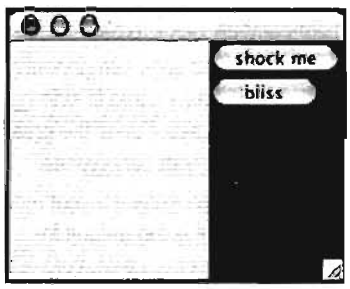
Unlike FlowLayout, BoxLayout can force a 'new line' to make the components wrap to the next line, even if there's room for them to fit horizontally.

But now you'll have to change the panel's layout manager from the default FlowLayout to BoxLayout.

```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");
    panel.add(button);
    panel.add(buttonTwo);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

Change the layout manager to be a new instance of BoxLayout

The BoxLayout constructor needs to know the component its laying out (i.e., the panel) and which axis to use (we use Y_AXIS for a vertical stack).



Notice how the panel is narrower again, because it doesn't need to fit both buttons horizontally. So the panel told the frame it needed enough room for only the largest button, 'shock me'.

there are no Dumb Questions

Q: How come you can't add directly to a frame the way you can to a panel?

A: A JFrame is special because it's where the rubber meets the road in making something appear on the screen. While all your Swing components are pure Java, a JFrame has to connect to the underlying OS in order to access the display. Think of the content pane as a 100% pure Java layer that sits on *top* of the JFrame. Or think of it as though JFrame is the window frame and the content pane is the... glass. You know, the window *pane*. And you can even *swap* the content pane with your own JPanel, to make your JPanel the frame's content pane, using,

```
myFrame.setContentPane(myPanel);
```

Q: Can I change the layout manager of the frame? What if I want the frame to use flow instead of border?

A: The easiest way to do this is to make a panel, build the GUI the way you want in the panel, and then make that panel the frame's content pane using the code in the previous answer (rather than using the default content pane).

Q: What if I want a different preferred size? Is there a setSize() method for components?

A: Yes, there is a setSize(), but the layout managers will ignore it. There's a distinction between the *preferred size* of the component and the size *you* want it to be. The preferred size is based on the size the component actually *needs* (the component makes that decision for itself). The layout manager calls the component's getPreferredSize() method, and *that* method doesn't *care* if you've previously called setSize() on the component.

Q: Can't I just put things where I want them? Can I turn the layout managers off?

A: Yep. On a component by component basis, you can call `setLayout(null)` and then it's up to you to hard-code the exact screen locations and dimensions. In the long run, though, it's almost always easier to use layout managers.

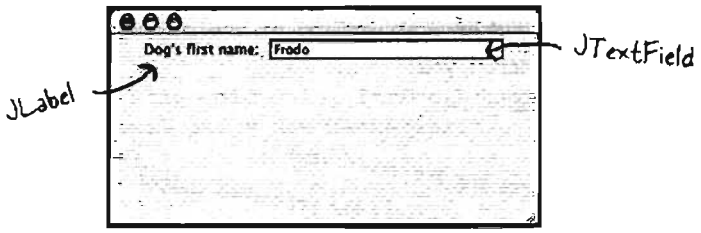
BULLET POINTS

- Layout managers control the size and location of components nested within other components.
- When you add a component to another component (sometimes referred to as a *background* component, but that's not a technical distinction), the added component is controlled by the layout manager of the *background* component.
- A layout manager asks components for their preferred size, before making a decision about the layout. Depending on the layout manager's policies, it might respect all, some, or none of the component's wishes.
- The BorderLayout manager lets you add a component to one of five regions. You must specify the region when you add the component, using the following syntax:
  `add(BorderLayout.EAST, panel);`
- With BorderLayout, components in the north and south get their preferred height, but not width. Components in the east and west get their preferred width, but not height. The component in the center gets whatever is left over (unless you use `pack()`).
- The pack() method is like shrink-wrap for the components; it uses the full preferred size of the center component, then determines the size of the frame using the center as a starting point, building the rest based on what's in the other regions.
- FlowLayout places components left to right, top to bottom, in the order they were added, wrapping to a new line of components only when the components won't fit horizontally.
- FlowLayout gives components their preferred size in both dimensions.
- BoxLayout lets you align components stacked vertically, even if they could fit side-by-side. Like FlowLayout, BoxLayout uses the preferred size of the component in both dimensions.
- BorderLayout is the default layout manager for a frame; FlowLayout is the default for a panel.
- If you want a panel to use something other than flow, you have to call `setLayout()` on the panel.

Playing with Swing components

You've learned the basics of layout managers, so now let's try out a few of the most common components: a text field, scrolling text area, checkbox, and list. We won't show you the whole darn API for each of these, just a few highlights to get you started.

JTextField



Constructors

```
JTextField field = new JTextField(20);
```

20 means 20 columns, not 20 pixels. This defines the preferred width of the text field.

```
JTextField field = new JTextField("Your name");
```

How to use it

- Get text out of it

```
System.out.println(field.getText());
```

- Put text in it

```
field.setText("whatever");
field.setText("");
```

This clears the field

- Get an ActionEvent when the user presses return or enter

You can also register for key events if you really want to hear about it every time the user presses a key.

```
field.addActionListener(myActionListener);
```

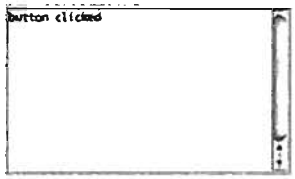
- Select/Highlight the text in the field

```
field.selectAll();
```

- Put the cursor back in the field (so the user can just start typing)

```
field.requestFocus();
```

JTextArea



Unlike JTextField, JTextArea can have more than one line of text. It takes a little configuration to make one, because it doesn't come out of the box with scroll bars or line wrapping. To make a JTextArea scroll, you have to stick it in a ScrollPane. A ScrollPane is an object that really loves to scroll, and will take care of the text area's scrolling needs.

Constructor

10 means 10 lines (sets the preferred height)

20 means 20 columns (sets the preferred width)

```
JTextArea text = new JTextArea(10,20);
```

How to use it

- Make it have a vertical scrollbar only

Make a JScrollPane and give it the text area that it's going to scroll for.

Tell the scroll pane to use only a vertical scrollbar

Turn on line wrapping

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

Important!! You give the text area to the scroll pane (through the scroll pane constructor), then add the scroll pane to the panel. You don't add the text area directly to the panel!

- Replace the text that's in it

```
text.setText("Not all who are lost are wandering");
```

- Append to the text that's in it

```
text.append("button clicked");
```

- Select/Highlight the text in the field

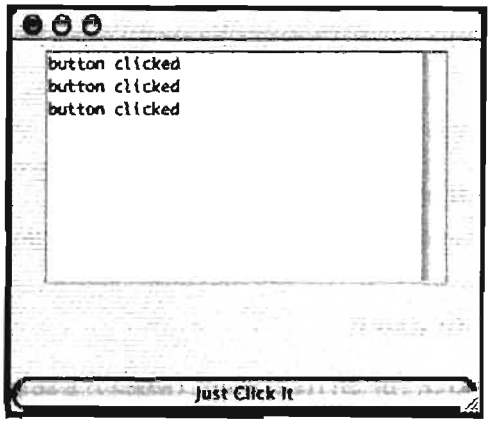
```
text.selectAll();
```

- Put the cursor back in the field (so the user can just start typing)

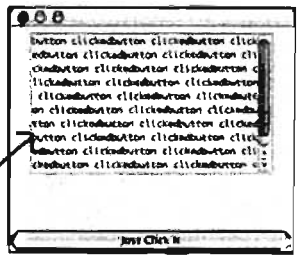
```
text.requestFocus();
```

JTextArea example

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


public class TextArea1 implements ActionListener {

    JTextArea text;

    public static void main (String[] args) {
       TextArea1 gui = new TextArea1();
       gui.go();
    }


   public void go() {
      JFrame frame = new JFrame();
      JPanel panel = new JPanel();
      JButton button = new JButton("Just Click It");
      button.addActionListener(this);
      text = new JTextArea(10,20);
      text.setLineWrap(true);

      JScrollPane scroller = new JScrollPane(text);
      scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
      scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

      panel.add(scroller);

      frame.getContentPane().add(BorderLayout.CENTER, panel);
      frame.getContentPane().add(BorderLayout.SOUTH, button);

      frame.setSize(350,300);
      frame.setVisible(true);
   }

   public void actionPerformed(ActionEvent ev) {
      text.append("button clicked \n ");
   }
}
```

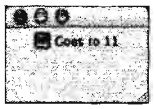


Insert a new line so the words go on a separate line each time the button is clicked. Otherwise, they'll run together.



check box

JCheckBox



Constructor

```
JCheckBox check = new JCheckBox("Goes to 11");
```

How to use it

1. Listen for an item event (when it's selected or deselected)

```
check.addItemListener(this);
```

2. Handle the event (and find out whether or not it's selected)

```
public void itemStateChanged(ItemEvent ev) {
   String onOrOff = "off";
   if (check.isSelected()) onOrOff = "on";
   System.out.println("Check box is " + onOrOff);
}
```

3. Select or deselect it in code

```
check.setSelected(true);
check.setSelected(false);
```

there are no Dumb Questions

Q: Aren't the layout managers just more trouble than they're worth? If I have to go to all this trouble, I might as well just hard-code the size and coordinates for where everything should go.

A: Getting the exact layout you want from a layout manager can be a challenge. But think about what the layout manager is really doing for you. Even the seemingly simple task of figuring out where things should go on the screen can be complex. For example, the layout manager takes care of keeping your components from overlapping one another. In other words, it knows how to manage the spacing between components (and between the edge of the frame). Sure you can do that yourself, but what happens if you want components to be very tightly packed? You might get them placed just right, by hand, but that's only good for your JVM!

Why? Because the components can be slightly different from platform to platform, especially if they use the underlying platform's native 'look and feel'. Subtle things like the bevel of the buttons can be different in such a way that components that line up neatly on one platform suddenly squish together on another.

And we're still not at the really Big Thing that layout managers do. Think about what happens when the user resizes the window! Or your GUI is dynamic, where components come and go. If you had to keep track of re-laying out all the components every time there's a change in the size or contents of a background component...yikes!

JList

JList constructor takes an array of any object type. They don't have to be Strings, but a String representation will appear in the list

Constructor

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta "};

list = new JList(listEntries);
```

How to use it

This is just like with JTextArea -- you make a JScrollPane (and give it the list), then add the scroll pane (NOT the list) to the panel.

- Make it have a vertical scrollbar

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

- Set the number of lines to show before scrolling

```
list.setVisibleRowCount(4);
```

- Restrict the user to selecting only ONE thing at a time

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

- Register for list selection events

