

## 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.

## flow control exercise



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

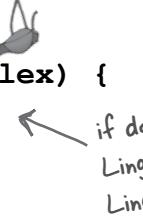


*This method declares two, count 'em, TWO exceptions.*

```
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.*

## polymorphic exceptions

# 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.

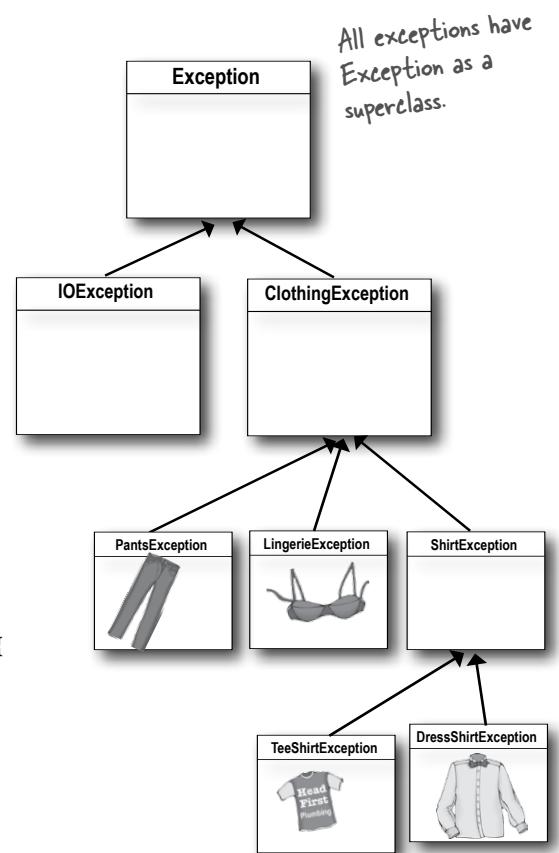
- ① You can **DECLARE** exceptions using a supertype of the exceptions you throw.



```
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.

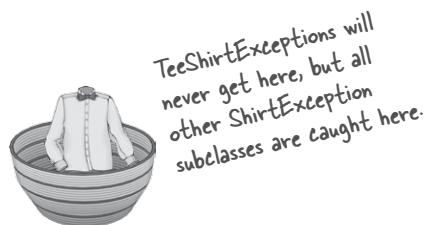
← All other ClothingExceptions are caught here.

## order of multiple catch blocks

**Multiple catch blocks must be ordered from smallest to biggest**



`catch (TeeShirtException tex)`

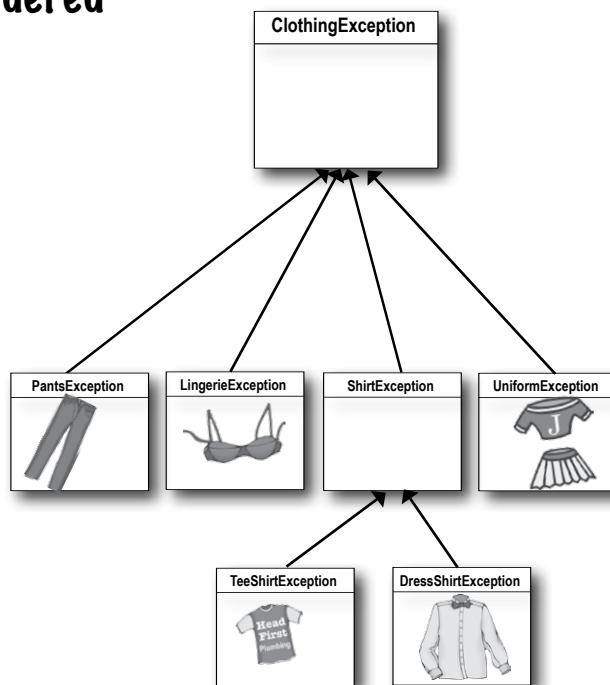


`catch (ShirtException sex)`



`catch (ClothingException cex)`

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



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

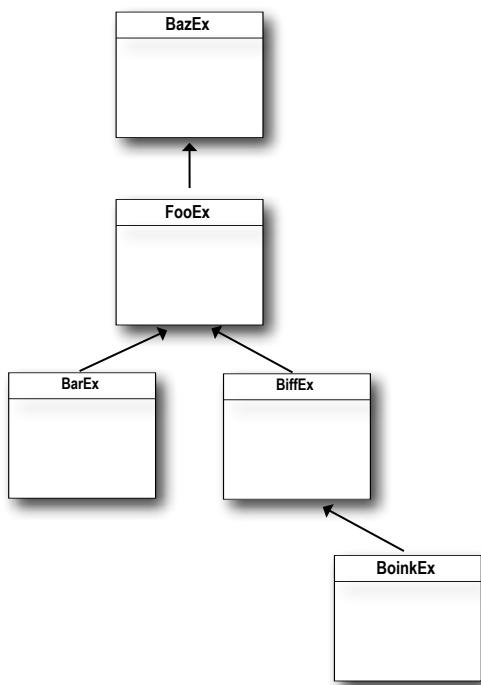


Assume the try/catch block here is legally coded. Your task is to draw two different class diagrams that can accurately reflect the Exception classes. In other words, what class inheritance structures would make the try/catch blocks in the sample code legal?

```
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.

handle or declare

## 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 and the exception is thrown back to main(). But main() doesn't have a try/catch so 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.**

### ① 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.

### ② 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();
    }
}
```

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!

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.

## fixing the Sequencer code

# 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 MidiUnavailableException 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?

## 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.

## 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.

## 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.

## JavaSound MIDI classes

# 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:

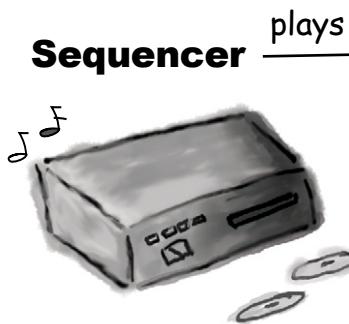
## You need FOUR things:

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.



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**.

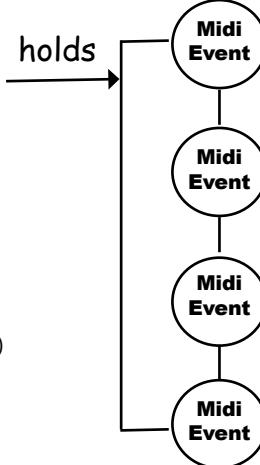
For this book, think of the Sequence as a single-song CD (has only one Track). The information about how to play the song lives on the Track, and the Track is part of the Sequence.

Sequencer → Sequence → Track

plays → has a → holds

Track

For this book, we only need one Track, so just imagine 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."

## 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);
```



```
player.start();
```

## a sound application

# 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 *someone*, but *something*.)

```
import javax.sound.midi.*;      ← Don't forget to import the midi package
public class MiniMiniMusicApp {
    public static void main(String[] args) {
        MiniMiniMusicApp mini = new MiniMiniMusicApp();
        mini.play();
    } // close main
    public void play() {
        try {
            ① Sequencer player = MidiSystem.getSequencer();
                player.open();          ← get a Sequencer and open it
                                         (so we can use it... a Sequencer
                                         doesn't come already open)
            ② Sequence seq = new Sequence(Sequence.PPQ, 4);   ← Don't worry about the arguments to the
                                                       Sequence constructor. Just copy these (think
                                                       of 'em as Ready-bake arguments).
            ③ Track track = seq.createTrack();               ← Ask the Sequence for a Track. Remember, the
                                                       Track lives in the Sequence, and the MIDI data
                                                       lives in the Track.
            ④ 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);           ← Give the Sequence to the Sequencer (like
                                              putting the CD in the CD player)
            player.start();                  ← Start() the Sequencer (like pushing PLAY)
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } // close play
} // close class
```



## 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.

## contents of a Midi event

# 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.

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

```
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.

### ① Message type



### ② 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.

### ③ Note to play

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



### ④ 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.

### ① Change the note

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

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



### ② 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);
```



### 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
```

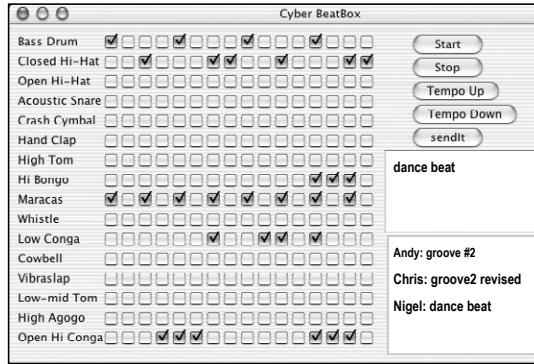
Run it with two int args from 0 to 127. Try these for starters:

```
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

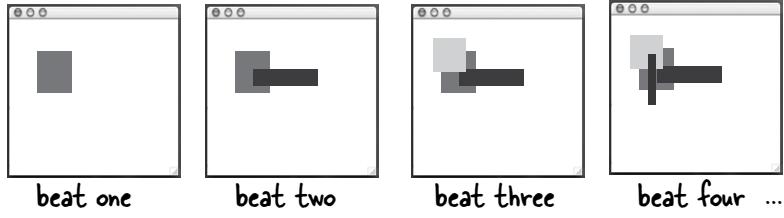
### Chapter 15: the goal

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.



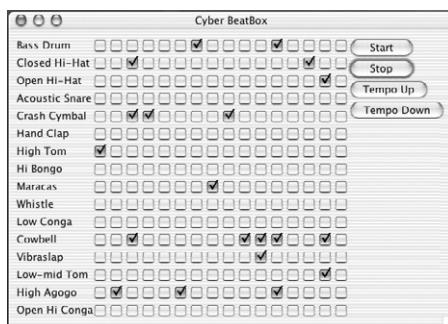
### Chapter 12: MIDI events

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).



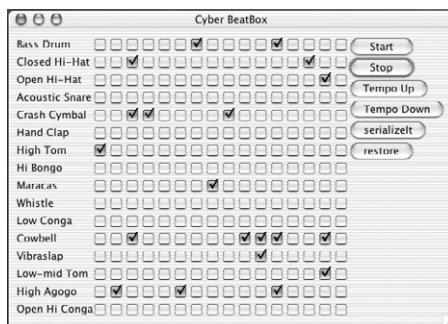
### Chapter 13: Stand-alone BeatBox

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*.



exception handling

## 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!

```
System.out.print("r");
try {
    System.out.print("t");
    doRisky(test);
} 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];
```

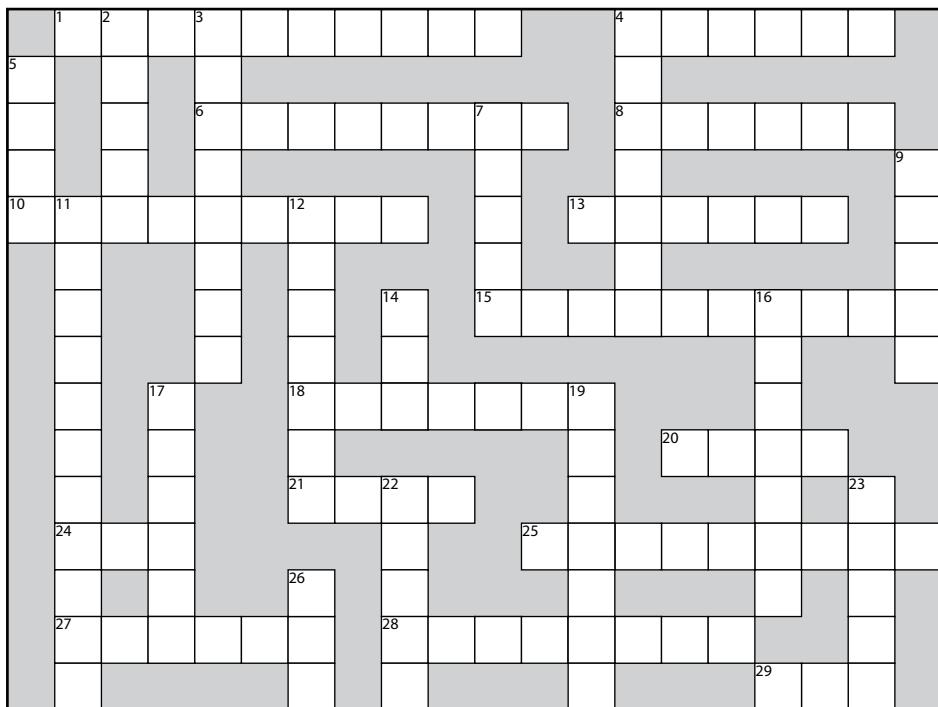
```
File Edit Window Help ThrowUp
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

**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'

**Down**

- 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

**Down**

- 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. Duck                     | 27. Starts a problem | 28. Not Abstract       |
| 6. A Java child   | 2. Or a mouthwash             | 3. For _____ (not example)   | 17. Not a getter     | 13. Instead of declare |
| 8. Start a method | 9. Only public or default     | 16. _____ the family fortune | 5. Numbers ...       | 20. Starts a problem   |



## 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 synonymous 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
```

**puzzle answers**

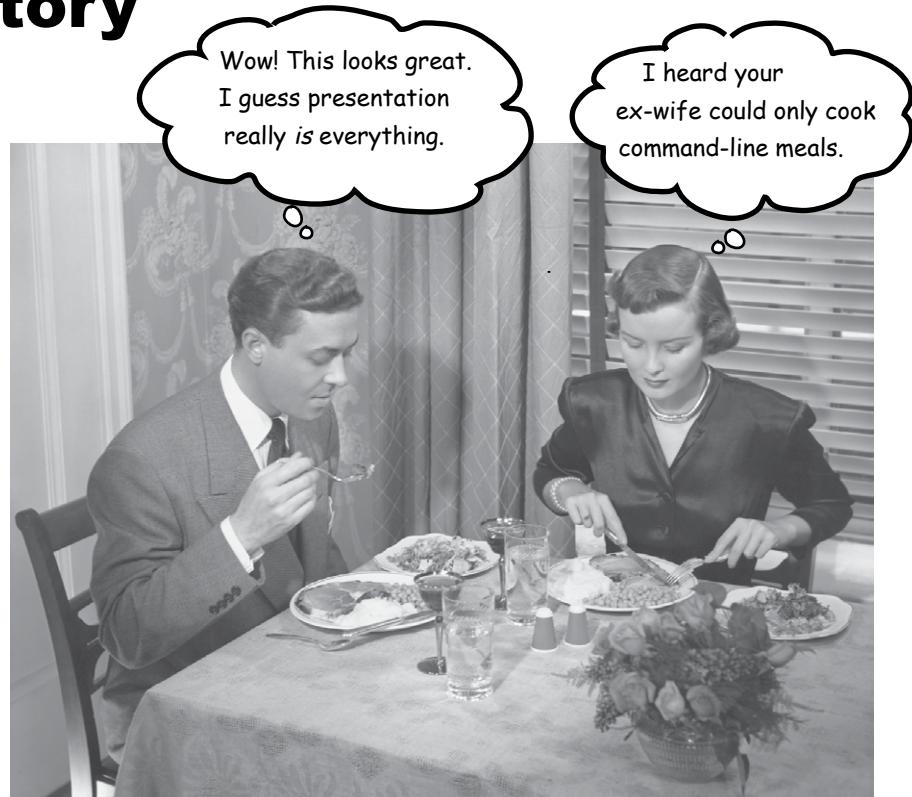


# JavaCross Answers

	<sup>1</sup> A	<sup>2</sup> S	<sup>3</sup> S	<sup>4</sup> I	<sup>5</sup> G	<sup>6</sup> N	<sup>7</sup> M	<sup>8</sup> E	<sup>9</sup> N	<sup>10</sup> T	<sup>11</sup> P	<sup>12</sup> O	<sup>13</sup> C	<sup>14</sup> N	<sup>15</sup> V	<sup>16</sup> R	<sup>17</sup> H	<sup>18</sup> I	<sup>19</sup> E	<sup>20</sup> C	<sup>21</sup> A	<sup>22</sup> O	<sup>23</sup> E	<sup>24</sup> T	<sup>25</sup> A	<sup>26</sup> T	<sup>27</sup> H	<sup>28</sup> C	<sup>29</sup> N	
	<sup>1</sup> A	<sup>2</sup> S	<sup>3</sup> S	<sup>4</sup> I	<sup>5</sup> G	<sup>6</sup> N	<sup>7</sup> M	<sup>8</sup> E	<sup>9</sup> N	<sup>10</sup> T	<sup>11</sup> P	<sup>12</sup> O	<sup>13</sup> C	<sup>14</sup> N	<sup>15</sup> V	<sup>16</sup> R	<sup>17</sup> H	<sup>18</sup> I	<sup>19</sup> E	<sup>20</sup> C	<sup>21</sup> A	<sup>22</sup> O	<sup>23</sup> E	<sup>24</sup> T	<sup>25</sup> A	<sup>26</sup> T	<sup>27</sup> H	<sup>28</sup> C	<sup>29</sup> N	
	<sup>1</sup> A	<sup>2</sup> S	<sup>3</sup> S	<sup>4</sup> I	<sup>5</sup> G	<sup>6</sup> N	<sup>7</sup> M	<sup>8</sup> E	<sup>9</sup> N	<sup>10</sup> T	<sup>11</sup> P	<sup>12</sup> O	<sup>13</sup> C	<sup>14</sup> N	<sup>15</sup> V	<sup>16</sup> R	<sup>17</sup> H	<sup>18</sup> I	<sup>19</sup> E	<sup>20</sup> C	<sup>21</sup> A	<sup>22</sup> O	<sup>23</sup> E	<sup>24</sup> T	<sup>25</sup> A	<sup>26</sup> T	<sup>27</sup> H	<sup>28</sup> C	<sup>29</sup> N	
	<sup>1</sup> A	<sup>2</sup> S	<sup>3</sup> S	<sup>4</sup> I	<sup>5</sup> G	<sup>6</sup> N	<sup>7</sup> M	<sup>8</sup> E	<sup>9</sup> N	<sup>10</sup> T	<sup>11</sup> P	<sup>12</sup> O	<sup>13</sup> C	<sup>14</sup> N	<sup>15</sup> V	<sup>16</sup> R	<sup>17</sup> H	<sup>18</sup> I	<sup>19</sup> E	<sup>20</sup> C	<sup>21</sup> A	<sup>22</sup> O	<sup>23</sup> E	<sup>24</sup> T	<sup>25</sup> A	<sup>26</sup> T	<sup>27</sup> H	<sup>28</sup> C	<sup>29</sup> N	
	<sup>1</sup> A	<sup>2</sup> S	<sup>3</sup> S	<sup>4</sup> I	<sup>5</sup> G	<sup>6</sup> N	<sup>7</sup> M	<sup>8</sup> E	<sup>9</sup> N	<sup>10</sup> T	<sup>11</sup> P	<sup>12</sup> O	<sup>13</sup> C	<sup>14</sup> N	<sup>15</sup> V	<sup>16</sup> R	<sup>17</sup> H	<sup>18</sup> I	<sup>19</sup> E	<sup>20</sup> C	<sup>21</sup> A	<sup>22</sup> O	<sup>23</sup> E	<sup>24</sup> T	<sup>25</sup> A	<sup>26</sup> T	<sup>27</sup> H	<sup>28</sup> C	<sup>29</sup> N	

## 12 getting gui

# 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.

## your first gui

### 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:



### 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.

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



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

### 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.*;           ← don't forget to import this
                                swing package

public class SimpleGuil {
    public static void main (String[] args) {
        JFrame frame = new JFrame();      ← make a frame and a button
        JButton button = new JButton("click me");   ← (you can pass the button constructor)
                                                    the text you want on the button

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                ← 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)

        frame.getContentPane().add(button);
                                                ← add the button to the frame's
                                                content pane

        frame.setSize(300,300);          ← 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)
    }
}

```

*Annotations:*

- `import javax.swing.*;`: don't forget to import this swing package
- `JFrame frame = new JFrame();`: make a frame and a button (you can pass the button constructor)
- `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`: 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)
- `frame.getContentPane().add(button);`: add the button to the frame's content pane
- `frame.setSize(300,300);`: give the frame a size, in pixels
- `frame.setVisible(true);`: 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 SimpleGuil



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.

there are no  
Dumb Questions

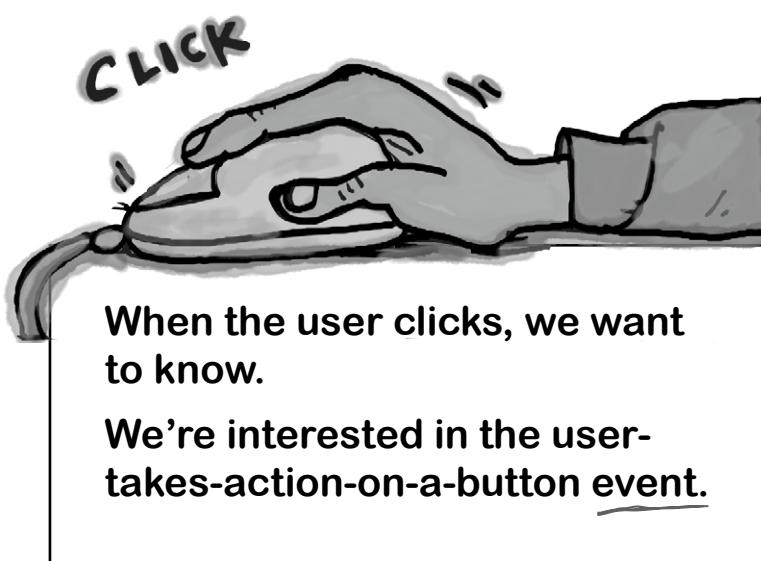
## But nothing happens when I click it...

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!



**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):

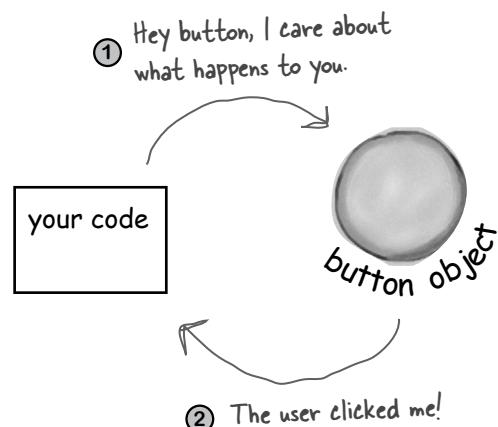
```
public void changeIt() {
    button.setText("I've been clicked!");
}
```

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]

## event listeners

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).

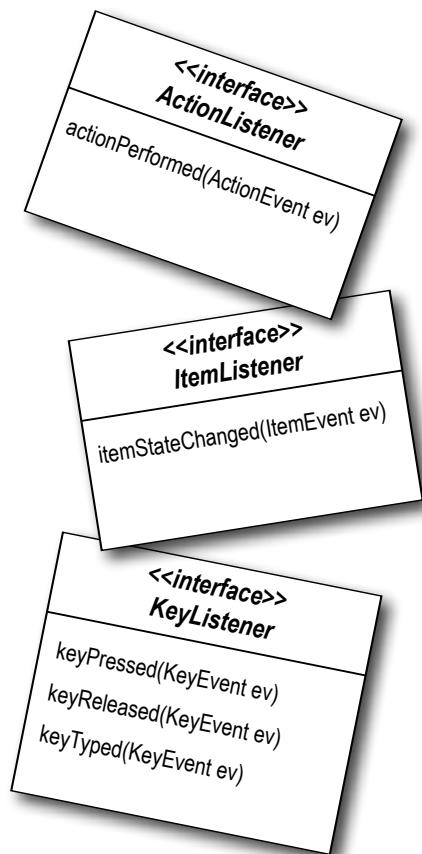
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through 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*.

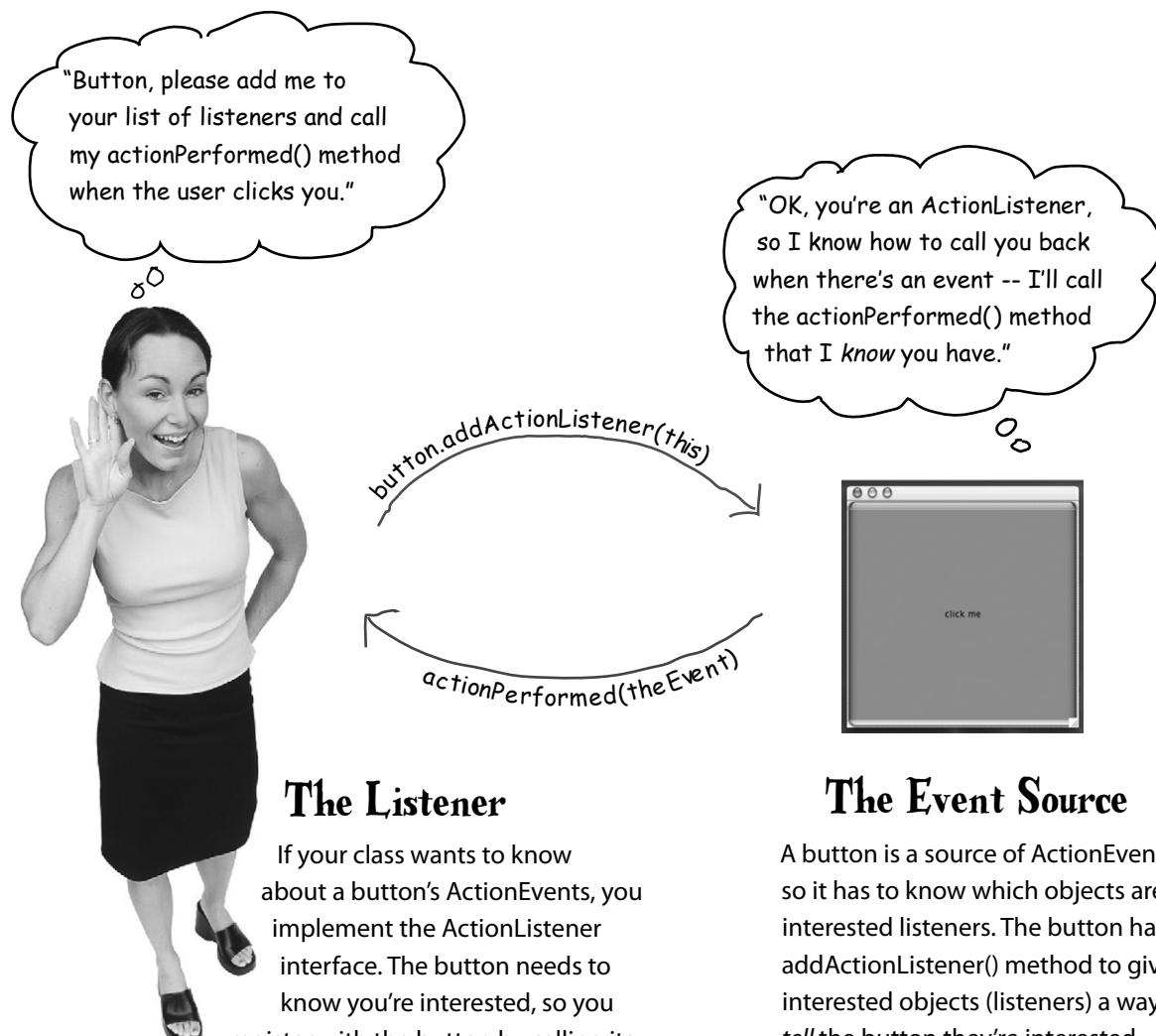
Every event type has a matching listener interface. If you want `MouseEvents`, implement the `MouseListener` interface. Want `WindowEvents`? Implement `WindowListener`. You get the idea. And remember your interface rules—to implement an interface you *declare* that you implement it (class `Dog` implements `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 events

### 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 interface)

```
import javax.swing.*;
import java.awt.event.*; // a new import statement for the package that
// ActionListener and ActionEvent are in.

public class SimpleGuilB implements ActionListener { // ①
    JButton button;

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

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

        // ② button.addActionListener(this); // 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!!
        frame.getContentPane().add(button);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,300);
        frame.setVisible(true);
    }

    // ③
    public void actionPerformed(ActionEvent event) {
        button.setText("I've been clicked!");
    }
}
```

Implement the interface. This says, "an instance of SimpleGuilB 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.

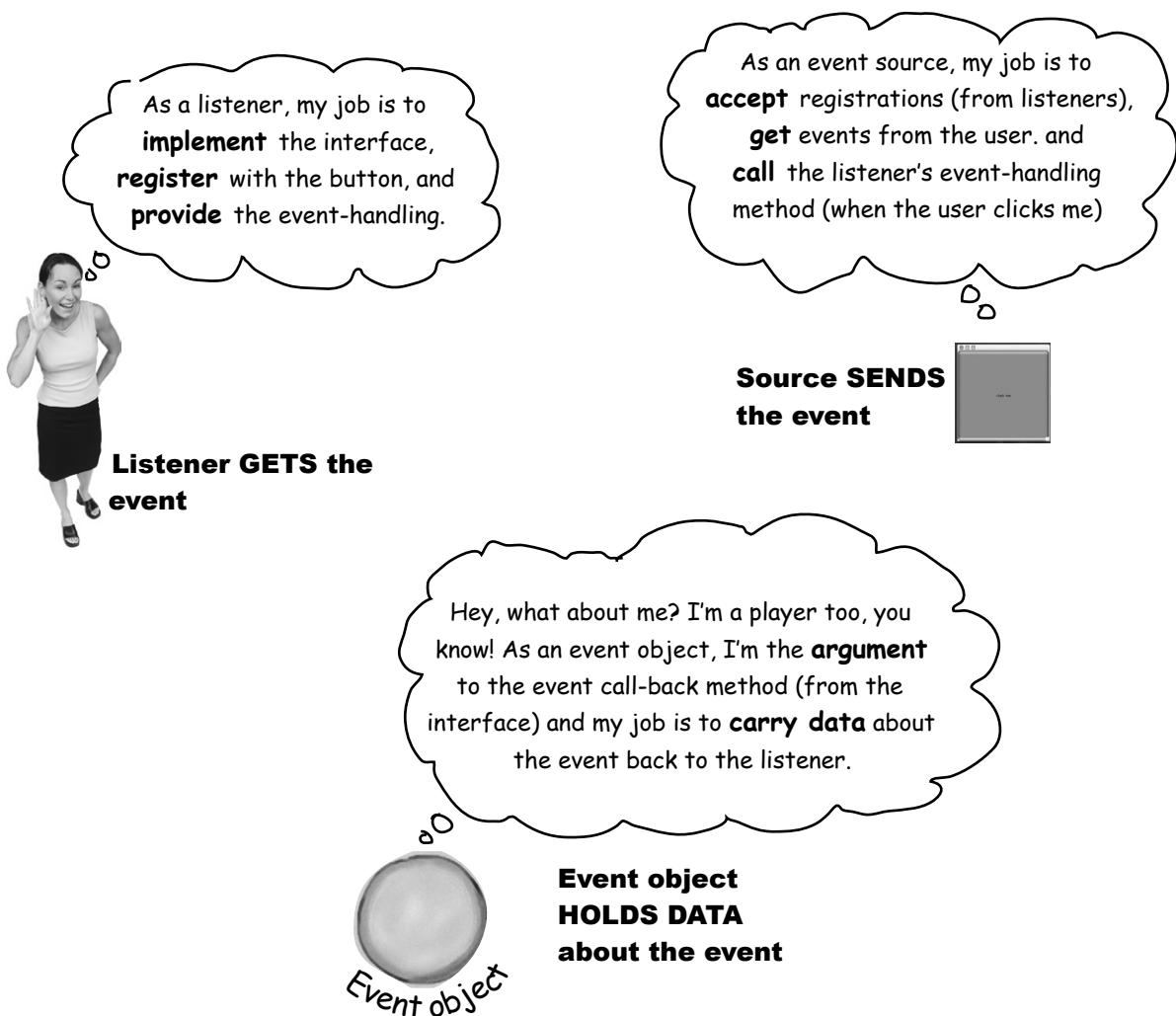
## Listeners, Sources, and Events

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.)



## event handling

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

- check box
- text field
- scrolling list
- button
- dialog box
- radio button
- menu item

#### Event methods

- `windowClosing()`
- `actionPerformed()`
- `itemStateChanged()`
- `mousePressed()`
- `keyTyped()`
- `mouseExited()`
- `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:

#### ① 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.

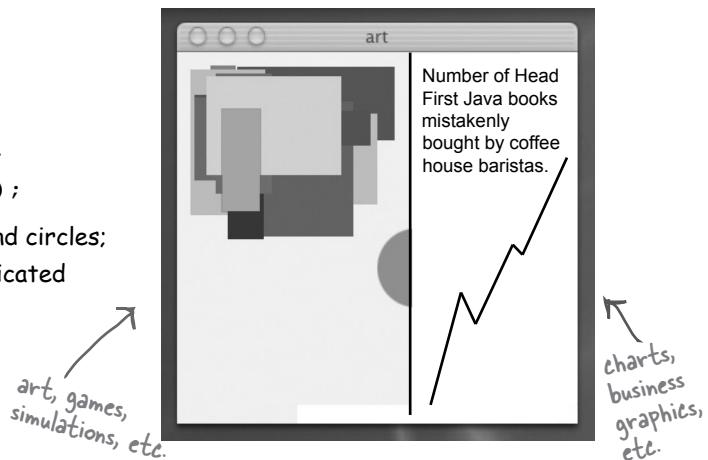


#### ② Draw 2D graphics on a widget

Use a graphics object to paint shapes.

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

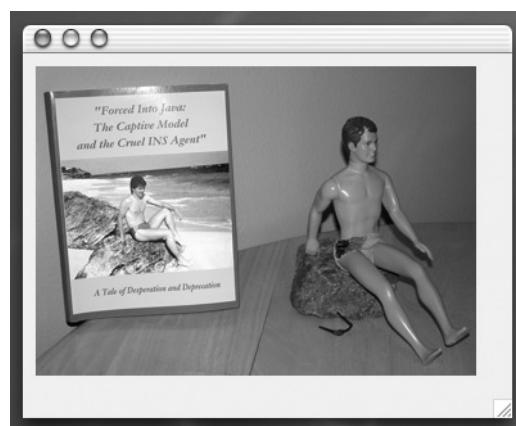
You can paint a lot more than boxes and circles; the Java2D API is full of fun, sophisticated graphics methods.



#### ③ Put a JPEG on a widget

You can put your own images on a widget.

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



## making a drawing panel

# Make your own drawing widget

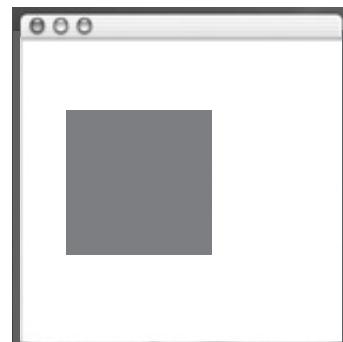
If you want to put your own graphics on the screen, your best bet is to make your own paintable widget. You plop that widget on the frame, just like a button or any other widget, but when it shows up it will have your images on it. You can even make those images move, in an animation, or make the colors 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.*;           ← you need both of these
import javax.swing.*;          ←
class MyDrawPanel extends JPanel {
    public void paintComponent(Graphics g) {           ← This is the Big Important Graphics method.
        g.setColor(Color.orange);                      ← You will NEVER call this yourself. The
        g.fillRect(20,50,100,100);                     ← system calls it and says, "Here's a nice
                                                       ← fresh drawing surface, of type Graphics,
                                                       ← that you may paint on now."
    }
}
```

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.

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. Note: If you're using an IDE and have difficulty, try this line of code instead:  
`Image image = new ImageIcon(getClass().getResource("catzilla.jpg")).getImage();`

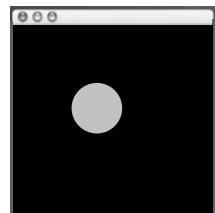


The x,y coordinates for where the picture's top left corner should go. This says "3 pixels from the left edge of the panel and 4 pixels from the top edge of the panel". These numbers are always relative to the widget (in this case your JPanel subclass), not the entire frame.

### 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() * 256);
    int green = (int) (Math.random() * 256);
    int blue = (int) (Math.random() * 256);
    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 top, make it 100 pixels wide, and 100 pixels tall.

## drawing gradients with Graphics2D

# 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 CANNOT 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:**

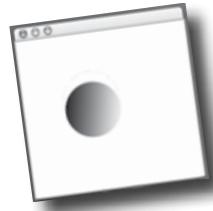
```
Graphics2D g2d = (Graphics2D) g;
```

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

    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);
    starting ↑      starting ↑      ending ↑      ending ↑
    point          color        point       color
    g2d.setPaint(gradient); this sets the virtual paint brush to a
    gradient instead of a solid color
    g2d.fillOval(70,70,100,100);

    }
}

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() * 256);
    int green = (int) (Math.random() * 256);
    int blue = (int) (Math.random() * 256);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 256);
    green = (int) (Math.random() * 256);
    blue = (int) (Math.random() * 256);
    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:  
`g.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 image 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;`

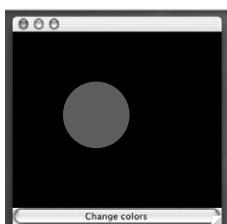
We can get an event.

We can paint graphics.

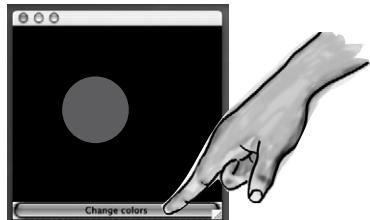
But can we paint graphics when we get an event?

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

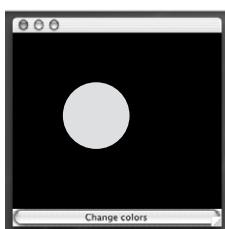


- 1 The frame is built with the two widgets (your drawing panel and a button). A listener is created and registered with the button. Then the frame is displayed and it just waits for the user to click.



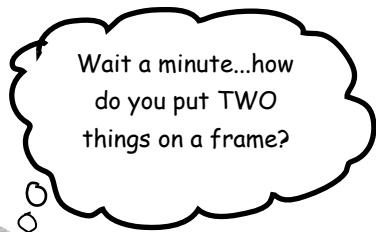
- 2 The user clicks the button and the button creates an event object and calls the listener's event handler.

- 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.

## building a GUI frame



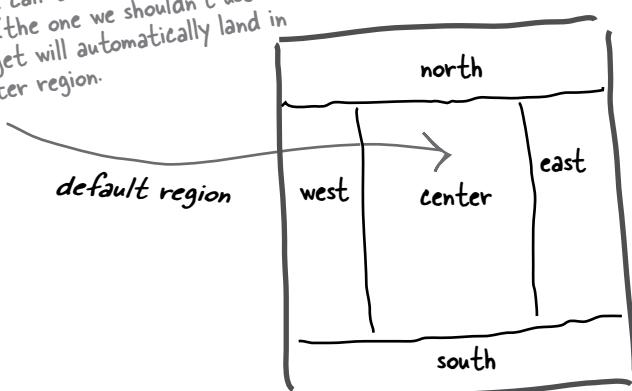
## GUI layouts: putting more than one widget on a frame

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 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.



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

```
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 369, write the code that adds the button and the panel to the frame.

## The circle changes color each time you click the button.

```

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();
    }

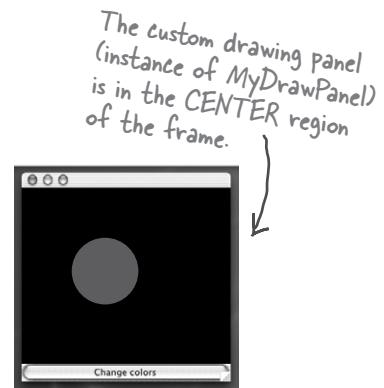
    public void go() {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        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();
    }
}

```



Button is in the SOUTH region of the frame

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 367 for the code
    }
}

```

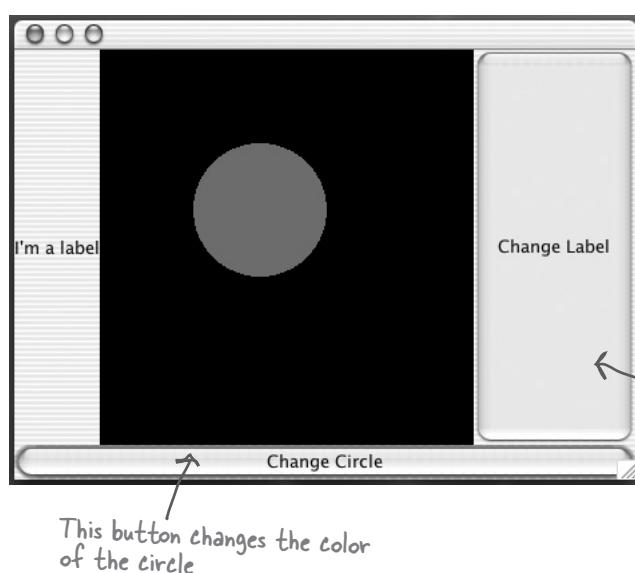
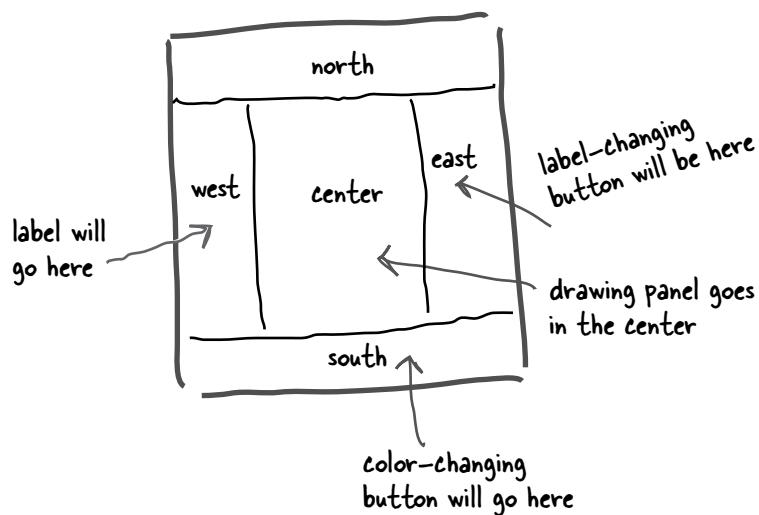
The drawing panel's paintComponent() method is called every time the user clicks.

**multiple listeners**

## 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?

### ① 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?

### ② 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); ← Register the same listener
        labelButton.addActionListener(this); ← with both buttons
        // more gui code here ...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint(); ← Query the event object
        } else {           to find out which button
            label.setText("That hurt!");   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.

multiple listeners

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

③ **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 classes

# 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 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.

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.

### Inner class using an outer class variable

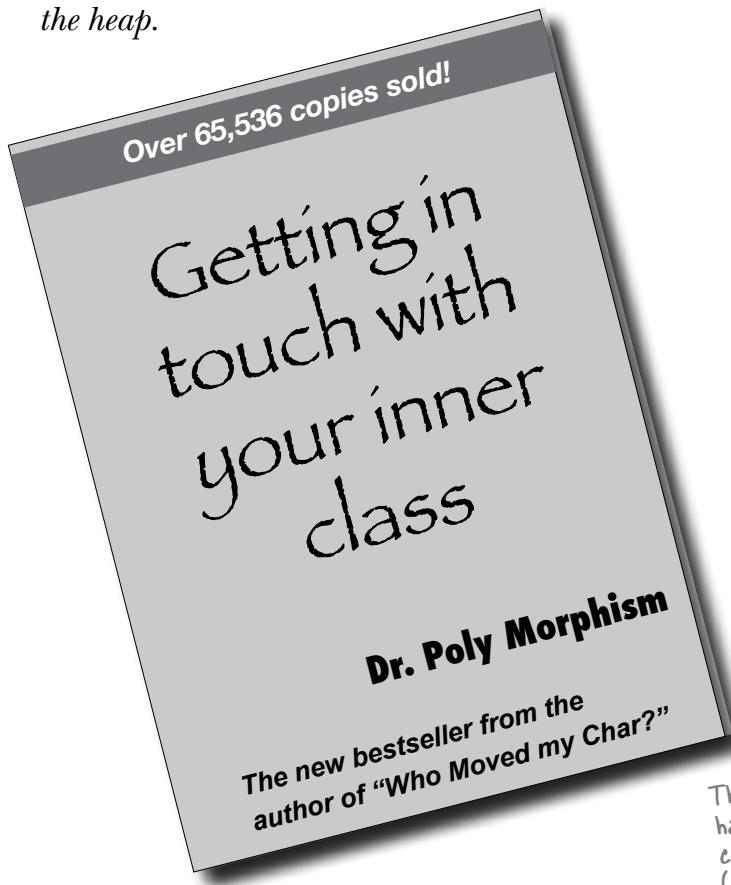
```
class MyOuterClass {  
  
    private int x;  
  
    class MyInnerClass {  
        void go() {  
            x = 42; ← use 'x' as if it were a variable  
        }  
    } // close inner class  
  
} // close outer class
```

## 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

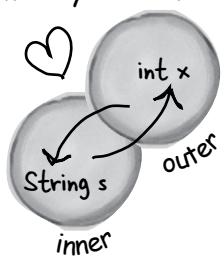


- ② Make an instance of the inner class, by using the instance of the outer class.



- 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.

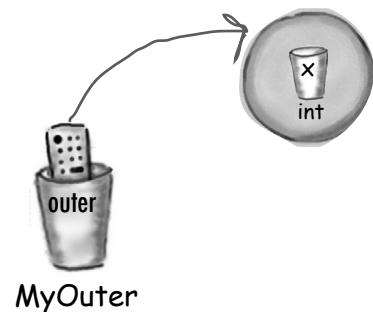
## inner class instances

# How to make an instance of an inner class

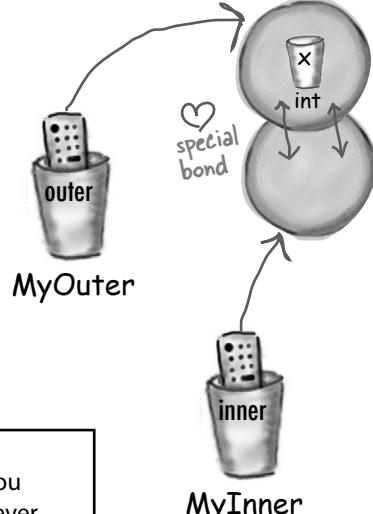
If you instantiate an inner class from code *within* an outer class, the instance of the outer class is the one that the inner object will ‘bond’ with. For example, if code within a method instantiates the inner class, the inner object will bond to the instance whose method is running.

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;           ← The outer class has a private  
    MyInner inner = new MyInner();   instance variable 'x'  
    public void doStuff() {  
        inner.go();          ← Make an instance of the  
    }                         inner class  
    class MyInner {  
        void go() {  
            x = 42;           ← call a method on the  
        } // close inner class  
    } // close outer 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);
```

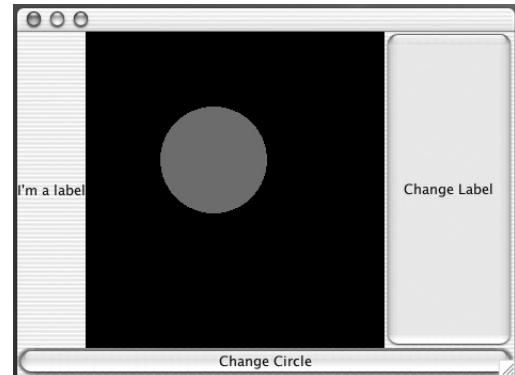
```
    frame.setSize(300,300);
    frame.setVisible(true);
```

```
}
```

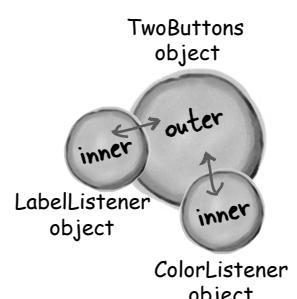
```
class LabelListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("Ouch!");
    }
} // close inner class
```

```
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

## inner classes



## 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... hmmmm. 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?

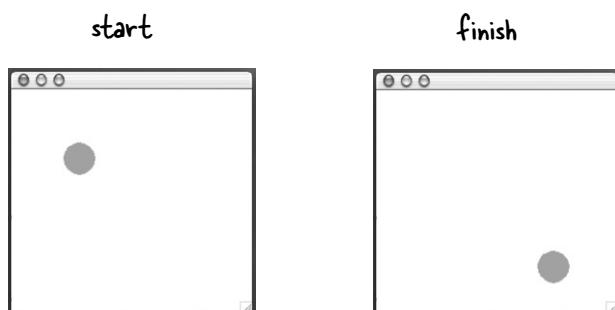
**HeadFirst:** Well, it does make sense. And let's not forget, it was *you* extolling the virtues of "multiple implementations of the same interface". So it makes sense that if the outer class has three buttons, it would need three different inner classes (and thus three different inner class objects) 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*

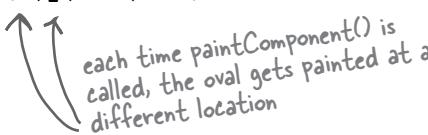
**Q:** Why are we learning about animation here? I doubt if I'm going to be making games.

**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:**

## animation using an inner class

### The complete simple animation code

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

public class SimpleAnimation {
    int x = 70; } ← make two instance variables in the
    int y = 70; } ← main GUI class, for the x and y
                    coordinates of the circle.

    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);

    This is where the
    action is!
    for (int i = 0; i < 130; i++) {           repeat this 130 times
        x++; ← increment the x and y
        y++; ← coordinates
        drawPanel.repaint(); ← tell the panel to repaint itself (so we
                            can see the circle in the new location)
        try {
            Thread.sleep(50);
        } catch(Exception ex) { }
    }
} // close go() method

Now it's an
inner class.
class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {
        g.setColor(Color.green);
        g.fillOval(x,y,40,40);           Use the continually-updated x and y
                                         coordinates of the outer class.
    }
} // close inner class
} // close outer class
```

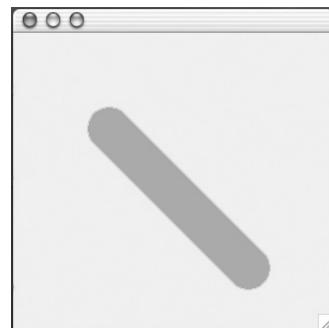
## Uh-oh. It didn't move... it smeared.

What did we do wrong?

There's one little flaw in the `paintComponent()` method.

## 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());

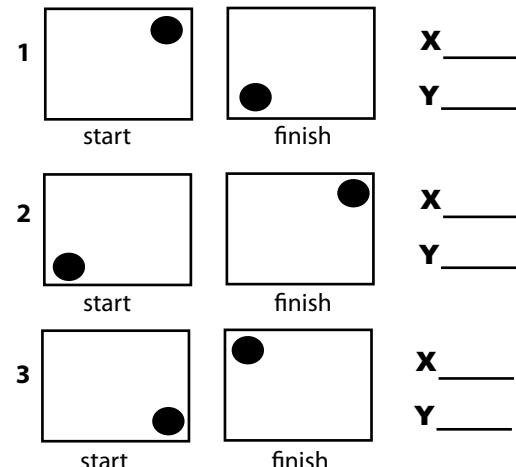
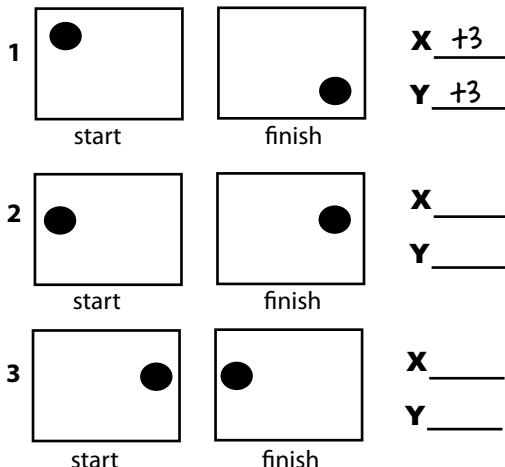
    g.setColor(Color.green);
    g.fillOval(x,y,40,40);
}
```

*getWidth() and getHeight() are methods inherited from JPanel.*

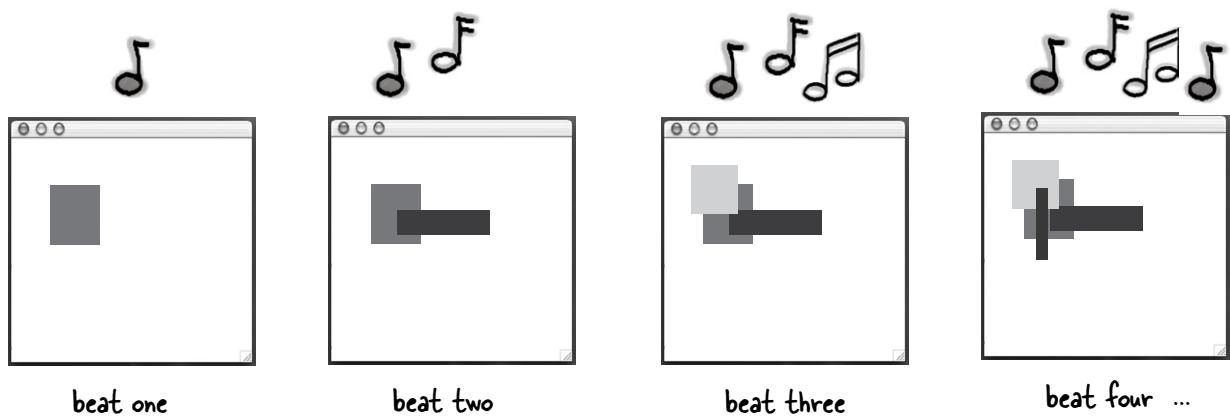


### Sharpen your pencil (optional, just for fun)

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



beat one

beat two

beat three

beat four ...

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:

- ① Make a series of MIDI messages/ events to play random notes on a piano (or whatever instrument you choose)
- ② Register a listener for the events
- ③ Start the sequencer playing
- ④ 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:

- ① Version One: Code that simplifies making and adding MIDI events, since we'll be making a lot of them.
- ② Version Two: Register and listen for the events, but without graphics. Prints a message at the command-line with each beat.
- ③ 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;
}
```

whoo! A method with five parameters.

the four arguments for the message

The event 'tick' for WHEN this message should happen

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.*; ← don't forget the import
public class MiniMusicPlayer1 {

    public static void main(String[] args) {

        try {
            Sequencer sequencer = MidiSystem.getSequencer(); ← make (and open) a sequencer
            sequencer.open(); ← and a track

            Sequence seq = new Sequence(Sequence.PPQ, 4); ← make a sequence
            Track track = seq.createTrack(); ← and a track

            for (int i = 5; i < 61; i+= 4) { ← make a bunch of events to make the notes keep
                going up (from piano note 5 to piano note 61)
                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); } start it running
            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

```

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

controller events

## Version Two: registering and getting ControllerEvents

```
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();

            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
```

We need to listen for ControllerEvents,  
so we implement the listener interface

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.

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 NOTH-  
ING! 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 Controller-  
Event 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; ← We set a flag to false, and we'll set it
                           to true only when we get an event.

    public void controlChange(ShortMessage event) {
        msg = true; ← We got an event, so we set the flag to
                      true and call repaint()
    }

    public void paintComponent(Graphics g) {
        if (msg) { ← We have to use a flag because OTHER things might trigger a repaint(),
                   and we want to paint ONLY when there's a ControllerEvent
        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250); ← The rest is code to generate
                                                a random color and paint a
                                                semi-random rectangle.

        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, width, ht);
        msg = false;

    } // close if
} // close method
} // close inner class

```

### MiniMusicPlayer3 code



This is the complete code listing for Version Three. It builds directly 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
}
```

```

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 method

class MyDrawPanel extends JPanel implements ControllerEventListener {
    boolean msg = false;

    public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
    }

    public void paintComponent(Graphics g) {
        if (msg) {

            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);

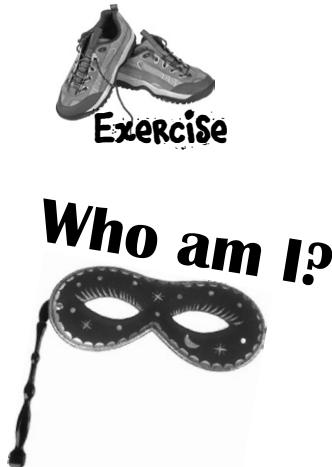
            g.fillRect(x,y,ht, width);
            msg = false;

        } // close if
    } // close method
} // close inner class

} // close class

```

**exercise:** 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.

---



```

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");
            }
        }
    }
}

```

## 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?



## puzzle: Pool Puzzle



## Pool Puzzle



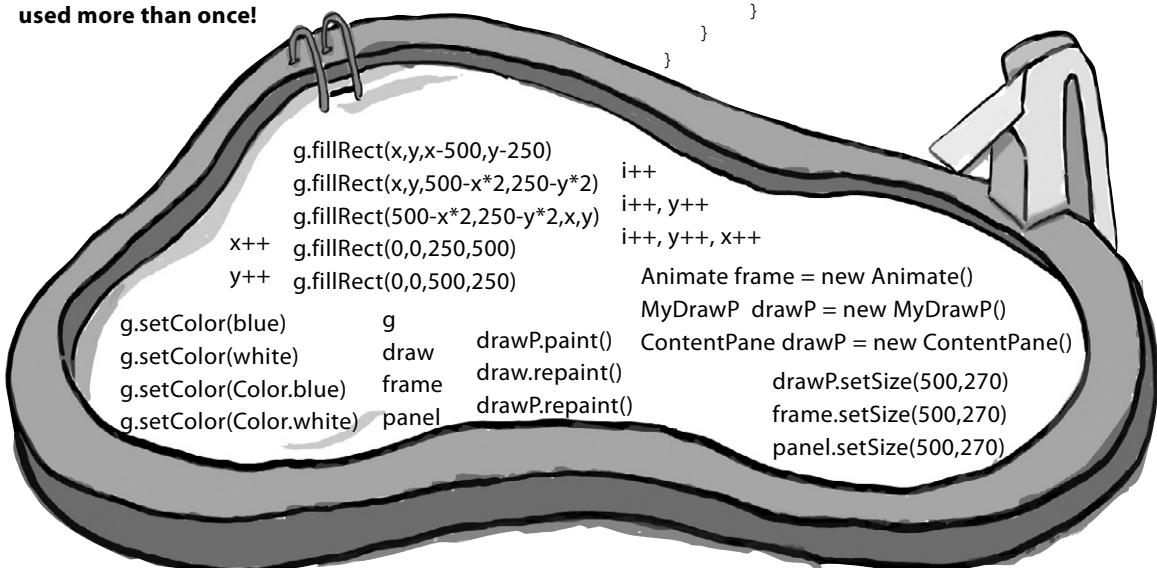
Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

### Output

The Amazing, Shrinking, Blue Rectangle. This program will produce a blue rectangle that will shrink and shrink and disappear into a field of white.



**Note: Each snippet from the pool can be used more than once!**



```
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 _____ = new JFrame();
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        _____;
        _____.getContentPane().add(drawP);
        _____;
        _____.setVisible(true);
        for (int i=0; i<124; _____) {
            _____;
            _____;
        }
        try {
            Thread.sleep(50);
        } catch(Exception ex) { }
    }
    class MyDrawP extends JPanel {
        public void paintComponent (Graphics
            _____) {
            _____;
            _____;
            _____;
            _____;
        }
    }
}
```



## Exercise Solutions

### Who am I?

I got the whole GUI, in my hands.	<b>JFrame</b>
Every event type has one of these.	<b>listener interface</b>
The listener's key method.	<b>actionPerformed( )</b>
This method gives JFrame its size.	<b>setSize( )</b>
You add code to this method but never call it.	<b>paintComponent( )</b>
When the user actually does something, it's an _____	<b>event</b>
Most of these are event sources.	<b>swing components</b>
I carry data back to the listener.	<b>event object</b>
An addXxxListener( ) method says an object is an _____	<b>event source</b>
How a listener signs up.	<b>addActionListener( )</b>
The method where all the graphics code goes.	<b>paintComponent( )</b>
I'm typically bound to an instance.	<b>inner class</b>
The 'g' in (Graphics g), is really of this class.	<b>Graphics2D</b>
The method that gets paintComponent( ) rolling.	<b>repaint( )</b>
The package where most of the Swingers reside.	<b>javax.swing</b>

## BE the compiler

```
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);
```

The **addActionListener()** method takes a class that implements the **ActionListener** interface

```
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");
        }
    }
}
```

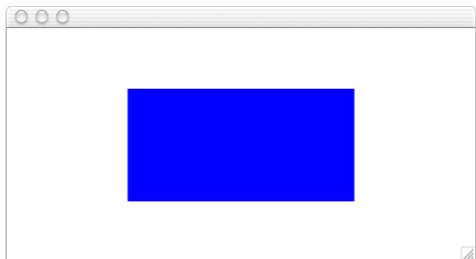
ActionListener is an interface, interfaces are implemented, not extended

**puzzle answers**



## Poo! Puzzle

The Amazing, Shrinking, Blue Rectangle.