```
list.addListSelectionListener(this);
```

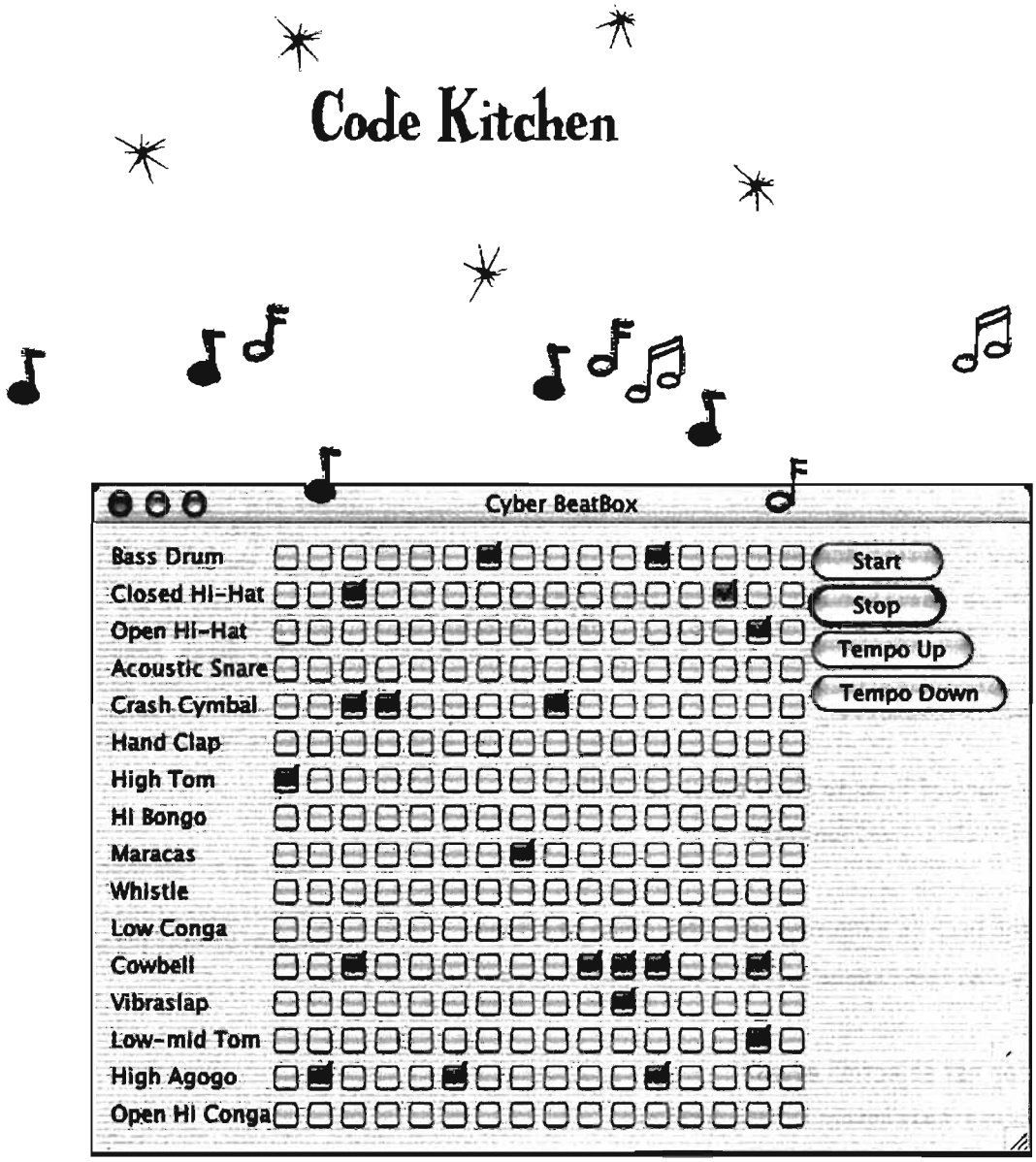
- Handle events (find out which thing in the list was selected)

You'll get the event TWICE if you don't put in this if test

```
public void valueChanged(ListSelectionEvent lse) {
      if( !lse.getValueIsAdjusting()) {
        String selection = (String) list.getSelectedValue();
        System.out.println(selection);
      }
  }
```

getSelectedValue() actually returns an Object. A list isn't limited to only String objects.

Code Kitchen



This part's optional. We're making the full BeatBox, GUI and all. In the Saving Objects chapter, we'll learn how to save and restore drum patterns. Finally, in the networking chapter (Make a Connection), we'll turn the BeatBox into a working chat client.

Making the BeatBox

This is the full code listing for this version of the BeatBox, with buttons for starting, stopping, and changing the tempo. The code listing is complete, and fully-annotated, but here's the overview:

- Build a GUI that has 256 checkboxes (JCheckBox) that start out unchecked, 16 labels (JLabel) for the instrument names, and four buttons.

- Register an ActionListener for each of the four buttons. We don't need listeners for the individual checkboxes, because we aren't trying to change the pattern sound dynamically (i.e. as soon as the user checks a box). Instead, we wait until the user hits the 'start' button, and then walk through all 256 checkboxes to get their state and make a MIDI track.

- Set-up the MIDI system (you've done this before) including getting a Sequencer, making a Sequence, and creating a track. We are using a sequencer method that's new to Java 5.0, setLoopCount( ). This method allows you to specify how many times you want a sequence to loop. We're also using the sequence's tempo factor to adjust the tempo up or down, and maintain the new tempo from one iteration of the loop to the next.

- When the user hits 'start', the real action begins. The event-handling method for the 'start' button calls the buildTrackAndStart() method. In that method, we walk through all 256 checkboxes (one row at a time, a single instrument across all 16 beats) to get their state, then use the information to build a MIDI track (using the handy makeEvent() method we used in the previous chapter). Once the track is built, we start the sequencer, which keeps playing (because we're looping it) until the user hits 'stop'.

BeatBox code

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;


public class BeatBox {

    JPanel mainPanel;
    ArrayList<JCheckBox> checkboxList;
    Sequencer sequencer;
    Sequence sequence;
    Track track;
    JFrame theFrame;

    String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
       "Open Hi-Hat","Acoustic Snare", "Crash Cymbal", "Hand Clap",
       "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
       "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
       "Open Hi Conga"};
    int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};


    public static void main (String[] args) {
        new BeatBox2().buildGUI();
    }

    public void buildGUI() {
        theFrame = new JFrame("Cyber BeatBox");
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        BorderLayout layout = new BorderLayout();
        JPanel background = new JPanel(layout);
        background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        checkboxList = new ArrayList<JCheckBox>();
        Box buttonBox = new Box(BoxLayout.Y_AXIS);

        JButton start = new JButton("Start");
        start.addActionListener(new MyStartListener());
        buttonBox.add(start);

        JButton stop = new JButton("Stop");
        stop.addActionListener(new MyStopListener());
        buttonBox.add(stop);

        JButton upTempo = new JButton("Tempo Up");
        upTempo.addActionListener(new MyUpTempoListener());
        buttonBox.add(upTempo);

        JButton downTempo = new JButton("Tempo Down");
```

We store the checkboxes in an ArrayList

These are the names of the instruments, as a String array, for building the GUI labels (on each row)

These represent the actual drum 'keys'. The drum channel is like a piano, except each 'key' on the piano is a different drum. So the number '35' is the key for the Bass drum, 42 is Closed Hi-Hat, etc.

An 'empty border' gives us a margin between the edges of the panel and where the components are placed. Purely aesthetic.

Nothing special here, just lots of GUI code. You've seen most of it before.

```
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
       nameBox.add(new Label(instrumentNames[i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);

    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        checkboxList.add(c);
        mainPanel.add(c);
    } // end loop

    setUpMidi();

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
} // close method


public void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // close method
```

Still more GUI set-up code. Nothing remarkable.

Make the checkboxes, set them to 'false' (so they aren't checked) and add them to the ArrayList AND to the GUI panel.

The usual MIDI set-up stuff for getting the Sequencer, the Sequence, and the Track. Again, nothing special.

This is where it all happens! Where we turn checkbox state into MIDI events, and add them to the Track.

```
public void buildTrackAndStart() {
  int[] trackList = null;

  sequence.deleteTrack(track);
  track = sequence.createTrack();

    for (int i = 0; i < 16; i++) {
      trackList = new int[16];

      int key = instruments[i];

      for (int j = 0; j < 16; j++ ) {

          JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
          if ( jc.isSelected()) {
             trackList[j] = key;
          } else {
             trackList[j] = 0;
          }
       } // close inner loop

       makeTracks(trackList);
       track.add(makeEvent(176,1,127,0,16));
   } // close outer

   track.add(makeEvent(192,9,1,0,15));
   try {

      sequencer.setSequence(sequence);
       sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
       sequencer.start();
       sequencer.setTempoInBPM(120);
   } catch(Exception e) {e.printStackTrace();}
} // close buildTrackAndStart method

public class MyStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        buildTrackAndStart();
    }
} // close inner class
```

We'll make a 16-element array to hold the values for one instrument, across all 16 beats. If the instrument is supposed to play on that beat, the value at that element will be the key. If that instrument is NOT supposed to play on that beat, put in a zero.

get rid of the old track, make a fresh one.

do this for each of the 16 ROWS (i.e. Bass, Congo, etc.)

Set the 'key' that represents which instrument this is (Bass, Hi-Hat, etc. The instruments array holds the actual MIDI numbers for each instrument.)

Do this for each of the BEATS for this row

Is the checkbox at this beat selected? If yes, put the key value in this slot in the array (the slot that represents this beat). Otherwise, the instrument is NOT supposed to play at this beat, so set it to zero.

For this instrument, and for all 16 beats, make events and add them to the track.

We always want to make sure that there IS an event at beat 16 (it goes 0 to 15). Otherwise, the BeatBox might not go the full 16 beats before it starts over.

Let's you specify the number of loop iterations, or in this case, continuous looping.

NOW PLAY THE THING!!

First of the inner classes, listeners for the buttons. Nothing special here.

```
public class MyStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequencer.stop();
    }
} // close inner class

public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
 } // close inner class

 public class MyDownTempoListener implements ActionListener {
     public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
} // close inner class
```

The other inner class listeners for the buttons

The Tempo Factor scales the sequencer's tempo by the factor provided. The default is 1.0, so we're adjusting +/– 3% per click.

```
public void makeTracks(int[] list) {

   for (int i = 0; i < 16; i++) {
      int key = list[i];

      if (key != 0) {
         track.add(makeEvent(144,9,key, 100, i));
         track.add(makeEvent(128,9,key, 100, i+1));
      }
   }
}

public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return event;
}

} // close class
```

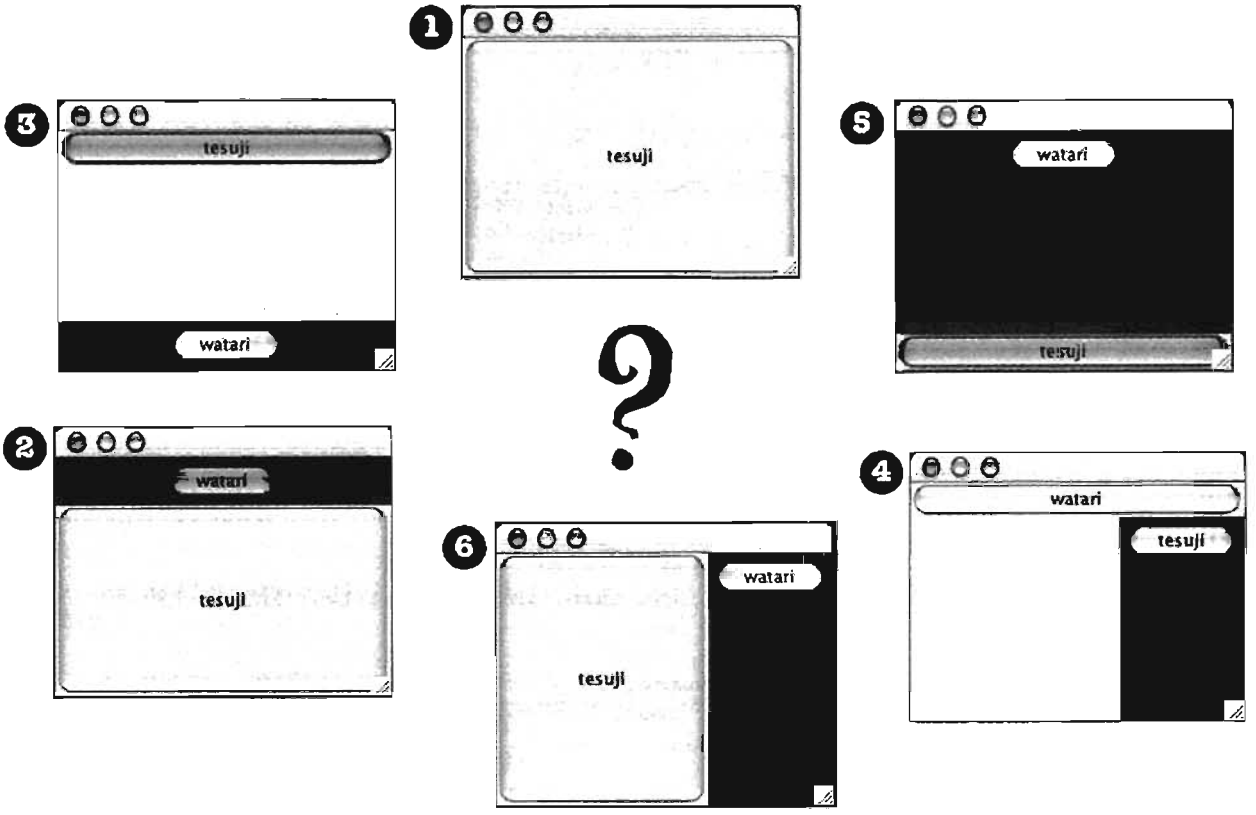
This makes events for one instrument at a time, for all 16 beats. So it might get an int[ ] for the Bass drum, and each index in the array will hold either the key of that instrument, or a zero. If it's a zero, the instrument isn't supposed to play at that beat. Otherwise, make an event and add it to the track.