```
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++,y++,x++ ) {
            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);
        }
    }
}
```

## 13 using swing

# 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.

## components and containers

# 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`.

## 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.

A **widget** is technically a **Swing Component**.  
Almost every thing you can stick in a GUI extends from `javax.swing.JComponent`.

## Four steps to making a GUI (review)

- ① Make a window (a JFrame)

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

- ② Make a component (button, text field, etc.)

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

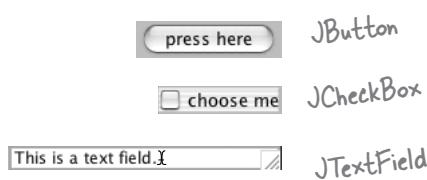
- ③ Add the component to the frame

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

- ④ 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.

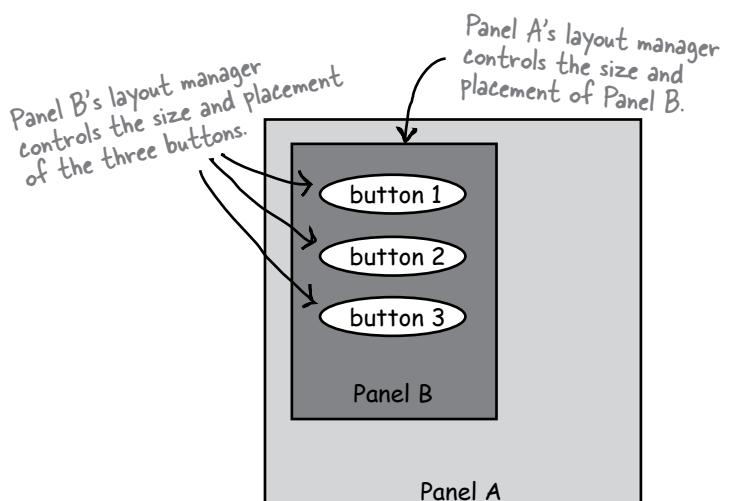
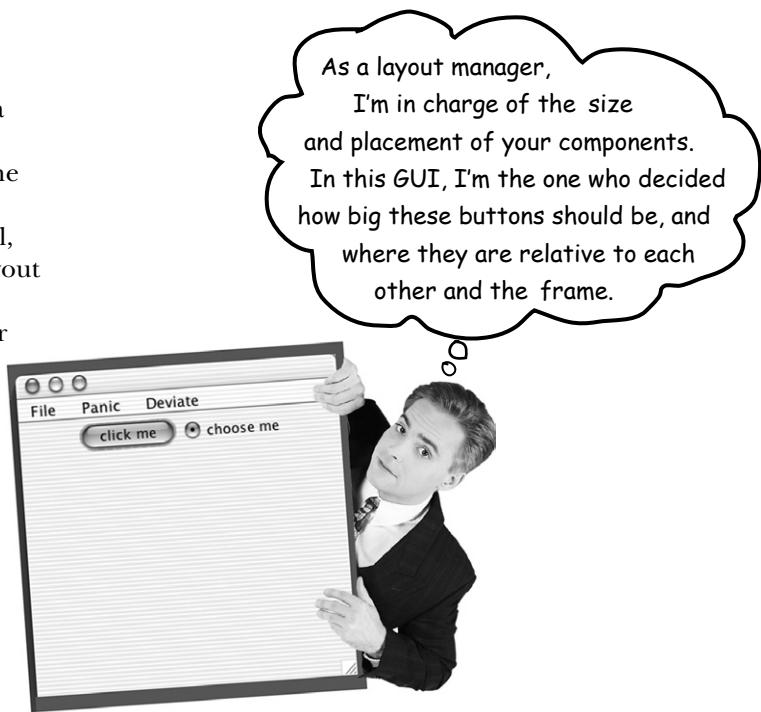
If a panel holds five things, even if those five things each have their own layout managers, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things, then those *other* things are placed according to the 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.

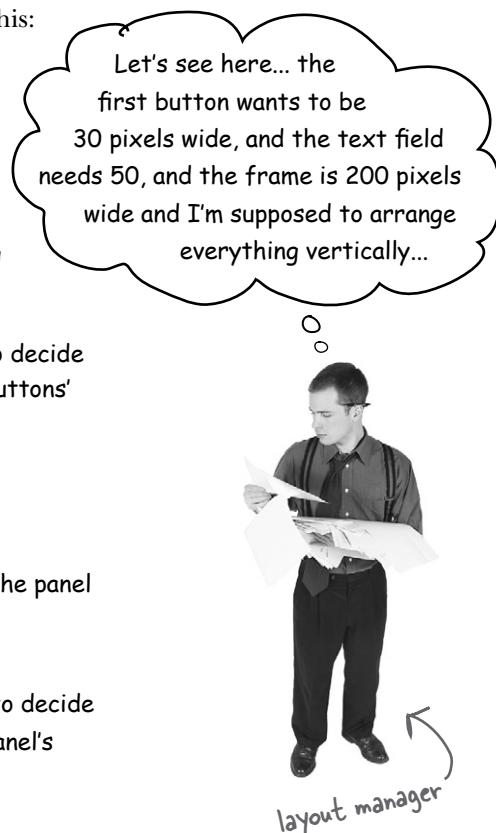
## layout managers

# 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 laid 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.



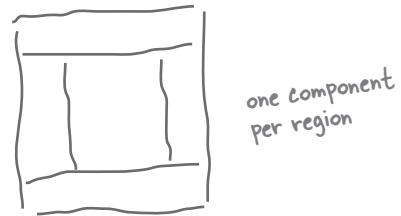
## Different layout managers have different policies

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 laid 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.

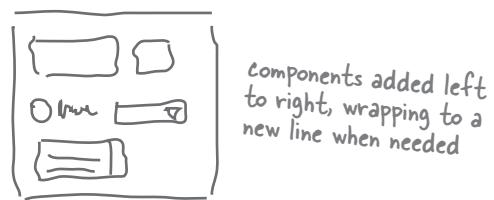
## BorderLayout

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!**



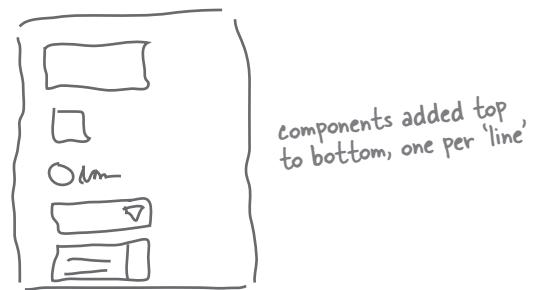
## FlowLayout

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!**

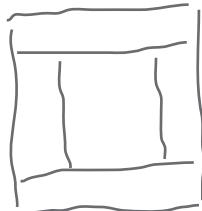


## BoxLayout

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.



border layout



**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

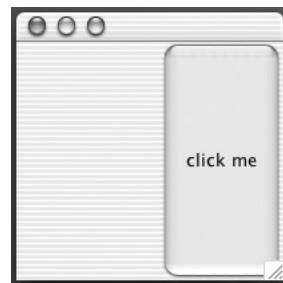


## Brain Barbell

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...

```
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);
}
```

We changed only the text on the button

First, I ask the button for its preferred size.



I have a lot of words now, so I'd prefer to be 60 pixels wide and 25 pixels tall.



Button object

Since it's in the east region of a border layout, I'll respect its preferred width. But I don't care how tall it wants to be; it's gonna be as tall as the frame, because that's my policy.



The button gets its preferred width, but not height.

Button object

Next time I'm goin' with flow layout. Then I get EVERYTHING I want.



Button object

## border layout

### 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

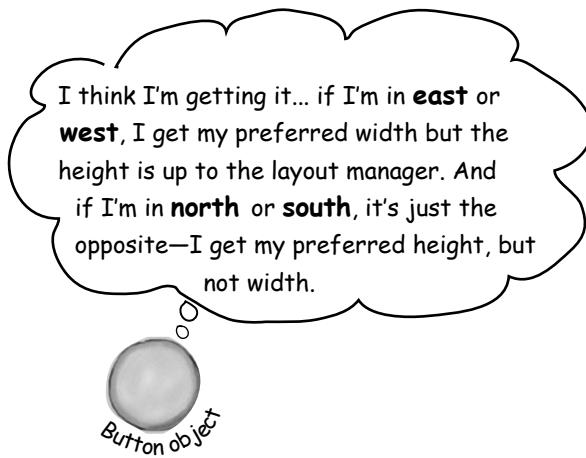
How do we do that? The button is already as wide as it can ever be—as wide as the frame. But we can try to make it taller by giving it a bigger font.

```
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 accommodate 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();

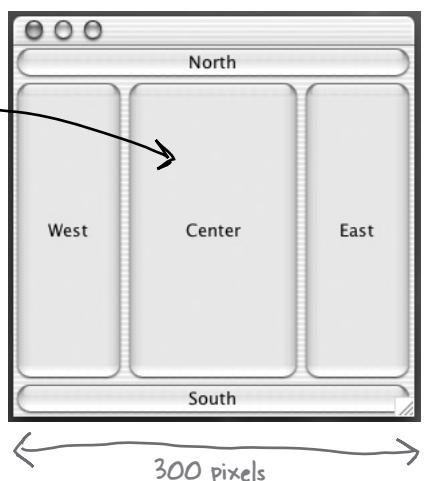
    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton center = new JButton("Center");

    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);
}
```

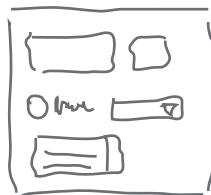
Components in the center get whatever space is left over, based on the frame dimensions (300 x 300 in this code).

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.

## flow layout



**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 Panell {

    public static void main (String[] args) {
        Panell gui = new Panell();
        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);

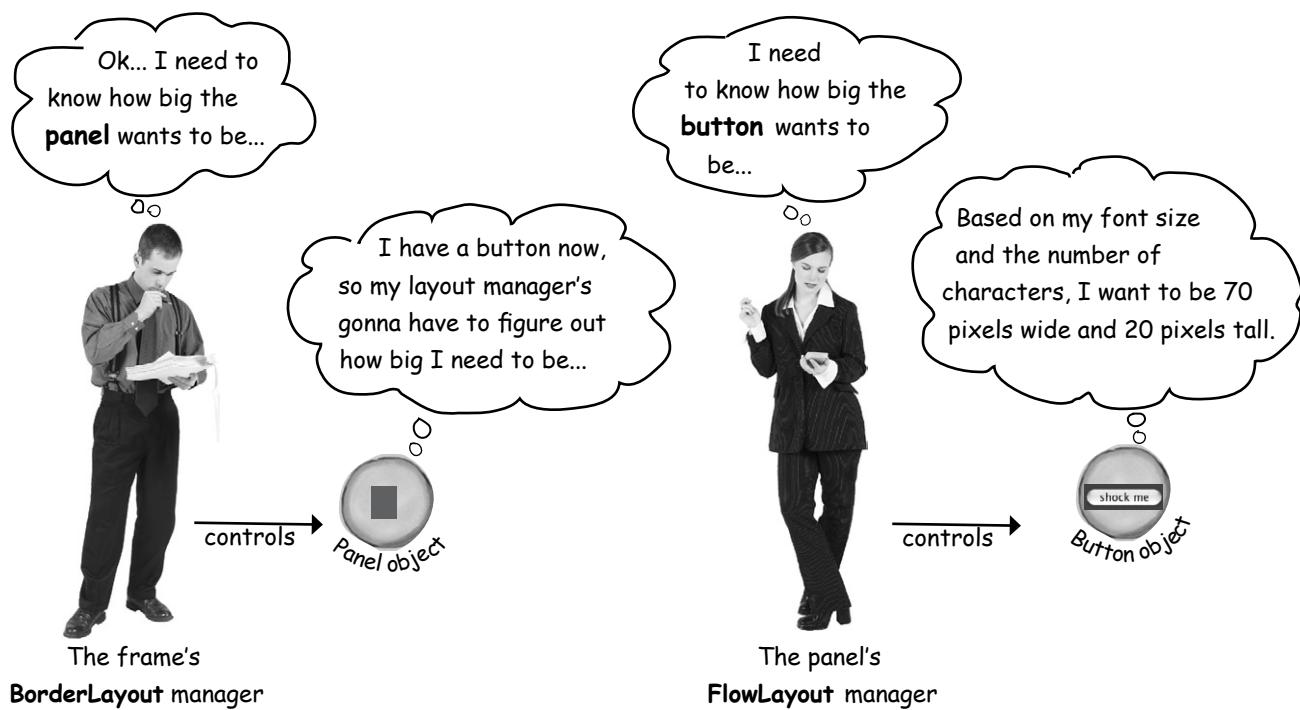
    frame.setSize(250, 200);
    frame.setVisible(true);
}
```

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).



## flow layout

### 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"); ← make TWO buttons  
    JButton buttonTwo = new JButton("bliss"); ←  
  
    panel.add(button); ← add BOTH to the panel  
    panel.add(buttonTwo); ←  
  
    frame.getContentPane().add(BorderLayout.EAST, panel);  
    frame.setSize(250, 200);  
    frame.setVisible(true);  
}
```

#### 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 it's 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'.

## layout managers

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.getContentPane(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.

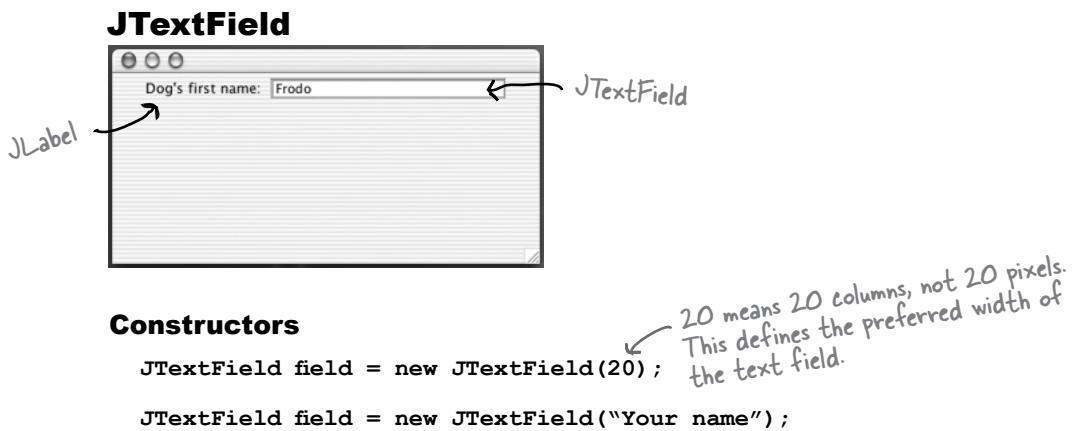
## BULLET POINTS



- 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.



### How to use it

#### ① Get text out of it

```
System.out.println(field.getText());
```

#### ② Put text in it

```
field.setText("whatever");
field.setText("");
```

This clears the field

#### ③ 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);
```

#### ④ Select/Highlight the text in the field

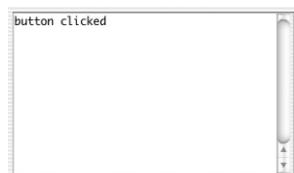
```
field.selectAll();
```

#### ⑤ Put the cursor back in the field (so the user can just start typing)

```
field.requestFocus();
```

## text area

### 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

```
JTextArea text = new JTextArea(10, 20);
```

10 means 10 lines (sets the preferred height)  
20 means 20 columns (sets the preferred width)

#### How to use it

- ① Make it have a vertical scrollbar only

```
JScrollPane scroller = new JScrollPane(text);  
text.setLineWrap(true); // Turn on line wrapping  
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);  
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

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

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!

- ② Replace the text that's in it

```
text.setText("Not all who are lost are wandering");
```

- ③ Append to the text that's in it

```
text.append("button clicked");
```

- ④ Select/Highlight the text in the field

```
text.selectAll();
```

- ⑤ 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);

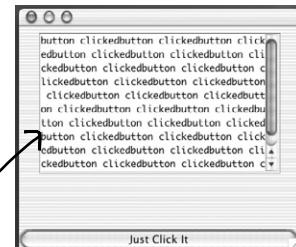
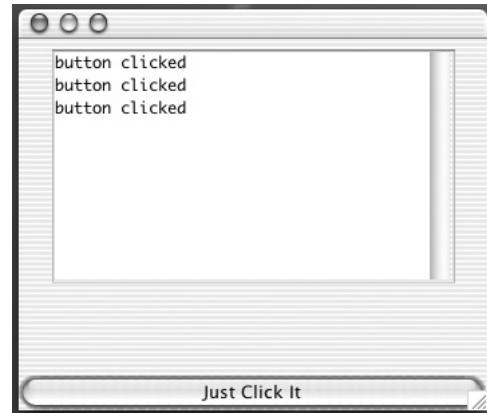
        frame.getContentPane().add(BorderLayout.CENTER, panel);
        frame.getContentPane().add(BorderLayout.SOUTH, button);

        frame.setSize(350,300);
        frame.setVisible(true);
    }
}

public void actionPerformed(ActionEvent ev) {
    text.append("button clicked \n ");
}

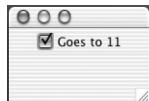
```

↑  
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*

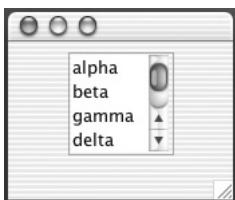
**Q:** Aren't the layout managers just more trouble than they're worth? If I have to go to all this trouble, I might as well just hard-code the size and coordinates for where everything should go.

**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 "};

JList list = new JList(listEntries);
```

### How to use it

- ① 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);
```

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.

- ② Set the number of lines to show before scrolling

```
list.setVisibleRowCount(4);
```

- ③ Restrict the user to selecting only ONE thing at a time

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

- ④ Register for list selection events

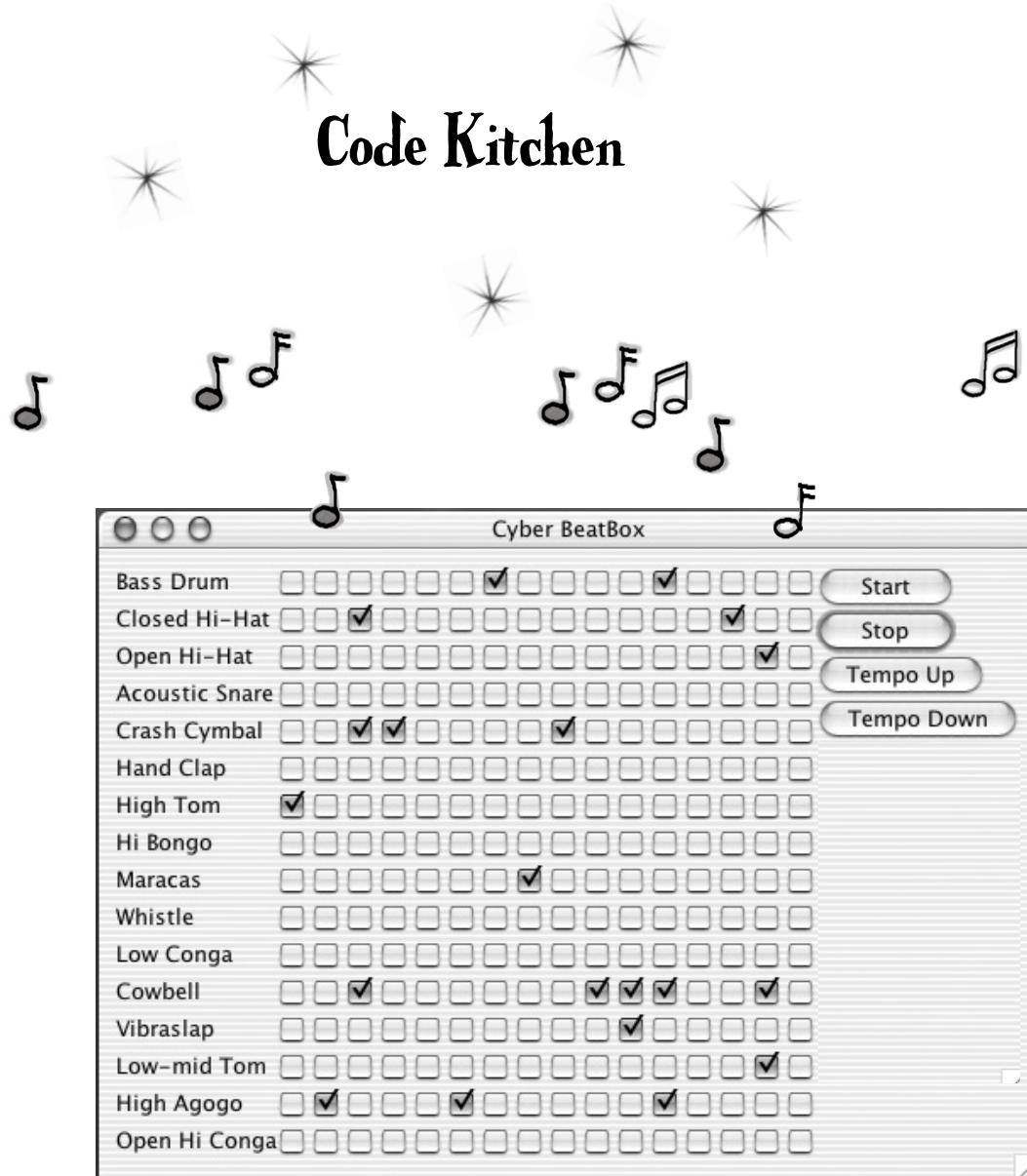
```
list.addListSelectionListener(this);
```

You'll get the event TWICE if you don't put in this if test.

- ⑤ Handle events (find out which thing in the list was selected)

```
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.



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:

- ① Build a GUI that has 256 checkboxes (`JCheckBox`) that start out unchecked, 16 labels (`JLabel`) for the instrument names, and four buttons.
- ② 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.
- ③ 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.
- ④ 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 BeatBox().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);
Still more GUI set-up code.  
Nothing remarkable.

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
} // close method

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
}

```

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.

## BeatBox code

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; ← We'll make a 16-element array to hold the values for
    sequence.deleteTrack(track); ← one instrument, across all 16 beats. If the instrument is
    track = sequence.createTrack(); ← 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.

    sequence.deleteTrack(track); } get rid of the old track, make a fresh one.

    track = sequence.createTrack(); ← do this for each of the 16 ROWS (i.e. Bass, Congo, etc.)

    for (int i = 0; i < 16; i++) { ← Set the 'key'. that represents which instrument this
        trackList = new int[16]; ← is (Bass, Hi-Hat, etc. The instruments array holds the
                                actual MIDI numbers for each instrument)

        int key = instruments[i]; ← 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 (int j = 0; j < 16; j++) { ← Do this for each of the BEATS for this row
            JCheckBox jc = checkboxList.get(j + 16*i);
            if ( jc.isSelected() ) {
                trackList[j] = key;
            } else {
                trackList[j] = 0;
            }
        } // close inner loop ← For this instrument, and for all 16 beats,
                                make events and add them to the track.

        makeTracks(trackList); ← We always want to make sure that there IS an event at
        track.add(makeEvent(176,1,127,0,16)); ← beat 16 (it goes 0 to 15). Otherwise, the BeatBox might
    } // close outer ← not go the full 16 beats before it starts over.

    track.add(makeEvent(192,9,1,0,15));
    try {
        sequencer.setSequence(sequence);
        sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY); ← Lets you specify the number of
        sequencer.start(); ← loop iterations, or in this case,
        sequencer.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}
} // close buildTrackAndStart method ← NOW PLAY THE THING!!

public class MyStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        buildTrackAndStart();
    }
} // close inner class ← First of the inner classes,
                           listeners for the buttons.
                           Nothing special here.

```

## using swing

```

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 NOTE OFF events, and add them to the Track.

This is the utility method from last chapter's CodeKitchen. Nothing new.

**exercise:** Which Layout?



## 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.

The figure shows six mobile device screen layouts arranged in a 3x2 grid. A large question mark is centered between the second and third columns. Each screen has a number in a circle at its top-left corner.

- 1**: A single-column layout with rounded corners. The top section is light gray with the word "tesuji". The bottom section is dark gray with the word "watari".
- 2**: A single-column layout with rounded corners. The top section is dark gray with the word "watari". The bottom section is light gray with the word "tesuji".
- 3**: A single-column layout with rounded corners. The top section is light gray with the word "tesuji". The bottom section is dark gray with the word "watari".
- 4**: A two-column layout. The left column is light gray with the word "tesuji". The right column is dark gray with the word "watari".
- 5**: A single-column layout with rounded corners. The top section is dark gray with the word "watari". The bottom section is light gray with the word "tesuji".
- 6**: A two-column layout. The left column is light gray with the word "tesuji". The right column is dark gray with the word "watari".

## 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);
```

---

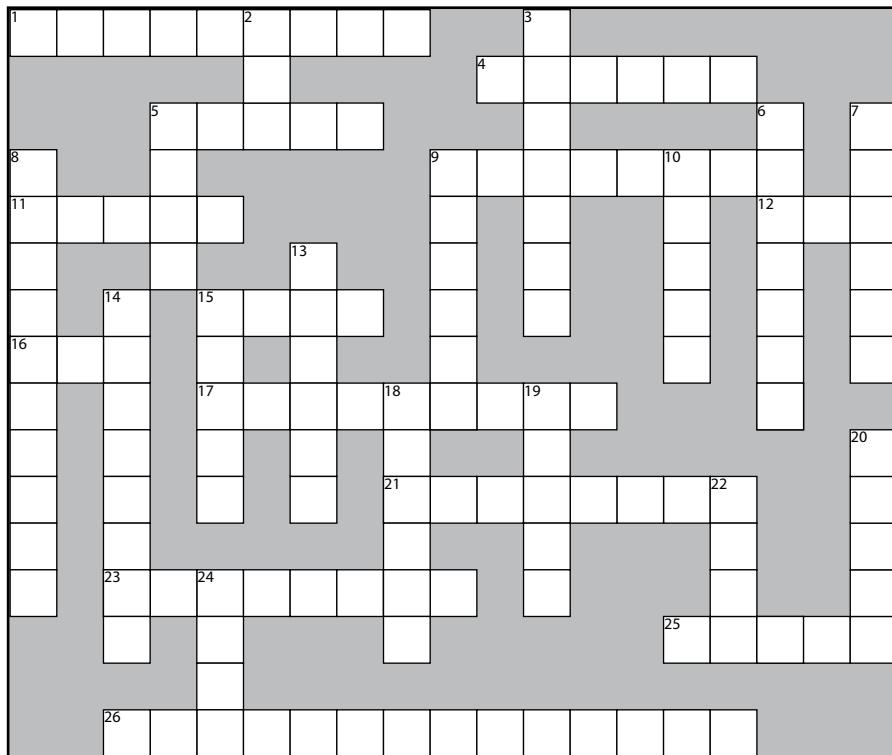
**E**

```
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



# GUIL-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

## Down

- 17. Shake it baby
- 21. Lots to say
- 23. Choose many
- 25. Button's pal
- 26. Home of actionPerformed
- 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**

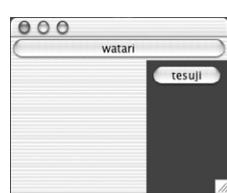
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

**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****F**

```
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);
```

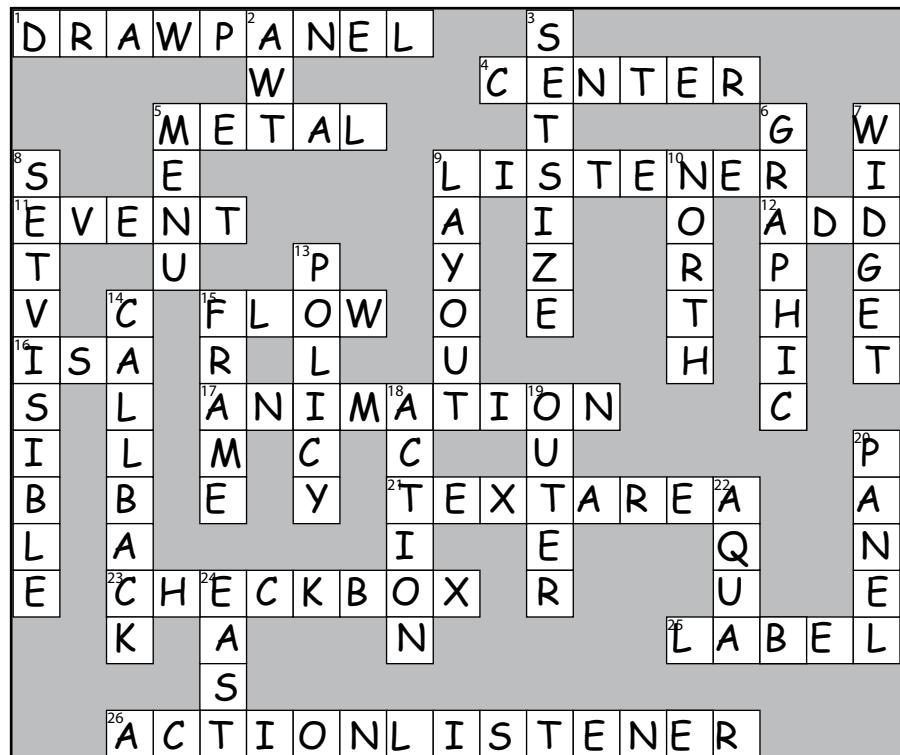
**6****G**

```
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**



# Puzzle Answers GUI-Cross 7.0



## Saving Objects



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

**Objects can be flattened and inflated.** Objects have state and behavior.

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

## saving objects

# 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.

Bass Drum	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
Closed Hi-Hat	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Open Hi-Hat	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>											
Acoustic Snare	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>											
Crash Cymbal	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>											
Hand Clap	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>											
High Tom	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>											
Hi Bongo	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Maracas	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>										
Whistle	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										
Low Conga	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										
Cowbell	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										
Vibraslap	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										
Low-mid Tom	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										
High Agogo	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>										
Open Hi Conga	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>										

### 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.

### ① 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:

```
"IsrGameCharacter
%g 8M IpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xptlfur[Ljava/lang/String;*“V 
 {Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

### ② Option two

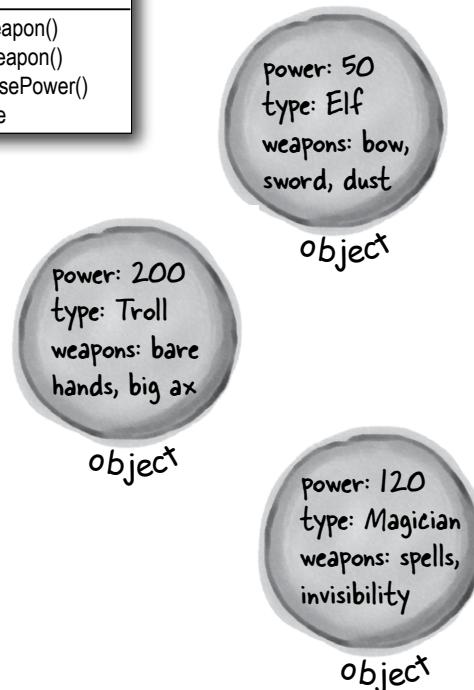
#### **Write a plain text file**

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
```

*Imagine you  
have three game  
characters to save...*

<b>GameCharacter</b>
int power
String type
Weapon[] weapons
getWeapon()
useWeapon()
increasePower()
// more



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...

saving objects

## 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);
```

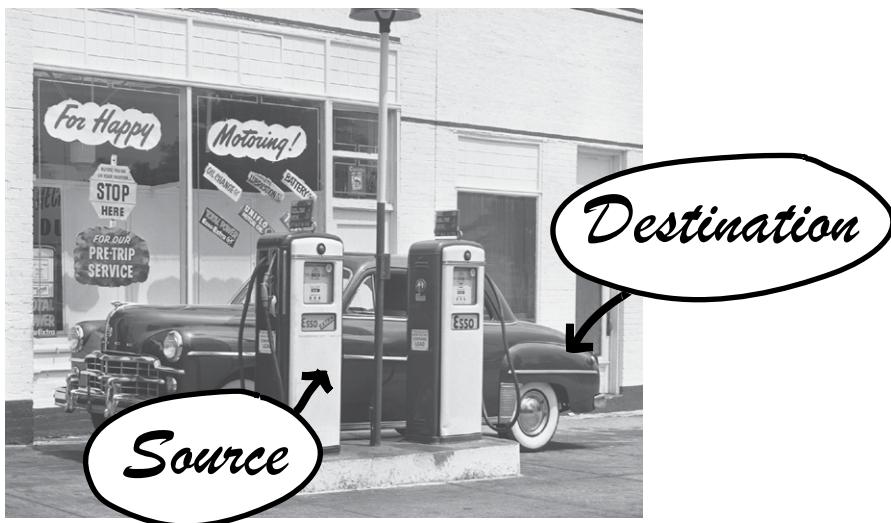
serializes the objects referenced by characterOne, characterTwo, and characterThree, and writes them to the file "MyGame.ser".

### 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.



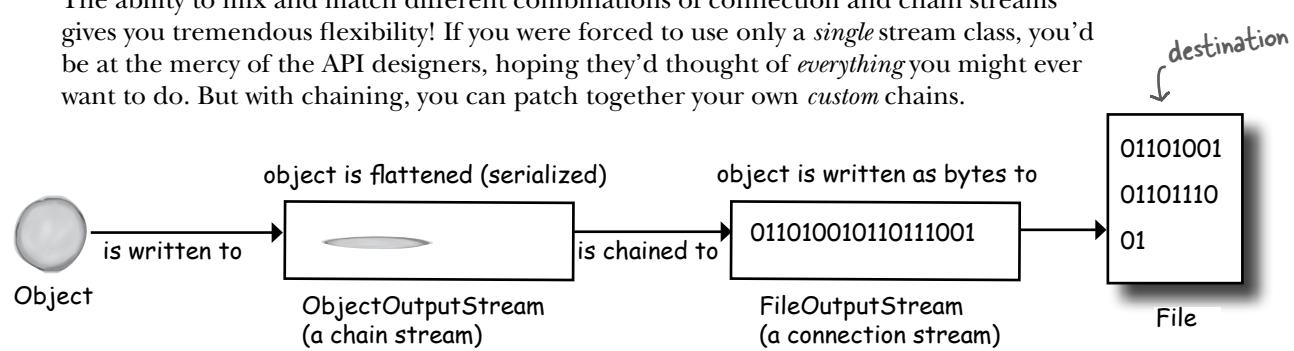
Connection streams represent a connection to a source or destination (file, socket, etc.) while chain streams can't connect on their own and must be chained to a connection stream.

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.



## serialized objects

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

### 1 Object on the heap

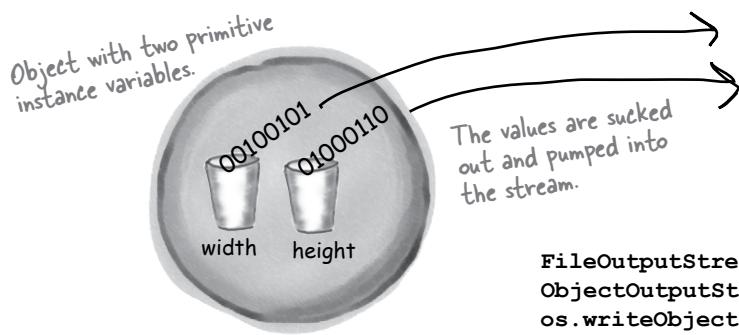


Objects on the heap have state—the value of the object's instance variables. These values make one instance of a class different from another instance of the same class.

### 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);
```

The instance variable values for width and height are saved to the file "foo.ser", along with a little more info the JVM needs to restore the object (like what its class type is).

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.



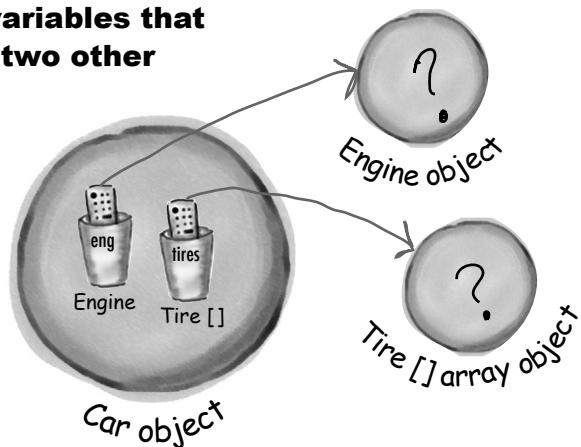
### Brain Barbell

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 Carburetor? 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?**

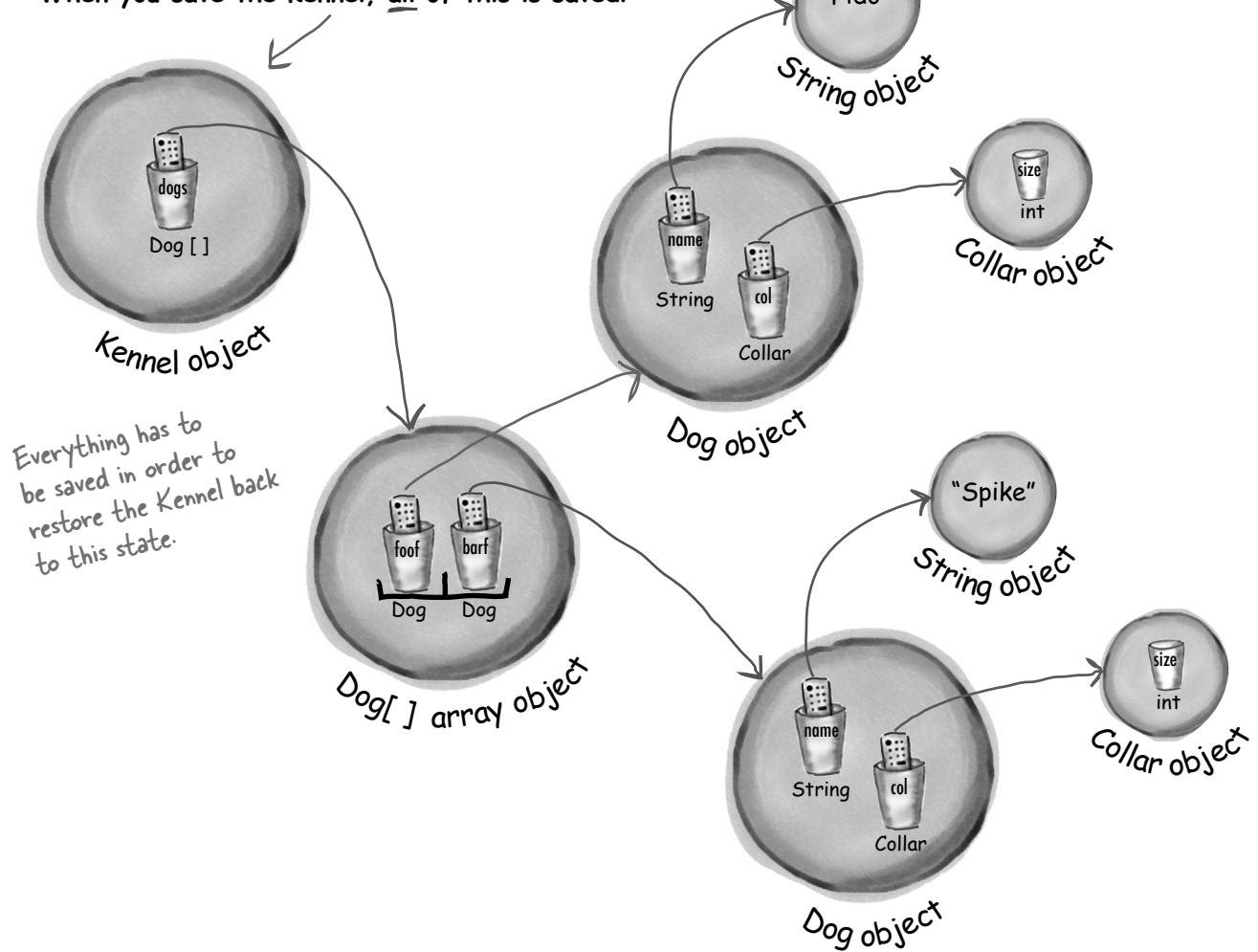
## serialized objects

**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.

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



## If you want your class to be serializable, implement Serializable

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

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

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

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

## serialized objects

### Serialization is all or nothing.

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



Eeeeeew! 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();  
        }  
    }  
  
    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.
```

Pond objects can be serialized.

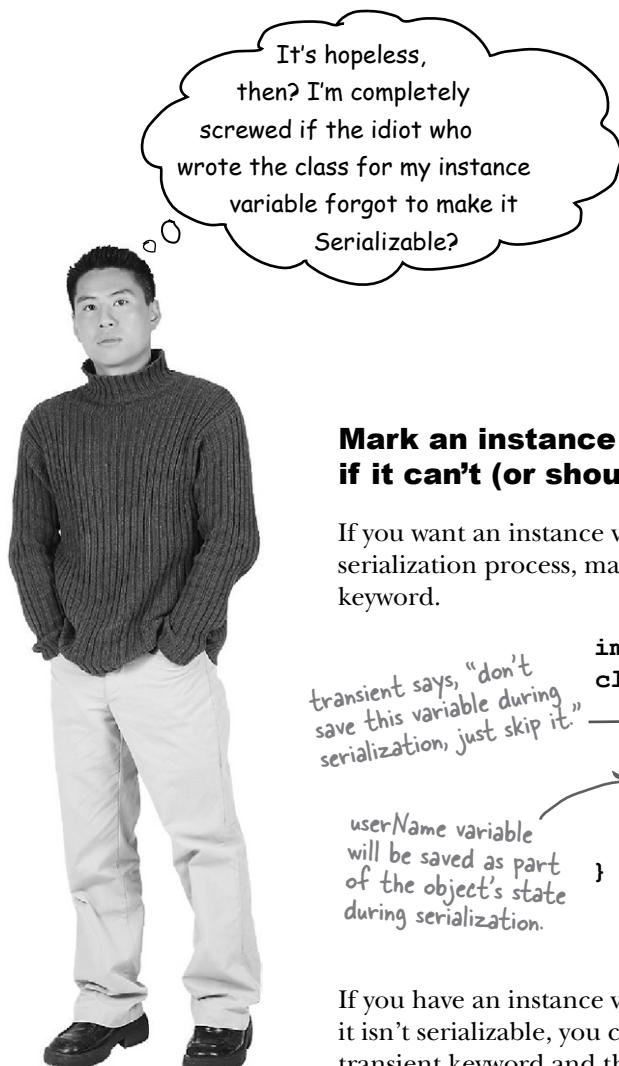
Class Pond has one instance variable, a Duck.

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

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.

```
import java.net.*;
class Chat implements Serializable {
    transient String currentID;
    String userName;
    // more code
}
```

Annotations on the code:

- A handwritten note next to the `transient` keyword says: "transient says, 'don't save this variable during serialization, just skip it.'"
- An arrow points from the handwritten note to the `transient` keyword.
- A handwritten note next to the `userName` variable says: "userName variable will be saved as part of the object's state during serialization."
- An arrow points from the handwritten note to the `userName` variable.

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.

## serialized objects

# 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:

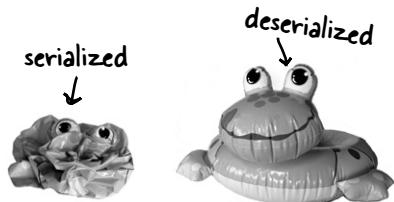
- 1) When the object is brought back, reinitialize that null instance variable back to some default state. This works if your deserialized object isn't dependent on a particular value for that transient variable. In other words, it might be important that the Dog have a Collar, but perhaps all Collar 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.



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

### 1 Make a FileInputStream

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

↑  
Make a FileInputStream object. It 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

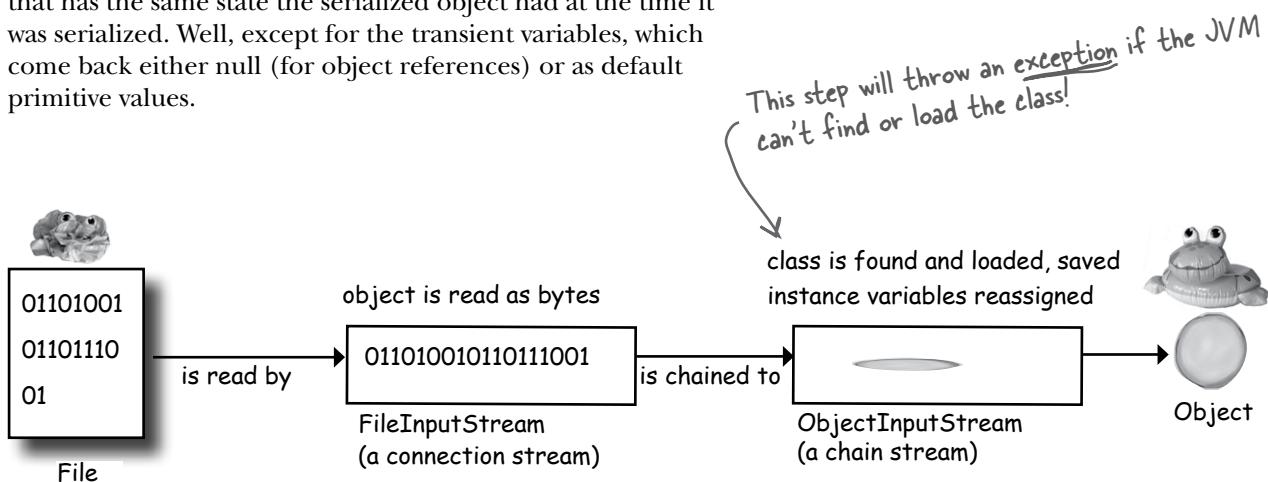
```
os.close();
```

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

## deserializing objects

# 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.



- ❶ The object is **read** from the stream.
- ❷ The JVM determines (through info stored with the serialized object) the object's **class type**.
- ❸ 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.
- ❹ 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.

## <sup>there are no</sup> 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.

serialization example

## 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; ← We set them to null so we can't  
        two = null; ← access the objects on the heap.  
        three = null;  
  
        Now read them back in from the file...  
  
        try {  
            ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));  
            GameCharacter oneRestore = (GameCharacter) is.readObject();  
            GameCharacter twoRestore = (GameCharacter) is.readObject();  
            GameCharacter threeRestore = (GameCharacter) is.readObject();  
  
            System.out.println("One's type: " + oneRestore.getType()); ← Check to see if it worked.  
            System.out.println("Two's type: " + twoRestore.getType());  
            System.out.println("Three's type: " + threeRestore.getType());  
        } catch(Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

File Edit Window Help Resuscitate

% java GameSaverTest

One's type: Elf

Two's type: Troll

Three's type: Magician

power: 50  
type: Elf  
weapons: bow, sword, dust

object

power: 200  
type: Troll  
weapons: bare hands, big ax

object

power: 120  
type: Magician  
weapons: spells, invisibility

object

## The GameCharacter class

```

import java.io.*;

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

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

    public int getPower() {
        return power;
    }

    public String getType() {
        return type;
    }

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

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

```

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

# Object Serialization

## BULLET POINTS

- ▶ You can save an object's state by serializing the object.
- ▶ To serialize an object, you need an ObjectOutputStream (from the java.io package)
- ▶ Streams are either connection streams or chain streams
- ▶ Connection streams can represent a connection to a source or destination, typically a file, network socket connection, or the console.
- ▶ Chain streams cannot connect to a source or destination and must be chained to a connection (or other) stream.
- ▶ To serialize an object to a file, make a FileOutputStream 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");
            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!!*

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

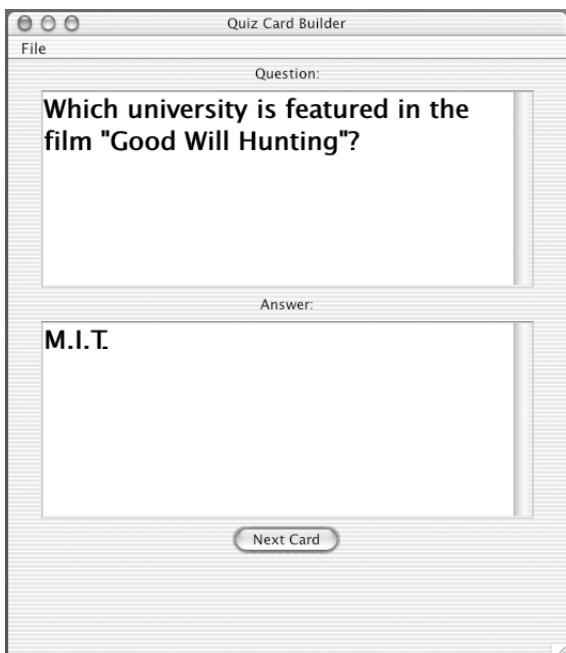
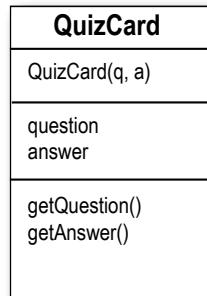
## writing a text file

# 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.

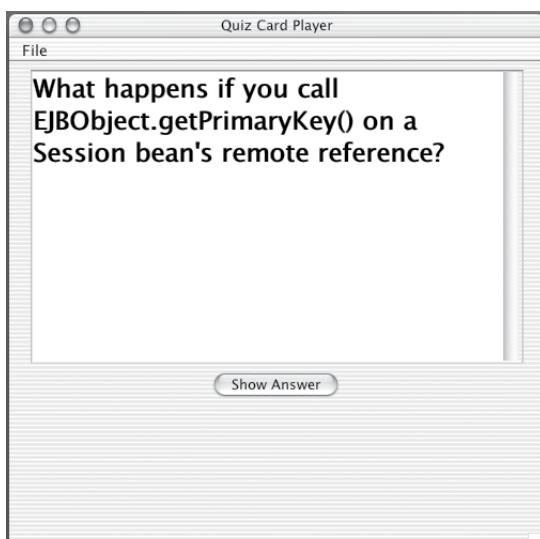
We're going to make an electronic version that has three classes:

- 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() {           Builds and displays the GUI, including
        // build and display gui   making and registering event listeners.
    }

    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)
    }
}

```

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

*Triggered when user 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.).*

*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 MenuItem's code.
```

## serialization and file I/O

```

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")

## writing files

# 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`.

### 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());
```

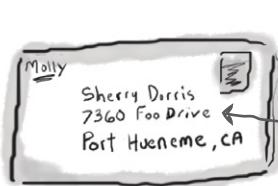
- ⑤ Delete a file or directory (returns true if successful)

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

**A File object represents the 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.

A File object represents the filename "GameFile.txt"

**GameFile.txt**

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

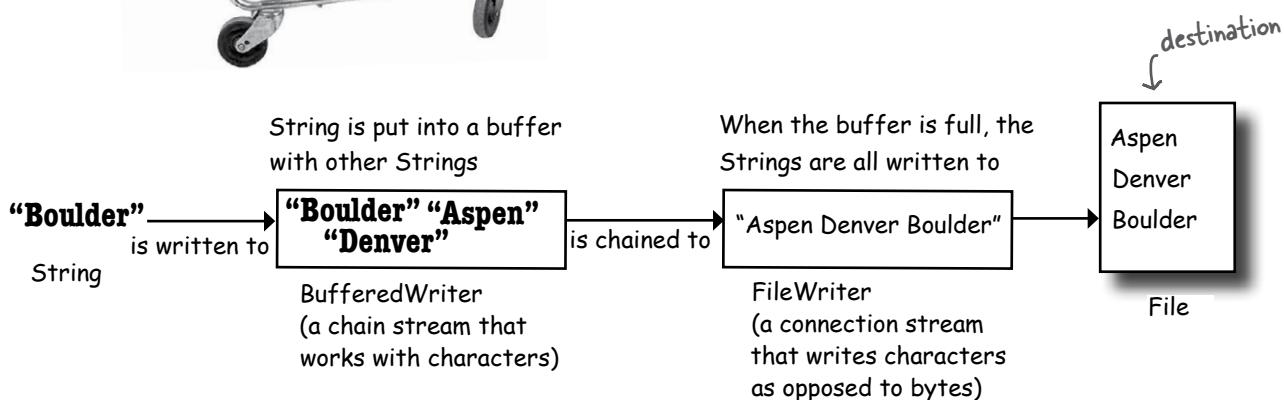
A File object does NOT represent (or give you direct access to) the data inside the file!

## 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));
```

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!**"

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.

## reading files

# Reading from a Text File

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

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

A file with two lines of text.

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

**MyText.txt**

```
import java.io.*;  Don't forget the import.  
  
class ReadAfile {  
    public static void main (String[] args) {  
        try {  
            File myFile = new File("MyText.txt");  
            FileReader fileReader = new FileReader(myFile);  
  
            BufferedReader reader = new BufferedReader(fileReader);  
  
            Make a String variable to hold  
            each line as the line is read  
            String line = null;  
  
            while ((line = reader.readLine()) != null) {  
                System.out.println(line);  
            }  
            reader.close();  
  
        } catch(Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

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).

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*

## serialization and file I/O

```

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.

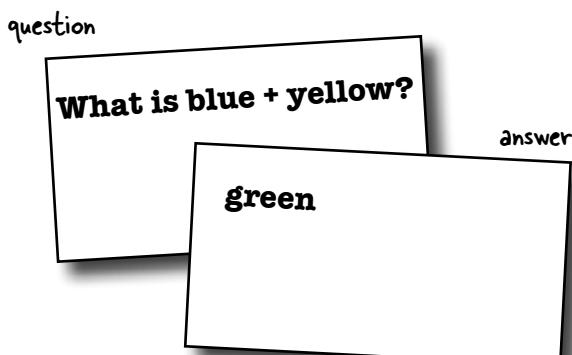
Make a `BufferedReader` chained to a new `FileReader`, giving the `FileReader` the `File` object the user chose from the open file dialog.  
Read a line at a time, passing the line to the `makeCard()` method that parses it and turns it into a real `QuizCard` and adds it to the `ArrayList`.

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 Strings with `split()`

## Parsing with String `split()`

**Imagine you have a flashcard like this:**



**Saved in a question file like this:**

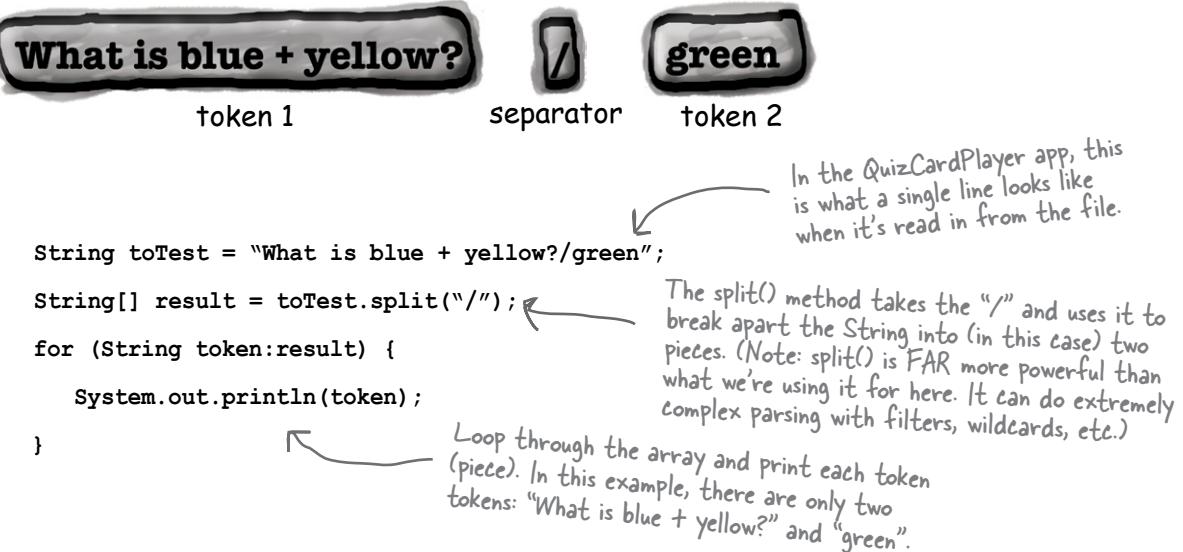
What is blue + yellow?/green  
What is red + blue?/purple

**How do you separate the question and answer?**

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

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

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



## <sup>there are no</sup> 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.

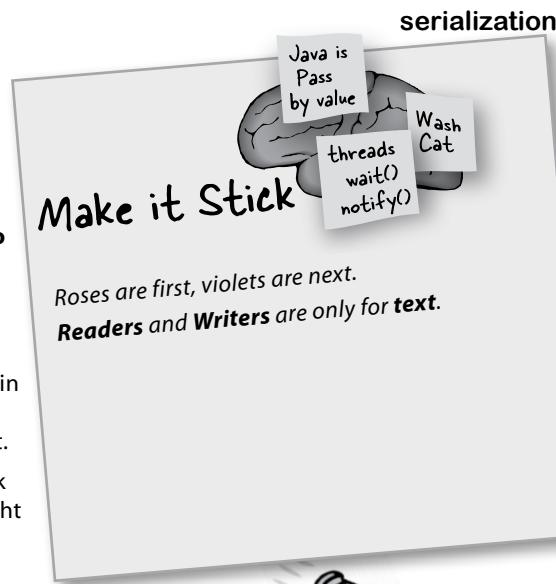
Most of the time, though, you’ll use the same small handful of classes. If you’re writing text files, BufferedReader and BufferedWriter (chained to FileReader and FileWriter) are probably all you need. If you’re writing serialized objects, you can use ObjectOutputStream and ObjectInputStream (chained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

## saving objects

# 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 effect on the ability of deserialization to assign a value to the variable
- Changing an instance variable from transient to non-transient (previously-serialized objects will simply have a default value for the previously-transient variables)

#### ① You write a Dog class

101101  
101101  
101000010  
1010 10 0  
01010 1  
1010101  
10101010  
1001010101  
#343

Dog.class

#### ② You serialize a Dog object using that class

Dog object  
Object is stamped with version #343

#### ③ You change the Dog class

101101  
101101  
101000010  
1010 10 0  
01010 1  
100001 1010  
0 00110101  
1 0 1 10 10  
#728

Dog.class

#### ④ You deserialize a Dog object using the changed class

Dog object  
Object is stamped with version #343

Dog.class  
class version is #728

#### ⑤ Serialization fails!!

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

## Using the serialVersionUID

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

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

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

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

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

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

```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

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

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

```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

 Based on the version of Java you’re using, this value might be different.

② Paste the output into your class

```
public class Dog {

    static final long serialVersionUID =
        -6849794470754667710L;

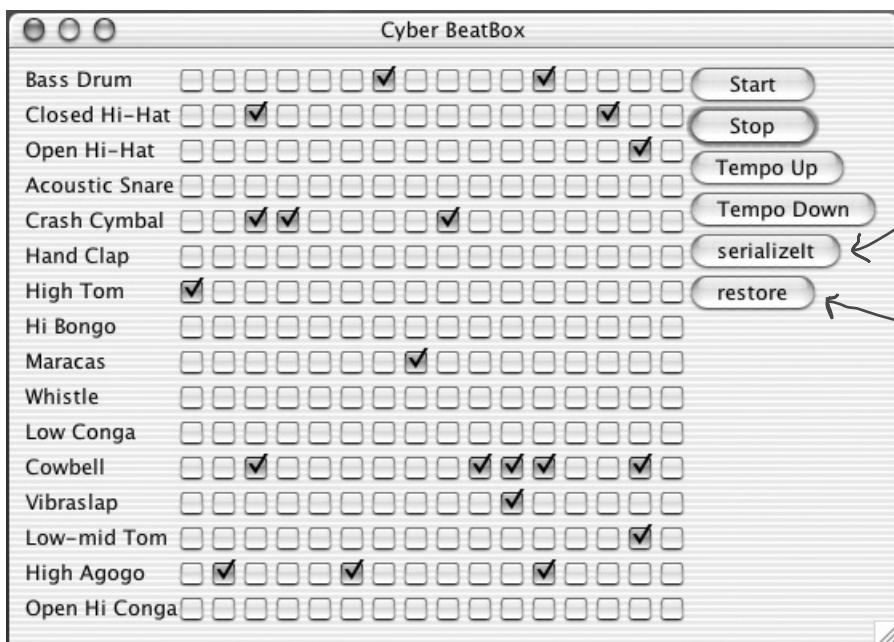
    private String name;
    private int size;

    // method code here
}
```

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

## Code Kitchen

# 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) { ← It all happens when the user clicks the
        boolean[] checkboxState = new boolean[256]; ← button and the ActionEvent fires.
        for (int i = 0; i < 256; i++) { ← Make a boolean array to hold the
            JCheckBox check = (JCheckBox) checkboxList.get(i); ← state of each checkbox.
            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

```

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!

## deserializing the pattern

# 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(); ← 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.)  
        } 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
```

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 FileInputStream("Game.ser"));
GameCharacter oneAgain = (GameCharacter) is.readObject();
```



**exercise: True or False**



This chapter explored the wonderful 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. BufferedWriter can be chained to FileWriter.
13. File objects represent files, but not directories.
14. You can't force a buffer to send its data before it's full.
15. Both file readers and file writers can be buffered.
16. The String split() method includes separators as tokens in the result array.
17. Any change to a class breaks previously serialized objects of that class.