Make the NOTE ON and NOT OFF events, and add them to the Track.

This is the utility method from last chapter's CodeKitchen. Nothing new.

Which code goes with which layout?

Five of the six screens below were made from one of the code fragments on the opposite page. Match each of the five code fragments with the layout that fragment would produce.



Code Fragments

D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

A

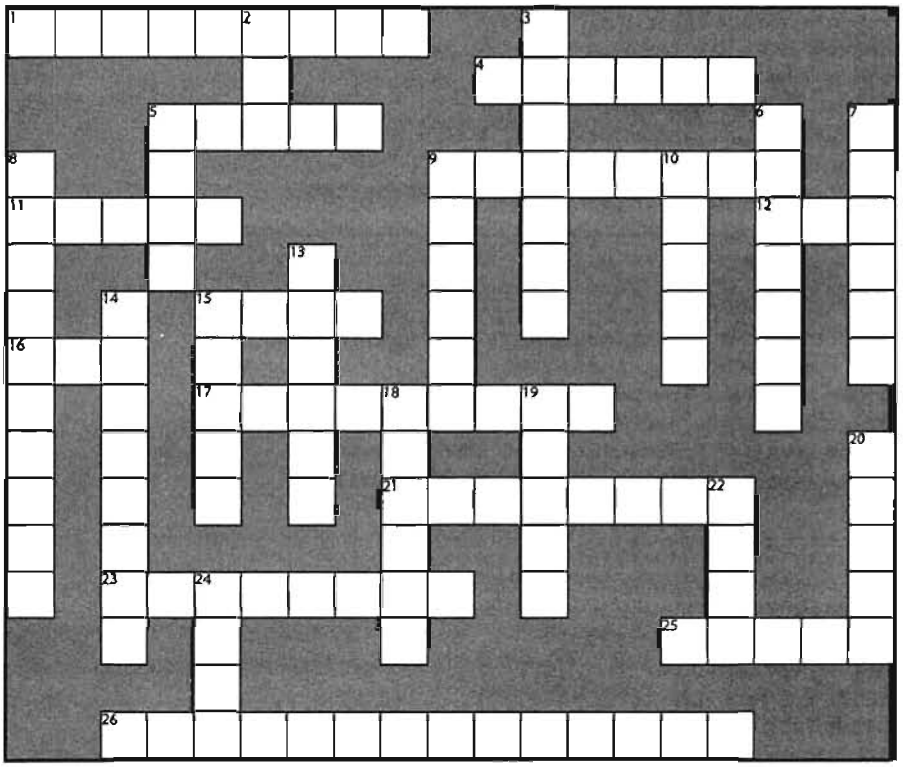
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

puzzle: crossword

GUI-Cross 7.0



You can do it.

Across

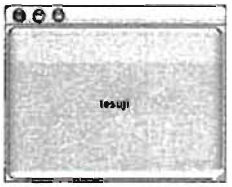
1. Artist's sandbox
4. Border's catchall
5. Java look
9. Generic waiter
11. A happening
12. Apply a widget
15. JPanel's default
16. Polymorphic test
17. Shake it baby
21. Lots to say
23. Choose many
25. Button's pal
26. Home of actionPerformed

Down

2. Swing's dad
3. Frame's purview
5. Help's home
6. More fun than text
7. Component slang
8. Romulin command
9. Arrange
10. Border's top
13. Manager's rules
14. Source's behavior
15. Border by default
18. User's behavior
19. Inner's squeeze
20. Backstage widget
22. Mac look
24. Border's right

Exercise Solutions

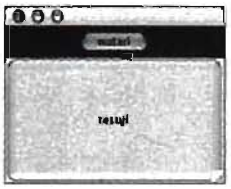
1



C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

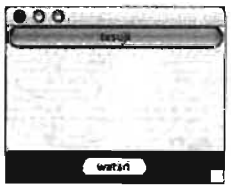
2



D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

3



E

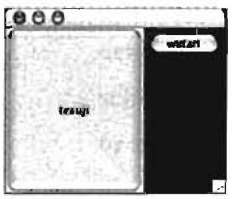
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

4

A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

6

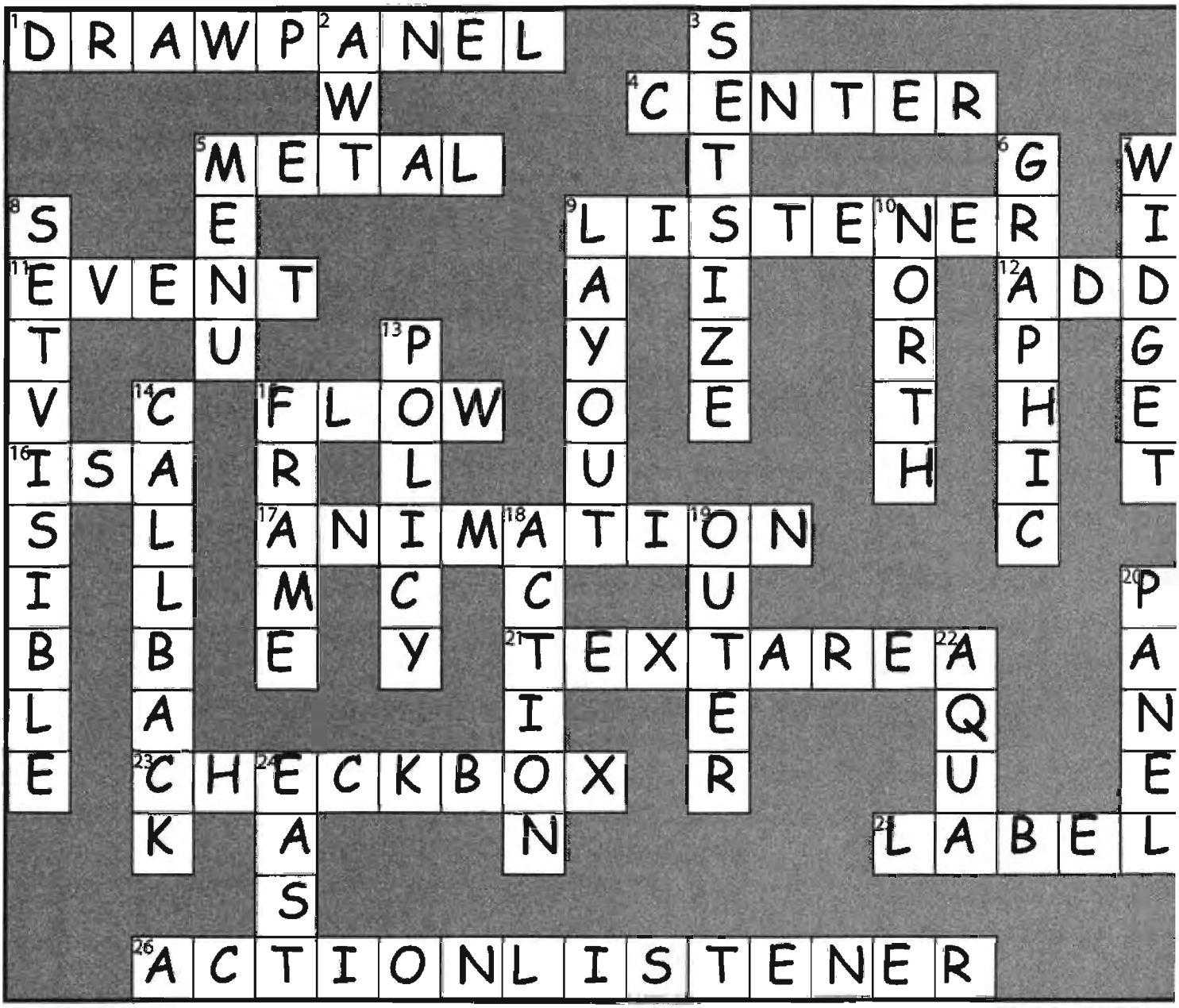


B

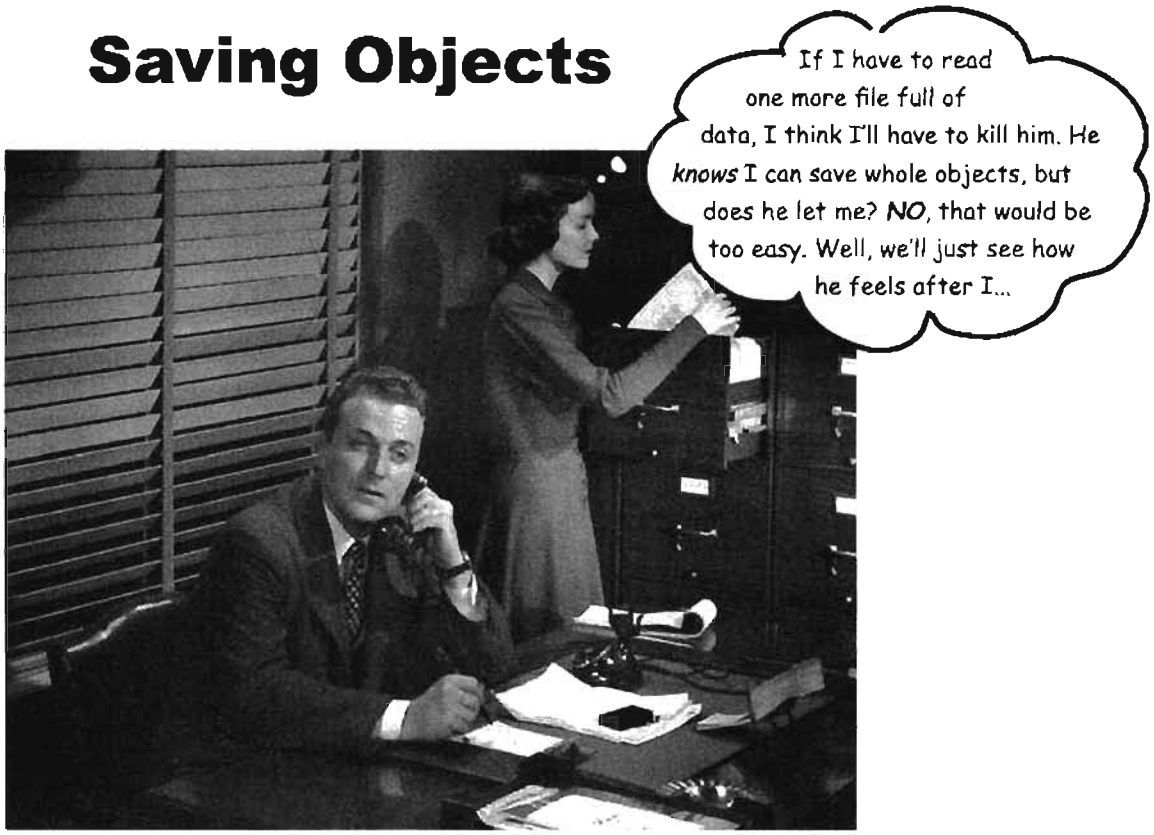
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

Puzzle Answers

GUI-Cross 7.0



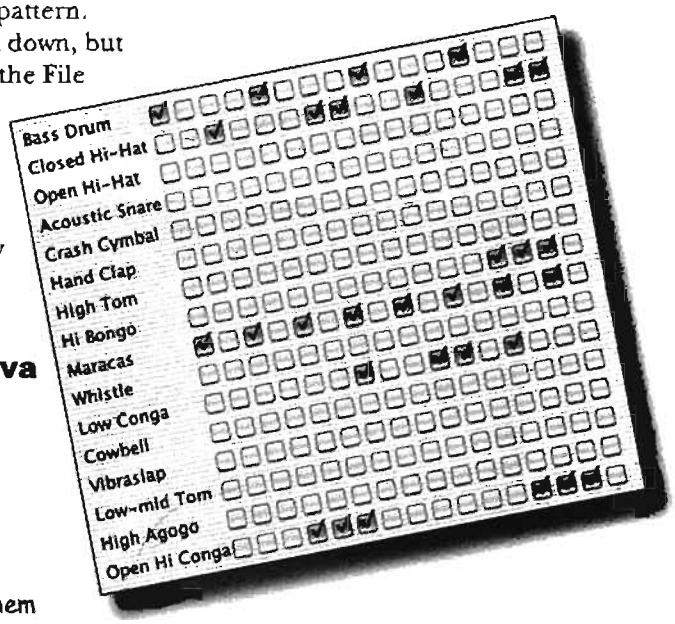
Saving Objects



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.