## Code Magnets

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

```

class DungeonGame implements Serializable {
    try {
        FileOutputStream fos = new
            FileOutputStream("dg.ser");
        fos.writeObject(d);
    } catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
}

class DungeonTest {
    public static void main(String [] args) {
        DungeonGame d = new DungeonGame();
        ObjectInputStream ois = new
            ObjectInputStream(fis);
        d = (DungeonGame) ois.readObject();
        int getX() {
            return x;
        }
        short getZ() {
            return z;
        }
        long getY() {
            return y;
        }
        public int x = 3;
        transient long y = 4;
        private short z = 5;
    }
}

```

```

File Edit Window Help Torture
% java DungeonTest
12
8

```

```

ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(d);

```

```

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

```

**exercise solutions**



## 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 used on 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. BufferedWriter can be chained to FileWriter. **True**
13. File objects represent files, but not directories. **False**
14. You can't force a buffer to send its data before it's full. **False**
15. Both file readers and file writers can optionally be buffered. **True**
16. The String split() method includes separators as tokens in the result array. **False**
17. Any change to a class breaks previously serialized objects of that class. **False**



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

```
import java.io.*;

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

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

File Edit Window Help Escape
% java DungeonTest
12
8



## 15 networking and threads

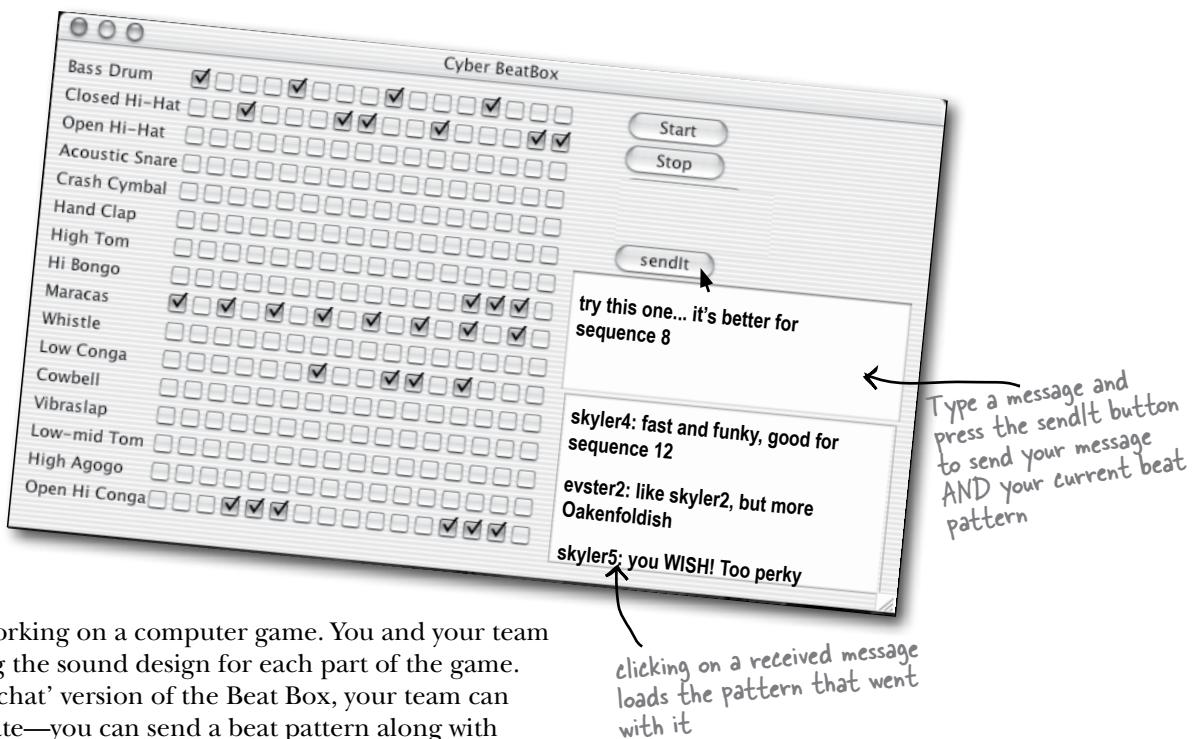
# Make a Connection



**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` couldn't 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.

## beat box chat

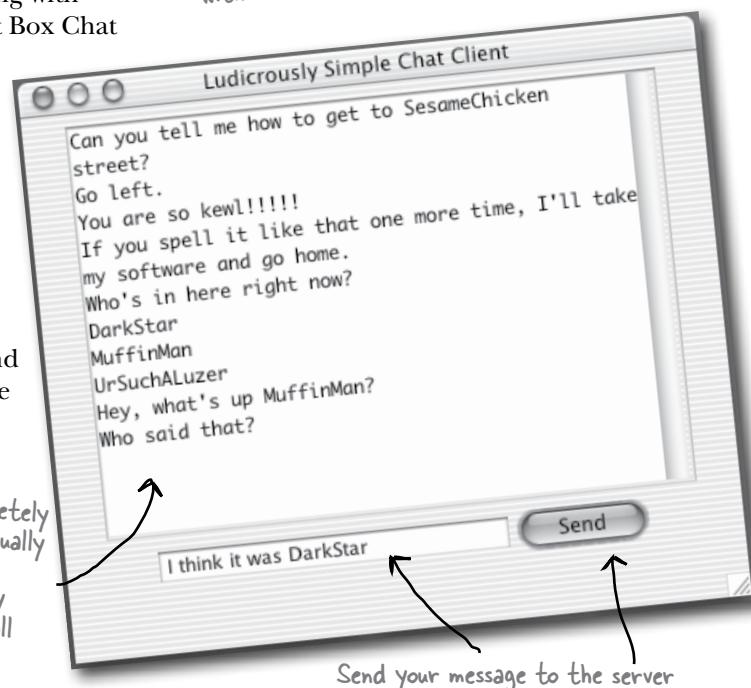
# Real-time Beat Box Chat



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.

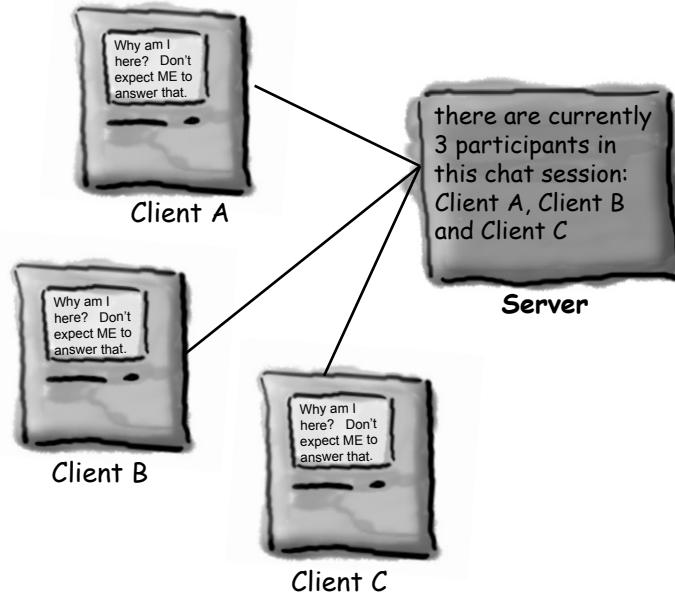
You can have completely authentic, intellectually stimulating chat conversations. Every message is sent to all participants.



## Chat Program Overview

The Client has to know about the Server.

The Server has to know about ALL the Clients.

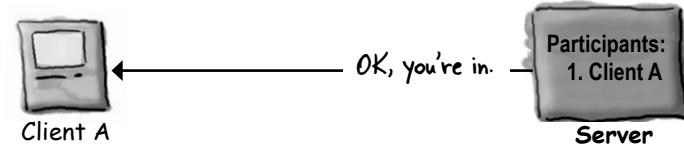


## How it Works:

- 1 Client connects to the server



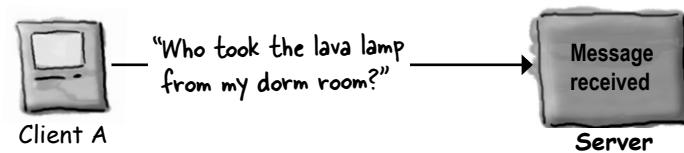
- 2 The server makes a connection and adds the client to the list of participants



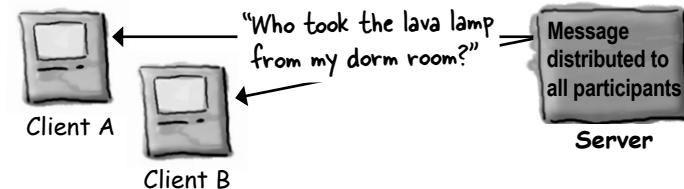
- 3 Another client connects



- 4 Client A sends a message to the chat service



- 5 The server distributes the message to ALL participants (including the original sender)



**socket connections**

## 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). Hmm... 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

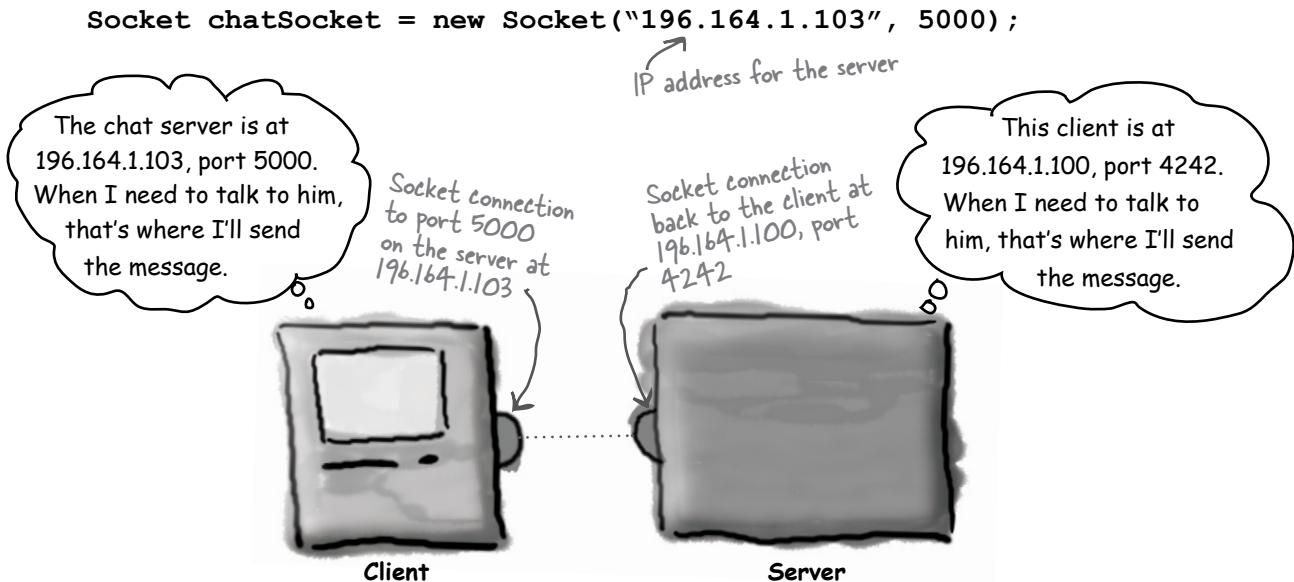
Client **gets** a message from the server



## 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?



A Socket connection means the two machines have information about each other, including network location (IP address) and TCP port.

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.**

## well-known ports

# A TCP port is just a number. A 16-bit number that identifies a specific program on the server.

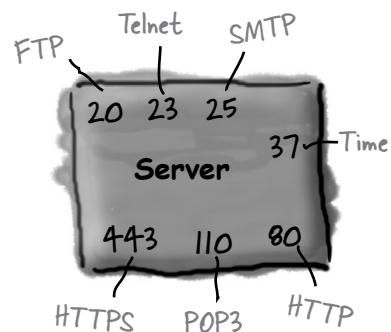
Your internet web (HTTP) server runs on port 80. That's a standard. If you've got a Telnet server, it's running on port 23. FTP? 20. POP3 mail server? 110. SMTP? 25. The Time server sits at 37. Think of port numbers as unique identifiers. They represent a logical connection to a particular piece of software 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.

## <sup>there are no</sup> 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.

IP address is the mall



Port number is the specific store in the mall



IP address is like specifying a particular shopping mall, say, "Flatirons Marketplace"

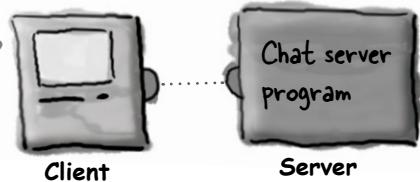
Port number is like naming a specific store, say, "Bob's CD Shop"



## 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.

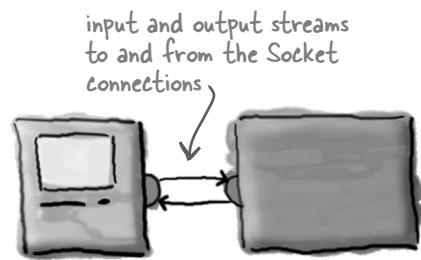
How do these two actually talk to each other?



reading from a socket

## 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);
```

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.

The port number, which you know because we TOLD you that 5000 is the port number for our chat server.

### 2 Make an InputStreamReader chained to the Socket's low-level (connection) input stream

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

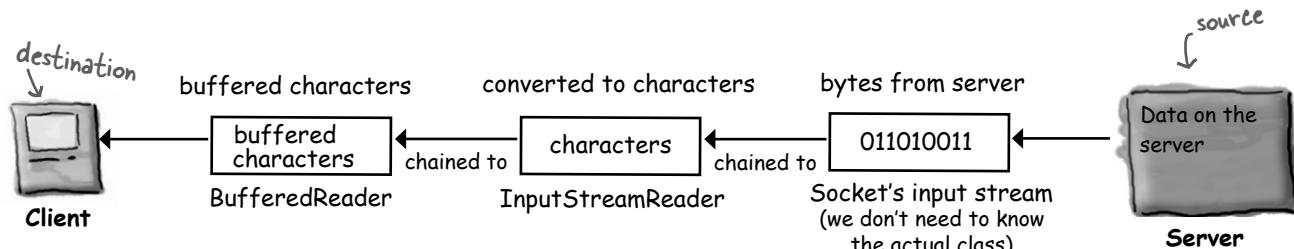
InputStreamReader is a 'bridge' between a low-level byte stream (like the one coming from the Socket) and a high-level character stream (like the BufferedReader we're after as our top of the chain stream).

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!

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```

Chain the BufferedReader to the InputStreamReader(which was chained to the low-level connection stream we got from the Socket.)



## 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

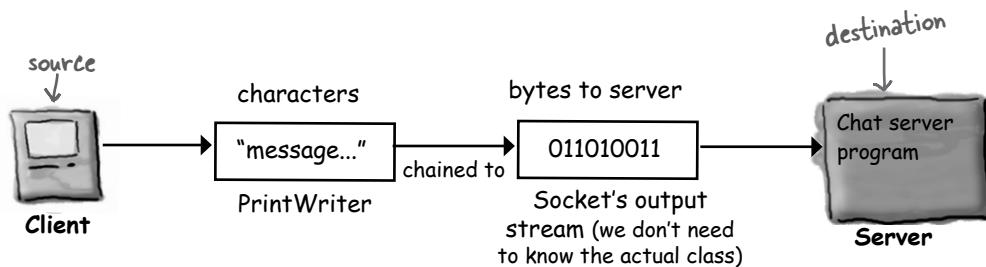
```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

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"); ← println() adds a new line at the end of what it sends.  
writer.print("another message"); ← print() doesn't add the new line.
```



## writing a client

# 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.

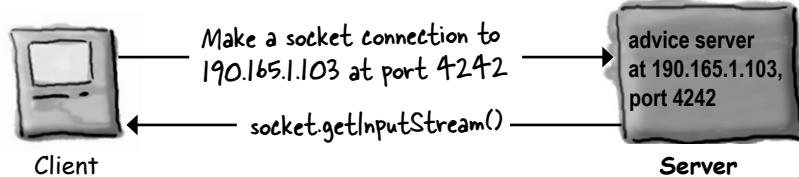
What are you waiting for? Who *knows* what opportunities you've missed without this app.



The Advice Guy

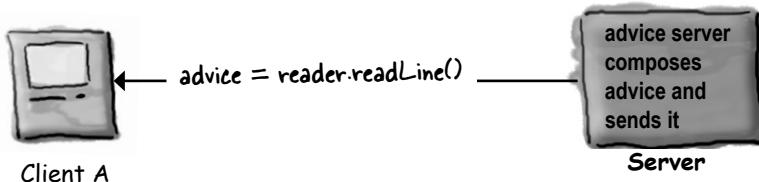
## ➊ Connect

Client connects to the server and gets an input stream from it



## ➋ 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.*; ← class Socket is in java.net

public class DailyAdviceClient {

    public void go() {
        try { ← a lot can go wrong here
            Socket s = new Socket("127.0.0.1", 4242);

            InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
            BufferedReader reader = new BufferedReader(streamReader); ← chain a BufferedReader to
                                                               an InputStreamReader to
                                                               the input stream from the
                                                               Socket.

            String advice = reader.readLine(); ← this readLine() is EXACTLY
            System.out.println("Today you should: " + advice); ← the same as if you were using a
                                                               BufferedReader chained to a FILE..
                                                               In other words, by the time you
                                                               call a BufferedReader method, the
                                                               reader doesn't know or care where
                                                               the characters came from.

            reader.close(); ← this closes ALL the streams

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

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

```

## socket connections



### 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



Client



write/draw in the chain of streams the client uses to send something to the 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);
```

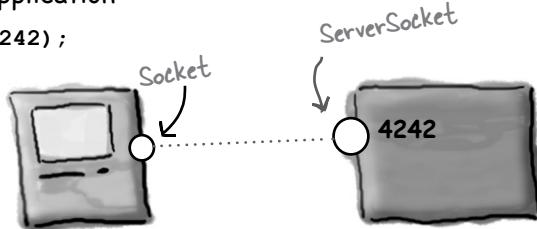
This starts the server application listening for client requests coming in for port 4242.



- 2 Client makes a Socket connection to the server application

```
Socket sock = new Socket("190.165.1.103", 4242);
```

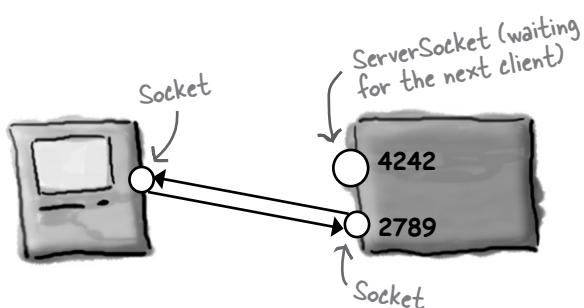
Client knows the IP address and port number (published or given to him by whomever configures the server app to be on that port)



- 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.



## writing a server

# DailyAdviceServer code

This program makes a ServerSocket and waits for client requests. When it gets a client request (i.e. client said new Socket() for this application), the server makes a new Socket connection to that client. The server makes a PrintWriter (using the Socket's output stream) and sends a message to the client.

```
import java.io.*; remember the imports
import java.net.*;

public class DailyAdviceServer {
    String[] adviceList = {"Take smaller bites", "Go for the tight jeans. No they do NOT
make you look fat.", "One word: inappropriate", "Just for today, be honest. Tell your
boss what you *really* think", "You might want to rethink that haircut."};

    public void go() {
        try {
            ServerSocket serverSock = new ServerSocket(4242);
            while(true) {
                Socket sock = serverSock.accept();
                PrintWriter writer = new PrintWriter(sock.getOutputStream());
                String advice = getAdvice();
                writer.println(advice);
                writer.close();
                System.out.println(advice);
            }
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }

    private String getAdvice() {
        int random = (int) (Math.random() * adviceList.length);
        return adviceList[random];
    }

    public static void main(String[] args) {
        DailyAdviceServer server = new DailyAdviceServer();
        server.go();
    }
}
```

remember the imports

daily advice comes from this array

(remember, these Strings were word-wrapped by the code editor. Never hit return in the middle of a String!)

The server goes into a permanent loop, waiting for (and servicing) client requests

ServerSocket makes this server application 'listen' for client requests on port 4242 on the machine this code is running on.

the accept method blocks (just sits there) until a request comes in, and then the method returns a Socket (on some anonymous port) for communicating with the client

now we use the Socket connection to the client to make a PrintWriter and send it (println()) a String advice message. Then we close the Socket because we're done with this client.



# Brain Barbell

**How does the server know how to communicate with the client?**

The client knows the IP address and port number of the server, but how is the server able to make a Socket connection with the client (and make input and output streams)?

Think about how / when / where the server gets knowledge about the client.

*there are no*  
**Dumb Questions**

**Q:** The advice server code on the opposite page has a VERY serious limitation—it looks like it can handle only one client at a time!

**A:** Yes, that's right. It can't accept a request from a client until it has finished with the current client and started the next iteration of the infinite loop (where it sits at the `accept()` call until a request comes in, at which time it makes a Socket with the new client and starts the process over again).

**Q:** Let me rephrase the problem: how can you make a server that can handle multiple clients concurrently??? This would never work for a chat server, for instance.

**A:** Ah, that's simple, really. Use separate threads, and give each new client Socket to a new thread. We're just about to learn how to do that!



## BULLET POINTS

- Client and server applications communicate over a Socket connection.
- A Socket represents a connection between two applications which may (or may not) be running on two different physical machines.
- A client must know the IP address (or domain name) and TCP port number of the server application.
- A TCP port is a 16-bit unsigned number assigned to a specific server application. TCP port numbers allow different clients to connect to the same machine but communicate with different applications running on that machine.
- The port numbers from 0 through 1023 are reserved for 'well-known services' including HTTP, FTP, SMTP, etc.
- A client connects to a server by making a Server socket  
`Socket s = new Socket("127.0.0.1", 4200);`
- Once connected, a client can get input and output streams from the socket. These are low-level 'connection' streams.  
`sock.getInputStream();`  
`sock.getOutputStream();`
- To read text data from the server, create a BufferedReader, chained to an InputStreamReader, which is chained to the input stream from the Socket.
- InputStreamReader is a 'bridge' stream that takes in bytes and converts them to text (character) data. It's used primarily to act as the middle chain between the high-level BufferedReader and the low-level Socket input stream.
- To write text data to the server, create a PrintWriter chained directly to the Socket's output stream. Call the `print()` or `println()` methods to send Strings to the server.
- Servers use a ServerSocket that waits for client requests on a particular port number.
- When a ServerSocket gets a request, it 'accepts' the request by making a Socket connection with the client.

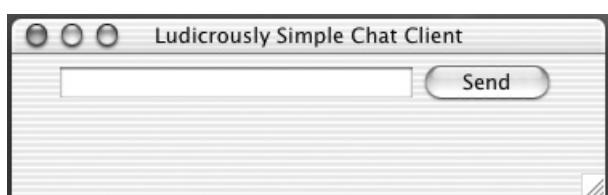
## a simple chat client

# Writing a Chat Client

We'll write the Chat client application in two stages. First we'll make a send-only version that sends messages to the server but doesn't get to read any of the messages from other participants (an exciting and mysterious twist to the whole chat room concept).

Then we'll go for the full chat monty and make one that both sends *and* receives chat messages.

## Version One: send-only



Type a message, then press 'Send' to send it to the server. We won't get any messages FROM the server in this version, so there's no scrolling text area.

## Code outline

```
public class SimpleChatClientA {  
  
    JTextField outgoing;  
    PrintWriter writer;  
    Socket sock;  
  
    public void go() {  
        // make gui and register a listener with the send button  
        // call the setUpNetworking() method  
    }  
  
    private void setUpNetworking() {  
        // make a Socket, then make a PrintWriter  
        // assign the PrintWriter to writer instance variable  
    }  
  
    public class SendButtonListener implements ActionListener {  
        public void actionPerformed(ActionEvent ev) {  
            // get the text from the text field and  
            // send it to the server using the writer (a PrintWriter)  
        }  
    } // close SendButtonListener inner class  
  
} // close outer class
```

## networking and threads

```

import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClientA {
    JTextField outgoing;
    PrintWriter writer;
    Socket sock;

    public void go() {
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        setUpNetworking();
        frame.setSize(400,500);
        frame.setVisible(true);
    } // close go

    private void setUpNetworking() {
        try {
            sock = new Socket("127.0.0.1", 5000);
            writer = new PrintWriter(sock.getOutputStream());
            System.out.println("networking established");
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // close setUpNetworking

    public class SendButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            try {
                writer.println(outgoing.getText());
                writer.flush();
            } catch(Exception ex) {
                ex.printStackTrace();
            }
            outgoing.setText("");
            outgoing.requestFocus();
        }
    } // close SendButtonListener inner class

    public static void main(String[] args) {
        new SimpleChatClientA().go();
    }
} // close outer class

```

imports for the streams (java.io),  
Socket (java.net) and the GUI  
stuff

build the GUI, nothing new  
here, and nothing related to  
networking or I/O.

we're using localhost so  
you can test the client  
and server on one machine

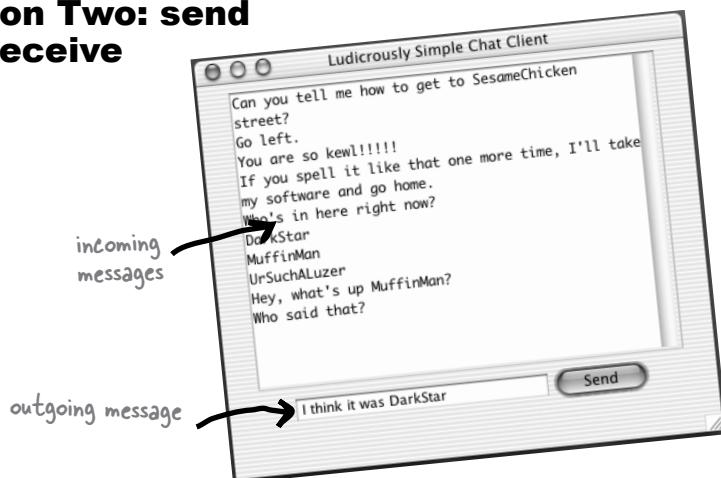
This is where we make the Socket  
and the PrintWriter (it's called  
from the go() method right before  
displaying the app GUI)

Now we actually do the writing.  
Remember, the writer is chained to  
the output stream from the Socket,  
so whenever we do a println(), it goes  
over the network to the server!

If you want to try this now, type in  
the Ready-bake chat server code  
listed at the end of this chapter.  
First, start the server in one terminal.  
Next, use another terminal to start  
this client.

## improving the chat client

### Version Two: send and receive



The Server sends a message to all client participants, as soon as the message is received by the server. When a client sends a message, it doesn't appear in the incoming message display area until the server sends it to everyone.

### Big Question: HOW do you get messages from the server?

Should be easy; when you set up the networking make an input stream as well (probably a BufferedReader). Then read messages using readLine().

### Bigger Question: WHEN do you get messages from the server?

Think about that. What are the options?

#### ① Option One: Poll the server every 20 seconds

**Pros:** Well, it's do-able

**Cons:** How does the server know what you've seen and what you haven't? The server would have to store the messages, rather than just doing a distribute-and-forget each time it gets one. And why 20 seconds? A delay like this affects usability, but as you reduce the delay, you risk hitting your server needlessly. Inefficient.

#### ② Option Two: Read something in from the server each time the user sends a message.

**Pros:** Do-able, very easy

**Cons:** Stupid. Why choose such an arbitrary time to check for messages? What if a user is a lurker and doesn't send anything?

#### ③ Option Three: Read messages as soon as they're sent from the server

**Pros:** Most efficient, best usability

**Cons:** How do you do two things at the same time? Where would you put this code? You'd need a loop somewhere that was always waiting to read from the server. But where would that go? Once you launch the GUI, nothing happens until an event is fired by a GUI component.



In Java you really CAN walk and chew gum at the same time.

### You know by now that we're going with option three.

We want something to run continuously, checking for messages from the server, but *without interrupting the user's ability to interact with the GUI!* So while the user is happily typing new messages or scrolling through the incoming messages, we want something *behind the scenes* to keep reading in new input from the server.

That means we finally need a new thread. A new, separate stack.

We want everything we did in the Send-Only version (version one) to work the same way, while a new *process* runs along side that reads information from the server and displays it in the incoming text area.

Well, not quite. Unless you have multiple processors on your computer, each new Java thread is not actually a separate process running on the OS. But it almost *feels* as though it is.

### Multithreading in Java

Java has multiple threading built right into the fabric of the language. And it's a snap to make a new thread of execution:

```
Thread t = new Thread();
t.start();
```

That's it. By creating a new Thread *object*, you've launched a separate *thread of execution*, with its very own call stack.

#### **Except for one problem.**

That thread doesn't actually *do* anything, so the thread "dies" virtually the instant it's born. When a thread dies, its new stack disappears again. End of story.

So we're missing one key component—the thread's *job*. In other words, we need the code that you want to have run by a separate thread.

Multiple threading in Java means we have to look at both the *thread* and the *job* that's *run* by the thread. And we'll also have to look at the Thread *class* in the java.lang package. (Remember, java.lang is the package you get imported for free, implicitly, and it's where the classes most fundamental to the language live, including String and System.)

## threads and Thread

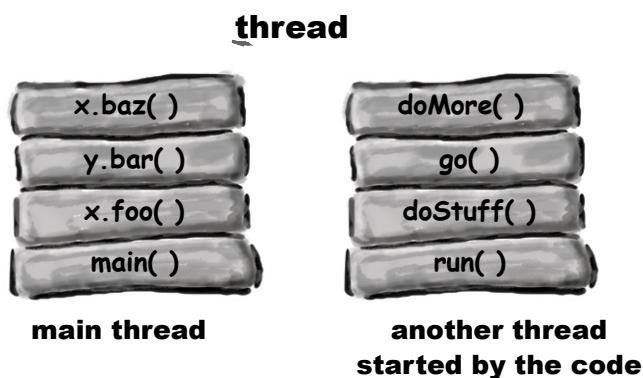
# Java has multiple threads but only one Thread class

We can talk about *thread* with a lower-case ‘t’ and **Thread** with a capital ‘T’. When you see *thread*, we’re talking about a separate thread of execution. In other words, a separate call stack. When you see **Thread**, think of the Java naming convention. What, in Java, starts with a capital letter? Classes and interfaces. In this case, **Thread** is a class in the `java.lang` package. A **Thread** object represents a *thread of execution*; you’ll create an instance of class **Thread** each time you want to start up a new *thread* of execution.

A **thread** is a separate ‘*thread of execution*’. In other words, a **separate call stack**.

A **Thread** is a Java class that represents a **thread**.

To make a *thread*, make a **Thread**.



A thread (lower-case ‘t’) is a separate thread of execution. That means a separate call stack. Every Java application starts up a main thread—the thread that puts the `main()` method on the bottom of the stack. The JVM is responsible for starting the main thread (and other threads, as it chooses, including the garbage collection thread). As a programmer, you can write code to start other threads of your own.

## Thread

Thread
void join()
void start()

## java.lang.Thread class

**Thread** (capital ‘T’) is a class that represents a thread of execution. It has methods for starting a thread, joining one thread with another, and putting a thread to sleep. (It has more methods; these are just the crucial ones we need to use now).

## What does it mean to have more than one call stack?

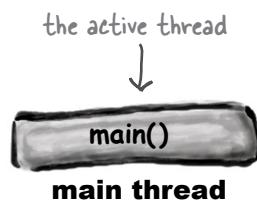
With more than one call stack, you get the *appearance* of having multiple things happen at the same time. In reality, only a true multiprocessor system can actually do more than one thing at a time, but with Java threads, it can *appear* that you're doing several things simultaneously. In other words, execution can move back and forth between stacks so rapidly that you feel as though all stacks are executing at the same time. Remember, Java is just a process running on your underlying OS. So first, Java *itself* has to be 'the currently executing process' on the OS. But once Java gets its turn to execute, exactly *what* does the JVM run? Which bytecodes execute? Whatever is on the top of the currently-running stack! And in 100 milliseconds, the currently executing code might switch to a *different* method on a *different* stack.