● 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:

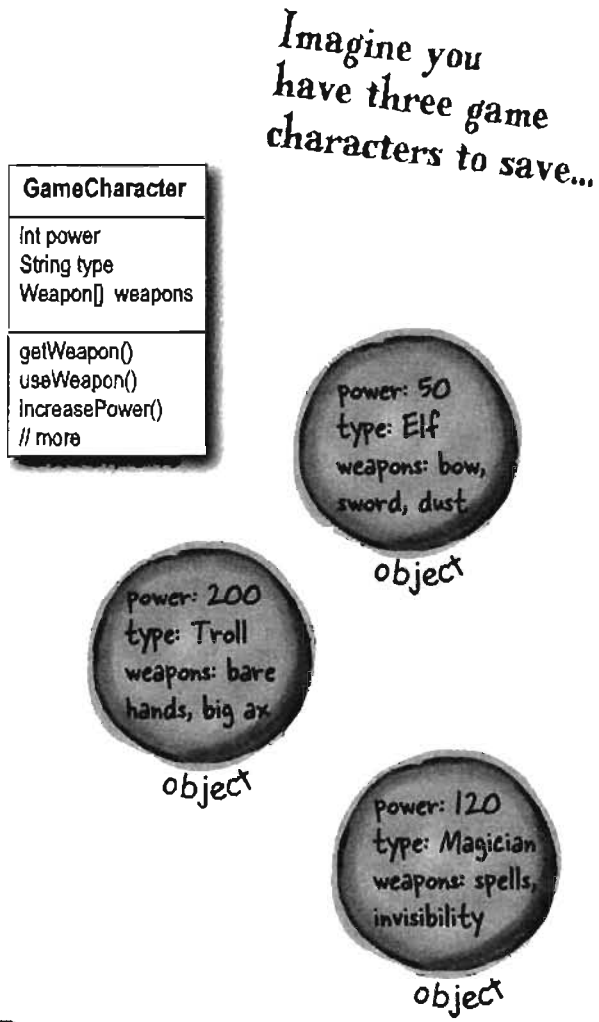
```
¬ísrGameCharacter
¬%g88MÛIpowerL¡java/lang/
String;[weaponst[L¡java/lang/
String;xpStifur[L¡java.lang.String;»"VÅ
ä{Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

● 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);
```

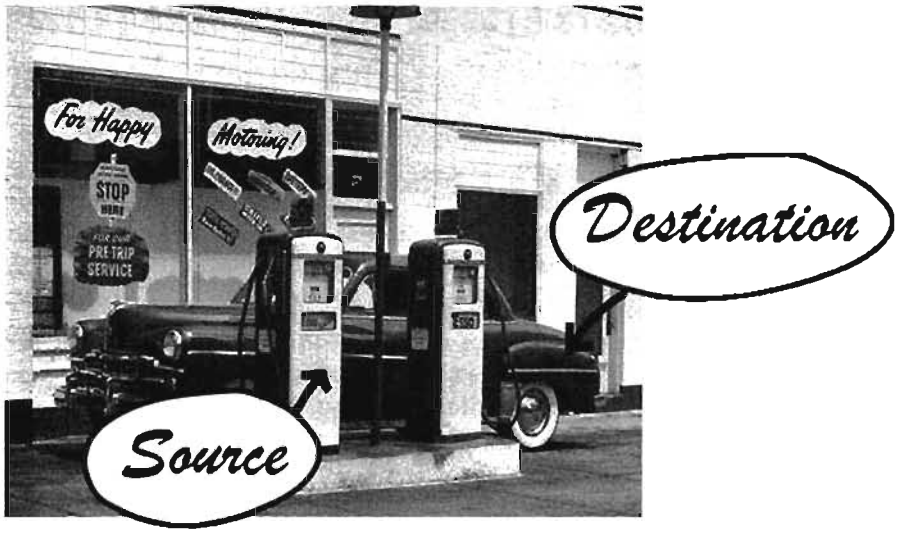
serializes the objects referenced by characterOne, characterTwo, and characterThree, and writes them to the file "MyGame.ser".

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.



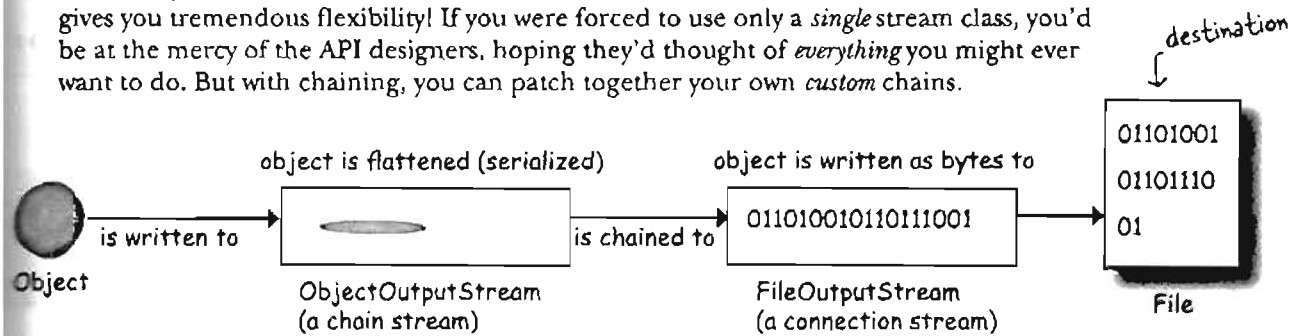
Connection streams represent a connection to a source or destination (file, socket, etc.) while chain streams can't connect on their own and must be chained to a connection stream.

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.



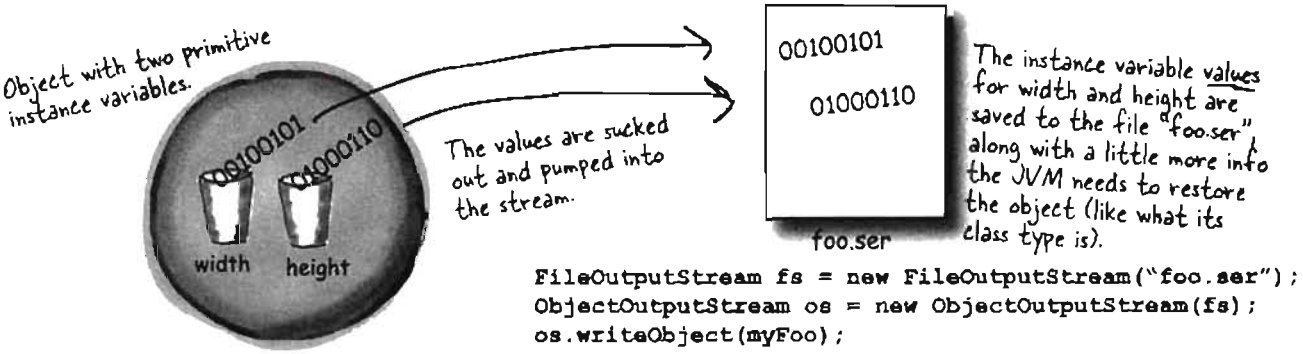
What really happens to an object when it's serialized?

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.




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

```
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.

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.

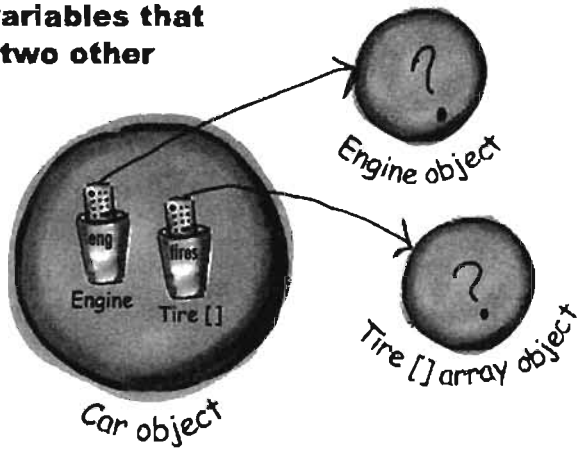
Brain Barbell

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.

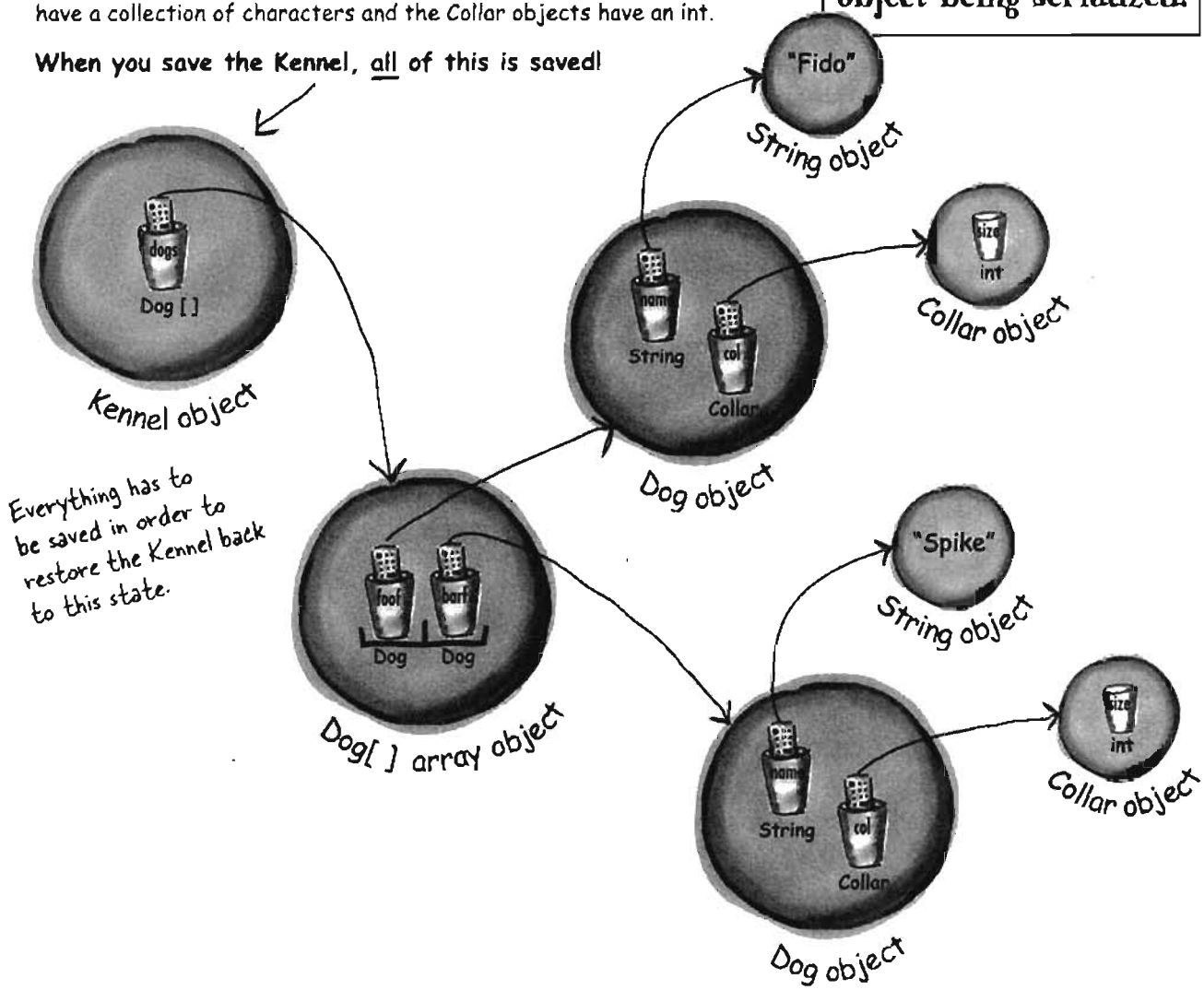


What does it take to save a Car object?

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!

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.

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



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.*;
```

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

```
public class Box implements Serializable {
```

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

```
    private int width;
    private int height;
```

these two values will be saved

```
    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();
      }
   }
}
```

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.