One of the things a thread must do is keep track of which statement (of which method) is currently executing on the thread's stack.

It might look something like this:

- 1** The JVM calls the main() method.

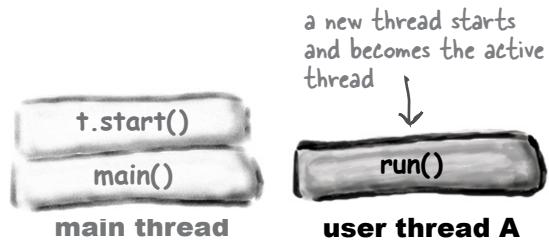
```
public static void main(String[] args) {
    ...
}
```



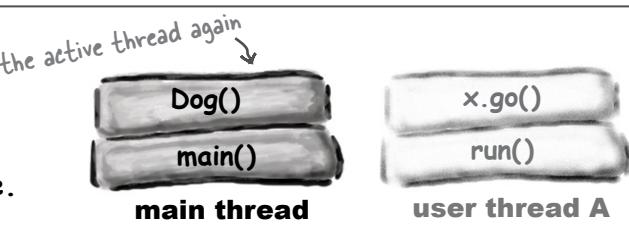
- 2** main() starts a new thread. The main thread is temporarily frozen while the new thread starts running.

```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

*you'll learn what this means in just a moment...*



- 3** The JVM switches between the new thread (user thread A) and the original main thread, until both threads complete.



## How to launch a new thread:

### ➊ Make a Runnable object (the thread's job)

```
Runnable threadJob = new MyRunnable();
```

Runnable is an interface you'll learn about on the next page. You'll write a class that implements the Runnable interface, and that class is where you'll define the work that a thread will perform. In other words, the method that will be run from the thread's new call stack.



### ➋ Make a Thread object (the worker) and give it a Runnable (the job)

```
Thread myThread = new Thread(threadJob);
```

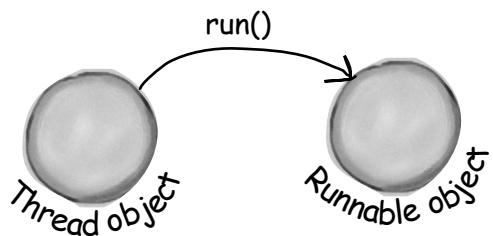
Pass the new Runnable object to the Thread constructor. This tells the new Thread object which method to put on the bottom of the new stack—the Runnable's run() method.



### ➌ Start the Thread

```
myThread.start();
```

Nothing happens until you call the Thread's start() method. That's when you go from having just a Thread instance to having a new thread of execution. When the new thread starts up, it takes the Runnable object's run() method and puts it on the bottom of the new thread's stack.



## Every Thread needs a job to do. A method to put on the new thread stack.



```
public void run() {
    // code that will be run by the new thread
}
```

How does the thread know which method to put at the bottom of the stack? Because Runnable defines a contract. Because Runnable is an interface. A thread's job can be defined in any class that implements the Runnable interface. The thread cares only that you pass the Thread constructor an object of a class that implements Runnable.

When you pass a Runnable to a Thread constructor, you're really just giving the Thread a way to get to a run() method. You're giving the Thread its job to do.

Runnable is to a Thread what a job is to a worker. A Runnable is the job a thread is supposed to run.

A Runnable holds the method that goes on the bottom of the new thread's stack: run().

The Runnable interface defines only one method, public void run(). (Remember, it's an interface so the method is public regardless of whether you type it in that way.)

## Runnable interface

To make a job for your thread,  
implement the Runnable interface

Runnable is in the `java.lang` package,  
so you don't need to import it.

```
public class MyRunnable implements Runnable {  
  
    public void run() {  
        go();  
    }  
  
    public void go() {  
        doMore();  
    }  
  
    public void doMore() {  
        System.out.println("top o' the stack");  
    }  
}
```

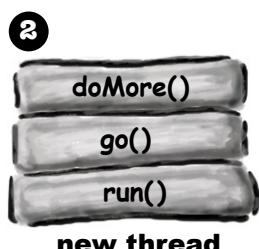
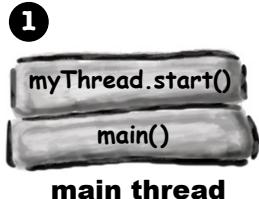
Runnable has only one method to implement: `public void run()` (with no arguments). This is where you put the JOB the thread is supposed to run. This is the method that goes at the bottom of the new stack.

```
class ThreadTester {  
  
    public static void main (String[] args) {  
  
        Runnable threadJob = new MyRunnable();  
        Thread myThread = new Thread(threadJob);  
    }  
}
```

Pass the new Runnable instance into the new Thread constructor. This tells the thread what method to put on the bottom of the new stack. In other words, the first method that the new thread will run.

```
1 myThread.start();  
    System.out.println("back in main");  
}  
}
```

You won't get a new thread of execution until you call `start()` on the Thread instance. A thread is not really a thread until you start it. Before that, it's just a Thread instance, like any other object, but it won't have any real 'threadness'.



## Brain Barbell

What do you think the output will be if you run the `ThreadTester` class? (we'll find out in a few pages)

## The three states of a new thread

```
Thread t = new Thread(r);
```



```
Thread t = new Thread(r);
```

A Thread instance has been created but not started. In other words, there is a Thread *object*, but no *thread of execution*.

```
t.start();
```

When you start the thread, it moves into the runnable state. This means the thread is ready to run and just waiting for its Big Chance to be selected for execution. At this point, there is a new call stack for this thread.

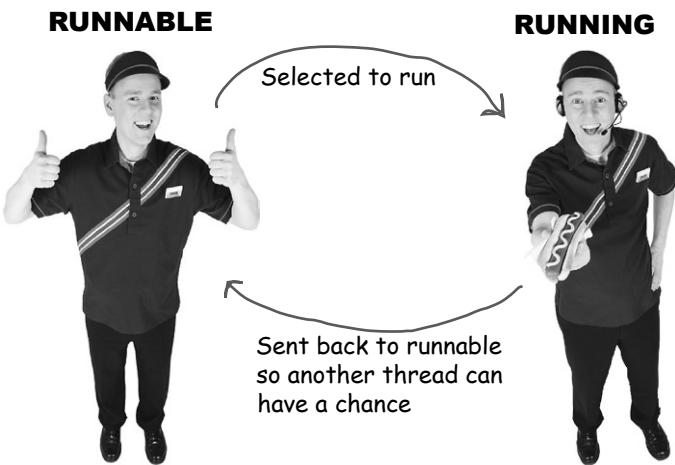
This is the state all threads lust after! To be The Chosen One. The Currently Running Thread. Only the JVM thread scheduler can make that decision. You can sometimes *influence* that decision, but you cannot force a thread to move from runnable to running. In the running state, a thread (and ONLY this thread) has an active call stack, and the method on the top of the stack is executing.

**But there's more. Once the thread becomes runnable, it can move back and forth between runnable, running, and an additional state: temporarily not runnable (also known as 'blocked').**

## thread states

### Typical runnable/running loop

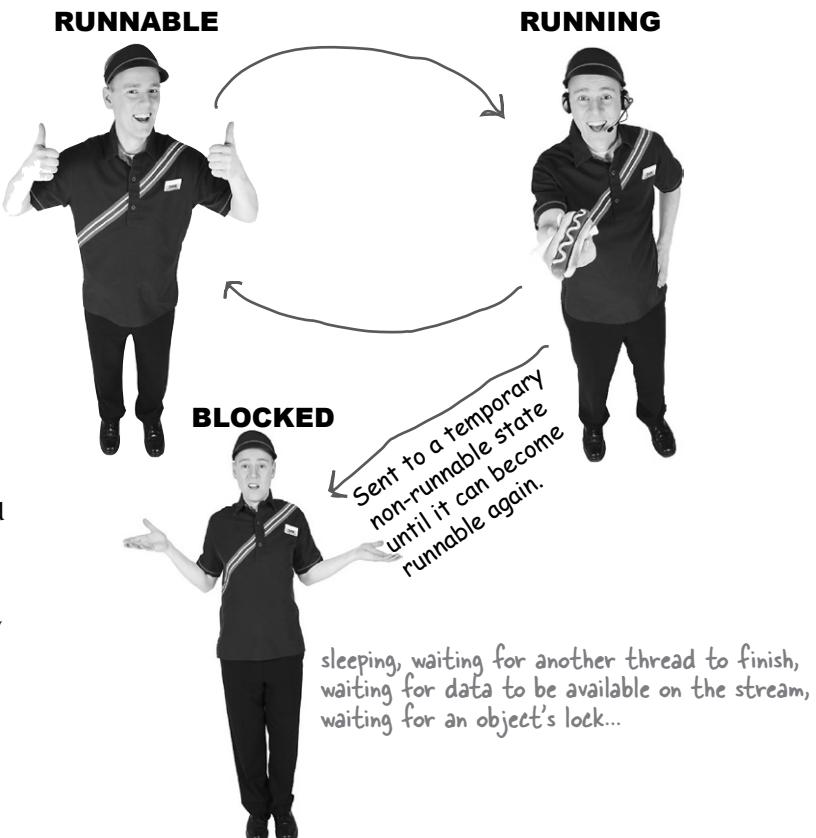
Typically, a thread moves back and forth between runnable and running, as the JVM thread scheduler selects a thread to run and then kicks it back out so another thread gets a chance.



### A thread can be made temporarily not-runnable

The thread scheduler can move a running thread into a blocked state, for a variety of reasons. For example, the thread might be executing code to read from a Socket input stream, but there isn't any data to read. The scheduler will move the thread out of the running state until something becomes available. Or the executing code might have told the thread to put itself to sleep (`sleep()`). Or the thread might be waiting because it tried to call a method on an object, and that object was 'locked'. In that case, the thread can't continue until the object's lock is freed by the thread that has it.

All of those conditions (and more) cause a thread to become temporarily not-runnable.



## The Thread Scheduler

The thread scheduler makes all the decisions about who moves from runnable to running, and about when (and under what circumstances) a thread leaves the running state. The scheduler decides who runs, and for how long, and where the threads go when the scheduler decides to kick them out of the currently-running state.

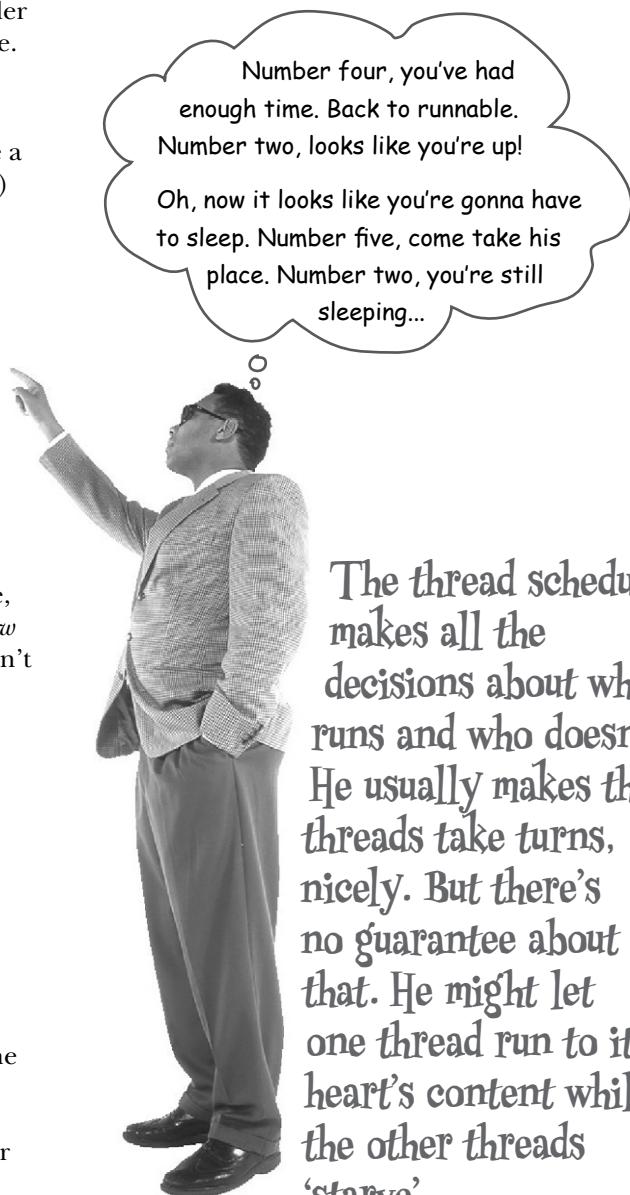
You can't control the scheduler. There is no API for calling methods on the scheduler. Most importantly, there are no guarantees about scheduling! (There are a few *almost*-guarantees, but even those are a little fuzzy.)

The bottom line is this: *do not base your program's correctness on the scheduler working in a particular way!*

The scheduler implementations are different for different JVM's, and even running the same program on the same machine can give you different results. One of the worst mistakes new Java programmers make is to test their multi-threaded program on a single machine, and assume the thread scheduler will always work that way, regardless of where the program runs.

So what does this mean for write-once-run-anywhere? It means that to write platform-independent Java code, your multi-threaded program must work no matter *how* the thread scheduler behaves. That means that you can't be dependent on, for example, the scheduler making sure all the threads take nice, perfectly fair and equal turns at the running state. Although highly unlikely today, your program might end up running on a JVM with a scheduler that says, "OK thread five, you're up, and as far as I'm concerned, you can stay here until you're done, when your run() method completes."

The secret to almost everything is *sleep*. That's right, *sleep*. Putting a thread to sleep, even for a few milliseconds, forces the currently-running thread to leave the running state, thus giving another thread a chance to run. The thread's sleep() method does come with *one* guarantee: a sleeping thread will *not* become the currently-running thread before the length of its sleep time has expired. For example, if you tell your thread to sleep for two seconds (2,000 milliseconds), that thread can never become the running thread again until sometime *after* the two seconds have passed.



**The thread scheduler makes all the decisions about who runs and who doesn't. He usually makes the threads take turns, nicely. But there's no guarantee about that. He might let one thread run to its heart's content while the other threads 'starve'.**

thread scheduling

## An example of how unpredictable the scheduler can be...

Running this code on one machine:

```
public class MyRunnable implements Runnable {  
  
    public void run() {  
        go();  
    }  
  
    public void go() {  
        doMore();  
    }  
  
    public void doMore() {  
        System.out.println("top o' the stack");  
    }  
}  
  
class ThreadTestDrive {  
  
    public static void main (String[] args) {  
  
        Runnable threadJob = new MyRunnable();  
        Thread myThread = new Thread(threadJob);  
  
        myThread.start();  
  
        System.out.println("back in main");  
    }  
}
```

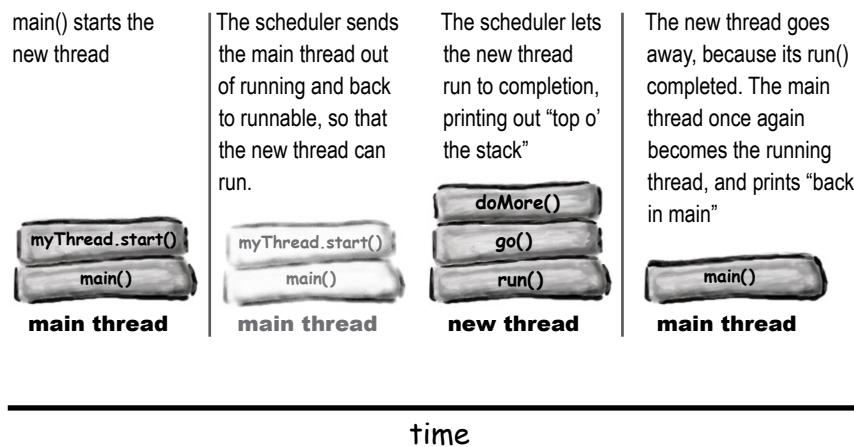
Notice how the order changes  
randomly. Sometimes the new thread  
finishes first, and sometimes the main  
thread finishes first.

Produced this output:

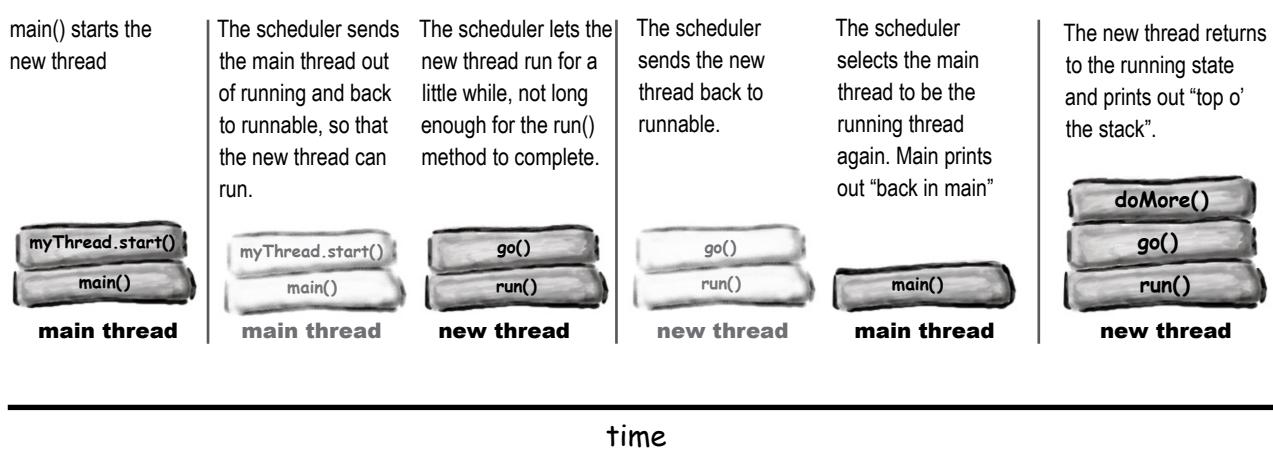
```
File Edit Window Help PickMe  
% java ThreadTestDrive  
back in main  
top o' the stack  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
top o' the stack  
back in main  
% java ThreadTestDrive  
back in main  
% java ThreadTestDrive  
back in main  
top o' the stack
```

## How did we end up with different results?

### Sometimes it runs like this:



### And sometimes it runs like this:



*there are no*  
**Dumb Questions**

**Q:** I've seen examples that don't use a separate Runnable implementation, but instead just make a subclass of Thread and override the Thread's run() method. That way, you call the Thread's no-arg constructor when you make the new thread;

```
Thread t = new Thread(); // no Runnable
```

**A:** Yes, that *is* another way of making your own thread, but think about it from an OO perspective. What's the purpose of subclassing? Remember that we're talking about two different things here—the Thread and the thread's job. From an OO view, those two are very separate activities, and belong in separate classes. The only time you want to subclass/extend the Thread class, is if you are making a new and more specific type of Thread. In other words, if you think of the Thread as the worker, don't extend the Thread class unless you need more specific worker behaviors. But if all you need is a new job to be run by a Thread/worker, then implement Runnable in a separate, job-specific (not worker-specific) class.

This is a design issue and not a performance or language issue. It's perfectly legal to subclass Thread and override the run() method, but it's rarely a good idea.

**Q:** Can you reuse a Thread object? Can you give it a new job to do and then restart it by calling start() again?

**A:** No. Once a thread's run() method has completed, the thread can never be restarted. In fact, at that point the thread moves into a state we haven't talked about—**dead**. In the dead state, the thread has finished its run() method and can never be restarted. The Thread object might still be on the heap, as a living object that you can call other methods on (if appropriate), but the Thread object has permanently lost its 'threadness'. In other words, there is no longer a separate call stack, and the Thread object is no longer a *thread*. It's just an object, at that point, like all other objects.

But, there are design patterns for making a pool of threads that you can keep using to perform different jobs. But you don't do it by restarting() a dead thread.



### BULLET POINTS

- A thread with a lower-case 't' is a separate thread of execution in Java.
- Every thread in Java has its own call stack.
- A Thread with a capital 'T' is the java.lang.Thread class. A Thread object represents a thread of execution.
- A Thread needs a job to do. A Thread's job is an instance of something that implements the Runnable interface.
- The Runnable interface has just a single method, run(). This is the method that goes on the bottom of the new call stack. In other words, it is the first method to run in the new thread.
- To launch a new thread, you need a Runnable to pass to the Thread's constructor.
- A thread is in the NEW state when you have instantiated a Thread object but have not yet called start().
- When you start a thread (by calling the Thread object's start() method), a new stack is created, with the Runnable's run() method on the bottom of the stack. The thread is now in the RUNNABLE state, waiting to be chosen to run.
- A thread is said to be RUNNING when the JVM's thread scheduler has selected it to be the currently-running thread. On a single-processor machine, there can be only one currently-running thread.
- Sometimes a thread can be moved from the RUNNING state to a BLOCKED (temporarily non-Runnable) state. A thread might be blocked because it's waiting for data from a stream, or because it has gone to sleep, or because it is waiting for an object's lock.
- Thread scheduling is not guaranteed to work in any particular way, so you cannot be certain that threads will take turns nicely. You can help influence turn-taking by putting your threads to sleep periodically.



## Putting a thread to sleep

One of the best ways to help your threads take turns is to put them to sleep periodically. All you need to do is call the static `sleep()` method, passing it the sleep duration, in milliseconds.

For example:

```
Thread.sleep(2000);
```

will knock a thread out of the running state, and keep it out of the runnable state for two seconds.

The thread *can't* become the running thread again until after at least two seconds have passed.

A bit unfortunately, the sleep method throws an `InterruptedException`, a checked exception, so all calls to sleep must be wrapped in a try/catch (or declared). So a sleep call really looks like this:

```
try {
    Thread.sleep(2000);
} catch(InterruptedException ex) {
    ex.printStackTrace();
}
```

Your thread will probably *never* be interrupted from sleep; the exception is in the API to support a thread communication mechanism that almost nobody uses in the Real World. But, you still have to obey the handle or declare law, so you need to get used to wrapping your `sleep()` calls in a try/catch.

Now you know that your thread won't wake up *before* the specified duration, but is it possible that it will wake up some time *after* the 'timer' has expired? Yes and no. It doesn't matter, really, because when the thread wakes up, *it always goes back to the runnable state!* The thread won't automatically wake up at the designated time and become the currently-running thread. When a thread wakes up, the thread is once again at the mercy of the thread scheduler. Now, for applications that don't require perfect timing, and that have only a few threads, it might appear as though the thread wakes up and resumes running right on schedule (say, after the 2000 milliseconds). But don't bet your program on it.

**Put your thread to sleep if you want to be sure that other threads get a chance to run.**

**When the thread wakes up, it always goes back to the runnable state and waits for the thread scheduler to choose it to run again.**

using Thread.sleep()

## Using sleep to make our program more predictable.

Remember our earlier example that kept giving us different results each time we ran it? Look back and study the code and the sample output. Sometimes main had to wait until the new thread finished (and printed “top o’ the stack”), while other times the new thread would be sent back to runnable before it was finished, allowing the main thread to come back in and print out “back in main”. How can we fix that? Stop for a moment and answer this question: “Where can you put a sleep() call, to make sure that “back in main” always prints before “top o’ the stack”?

We’ll wait while you work out an answer (there’s more than one answer that would work).

Figure it out?

```
public class MyRunnable implements Runnable {  
  
    public void run() {  
        go();  
    }  
  
    public void go() {  
  
        try {  
            Thread.sleep(2000);  
        } catch(InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        doMore();  
    }  
  
    public void doMore() {  
        System.out.println("top o' the stack");  
    }  
}  
  
class ThreadTestDrive {  
    public static void main (String[] args) {  
        Runnable theJob = new MyRunnable();  
        Thread t = new Thread(theJob);  
        t.start();  
        System.out.println("back in main");  
    }  
}
```

This is what we want—a consistent order of print statements:

```
File Edit Window Help SnoozeButton  
% java ThreadTestDrive  
back in main  
top o' the stack  
% java ThreadTestDrive  
back in main  
top o' the stack  
% java ThreadTestDrive  
back in main  
top o' the stack  
% java ThreadTestDrive  
back in main  
top o' the stack  
% java ThreadTestDrive  
back in main  
top o' the stack
```

Calling sleep here will force the new thread to leave the currently-running state!

The main thread will become the currently-running thread again, and print out “back in main”. Then there will be a pause (for about two seconds) before we get to this line, which calls doMore() and prints out “top o’ the stack”

## Making and starting two threads

Threads have names. You can give your threads a name of your choosing, or you can accept their default names. But the cool thing about names is that you can use them to tell which thread is running. The following example starts two threads. Each thread has the same job: run in a loop, printing the currently-running thread's name with each iteration.

```

public class RunThreads implements Runnable {

    public static void main(String[] args) {
        RunThreads runner = new RunThreads(); ← Make one Runnable instance.
        Thread alpha = new Thread(runner); ← Make two threads, with the same Runnable (the
        Thread beta = new Thread(runner); ← same job—we'll talk more about the "two threads
        alpha.setName("Alpha thread"); ← and one Runnable" in a few pages).
        beta.setName("Beta thread"); ← Name the threads.
        alpha.start(); ←
        beta.start(); ← Start the threads.
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running");
        }
    }
}

```

Each thread will run through this loop,  
printing its name each time.

Part of the output when  
the loop iterates 25 times.

## What will happen?

Will the threads take turns? Will you see the thread names alternating? How often will they switch? With each iteration? After five iterations?

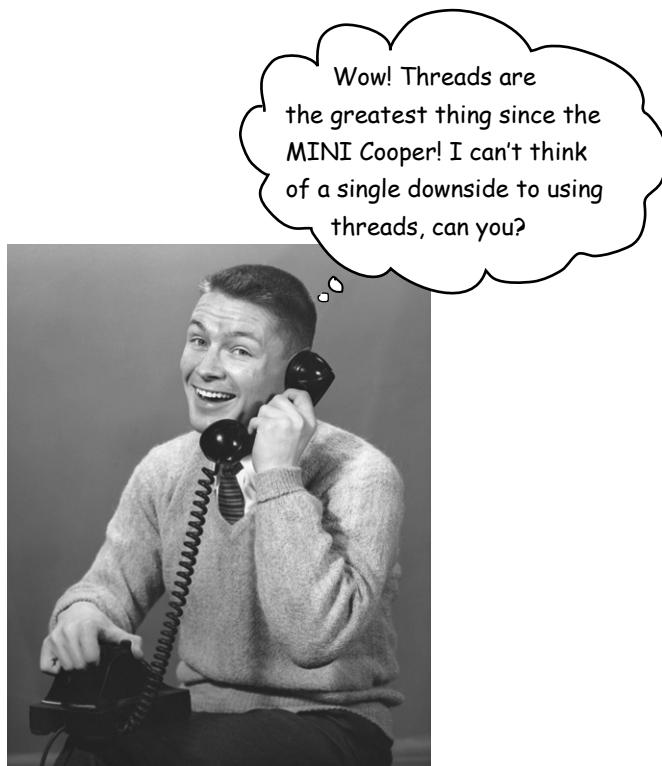
You already know the answer: *we don't know!* It's up to the scheduler. And on your OS, with your particular JVM, on your CPU, you might get very different results.

Running under OS X 10.2 (Jaguar), with five or fewer iterations, the Alpha thread runs to completion, then the Beta thread runs to completion. Very consistent. Not guaranteed, but very consistent.

But when you up the loop to 25 or more iterations, things start to wobble. The Alpha thread might not get to complete all 25 iterations before the scheduler sends it back to runnable to let the Beta thread have a chance.

File Edit Window Help Centauri
Alpha thread is running
Alpha thread is running
Alpha thread is running
Beta thread is running
Alpha thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Alpha thread is running

**aren't threads wonderful?**



## **Um, yes. There IS a dark side. Threads can lead to concurrency 'issues'.**

Concurrency issues lead to race conditions. Race conditions lead to data corruption. Data corruption leads to fear... you know the rest.

It all comes down to one potentially deadly scenario: two or more threads have access to a single object's *data*. In other words, methods executing on two different stacks are both calling, say, getters or setters on a single object on the heap.

It's a whole 'left-hand-doesn't-know-what-the-right-hand-is-doing' thing. Two threads, without a care in the world, humming along executing their methods, each thread thinking that he is the One True Thread. The only one that matters. After all, when a thread is not running, and in runnable (or blocked) it's essentially knocked unconscious. When it becomes the currently-running thread again, it doesn't know that it ever stopped.

## Marriage in Trouble. Can this couple be saved?

*Next, on a very special Dr. Steve Show*

[Transcript from episode #42]

Welcome to the Dr. Steve show.



We've got a story today that's centered around the top two reasons why couples split up—finances and sleep.

Today's troubled pair, Ryan and Monica, share a bed and a bank account. But not for long if we can't find a solution. The problem? The classic "two people—one bank account" thing.

Here's how Monica described it to me:

"Ryan and I agreed that neither of us will overdraw the checking account. So the procedure is, whoever wants to withdraw money *must* check the balance in the account *before* making the withdrawal. It all seemed so simple. But suddenly we're bouncing checks and getting hit with overdraft fees!"

I thought it wasn't possible, I thought our procedure was safe. But then *this* happened:

Ryan needed \$50, so he checked the balance in the account, saw that it was \$100. No problem. So, he plans to withdraw the money. **But first he falls asleep!**

And that's where I come in, while Ryan's still asleep, and now I want to withdraw \$100. I check the balance, and it's \$100 (because Ryan's still asleep and hasn't made his withdrawal), so I think, no problem. So I make the withdrawal, and again no problem. But then Ryan wakes up, completes *his* withdrawal, and we're suddenly overdrawn! He didn't even know that he fell asleep, so he just went ahead and completed his transaction without checking the balance again. You've got to help us Dr. Steve!"

Is there a solution? Are they doomed? We can't stop Ryan from falling asleep, but can we make sure that Monica can't get her hands on the bank account until after he wakes up?

Take a moment and think about that while we go to a commercial break.



Ryan and Monica: victims of the "two people, one account" problem.

Ryan falls asleep after he checks the balance but before he makes the withdrawal. When he wakes up, he immediately makes the withdrawal without checking the balance again.

## Ryan and Monica code

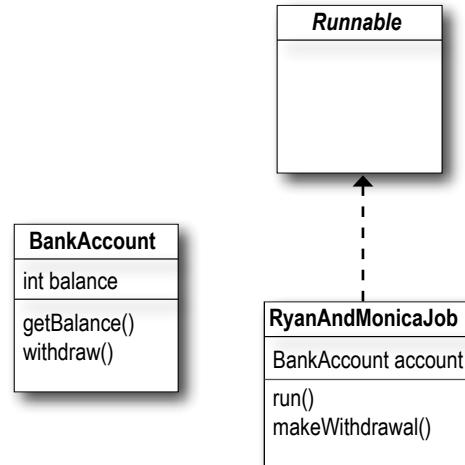
# The Ryan and Monica problem, in code

The following example shows what can happen when *two* threads (Ryan and Monica) share a *single* object (the bank account).

The code has two classes, BankAccount, and MonicaAndRyanJob. The MonicaAndRyanJob class implements Runnable, and represents the behavior that Ryan and Monica both have—checking the balance and making withdrawals. But of course, each thread falls asleep *in between* checking the balance and actually making the withdrawal.

The MonicaAndRyanJob class has an instance variable of type BankAccount, that represents their shared account.

The code works like this:



### ① Make one instance of RyanAndMonicaJob.

The RyanAndMonicaJob class is the Runnable (the job to do), and since both Monica and Ryan do the same thing (check balance and withdraw money), we need only one instance.