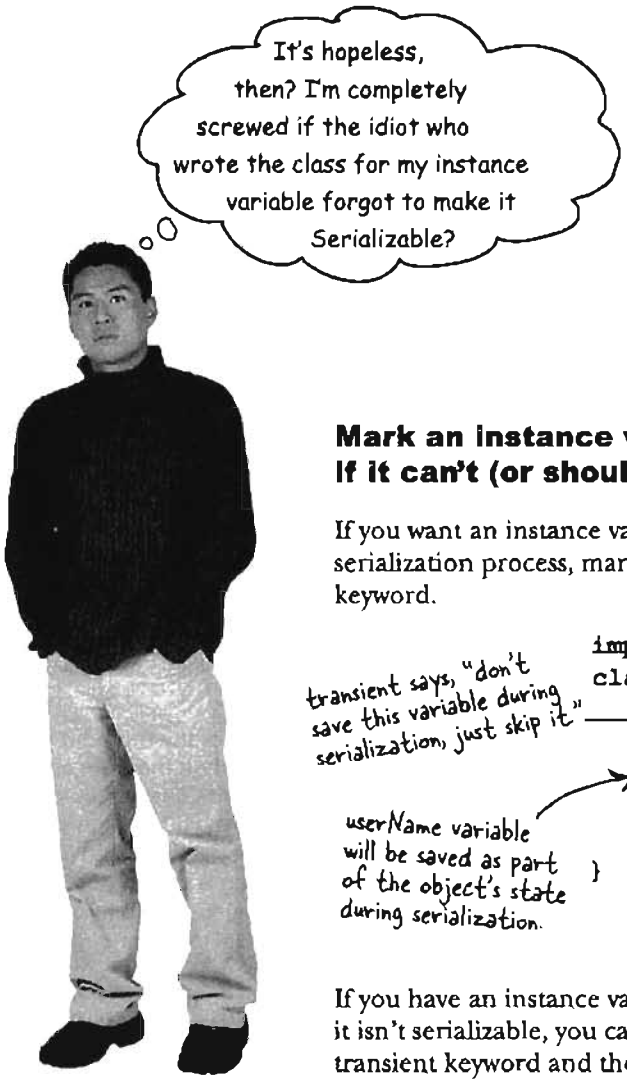
When you serialize myPond (a Pond object), its Duck instance variable automatically gets serialized.

```
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:

```
File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
        at Pond.main(Pond.java:13)
```



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.

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

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

  // more code
}
```

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.

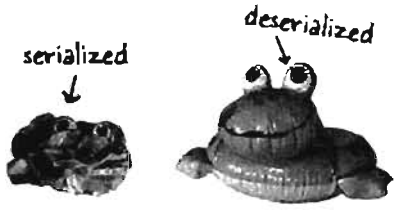
2) If the value of the transient variable *does* matter (say, if the color and design of the transient Collar are unique for each Dog) then you need to save the key values of the Collar and use them when the Dog is brought back to essentially re-create a brand new Collar that's identical to the original.

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.

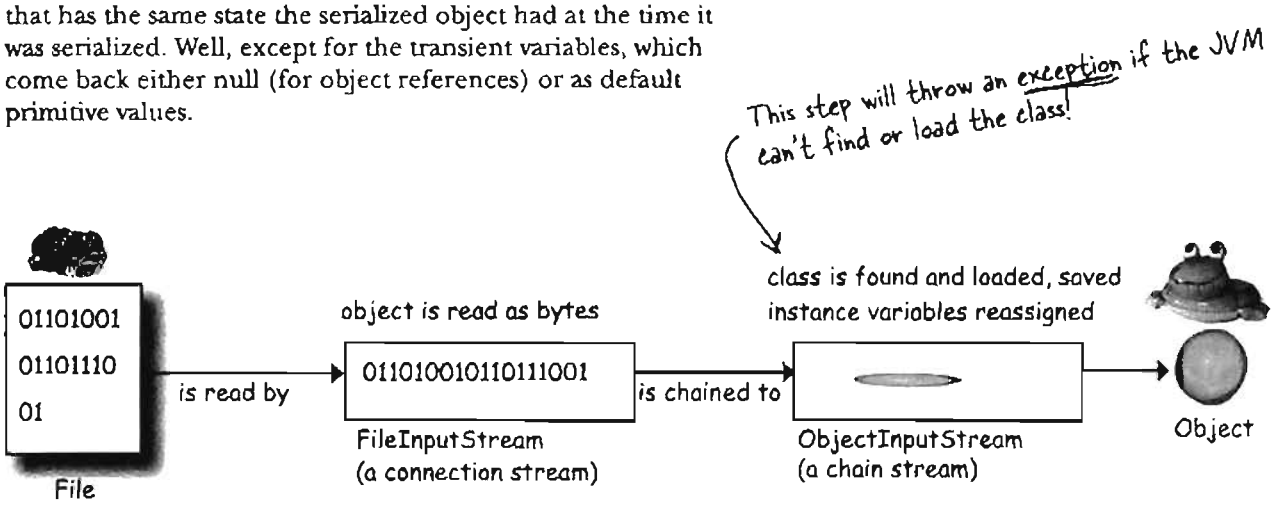
5 Close the ObjectInputStream

```
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

```
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;

     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();
     }
   }
}
```

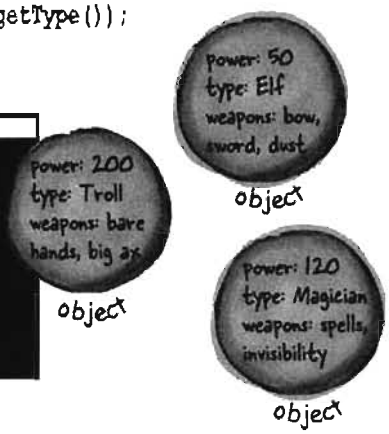
Make some characters...

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

Now read them back in from the file...

Check to see if it worked.

```
File Edit Window Help Resuscitate
% java GameSaver
Elf
Troll
Magician
```



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);
```

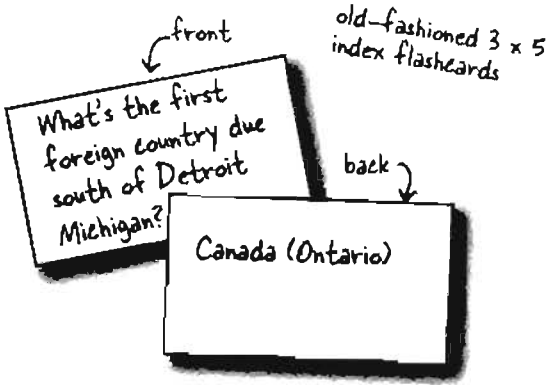
To write a String:

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

```
import java.io.*;                          // We need the java.io package for FileWriter

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

        try {
            FileWriter writer = new FileWriter("Foo.txt");   // If the file "Foo.txt" does not exist, FileWriter will create it

            writer.write("hello foo!");    // The write() method takes a String

            writer.close();                // Close it when you're done!

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

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

Text File Example: e-Flashcards



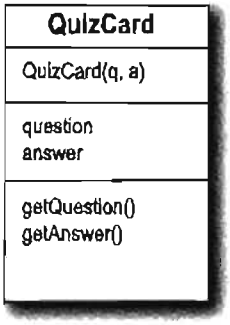
Remember those flashcards you used in school? Where you had a question on one side and the answer on the back? They aren't much help when you're trying to understand something, but nothing beats 'em for raw drill-and-practice and rote memorization. *When you have to burn in a fact.* And they're also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

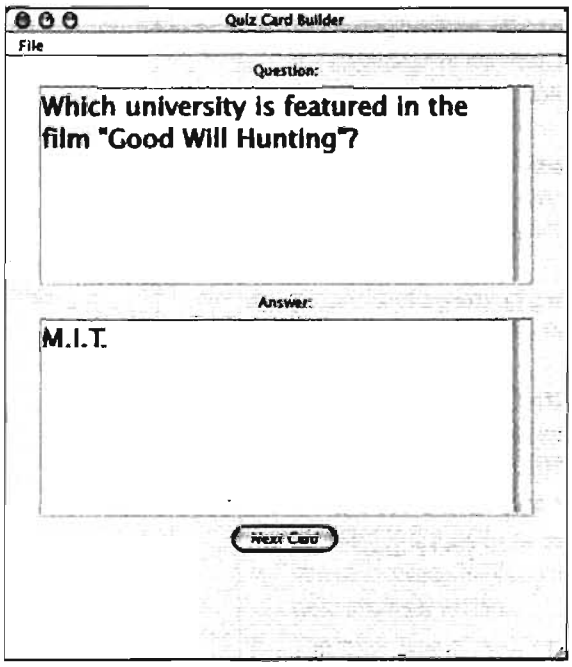
We're going to make an electronic version that has three classes:

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.

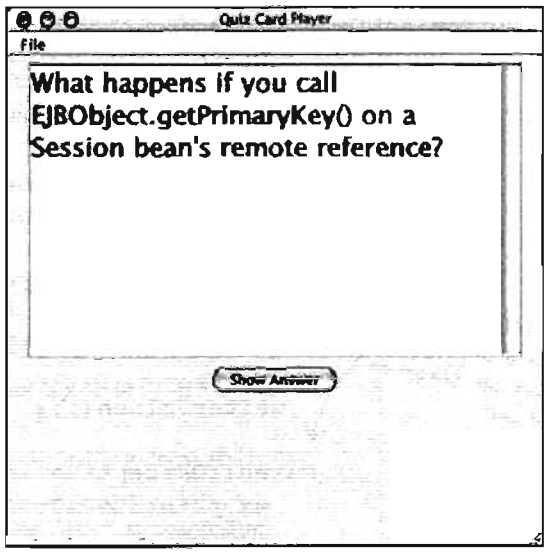
3) ***QuizCard***, a simple class representing card data. We'll walk through the code for the builder and the player, and have you make the QuizCard class yourself, using this →





QuizCardBuilder

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



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);
    }

    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());
       }
    }

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

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

    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 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

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.

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.

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")

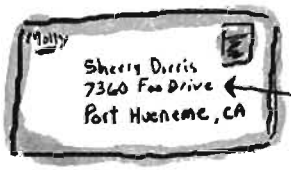
The java.io.File class

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!



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.

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();
```

A File object represents the filename "GameFile.txt"

GameFile.txt

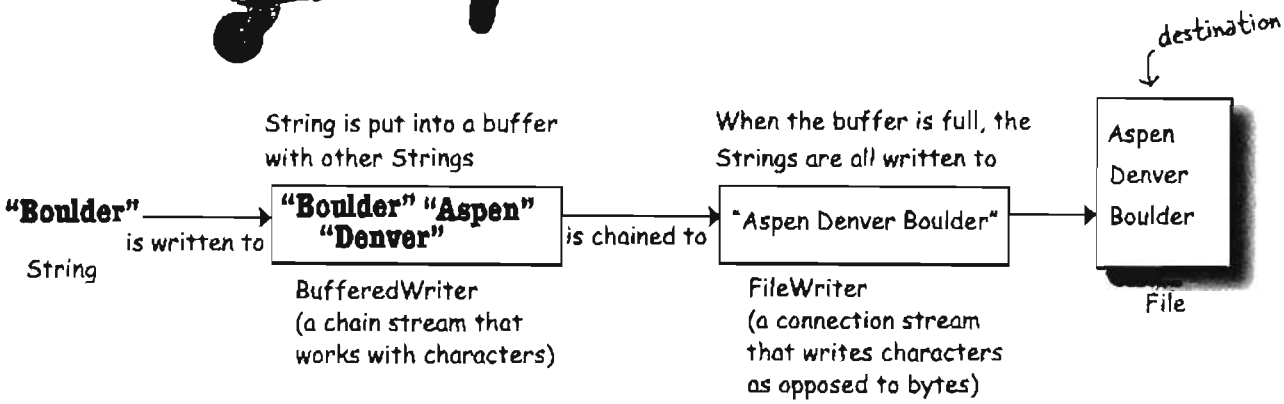
```
80,Elf,bow, sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

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.




```
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.*

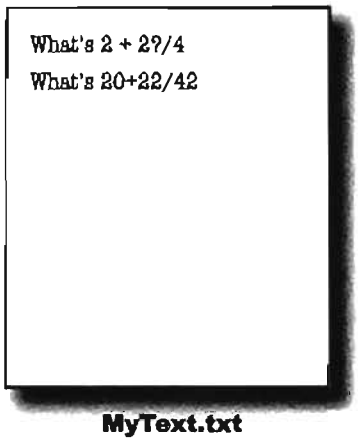
If you do want to send data *before* the buffer is full, you do have control. ***Just Flush It.*** Calls to writer.flush() say, "send whatever's in the buffer, ***now***!"

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



Don't forget the import

```
import java.io.*;

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

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

            BufferedReader reader = new BufferedReader(fileReader);

            String line = null;

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

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

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

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

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);
          }
       }
    }
  }

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

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();
}

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
```

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.

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

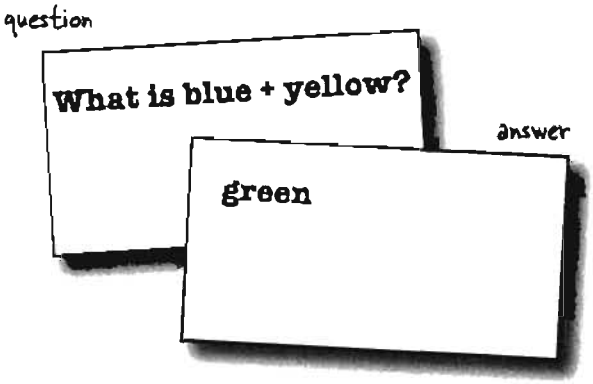
Make a BufferedReader chained to a new FileReader, giving the FileReader the File object the user chose from the open file dialog.

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

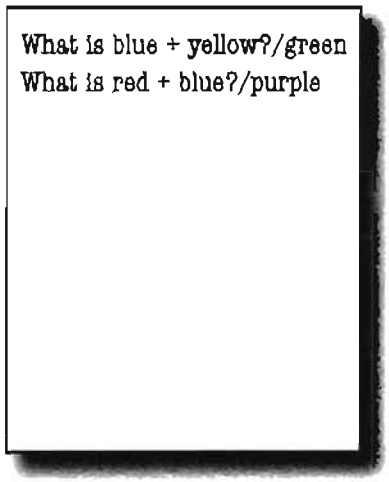
Each line of text corresponds to a single flashcard, but we have to parse out the question and answer as separate pieces. We use the String split() method to break the line into two tokens (one for the question and one for the answer). We'll look at the split() method on the next page.

Parsing with String split()

Imagine you have a flashcard like this:



Saved in a question file like this:



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.

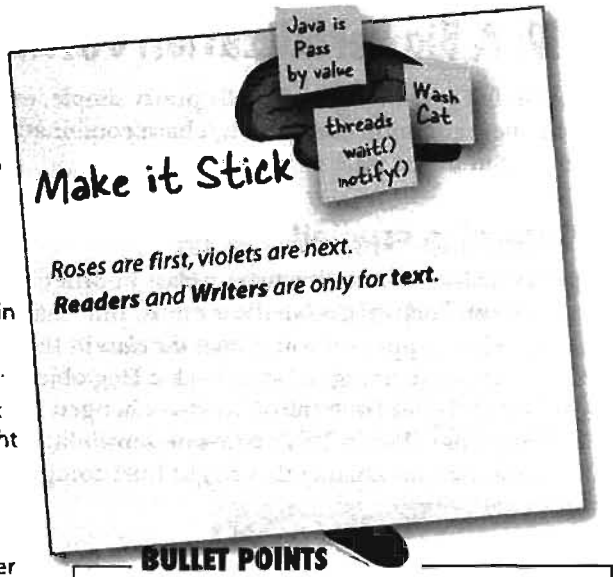
Most of the time, though, you'll use the same small handful of classes. If you're writing text files, BufferedReader and BufferedWriter (chained to FileReader and FileWriter) are probably all you need. If you're writing serialized objects, you can use ObjectOutputStream and ObjectInputStream (chained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



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:

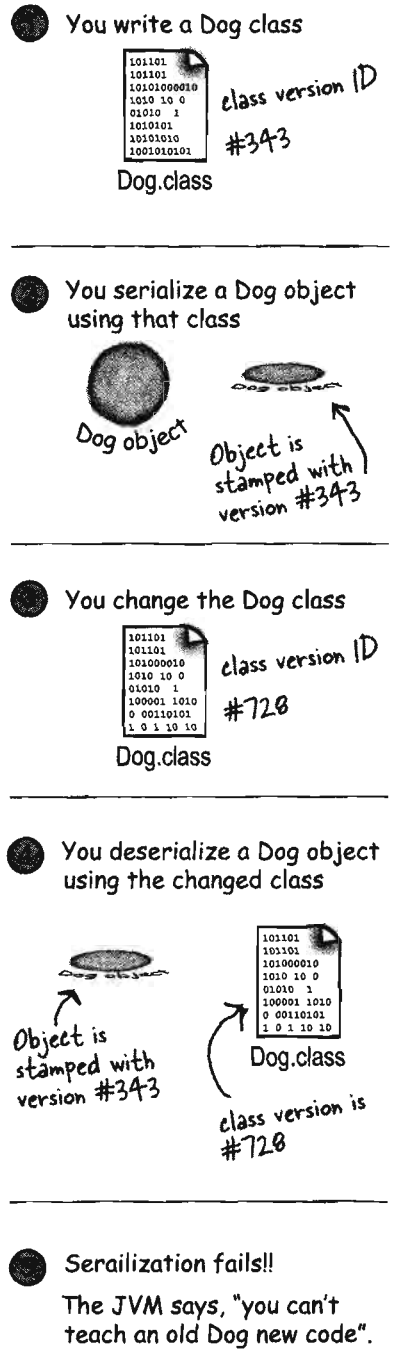
Adding new instance variables to the class (existing objects will deserialize with default values for the instance variables they didn't have when they were serialized)

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)



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

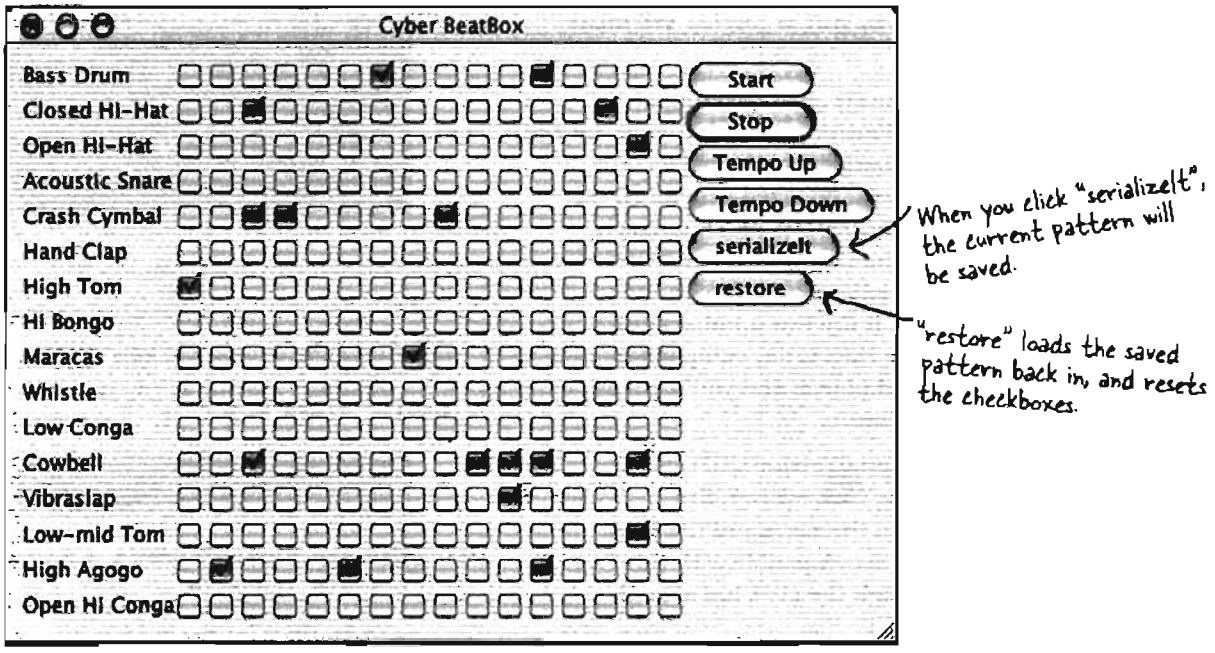
```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

2. Paste the output into your class

```
public class Dog {

   static final long serialVersionUID =
                        -5849794470754667710L;

   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();
```

exercise: True or False

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.)

```
File Edit Window Help Torture
java DungeonTest
12
```

```
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");
public int x = 3;
transient 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();
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**

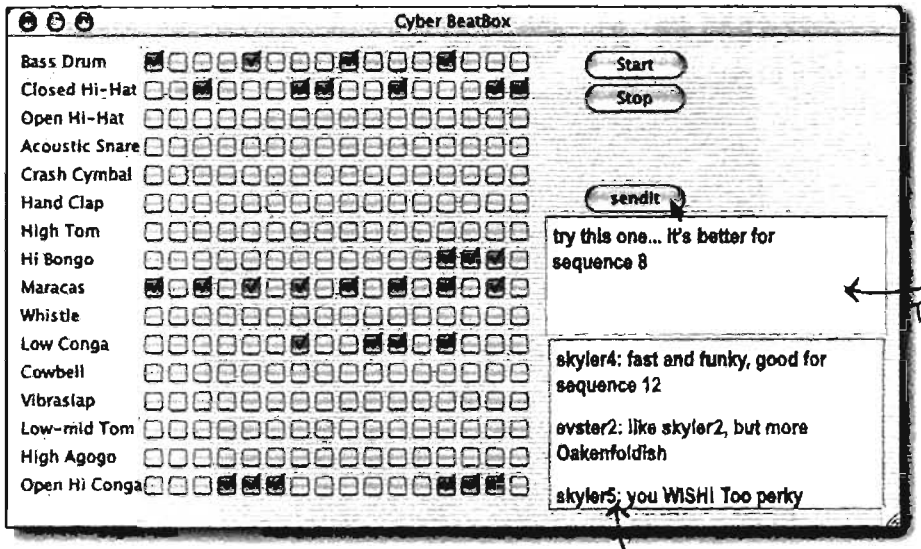

```
import java.io.*;

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());
  }
}
```

```
File Edit Window Help Escape
% java DungeonTest
12
8
```