```
RyanAndMonicaJob theJob = new RyanAndMonicaJob();
```

**In the run() method, do exactly what Ryan and Monica would do—check the balance and, if there's enough money, make the withdrawal.**

### ② Make two threads with the same Runnable (the RyanAndMonicaJob instance)

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

**This should protect against overdrawing the account.**

### ③ Name and start the threads

```
one.setName("Ryan");
two.setName("Monica");
one.start();
two.start();
```

**Except... Ryan and Monica always fall asleep after they check the balance but before they finish the withdrawal.**

### ④ Watch both threads execute the run() method (check the balance and make a withdrawal)

One thread represents Ryan, the other represents Monica. Both threads continually check the balance and then make a withdrawal, but only if it's safe!

```
if (account.getBalance() >= amount) {
    try {
        Thread.sleep(500);
    } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

## The Ryan and Monica example

```

class BankAccount {
    private int balance = 100; ← The account starts with a
                                balance of $100.

    public int getBalance() {
        return balance;
    }

    public void withdraw(int amount) {
        balance = balance - amount;
    }
}

public class RyanAndMonicaJob implements Runnable {
    private BankAccount account = new BankAccount(); ← There will be only ONE instance of the
                                                       RyanAndMonicaJob. That means only
                                                       ONE instance of the bank account. Both
                                                       threads will access this one account.

    public static void main (String [] args) {
        RyanAndMonicaJob theJob = new RyanAndMonicaJob(); ← Instantiate the Runnable (job)
        Thread one = new Thread(theJob); ← Make two threads, giving each thread the same Runnable
        Thread two = new Thread(theJob); ← job. That means both threads will be accessing the one
        one.setName("Ryan");
        two.setName("Monica");
        one.start();
        two.start();
    }

    public void run() {
        for (int x = 0; x < 10; x++) {
            makeWithdrawal(10);
            if (account.getBalance() < 0) {
                System.out.println("Overdrawn!");
            }
        }
    }

    private void makeWithdrawal(int amount) {
        if (account.getBalance() >= amount) { ← In the run() method, a thread loops through and tries
                                                to make a withdrawal with each iteration. After the
                                                withdrawal, it checks the balance once again to see if
                                                the account is overdrawn.

            System.out.println(Thread.currentThread().getName() + " is about to withdraw");
            try {
                System.out.println(Thread.currentThread().getName() + " is going to sleep");
                Thread.sleep(500);
            } catch(InterruptedException ex) {ex.printStackTrace();}
            System.out.println(Thread.currentThread().getName() + " woke up.");
            account.withdraw(amount);
            System.out.println(Thread.currentThread().getName() + " completes the withdrawal");
        }
        else {
            System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
        }
    }
}

```

We put in a bunch of print statements so we can see what's happening as it runs.

## Ryan and Monica output

```
File Edit Window Help Visa
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawal
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawal
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawal
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawal
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawal
Sorry, not enough for Monica
Ryan woke up.
Ryan completes the withdrawal
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
```

How did this happen? →

**The makeWithdrawal() method always checks the balance before making a withdrawal, but still we overdraw the account.**

### Here's one scenario:

Ryan checks the balance, sees that there's enough money, and then falls asleep.

Meanwhile, Monica comes in and checks the balance. She, too, sees that there's enough money. She has no idea that Ryan is going to wake up and complete a withdrawal.

Monica falls asleep.

Ryan wakes up and completes his withdrawal.

Monica wakes up and completes her withdrawal. Big Problem! In between the time when she checked the balance and made the withdrawal, Ryan woke up and pulled money from the account.

**Monica's check of the account was not valid, because Ryan had already checked and was still in the middle of making a withdrawal.**

Monica must be stopped from getting into the account until Ryan wakes up and finishes his transaction. And vice-versa.

## They need a lock for account access!

The lock works like this:

- ① There's a lock associated with the bank account transaction (checking the balance and withdrawing money). There's only one key, and it stays with the lock until somebody wants to access the account.



**The bank account transaction is unlocked when nobody is using the account.**

- ② When Ryan wants to access the bank account (to check the balance and withdraw money), he locks the lock and puts the key in his pocket. Now nobody else can access the account, since the key is gone.



**When Ryan wants to access the account, he secures the lock and takes the key.**

- ③ Ryan keeps the key in his pocket until he finishes the transaction. He has the only key, so Monica can't access the account (or the checkbook) until Ryan unlocks the account and returns the key.

Now, even if Ryan falls asleep after he checks the balance, he has a guarantee that the balance will be the same when he wakes up, because he kept the key while he was asleep!



**When Ryan is finished, he unlocks the lock and returns the key. Now the key is available for Monica (or Ryan again) to access the account.**

using synchronized

## We need the `makeWithdrawal()` method to run as one atomic thing.



We need to make sure that once a thread enters the `makeWithdrawal()` method, *it must be allowed to finish the method before any other thread can enter.*

In other words, we need to make sure that once a thread has checked the account balance, that thread has a guarantee that it can wake up and finish the withdrawal *before any other thread can check the account balance!*

Use the **synchronized** keyword to modify a method so that only one thread at a time can access it.

That's how you protect the bank account! You don't put a lock on the bank account itself; you lock the method that does the banking transaction. That way, one thread gets to complete the whole transaction, start to finish, even if that thread falls asleep in the middle of the method!

So if you don't lock the bank account, then what exactly *is* locked? Is it the method? The Runnable object? The thread itself?

We'll look at that on the next page. In code, though, it's quite simple—just add the **synchronized** modifier to your method declaration:

```
private synchronized void makeWithdrawal(int amount) {  
  
    if (account.getBalance() >= amount) {  
        System.out.println(Thread.currentThread().getName() + " is about to withdraw");  
        try {  
            System.out.println(Thread.currentThread().getName() + " is going to sleep");  
            Thread.sleep(500);  
        } catch(InterruptedException ex) {ex.printStackTrace();}  
        System.out.println(Thread.currentThread().getName() + " woke up.");  
        account.withdraw(amount);  
        System.out.println(Thread.currentThread().getName() + " completes the withdrawal");  
    } else {  
        System.out.println("Sorry, not enough for " + Thread.currentThread().getName());  
    }  
}
```

(Note for you physics-savvy readers: yes, the convention of using the word 'atomic' here does not reflect the whole subatomic particle thing. Think Newton, not Einstein, when you hear the word 'atomic' in the context of threads or transactions. Hey, it's not OUR convention. If WE were in charge, we'd apply Heisenberg's Uncertainty Principle to pretty much everything related to threads.)



**The `synchronized` keyword means that a thread needs a key in order to access the synchronized code.**

**To protect your data (like the bank account), synchronize the methods that act on that data.**

## Using an object's lock

Every object has a lock. Most of the time, the lock is unlocked, and you can imagine a virtual key sitting with it. Object locks come into play only when there are synchronized methods. When an object has one or more synchronized methods, *a thread can enter a synchronized method only if the thread can get the key to the object's lock!*

The locks are not per *method*, they are per *object*. If an object has two synchronized methods, it does not simply mean that you can't have two threads entering the same method. It means you can't have two threads entering *any* of the synchronized methods.

Think about it. If you have multiple methods that can potentially act on an object's instance variables, all those methods need to be protected with synchronized.

The goal of synchronization is to protect critical data. But remember, you don't lock the data itself, you synchronize the methods that *access* that data.

So what happens when a thread is cranking through its call stack (starting with the `run()` method) and it suddenly hits a synchronized method? The thread recognizes that it needs a key for that object before it can enter the method. It looks for the key (this is all handled by the JVM; there's no API in Java for accessing object locks), and if the key is available, the thread grabs the key and enters the method.

From that point forward, the thread hangs on to that key like the thread's life depends on it. The thread won't give up the key until it completes the synchronized method. So while that thread is holding the key, no other threads can enter *any* of that object's synchronized methods, because the one key for that object won't be available.



**Every Java object has a lock.  
A lock has only one key.**

**Most of the time, the lock is unlocked and nobody cares.**

**But if an object has synchronized methods, a thread can enter one of the synchronized methods ONLY if the key for the object's lock is available. In other words, only if another thread hasn't already grabbed the one key.**

## synchronization matters

# The dreaded “Lost Update” problem

Here's another classic concurrency problem, that comes from the database world. It's closely related to the Ryan and Monica story, but we'll use this example to illustrate a few more points.

The lost update revolves around one process:

Step 1: Get the balance in the account

```
int i = balance;
```

Step 2: Add 1 to that balance

```
balance = i + 1; ←
```

Probably not an atomic process

Even if we used the more common syntax: `balance++`; there is no guarantee that the compiled bytecode will be an “atomic process”. In fact, it probably won't.

In the “Lost Update” problem, we have two threads, both trying to increment the balance. Take a look at the code, and then we'll look at the real problem:

```
class TestSync implements Runnable {  
  
    private int balance;  
  
    public void run() {  
        for(int i = 0; i < 50; i++) { ←  
            increment();  
            System.out.println("balance is " + balance);  
        }  
    }  
  
    public void increment() {  
        int i = balance; ←  
        balance = i + 1;  
    }  
}  
  
public class TestSyncTest {  
    public static void main (String[] args) {  
        TestSync job = new TestSync();  
        Thread a = new Thread(job);  
        Thread b = new Thread(job);  
        a.start();  
        b.start();  
    }  
}
```

each thread runs 50 times,  
incrementing the balance on  
each iteration

Here's the crucial part! We increment the balance by  
adding 1 to whatever the value of balance was AT THE  
TIME WE READ IT (rather than adding 1 to whatever  
the CURRENT value is)

## Let's run this code...

### ① Thread A runs for awhile



Put the value of balance into variable i.  
Balance is 0, so i is now 0.  
Set the value of balance to the result of  $i + 1$ .  
Now balance is 1.  
Put the value of balance into variable i.  
Balance is 1, so i is now 1.  
Set the value of balance to the result of  $i + 1$ .  
Now balance is 2.

### ② Thread B runs for awhile



Put the value of balance into variable i.  
Balance is 2, so i is now 2.  
Set the value of balance to the result of  $i + 1$ .  
Now balance is 3.  
Put the value of balance into variable i.  
Balance is 3, so i is now 3.

*[now thread B is sent back to runnable,  
before it sets the value of balance to 4]*

### ③ Thread A runs again, picking up where it left off



Put the value of balance into variable i.  
Balance is 3, so i is now 3.  
Set the value of balance to the result of  $i + 1$ .  
Now balance is 4.  
Put the value of balance into variable i.  
Balance is 4, so i is now 4.  
Set the value of balance to the result of  $i + 1$ .  
Now balance is 5.

### ④ Thread B runs again, and picks up exactly where it left off!



Set the value of balance to the result of  $i + 1$ .  
**Now balance is 4.**

*Yikes!!*

Thread A updated it to 5, but  
now B came back and stepped  
on top of the update A made,  
as if A's update never happened.

**We lost the last updates  
that Thread A made!  
Thread B had previously  
done a 'read' of the value  
of balance, and when B  
woke up, it just kept going  
as if it never missed a beat.**

## synchronizing methods

# Make the increment() method atomic. Synchronize it!



Synchronizing the increment() method solves the “Lost Update” problem, because it keeps the two steps in the method as one unbreakable unit.

```
public synchronized void increment() {  
    int i = balance;  
    balance = i + 1;  
}
```

**Once a thread enters the method, we have to make sure that all the steps in the method complete (as one atomic process) before any other thread can enter the method.**

---

## there are no Dumb Questions

**Q:** Sounds like it's a good idea to synchronize everything, just to be thread-safe.

**A:** Nope, it's not a good idea. Synchronization doesn't come for free. First, a synchronized method has a certain amount of overhead. In other words, when code hits a synchronized method, there's going to be a performance hit (although typically, you'd never notice it) while the matter of “is the key available?” is resolved.

Second, a synchronized method can slow your program down because synchronization restricts concurrency. In other words, a synchronized method forces other threads to get in line and wait their turn. This might not be a problem in your code, but you have to consider it.

Third, and most frightening, synchronized methods can lead to deadlock! (See page 516.)

A good rule of thumb is to synchronize only the bare minimum that should be synchronized. And in fact, you can synchronize at a granularity that's even smaller than a method. We don't use it in the book, but you can use the synchronized keyword to synchronize at the more fine-grained level of one or more statements, rather than at the whole-method level.

```
public void go() {  
    doStuff();  
  
    synchronized(this) {  
        criticalStuff();  
        moreCriticalStuff();  
    }  
}
```

doStuff() doesn't need to be synchronized, so we don't synchronize the whole method.

Now, only these two method calls are grouped into one atomic unit. When you use the synchronized keyword WITHIN a method, rather than in a method declaration, you have to provide an argument that is the object whose key the thread needs to get.

Although there are other ways to do it, you will almost always synchronize on the current object (this). That's the same object you'd lock if the whole method were synchronized.

① Thread A runs for awhile



Attempt to enter the increment() method.

The method is synchronized, so **get the key** for this object

Put the value of balance into variable i.

Balance is 0, so i is now 0.

Set the value of balance to the result of  $i + 1$ .

Now balance is 1.

**Return the key** (it completed the increment() method).

Re-enter the increment() method and **get the key**.

Put the value of balance into variable i.

Balance is 1, so i is now 1.

*[now thread A is sent back to runnable, but since it has not completed the synchronized method, Thread A keeps the key]*

② Thread B is selected to run



Attempt to enter the increment() method. The method is synchronized, so we need to get the key.

**The key is not available.**

*[now thread B is sent into a 'object lock not available' lounge]*

③ Thread A runs again, picking up where it left off  
(remember, it still has the key)



Set the value of balance to the result of  $i + 1$ .

Now balance is 2.

**Return the key.**

*[now thread A is sent back to runnable, but since it has completed the increment() method, the thread does NOT hold on to the key]*

④ Thread B is selected to run



Attempt to enter the increment() method. The method is synchronized, so we need to get the key.

This time, the key IS available, get the key.

Put the value of balance into variable i.

[continues to run...]

## thread deadlock

# The deadly side of synchronization

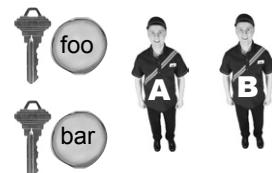
Be careful when you use synchronized code, because nothing will bring your program to its knees like thread deadlock.

Thread deadlock happens when you have two threads, both of which are holding a key the other thread wants. There's no way out of this scenario, so the two threads will simply sit and wait. And wait. And wait.

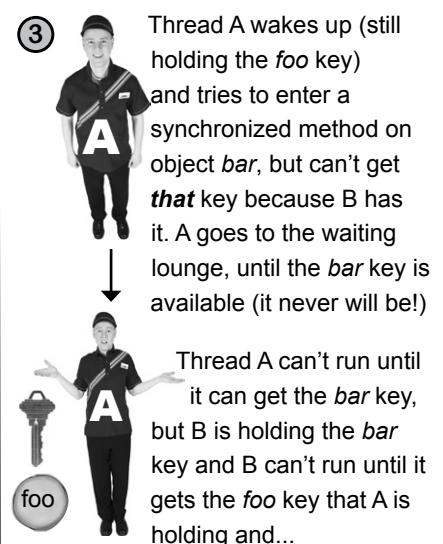
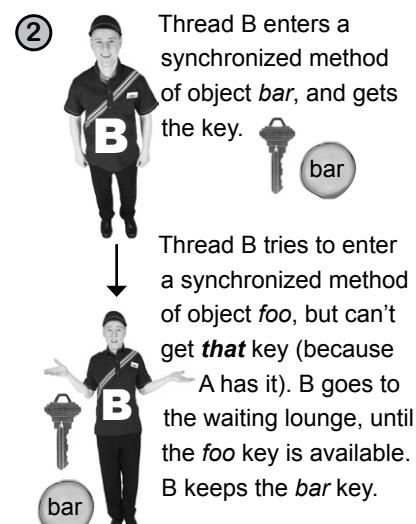
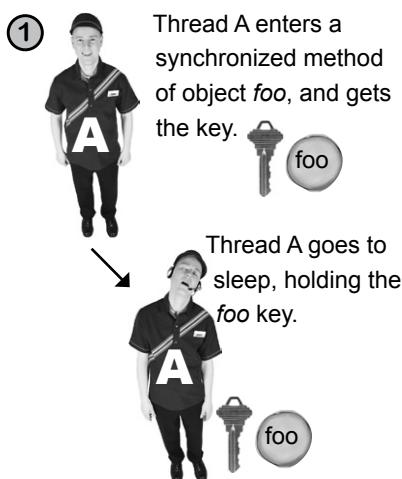
If you're familiar with databases or other application servers, you might recognize the problem; databases often have a locking mechanism somewhat like synchronization. But a real transaction management system can sometimes deal with deadlock. It might assume, for example, that deadlock might have occurred when two transactions are taking too long to complete. But unlike Java, the application server can do a "transaction rollback" that returns the state of the rolled-back transaction to where it was before the transaction (the atomic part) began.

Java has no mechanism to handle deadlock. It won't even *know* deadlock occurred. So it's up to you to design carefully. If you find yourself writing much multithreaded code, you might want to study "Java Threads" by Scott Oaks and Henry Wong for design tips on avoiding deadlock. One of the most common tips is to pay attention to the order in which your threads are started.

**All it takes for deadlock are two objects and two threads.**



## A simple deadlock scenario:





### BULLET POINTS

- The static Thread.sleep() method forces a thread to leave the running state for at least the duration passed to the sleep method. Thread.sleep(200) puts a thread to sleep for 200 milliseconds.
- The sleep() method throws a checked exception (InterruptedException), so all calls to sleep() must be wrapped in a try/catch, or declared.
- You can use sleep() to help make sure all threads get a chance to run, although there's no guarantee that when a thread wakes up it'll go to the end of the runnable line. It might, for example, go right back to the front. In most cases, appropriately-timed sleep() calls are all you need to keep your threads switching nicely.
- You can name a thread using the (yet another surprise) setName() method. All threads get a default name, but giving them an explicit name can help you keep track of threads, especially if you're debugging with print statements.
- You can have serious problems with threads if two or more threads have access to the same object on the heap.
- Two or more threads accessing the same object can lead to data corruption if one thread, for example, leaves the running state while still in the middle of manipulating an object's critical state.
- To make your objects thread-safe, decide which statements should be treated as one atomic process. In other words, decide which methods must run to completion before another thread enters the same method on the same object.
- Use the keyword **synchronized** to modify a method declaration, when you want to prevent two threads from entering that method.
- Every object has a single lock, with a single key for that lock. Most of the time we don't care about that lock; locks come into play only when an object has synchronized methods.
- When a thread attempts to enter a synchronized method, the thread must get the key for the object (the object whose method the thread is trying to run). If the key is not available (because another thread already has it), the thread goes into a kind of waiting lounge, until the key becomes available.
- Even if an object has more than one synchronized method, there is still only one key. Once any thread has entered a synchronized method on that object, no thread can enter any other synchronized method on the same object. This restriction lets you protect your data by synchronizing any method that manipulates the data.

```
final chat client
```

## New and improved SimpleChatClient

Way back near the beginning of this chapter, we built the SimpleChatClient that could *send* outgoing messages to the server but couldn't receive anything. Remember? That's how we got onto this whole thread topic in the first place, because we needed a way to do two things at once: send messages *to* the server (interacting with the GUI) while simultaneously reading incoming messages *from* the server, displaying them in the scrolling text area.

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

public class SimpleChatClient {

    JTextArea incoming;
    JTextField outgoing;
    BufferedReader reader;
    PrintWriter writer;
    Socket sock;

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

    public void go() {

        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        incoming = new JTextArea(15, 50);
        incoming.setLineWrap(true);
        incoming.setWrapStyleWord(true);
        incoming.setEditable(false);
        JScrollPane qScroller = new JScrollPane(incoming);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(qScroller);
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        setUpNetworking();

        Thread readerThread = new Thread(new IncomingReader());
        readerThread.start();

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(800, 500);
        frame.setVisible(true);
    }
}
```

Yes, there really IS an end to this chapter.  
But not yet...

This is mostly GUI code you've seen before. Nothing special except the highlighted part where we start the new 'reader' thread.

We're starting a new thread, using a new inner class as the Runnable (job) for the thread. The thread's job is to read from the server's socket stream, displaying any incoming messages in the scrolling text area.

## networking and threads

```
private void setUpNetworking() {  
  
    try {  
        sock = new Socket("127.0.0.1", 5000);  
        InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());  
        reader = new BufferedReader(streamReader);  
        writer = new PrintWriter(sock.getOutputStream());  
        System.out.println("networking established");  
    } catch(IOException ex) {  
        ex.printStackTrace();  
    }  
} // close setUpNetworking  
  
public class SendButtonListener implements ActionListener {  
    public void actionPerformed(ActionEvent ev) {  
        try {  
            writer.println(outgoing.getText());  
            writer.flush();  
        } catch(Exception ex) {  
            ex.printStackTrace();  
        }  
        outgoing.setText("");  
        outgoing.requestFocus();  
    }  
} // close inner class  
  
public class IncomingReader implements Runnable {  
    public void run() {  
        String message;  
        try {  
            while ((message = reader.readLine()) != null) {  
                System.out.println("read " + message);  
                incoming.append(message + "\n");  
            } // close while  
        } catch(Exception ex) {ex.printStackTrace();}  
    } // close run  
} // close inner class  
} // close outer class
```

We're using the socket to get the input and output streams. We were already using the output stream to send to the server, but now we're using the input stream so that the new 'reader' thread can get messages from the server.

Nothing new here. When the user clicks the send button, this method sends the contents of the text field to the server.

This is what the thread does!!  
In the run() method, it stays in a loop (as long as what it gets from the server is not null), reading a line at a time and adding each line to the scrolling text area (along with a new line character).



## Ready-bake Code

### The really really simple Chat Server

You can use this server code for both versions of the Chat Client. Every possible disclaimer ever disclaimed is in effect here. To keep the code stripped down to the bare essentials, we took out a lot of parts that you'd need to make this a real server. In other words, it works, but there are at least a hundred ways to break it. If you want a Really Good Sharpen Your Pencil for after you've finished this book, come back and make this server code more robust.

Another possible Sharpen Your Pencil, that you could do right now, is to annotate this code yourself. You'll understand it much better if you work out what's happening than if we explained it to you. Then again, this is Ready-bake code, so you really don't have to understand it at all. It's here just to support the two versions of the Chat Client.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

    ArrayList clientOutputStreams;

    public class ClientHandler implements Runnable {
        BufferedReader reader;
        Socket sock;

        public ClientHandler(Socket clientSocket) {
            try {
                sock = clientSocket;
                InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
                reader = new BufferedReader(isReader);

            } catch(Exception ex) {ex.printStackTrace();}
        } // close constructor

        public void run() {
            String message;
            try {
                while ((message = reader.readLine()) != null) {
                    System.out.println("read " + message);
                    tellEveryone(message);

                } // close while
            } catch(Exception ex) {ex.printStackTrace();}
        } // close run
    } // close inner class
}
```

To run the chat client, you need two terminals. First, launch this server from one terminal, then launch the client from another terminal

```
public static void main (String[] args) {
    new VerySimpleChatServer().go();
}

public void go() {
    clientOutputStreams = new ArrayList();
    try {
        ServerSocket serverSock = new ServerSocket(5000);

        while(true) {
            Socket clientSocket = serverSock.accept();
            PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
            clientOutputStreams.add(writer);

            Thread t = new Thread(new ClientHandler(clientSocket));
            t.start();
            System.out.println("got a connection");
        }
    } catch(Exception ex) {
        ex.printStackTrace();
    }
} // close go

public void tellEveryone(String message) {

    Iterator it = clientOutputStreams.iterator();
    while(it.hasNext()) {
        try {
            PrintWriter writer = (PrintWriter) it.next();
            writer.println(message);
            writer.flush();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
} // end while

} // close tellEveryone
} // close class
```

## synchronization questions

*there are no*  
Dumb Questions

**Q:** What about protecting static variable state? If you have static methods that change the static variable state, can you still use synchronization?

**A:** Yes! Remember that static methods run against the class and not against an individual instance of the class. So you might wonder whose object's lock would be used on a static method? After all, there might not even be any instances of that class. Fortunately, just as each *object* has its own lock, each loaded *class* has a lock. That means that if you have three Dog objects on your heap, you have a total of four Dog-related locks. Three belonging to the three Dog instances, and one belonging to the Dog class itself. When you synchronize a static method, Java uses the lock of the class itself. So if you synchronize two static methods in a single class, a thread will need the class lock to enter *either* of the methods.

**Q:** What are thread priorities? I've heard that's a way you can control scheduling.

**A:** Thread priorities *might* help you influence the scheduler, but they still don't offer any guarantee. Thread priorities are numerical values that tell the scheduler (if it cares) how important a thread is to you. In general, the scheduler will kick a lower priority thread out of the running state if a higher priority thread suddenly becomes runnable. But... one more time, say it with me now, "there is no guarantee." We recommend that you use priorities only if you want to influence *performance*, but never, ever rely on them for program correctness.

**Q:** Why don't you just synchronize all the getters and setters from the class with the data you're trying to protect? Like, why couldn't we have synchronized just the checkBalance() and withdraw() methods from class BankAccount, instead of synchronizing the makeWithdrawal() method from the Runnable's class?

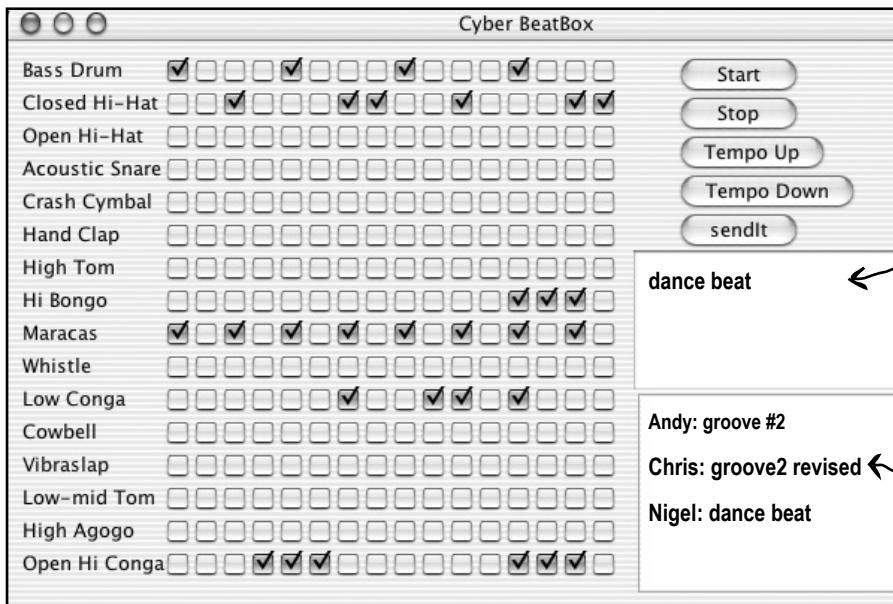
**A:** Actually, we *should* have synchronized those methods, to prevent other threads from accessing those methods in other ways. We didn't bother, because our example didn't have any other code accessing the account.

But synchronizing the getters and setters (or in this case the checkBalance() and withdraw()) isn't enough. Remember, the point of synchronization is to make a specific section of code work ATOMICALLY. In other words, it's not just the individual methods we care about, it's methods that require **more than one step to complete!** Think about it. If we had not synchronized the makeWithdrawal() method, Ryan would have checked the balance (by calling the synchronized checkBalance()), and then immediately exited the method and returned the key!

Of course he would grab the key again, after he wakes up, so that he can call the synchronized withdraw() method, but this still leaves us with the same problem we had before synchronization! Ryan can check the balance, go to sleep, and Monica can come in and also check the balance before Ryan has a chance to wakes up and completes his withdrawal.

So synchronizing all the access methods is probably a good idea, to prevent other threads from getting in, but you still need to synchronize the methods that have statements that must execute as one atomic unit.

# Code Kitchen



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

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

This is the last version of the BeatBox!

It connects to a simple MusicServer so that you can send and receive beat patterns with other clients.

The code is really long, so the complete listing is actually in Appendix A.

**exercise:** Code Magnets



## Code Magnets

A working Java program is scrambled up on the fridge. Can you add the code snippets on the next page to the empty classes below, to make a working Java program that produces the output listed? 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!

```
public class TestThreads {
```

```
class ThreadOne
```

```
class Accum {
```

```
class ThreadTwo
```

**Bonus Question:** Why do you think we used the modifiers we did in the Accum class?

```
File Edit Window Help Sewing
% java TestThreads
one 98098
two 98099
```

# Code Magnets, continued..



## exercise solutions

```
public class TestThreads {  
    public static void main(String [] args) {  
        ThreadOne t1 = new ThreadOne();  
        ThreadTwo t2 = new ThreadTwo();  
        Thread one = new Thread(t1);  
        Thread two = new Thread(t2);  
        one.start();  
        two.start();  
    }  
  
    class Accum {  
        private static Accum a = new Accum();  
        private int counter = 0;  
  
        private Accum() {} A private constructor  
        public static Accum getAccum() {  
            return a;  
        }  
  
        public void updateCounter(int add) {  
            counter += add;  
        }  
  
        public int getCount() {  
            return counter;  
        }  
    }  
  
    class ThreadOne implements Runnable {  
        Accum a = Accum.getAccum();  
        public void run() {  
            for(int x=0; x < 98; x++) {  
                a.updateCounter(1000);  
                try {  
                    Thread.sleep(50);  
                } catch(InterruptedException ex) {}  
            }  
            System.out.println("one "+a.getCount());  
        }  
    }
```

## Exercise Solutions

Threads from two different classes are updating the same object in a third class, because both threads are accessing a single instance of Accum. The line of code:

`private static Accum a = new Accum();` creates a static instance of Accum (remember static means one per class), and the private constructor in Accum means that no one else can make an Accum object. These two techniques (private constructor and static getter method) used together, create what's known as a 'Singleton' - an OO pattern to restrict the number of instances of an object that can exist in an application. (Usually, there's just a single instance of a Singleton—hence the name), but you can use the pattern to restrict the instance creation in whatever way you choose.)

```
class ThreadTwo implements Runnable {  
    Accum a = Accum.getAccum();  
    public void run() {  
        for(int x=0; x < 99; x++) {  
            a.updateCounter(1);  
            try {  
                Thread.sleep(50);  
            } catch(InterruptedException ex) {}  
        }  
        System.out.println("two "+a.getCount());  
    }  
}
```



## Near-miss at the Airlock

As Sarah joined the on-board development team's design review meeting , she gazed out the portal at sunrise over the Indian Ocean. Even though the ship's conference room was incredibly claustrophobic, the sight of the growing blue and white crescent overtaking night on the planet below filled Sarah with awe and appreciation.