Make a Connection

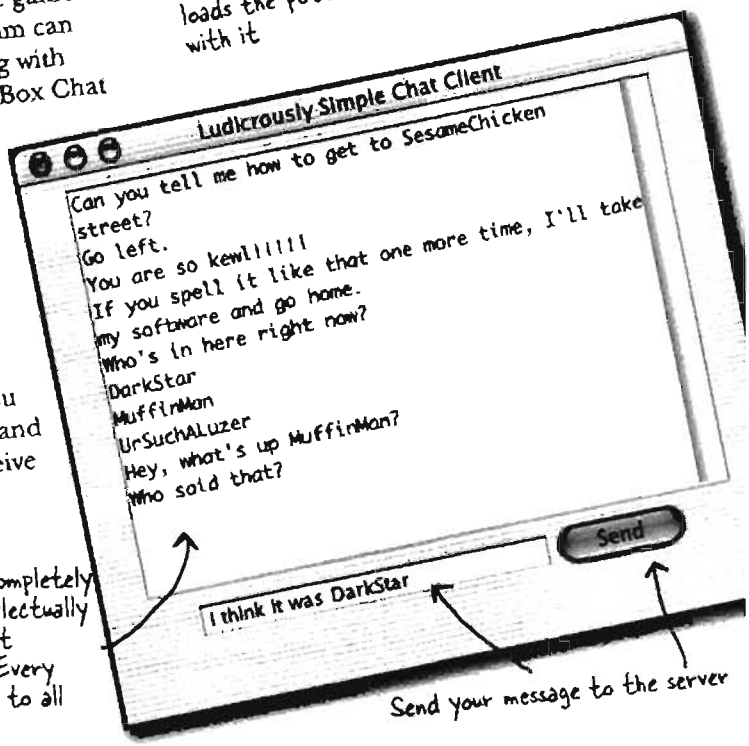
Connect with the outside world. Your Java program can reach out and touch a program on another machine. It's easy. All the low-level networking details are taken care of by classes in the java.net library. One of Java's big benefits is that sending and receiving data over a network is just I/O with a slightly different connection stream at the end of the chain. If you've got a BufferedReader, you can *read*. And the BufferedReader could care less if the data came out of a file or flew down an ethernet cable. In this chapter we'll connect to the outside world with sockets. We'll make *client* sockets. We'll make *server* sockets. We'll make *clients* and *servers*. And we'll make them talk to each other. Before the chapter's done, you'll have a fully-functional, multithreaded chat client. Did we just say *multithreaded*? Yes, now you *will* learn the secret of how to talk to Bob while simultaneously listening to Suzy.

:al-time Beat Box Chat



You're working on a computer game. You and your team are doing the sound design for each part of the game. Using a 'chat' version of the Beat Box, your team can collaborate—you can send a beat pattern along with your chat message, and everybody in the Beat Box Chat gets it. So you don't just get to *read* the other participants' messages, you get to load and *play* a beat pattern simply by clicking the message in the incoming messages area.

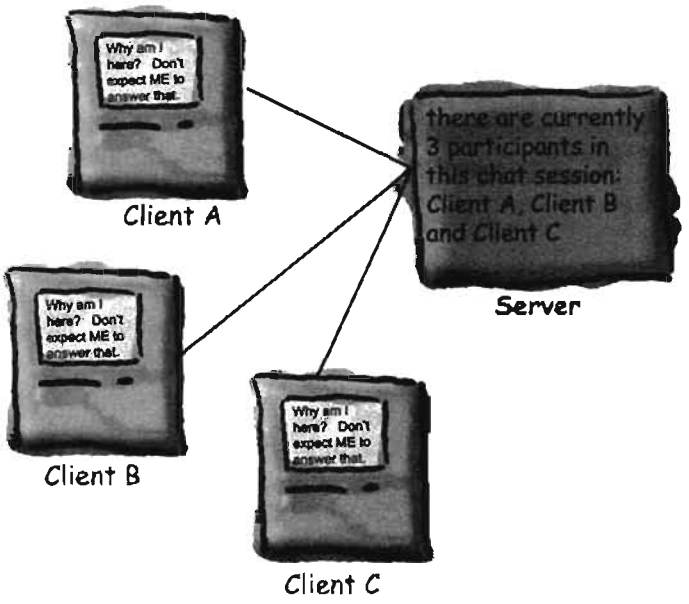
In this chapter we're going to learn what it takes to make a chat client like this. We're even going to learn a little about making a chat *server*. We'll save the full Beat Box Chat for the Code Kitchen, but in this chapter you *will* write a Ludicrously Simple Chat Client and Very Simple Chat Server that send and receive text messages.



Chat Program Overview

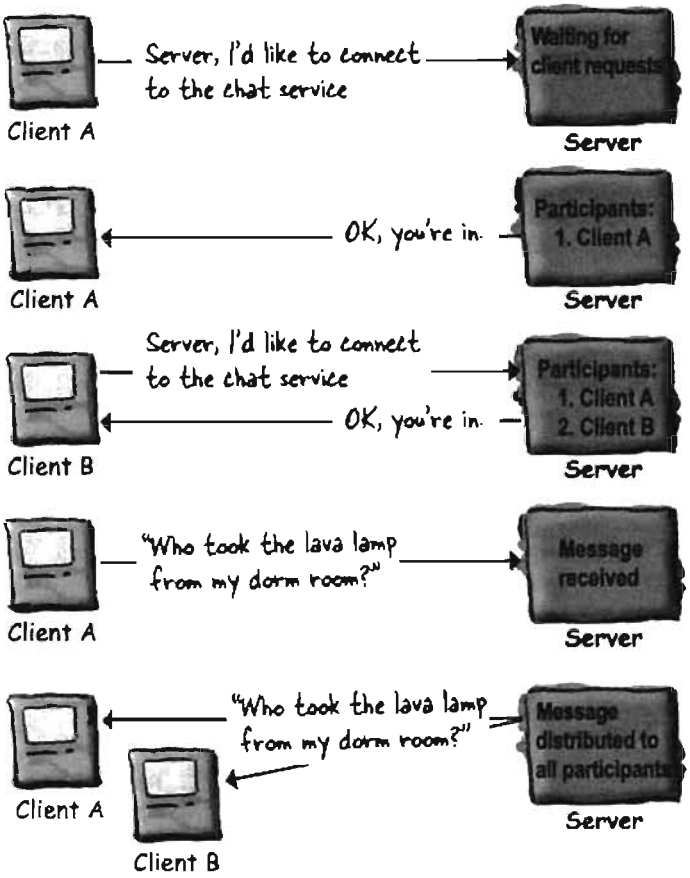
The Client has to know about the Server.

The Server has to know about ALL the Clients.



How it Works:

1. Client connects to the server
2. The server makes a connection and adds the client to the list of participants
3. Another client connects
4. Client A sends a message to the chat service
5. The server distributes the message to ALL participants (including the original sender)



Connecting, Sending, and Receiving

The three things we have to learn to get the client working are :

1) How to establish the initial **connection** between the client and server

2) How to **send** messages *to* the server

3) How to **receive** messages *from* the server

There's a lot of low-level stuff that has to happen for these things to work. But we're lucky, because the Java API networking package (java.net) makes it a piece of cake for programmers. You'll see a lot more GUI code than networking and I/O code.

And that's not all.

Lurking within the simple chat client is a problem we haven't faced so far in this book: doing two things at the same time. Establishing a connection is a one-time operation (that either works or fails). But after that, a chat participant wants to *send outgoing messages* and **simultaneously** *receive incoming messages* from the other participants (via the server). Hmmmm... that one's going to take a little thought, but we'll get there in just a few pages.

1 Connect

Client connects to the server by establishing a **Socket** connection.

2 Send

Client **sends** a message to the server

3 Receive

Client **gets** a message from the server

Make a network Socket connection

To connect to another machine, we need a Socket connection. A Socket (java.net.Socket class) is an object that represents a network connection between two machines. What's a connection? A *relationship* between two machines, where **two pieces of software know about each other.** Most importantly, those two pieces of software know how to *communicate* with each other. In other words, how to send *bits* to each other.

We don't care about the low-level details, thankfully, because they're handled at a much lower place in the 'networking stack'. If you don't know what the 'networking stack' is, don't worry about it. It's just a way of looking at the layers that information (bits) must travel through to get from a Java program running in a JVM on some OS, to physical hardware (ethernet cables, for example), and back again on some other machine. *Somebody* has to take care of all the dirty details. But not you. That somebody is a combination of OS-specific software and the Java networking API. The part that you have to worry about is high-level—make that *very* high-level—and shockingly simple. Ready?

To make a Socket connection, you need to know two things about the server: who it is, and which port it's running on.

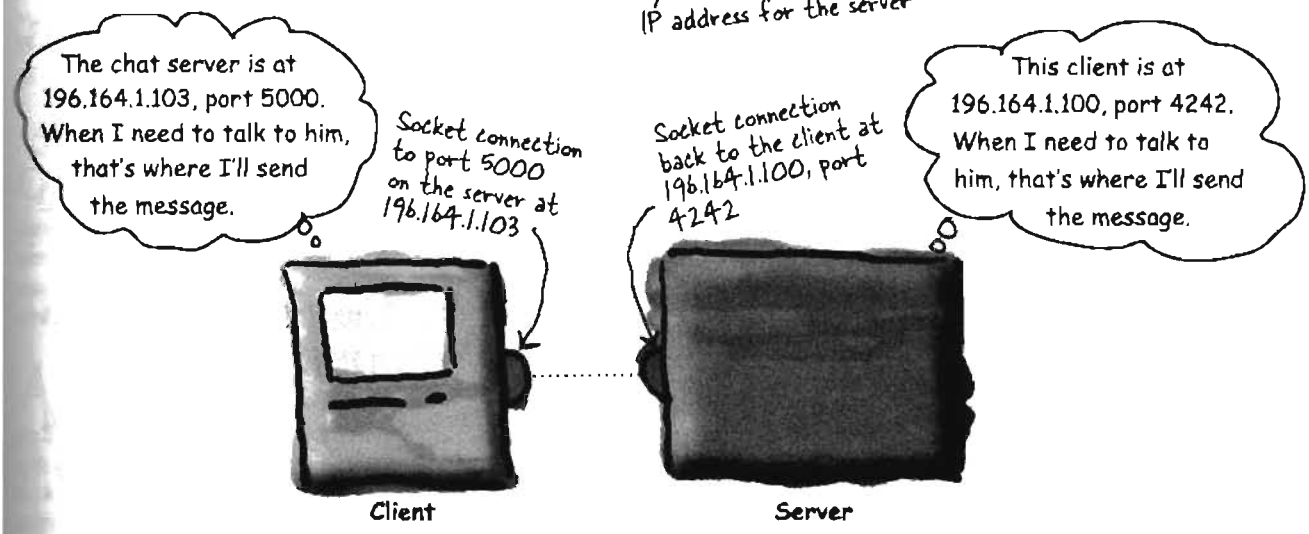
In other words,

IP address and TCP port number.

TCP port number

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

IP address for the server



A Socket connection means the two machines have information about each other, including network location (IP address) and TCP port.

A TCP port is just a number. A 16-bit number that identifies a specific program on the server.

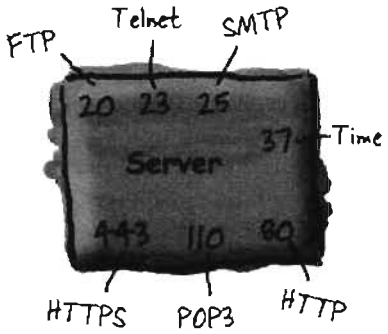
Your internet web (HTTP) server runs on port 80. That's a standard. If you've got a Telnet server, its running on port 23. FTP? 20. POP3 mail server? 110. SMTP? 25. The Time server sits at 37. Think of port numbers as unique identifiers. They represent a logical connection to a particular piece of software running on the server. That's it. You can't spin your hardware box around and find a TCP port. For one thing, you have 65536 of them on a server (0 - 65535). So they obviously don't represent a place to plug in physical devices. They're just a number representing an application.

Without port numbers, the server would have no way of knowing which application a client wanted to connect to. And since each application might have its own unique protocol, think of the trouble you'd have without these identifiers. What if your web browser, for example, landed at the POP3 mail server instead of the HTTP server? The mail server won't know how to parse an HTTP request! And even if it did, the POP3 server doesn't know anything about servicing the HTTP request.

When you write a server program, you'll include code that tells the program which port number you want it to run on (you'll see how to do this in Java a little later in this chapter). In the Chat program we're writing in this chapter, we picked 5000. Just because we wanted to. And because it met the criteria that it be a number between 1024 and 65535. Why 1024? Because 0 through 1023 are reserved for the well-known services like the ones we just talked about.

And if you're writing services (server programs) to run on a company network, you should check with the sys-admins to find out which ports are already taken. Your sys-admins might tell you, for example, that you can't use any port number below, say, 3000. In any case, if you value your limbs, you won't assign port numbers with abandon. Unless it's your *home* network. In which case you just have to check with your *kids*.

Well-known TCP port numbers for common server applications



A server can have up to 65536 different server apps running, one per port.

The TCP port numbers from 0 to 1023 are reserved for well-known services. Don't use them for your own server programs!*

The chat server we're writing uses port 5000. We just picked a number between 1024 and 65535.

*Well, you *might* be able to use one of these, but the sys-admin where you work will probably kill you.

there are no Dumb Questions

Q: How do you know the port number of the server program you want to talk to?

A: That depends on whether the program is one of the well-known services. If you're trying to connect to a well-known service, like the ones on the opposite page (HTTP, SMTP, FTP, etc.) you can look these up on the Internet (Google "Well-Known TCP Port"). Or ask your friendly neighborhood sys-admin.

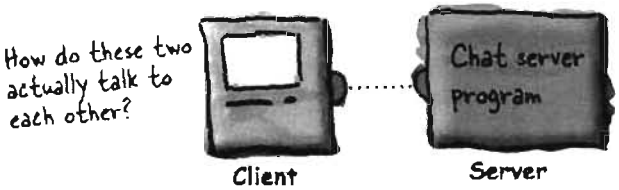
But if the program isn't one of the well-known services, you need to find out from whoever is deploying the service. Ask him. Or her. Typically, if someone writes a network service and wants others to write clients for it, they'll publish the IP address, port number, and protocol for the service. For example, if you want to write a client for a GO game server, you can visit one of the GO server sites and find information about how to write a client for that particular server.

Q: Can there ever be more than one program running on a single port? In other words, can two applications on the same server have the same port number?

A: No! If you try to bind a program to a port that is already in use, you'll get a BindException. To *bind* a program to a port just means starting up a server application and telling it to run on a particular port. Again, you'll learn more about this when we get to the server part of this chapter.

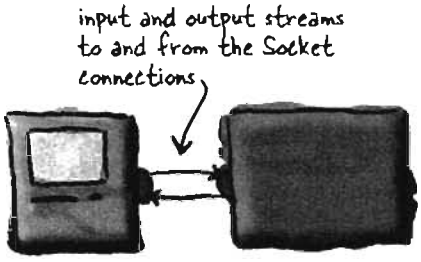
Brain Barbell

OK, you got a Socket connection. The client and the server know the IP address and TCP port number for each other. Now what? How do you communicate over that connection? In other words, how do you move bits from one to the other? Imagine the kinds of messages your chat client needs to send and receive.



To read data from a Socket, use a BufferedReader

To communicate over a Socket connection, you use streams. Regular old I/O streams, just like we used in the last chapter. One of the coolest features in Java is that most of your I/O work won't care what your high-level chain stream is actually connected to. In other words, you can use a BufferedReader just like you did when you were writing to a file, the difference is that the underlying connection stream is connected to a *Socket* rather than a *File!*



1 Make a Socket connection to the server

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

The port number, which you know because we TOLD you that 5000 is the port number for our chat server.

127.0.0.1 is the IP address for "localhost", in other words, the one this code is running on. You can use this when you're testing your client and server on a single, stand-alone machine.

2 Make an InputStreamReader chained to the Socket's low-level (connection) input stream

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

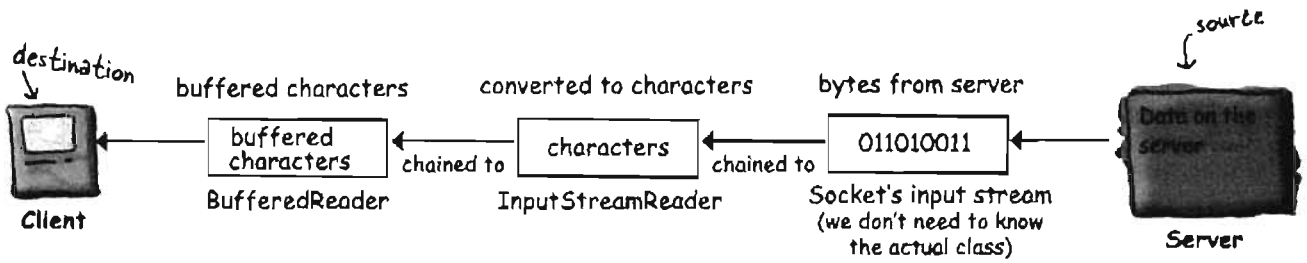
InputStreamReader is a 'bridge' between a low-level byte stream (like the one coming from the Socket) and a high-level character stream (like the BufferedReader we're after as our top of the chain stream).

All we have to do is ASK the socket for an input stream! It's a low-level connection stream, but we're just gonna chain it to something more text-friendly.

3 Make a BufferedReader and read!

Chain the BufferedReader to the InputStreamReader(which was chained to the low-level connection stream we got from the Socket)

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```



To write data to a Socket, use a PrintWriter

We didn't use PrintWriter in the last chapter, we used BufferedWriter. We have a choice here, but when you're writing one String at a time, PrintWriter is the standard choice. And you'll recognize the two key methods in PrintWriter, print() and println()! Just like good ol' System.out.

this part's the same as it was on the opposite page -- to write to the server, we still have to connect to it

1 Make a Socket connection to the server

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

2 Make a PrintWriter chained to the Socket's low-level (connection) output stream

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

PrintWriter acts as its own bridge between character data and the bytes it gets from the Socket's low-level output stream. By chaining a PrintWriter to the Socket's output stream, we can write Strings to the Socket connection.

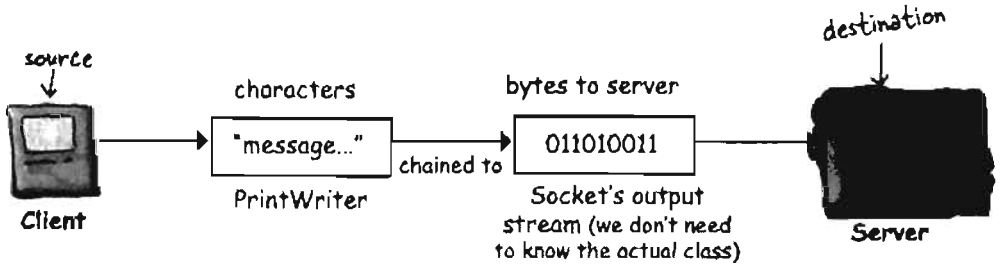
The Socket gives us a low-level connection stream and we chain it to the PrintWriter by giving it to the PrintWriter constructor.

3 Write (print) something

```
writer.println("message to send");
writer.print("another message");
```

println() adds a new line at the end of what it sends.

print() doesn't add the new line.



The DailyAdviceClient

Before we start building the Chat app, let's start with something a little smaller. The Advice Guy is a server program that offers up practical, inspirational tips to get you through those long days of coding.

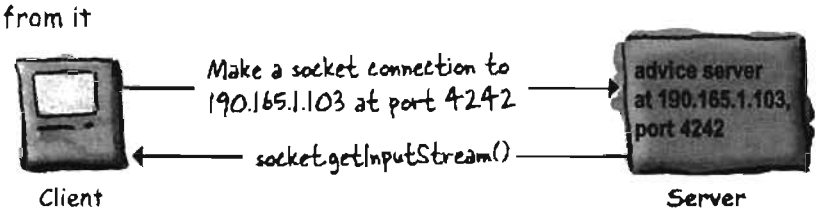
We're building a client for The Advice Guy program, which pulls a message from the server each time it connects.

What are you waiting for? Who *knows* what opportunities you've missed without this app.

The Advice Guy

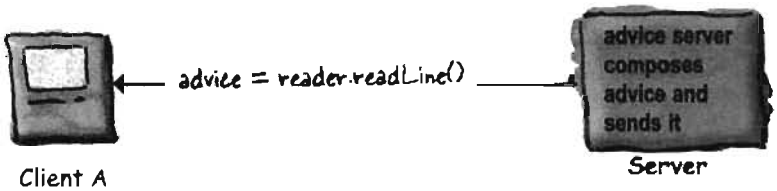
❶ Connect

Client connects to the server and gets an input stream from it



❷ Read

Client reads a message from the server



DailyAdviceClient code

This program makes a Socket, makes a BufferedReader (with the help of other streams), and reads a single line from the server application (whatever is running at port 4242).

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

public class DailyAdviceClient {

   public void go() {
      try {
         Socket s = new Socket("127.0.0.1", 4242);

         InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
         BufferedReader reader = new BufferedReader(streamReader);

         String advice = reader.readLine();
         System.out.println("Today you should: " + advice);

         reader.close();

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

   public static void main(String[] args) {
      DailyAdviceClient client = new DailyAdviceClient();
      client.go();
   }
}
```

class Socket is in java.net

a lot can go wrong here

make a Socket connection to whatever is running on port 4242, on the same host this code is running on. (The 'localhost')

chain a BufferedReader to an InputStreamReader to the input stream from the Socket.

this readLine() is EXACTLY the same as if you were using a BufferedReader chained to a FILE.. In other words, by the time you call a BufferedWriter method, the writer doesn't know or care where the characters came from.

this closes ALL the streams

Sharpen your pencil

Test your memory of the streams/classes for reading and writing from a Socket. Try not to look at the opposite page!

To ***read*** text from a Socket:

Client

write/draw in the chain of streams the client uses to read from the server

source

Server

To ***send*** text to a Socket:

Client

write/draw in the chain of streams the client uses to send something to the server

destination

Server

Sharpen your pencil

Fill in the blanks:

What two pieces of information does the client need in order to make a Socket connection with a server? _____ _____

Which TCP port numbers are reserved for 'well-known services' like HTTP and FTP? _____

TRUE or FALSE: The range of valid TCP port numbers can be represented by a short primitive? _____

Writing a simple server

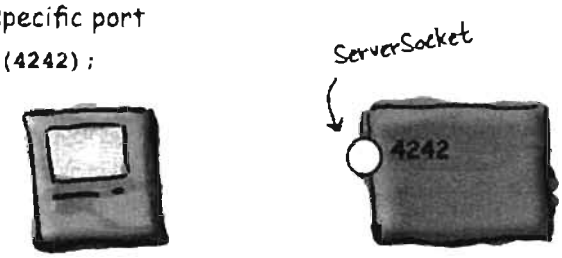
So what's it take to write a server application? Just a couple of Sockets. Yes, a couple as in *two*. A ServerSocket, which waits for client requests (when a client makes a new Socket()) and a plain old Socket socket to use for communication with the client.

How it Works:

1. Server application makes a ServerSocket, on a specific port

```
ServerSocket serverSock = new ServerSocket(4242);
```

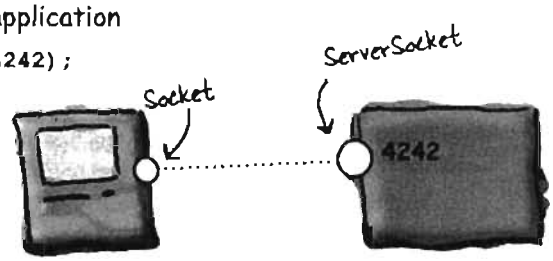
This starts the server application listening for client requests coming in for port 4242.



2. Client makes a Socket connection to the server application

```
Socket sock = new Socket("190.165.1.103", 4242);
```

Client knows the IP address and port number (published or given to him by whomever configures the server app to be on that port)



3. Server makes a new Socket to communicate with this client

```
Socket sock = serverSock.accept();
```

The accept() method blocks (just sits there) while it's waiting for a client Socket connection. When a client finally tries to connect, the method returns a plain old Socket (on a *different* port) that knows how to communicate with the client (i.e., knows the *client's* IP address and port number). The Socket is on a different port than the ServerSocket, so that the ServerSocket can go back to waiting for other clients.