## Five-Minute Mystery



This morning's meeting was focused on the control systems for the orbiter's airlocks. As the final construction phases were nearing their end, the number of spacewalks was scheduled to increase dramatically, and traffic was high both in and out of the ship's airlocks. "Good morning Sarah", said Tom, "Your timing is perfect, we're just starting the detailed design review."

"As you all know", said Tom, "Each airlock is outfitted with space-hardened GUI terminals, both inside and out. Whenever spacewalkers are entering or exiting the orbiter they will use these terminals to initiate the airlock sequences." Sarah nodded, "Tom can you tell us what the method sequences are for entry and exit?" Tom rose, and floated to the whiteboard, "First, here's the exit sequence method's pseudocode", Tom quickly wrote on the board.

```
orbiterAirlockExitSequence()

    verifyPortalStatus();

    pressurizeAirlock();

    openInnerHatch();

    confirmAirlockOccupied();

    closeInnerHatch();

    decompressAirlock();

    openOuterHatch();

    confirmAirlockVacated();

    closeOuterHatch();
```

"To ensure that the sequence is not interrupted, we have synchronized all of the methods called by the orbiterAirlockExitSequence() method", Tom explained. "We'd hate to see a returning spacewalker inadvertently catch a buddy with his space pants down!"

Everyone chuckled as Tom erased the whiteboard, but something didn't feel right to Sarah and it finally clicked as Tom began to write the entry sequence pseudocode on the whiteboard. "Wait a minute Tom!", cried Sarah, "I think we've got a big flaw in the exit sequence design, let's go back and revisit it, it could be critical!"

*Why did Sarah stop the meeting? What did she suspect?*

## puzzle answers



### What did Sarah know?

Sarah realized that in order to ensure that the entire exit sequence would run without interruption the

`orbiterAirlockExitSequence()` method needed to be synchronized. As the design stood, it would be possible for a returning spacewalker to interrupt the Exit Sequence! The Exit Sequence thread couldn't be interrupted in the middle of any of the lower level method calls, but it *could* be interrupted in *between* those calls. Sarah knew that the entire sequence should be run as one atomic unit, and if the `orbiterAirlockExitSequence()` method was synchronized, it could not be interrupted at any point.

## 16 collections and generics

# Data structures



Sheesh... and all this time I could have just let Java put things in alphabetical order? Third grade really sucks. We never learn anything useful...

**Sorting is a snap in Java.** You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've stabbed you in the back? Sort your pets by number of tricks learned? It's all here...

## sorting a list

# Tracking song popularity on your jukebox

Congratulations on your new job—managing the automated jukebox system at Lou's Diner. There's no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.



Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You're not writing the entire app—some of the other software developer/waiters are involved as well, but you're responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. All you get is the file the jukebox keeps adding to. Your job is to take it from there.

You've already figured out how to read and parse the file, and so far you've been storing the data in an `ArrayList`.

### SongList.txt

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

This is the file the jukebox device writes. Your code must read the file, then manipulate the song data.

## Challenge #1

### Sort the songs in alphabetical order

You have a list of songs in a file, where each line represents one song, and the title and artist are separated with a forward slash. So it should be simple to parse the line, and put all the songs in an `ArrayList`.

Your boss cares only about the song titles, so for now you can simply make a list that just has the song titles. But you can see that the list is not in alphabetical order... what can you do?

You know that with an `ArrayList`, the elements are kept in the order in which they were inserted into the list, so putting them in an `ArrayList` won't take care of alphabetizing them, unless... maybe there's a `sort()` method in the `ArrayList` class?

## Here's what you have so far, without the sort:

```

import java.util.*;
import java.io.*;

public class Jukebox1 {
    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line = reader.readLine()) != null) {
                addSong(line);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}

```

We'll store the song titles in an ArrayList of Strings.

The method that starts loading the file and then prints the contents of the songList ArrayList.

Nothing special here... just read the file and call the addSong() method for each line.

The addSong method works just like the QuizCard in the I/O chapter--you break the line pieces (tokens) using the split() method.

We only want the song title, so add only the first token to the SongList (the ArrayList).

```

File Edit Window Help Dance
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]

```

The songList prints out with the songs in the order in which they were added to the ArrayList (which is the same order the songs are in within the original text file).  
This is definitely NOT alphabetical!

## ArrayList API

# But the ArrayList class does NOT have a sort() method!

When you look in ArrayList, there doesn't seem to be any method related to sorting. Walking up the inheritance hierarchy didn't help either—it's clear that *you can't call a sort method on the ArrayList*.

The screenshot shows a Mac OS X desktop with a web browser displaying the Java API documentation for ArrayList. The title bar says "ArrayList (Java 2 Platform SE 5.0)". The URL in the address bar is "http://java.sun.com/j2se/1.5.0/docs/api/index.html". The page content is titled "Method Summary" and lists various methods of the ArrayList class. A callout bubble with handwritten text points to the "sort" method, which is not listed.

**Method Summary**

- boolean `add(E o)`  
Appends the specified element to the end of this list.
- void `add(int index, E element)`  
Inserts the specified element at the specified position in this list.
- boolean `addAll(Collection<? extends E> c)`  
Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's iterator.
- boolean `addAll(int index, Collection<? extends E> c)`  
Inserts all of the elements in the specified Collection into this list, starting at the specified position.
- void `clear()`  
Removes all of the elements from this list.
- Object `clone()`  
Returns a shallow copy of this ArrayList instance.
- boolean `contains(Object elem)`  
Returns true if this list contains the specified element.
- void `ensureCapacity(int minCapacity)`  
Increases the capacity of this ArrayList instance to contain the minimum number of elements specified by the minimum capacity argument.
- E `get(int index)`  
Returns the element at the specified position in this list.
- int `indexOf(Object elem)`  
Searches for the first occurrence of the given element in this list.
- boolean `isEmpty()`  
Tests if this list has no elements.
- int `lastIndexOf(Object elem)`  
Returns the index of the last occurrence of the given element in this list.
- E `remove(int index)`  
Removes the element at the specified position in this list.
- boolean `remove(Object o)`  
Removes a single instance of the specified element from this list.
- protected void `removeRange(int fromIndex, int toIndex)`  
Removes from this List all of the elements between the specified indices.
- E `set(int index, E element)`  
Replaces the element at the specified position in this list with the specified element.
- int `size()`  
Returns the number of elements in this list.
- Object[] `toArray()`  
Returns an array containing all of the elements in this list in the correct order.
- <T> T[] `toArray(T[] a)`  
Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
- void `trimToSize()`  
Trims the capacity of this ArrayList instance to be the list's current size.

**Methods inherited from class java.util.AbstractList**

- `equals`, `hashCode`, `iterator`, `listIterator`, `listIterator`, `subList`

I do see a collection class called TreeSet... and the docs say that it keeps your data sorted. I wonder if I should be using a TreeSet instead of an ArrayList...



## ArrayList is not the only collection

Although ArrayList is the one you'll use most often, there are others for special occasions. Some of the key collection classes include:

*Don't worry about trying to learn these other ones right now. We'll go into more details a little later.*

- ▶ **TreeSet**  
Keeps the elements sorted and prevents duplicates.
- ▶ **HashMap**  
Lets you store and access elements as name/value pairs.
- ▶ **LinkedList**  
Makes it easy to create structures like stacks or queues.
- ▶ **HashSet**  
Prevents duplicates in the collection, and given an element, can find that element in the collection quickly.
- ▶ **LinkedHashMap**  
Like a regular HashMap, except it can remember the order in which elements (name/value pairs) were inserted, or it can be configured to remember the order in which elements were last accessed.

`Collections.sort()`

## You could use a `TreeSet`... Or you could use the `Collections.sort()` method

If you put all the Strings (the song titles) into a `TreeSet` instead of an `ArrayList`, the Strings would automatically land in the right place, alphabetically sorted. Whenever you printed the list, the elements would always come out in alphabetical order.

And that's great when you need a *set* (we'll talk about sets in a few minutes) or when you know that the list must *always* stay sorted alphabetically.

On the other hand, if you don't need the list to stay sorted, `TreeSet` might be more expensive than you need—*every time you insert into a TreeSet, the TreeSet has to take the time to figure out where in the tree the new element must go*. With `ArrayList`, inserts can be blindingly fast because the new element just goes in at the end.

**Q:** But you CAN add something to an `ArrayList` at a specific index instead of just at the end—there's an overloaded `add()` method that takes an `int` along with the element to add. So wouldn't it be slower than inserting at the end?

**A:** Yes, it's slower to insert something in an `ArrayList` somewhere *other* than at the end. So using the overloaded `add(index, element)` method doesn't work as quickly as calling the `add(element)`—which puts the added element at the end. But most of the time you use `ArrayLists`, you won't need to put something at a specific index.

**Q:** I see there's a `LinkedList` class, so wouldn't that be better for doing inserts somewhere in the middle? At least if I remember my Data Structures class from college...

**A:** Yes, good spot. The `LinkedList` can be quicker when you insert or remove something from the middle, but for most applications, the difference between middle inserts into a `LinkedList` and `ArrayList` is usually not enough to care about unless you're dealing with a *huge* number of elements. We'll look more at `LinkedList` in a few minutes.

`java.util.Collections`

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listToFill, Object objToFillItWith)
public static int frequency(Collection c, Object o)
public static void reverse(List list)
public static void rotate(List list, int distance)
public static void shuffle(List list)
public static void sort(List list)
public static boolean swap(List list, int i, int j, Object oldVal, Object newVal)
// many more methods...
```

Hmmm... there IS a `sort()` method in the `Collections` class. It takes a `List`, and since `ArrayList` implements the `List` interface, `ArrayList` IS-A `List`. Thanks to polymorphism, you can pass an `ArrayList` to a method declared to take `List`.

Note: this is NOT the real `Collections` class API; we simplified it here by leaving out the generic type information (which you'll see in a few pages).

## Adding Collections.sort() to the Jukebox code

```

import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line = reader.readLine()) != null) {
                addSong(line);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}

```

```

File Edit Window Help Chill
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]

```

Call the static Collections.sort() method, then print the list again. The second print out is in alphabetical order!

The Collections.sort() method sorts a list of Strings alphabetically.

Before calling sort().

After calling sort().

## sorting your own objects

# But now you need Song objects, not just simple Strings.

Now your boss wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so this time the file will have *four* pieces (tokens) instead of just two.

The Song class is really simple, with only one interesting feature—the overridden `toString()` method. Remember, the `toString()` method is defined in class `Object`, so every class in Java inherits the method. And since the `toString()` method is called on an object when it's printed (`System.out.println(anObject)`), you should override it to print something more readable than the default unique identifier code. When you print a list, the `toString()` method will be called on each object.

```
class Song {  
    String title;  
    String artist;  
    String rating;  
    String bpm;  
  
    Song(String t, String a, String r, String b) {  
        title = t;  
        artist = a;  
        rating = r;  
        bpm = b;  
    }  
  
    public String getTitle() {  
        return title;  
    }  
  
    public String getArtist() {  
        return artist;  
    }  
  
    public String getRating() {  
        return rating;  
    }  
  
    public String.getBpm() {  
        return bpm;  
    }  
  
    public String.toString() {  
        return title;  
    }  
}
```

Four instance variables for the four song attributes in the file.

The variables are all set in the constructor when the new Song is created.

The getter methods for the four attributes.

We override `toString()`, because when you do a `System.out.println(aSongObject)`, we want to see the title. When you do a `System.out.println(aListOfSongs)`, it calls the `toString()` method of EACH element in the list.

## SongListMore.txt

```
Pink Moon/Nick Drake/5/80  
Somersault/Zero 7/4/84  
Shiva Moon/Prem Joshua/6/120  
Circles/BT/5/110  
Deep Channel/Afro Celts/4/120  
Passenger/Headmix/4/100  
Listen/Tahiti 80/5/90
```

The new song file holds four attributes instead of just two. And we want ALL of them in our list, so we need to make a Song class with instance variables for all four song attributes.

## Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little—the file I/O code is the same, and the parsing is the same (`String.split()`), except this time there will be *four* tokens for each song/line, and all four will be used to create a new `Song` object. And of course the `ArrayList` will be of type `<Song>` instead of `<String>`.

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

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongListMore.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

*Change to an ArrayList of Song objects instead of String.*

*Create a new Song object using the four tokens (which means the four pieces of info in the song file for this line), then add the Song to the list.*

`Collections.sort()`

## It won't compile!

Something's wrong... the Collections class clearly shows there's a `sort()` method, that takes a `List`.

`ArrayList` is-a `List`, because `ArrayList` implements the `List` interface, so... it *should* work.

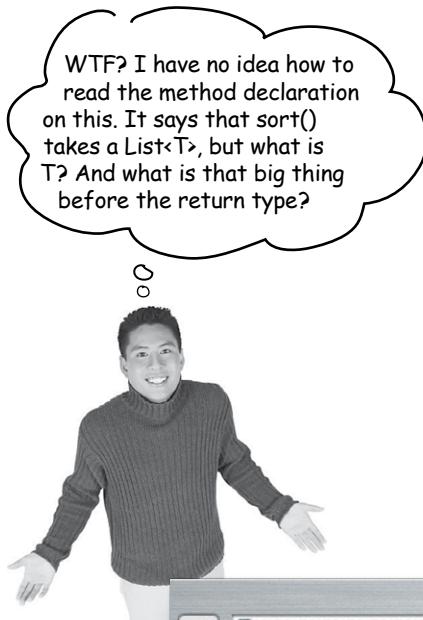
***But it doesn't!***

The compiler says it can't find a `sort` method that takes an `ArrayList<Song>`, so maybe it doesn't like an `ArrayList` of `Song` objects? It didn't mind an `ArrayList<String>`, so what's the important difference between `Song` and `String`? What's the difference that's making the compiler fail?

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
        Collections.sort(songList);
                           ^
1 error
```

And of course you probably already asked yourself, “What would it be sorting *on*?” How would the `sort` method even *know* what made one `Song` greater or less than another `Song`? Obviously if you want the song's *title* to be the value that determines how the songs are sorted, you'll need some way to tell the `sort` method that it needs to use the title and not, say, the beats per minute.

We'll get into all that a few pages from now, but first, let's find out why the compiler won't even let us pass a `Song` `ArrayList` to the `sort()` method.



## The sort() method declaration

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

**Method Detail**

**sort**

```
public static <T extends Comparable<? super T>> void sort List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

From the API docs (looking up the `java.util.Collections` class, and scrolling to the `sort()` method), it looks like the `sort()` method is declared... *strangely*. Or at least different from anything we've seen so far.

That's because the `sort()` method (along with other things in the whole collection framework in Java) makes heavy use of *generics*. Anytime you see something with angle brackets in Java source code or documentation, it means generics—a feature added to Java 5.0. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort `String` objects in an `ArrayList`, but not an `ArrayList` of `Song` objects.

## generic types

# Generics means more type-safety

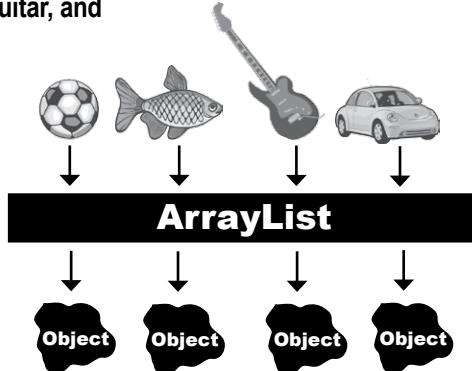
We'll just say it right here—*virtually all of the code you write that deals with generics will be collection-related code*. Although generics can be used in other ways, the main point of generics is to let you write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Before generics (which means before Java 5.0), the compiler could not care less what you put into a collection, because all collection implementations were declared to hold type Object. You could put *anything* in any ArrayList; it was like all ArrayLists were declared as ArrayList<Object>.

### WITHOUT generics

Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects

Before generics, there was no way to declare the type of an ArrayList, so its add() method took type Object.



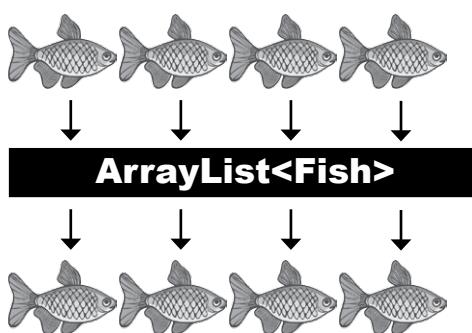
And come OUT as a reference of type Object

With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects.

### WITH generics

Objects go IN as a reference to only Fish objects



And come out as a reference of type Fish

Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references. You don't have to worry about someone sticking a Volkswagen in there, or that what you get out won't really be castable to a Fish reference.

## Learning generics

Of the dozens of things you could learn about generics, there are really only three that matter to most programmers:

### ① Creating instances of generified classes (like ArrayList)

When you make an ArrayList, you have to tell it the type of objects you'll allow in the list, just as you do with plain old arrays.

### ② Declaring and assigning variables of generic types

How does polymorphism really work with generic types? If you have an ArrayList<Animal> reference variable, can you assign an ArrayList<Dog> to it? What about a List<Animal> reference? Can you assign an ArrayList<Animal> to it? You'll see...

### ③ Declaring (and invoking) methods that take generic types

If you have a method that takes as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky polymorphism issues that are very different from the way you write methods that take plain old arrays.

(This is actually the same point as #2, but that shows you how important we think it is.)

**Q:** But don't I also need to learn how to create my OWN generic classes? What if I want to make a class type that lets people instantiating the class decide the type of things that class will use?

**A:** You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and virtually the only type of classes that really need to be generic are collection classes. In other words, classes designed to hold other elements, and you want programmers using it to specify what type those elements are when they declare and instantiate the collection class.

Yes, it is possible that you might want to *create* generic classes, but that's the exception, so we won't cover it here. (But you'll figure it out from the things we *do* cover, anyway.)

```
new ArrayList<Song>()
```

```
List<Song> songList =
    new ArrayList<Song>()
```

```
void foo(List<Song> list)
x.foo(songList)
```

## generic classes

# Using generic CLASSES

Since ArrayList is our most-used generified type, we'll start by looking at its documentation. The two key areas to look at in a generified class are:

- 1) The *class declaration*
- 2) The *method declarations* that let you add elements

## Understanding ArrayList documentation (Or, what's the true meaning of "E"?)

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {  
    public boolean add(E o)  
}  
The "E" is a placeholder for the  
REAL type you use when you  
declare and create an ArrayList  
↓  
Here's the important part! Whatever "E" is  
determines what kind of things you're allowed  
to add to the ArrayList.  
↑  
ArrayList is a subclass of AbstractList,  
so whatever type you specify for the  
type of the ArrayList  
↓  
The type (the value of <E>)  
becomes the type of the List  
interface as well.  
↑
```

The "E" represents the type used to create an instance of ArrayList. When you see an "E" in the ArrayList documentation, you can do a mental find/replace to exchange it for whatever <type> you use to instantiate ArrayList.

So, new ArrayList<Song> means that "E" becomes "Song", in any method or variable declaration that uses "E".

# Using type parameters with ArrayList

**THIS code:**

```
ArrayList<String> thisList = new ArrayList<String>
Means ArrayList:
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
    // more code
}
```

**Is treated by the compiler as:**

```
public class ArrayList<String> extends AbstractList<String>... {
    public boolean add(String o)
    // more code
}
```

In other words, the “E” is replaced by the *real* type (also called the *type parameter*) that you use when you create the ArrayList. And that’s why the add() method for ArrayList won’t let you add anything except objects of a reference type that’s compatible with the type of “E”. So if you make an ArrayList<String>, the add() method suddenly becomes **add(String o)**. If you make the ArrayList of type **Dog**, suddenly the add() method becomes **add(Dog o)**.

**Q:** Is “E” the only thing you can put there? Because the docs for sort used “T”....

**A:** You can use anything that’s a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But the convention is to use a single letter (so that’s what you should use), and a further convention is to use “T” unless you’re specifically writing a collection class, where you’d use “E” to represent the “type of the Element the collection will hold”.

## generic methods

# Using generic METHODS

A generic *class* means that the *class declaration* includes a type parameter. A generic *method* means that the method declaration uses a type parameter in its signature.

You can use type parameters in a method in several different ways:

### ① Using a type parameter defined in the class declaration

```
public class ArrayList<E> extends AbstractList<E> ... {  
    public boolean add(E o) You can use the "E" here ONLY because it's  
    already been defined as part of the class.
```

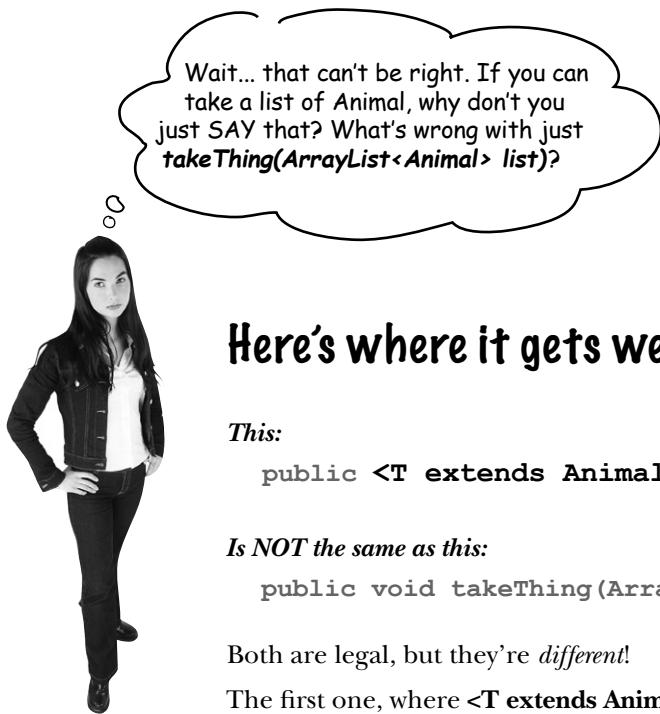
When you declare a type parameter for the class, you can simply use that type anywhere that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

### ② Using a type parameter that was NOT defined in the class declaration

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

If the class itself doesn't use a type parameter, you can still specify one for a method, by declaring it in a really unusual (but available) space—*before the return type*. This method says that T can be “any type of Animal”.

*Here we can use <T> because we declared  
“T” earlier in the method declaration.*



## Here's where it gets weird...

*This:*

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

*Is NOT the same as this:*

```
public void takeThing(ArrayList<Animal> list)
```

Both are legal, but they're *different!*

The first one, where `<T extends Animal>` is part of the method declaration, means that any `ArrayList` declared of a type that is `Animal`, or one of `Animal`'s subtypes (like `Dog` or `Cat`), is legal. So you could invoke the top method using an `ArrayList<Dog>`, `ArrayList<Cat>`, or `ArrayList<Animal>`.

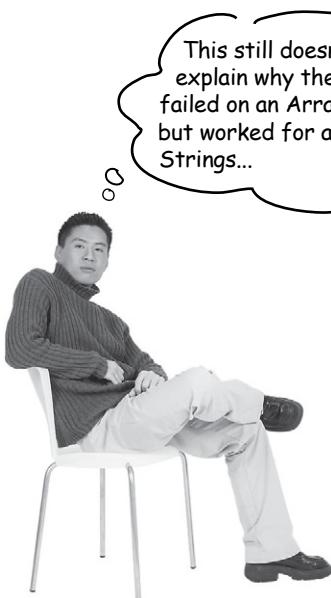
But... the one on the bottom, where the method argument is `(ArrayList<Animal> list)` means that *only* an `ArrayList<Animal>` is legal. In other words, while the first version takes an `ArrayList` of any type that is a type of `Animal` (`Animal`, `Dog`, `Cat`, etc.), the second version takes *only* an `ArrayList` of type `Animal`. Not `ArrayList<Dog>`, or `ArrayList<Cat>` but only `ArrayList<Animal>`.

And yes, it does appear to violate the point of polymorphism, but it will become clear when we revisit this in detail at the end of the chapter. For now, remember that we're only looking at this because we're still trying to figure out how to `sort()` that `SongList`, and that led us into looking at the API for the `sort()` method, which had this strange generic type declaration.

*For now, all you need to know is that the syntax of the top version is legal, and that it means you can pass in a `ArrayList` object instantiated as `Animal` or any `Animal` subtype.*

And now back to our `sort()` method...

## sorting a Song



This still doesn't explain why the sort method failed on an ArrayList of Songs but worked for an ArrayList of Strings...

### Remember where we were...

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
          Collections.sort(songList);
                                         ^
1 error
```

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

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongListMore.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

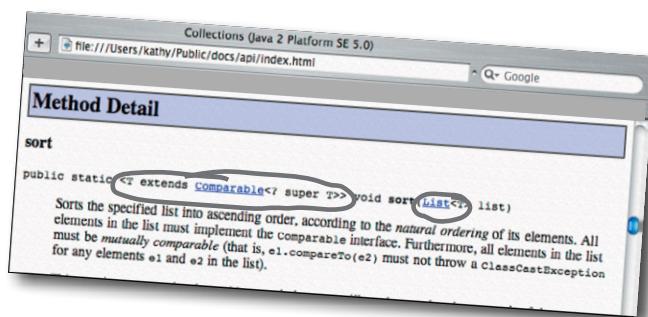
This is where it breaks! It worked fine when passed in an ArrayList<String>, but as soon as we tried to sort an ArrayList<Song>, it failed.

## Revisiting the sort() method

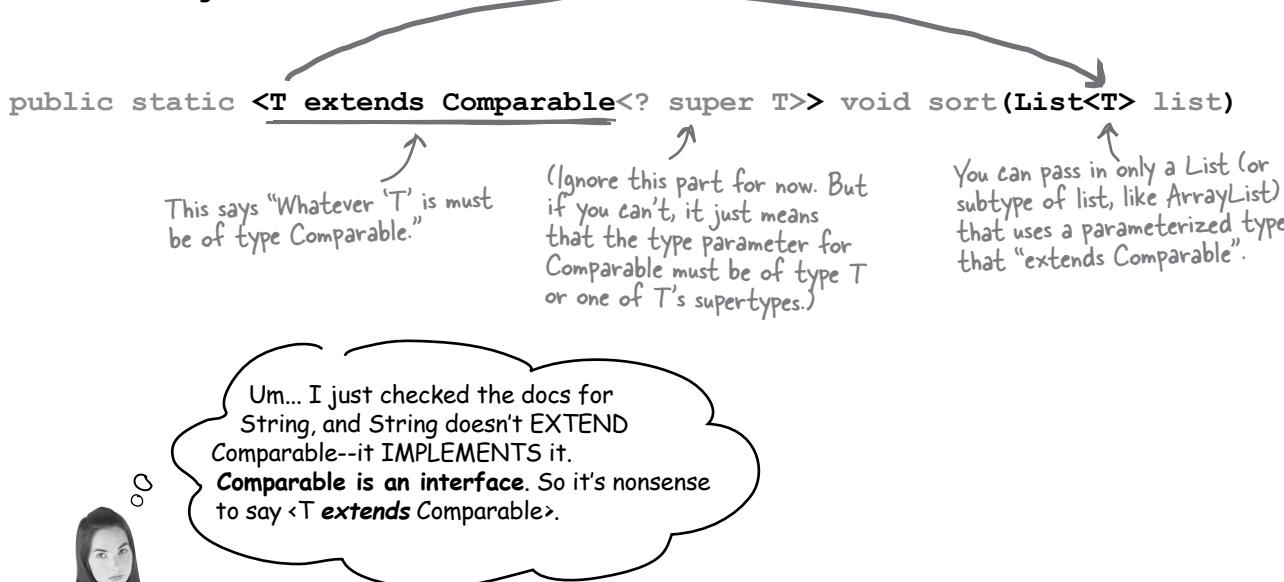
So here we are, trying to read the sort() method docs to find out why it was OK to sort a list of Strings, but not a list of Song objects. And it looks like the answer is...

**The sort() method can take only lists of Comparable objects.**

**Song is NOT a subtype of Comparable, so you cannot sort() the list of Songs.**



**At least not yet...**



```
public final class String extends Object implements Serializable,
    Comparable<String>, CharSequence
```

## the sort() method

### In generics, “extends” means “extends or implements”

The Java engineers had to give you a way to put a constraint on a parameterized type, so that you can restrict it to, say, only subclasses of Animal. But you also need to constrain a type to allow only classes that implement a particular interface. So here's a situation where we need one kind of syntax to work for both situations—inheritance and implementation. In other words, that works for both *extends* and *implements*.

And the winning word was... *extends*. But it really means “is-a”, and works regardless of whether the type on the right is an interface or a class.

Comparable is an interface, so this  
REALLY reads, “T must be a type that  
implements the Comparable interface”.

public static <T extends Comparable<? super T>> void sort(List<T> list)

↑  
It doesn't matter whether the thing on the right is  
a class or interface... you still say “extends”.

**Q:** Why didn't they just make a new keyword, “is”?

**A:** Adding a new keyword to the language is a REALLY big deal because it risks breaking Java code you wrote in an earlier version. Think about it—you might be using a variable “is” (which we do use in this book to represent input streams). And since you're not allowed to use keywords as identifiers in your code, that means any earlier code that used the keyword *before* it was a reserved word, would break. So whenever there's a chance for the Sun engineers to reuse an existing keyword, as they did here with “extends”, they'll usually choose that. But sometimes they don't have a choice...

A few (very few) new keywords *have* been added to the language, such as **assert** in Java 1.4 and **enum** in Java 5.0 (we look at enum in the appendix). And this does break people's code, however you sometimes have the option of compiling and running a *newer* version of Java so that it behaves as though it were an older one. You do this by passing a special flag to the compiler or JVM at the command-line, that says, “Yeah, yeah, I KNOW this is Java 1.4, but please pretend it's really 1.3, because I'm using a variable in my code named *assert* that I wrote back when you guys said it would OK!#\$%”.

(To see if you have a flag available, type *javac* (for the compiler) or *java* (for the JVM) at the command-line, without anything else after it, and you should see a list of available options. You'll learn more about these flags in the chapter on deployment.)

In generics, the keyword “extends” really means “is-a”, and works for BOTH classes and interfaces.

## Finally we know what's wrong... The Song class needs to implement Comparable

We can pass the `ArrayList<Song>` to the `sort()` method only if the `Song` class implements `Comparable`, since that's the way the `sort()` method was declared. A quick check of the API docs shows the `Comparable` interface is really simple, with only one method to implement:

`java.lang.Comparable`

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

And the method documentation for `compareTo()` says

**Returns:**  
 a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

It looks like the `compareTo()` method will be called on one `Song` object, passing that `Song` a reference to a different `Song`. The `Song` running the `compareTo()` method has to figure out if the `Song` it was passed should be sorted higher, lower, or the same in the list.

Your big job now is to decide what makes one song greater than another, and then implement the `compareTo()` method to reflect that. A negative number (any negative number) means the `Song` you were passed is greater than the `Song` running the method. Returning a positive number says that the `Song` running the method is greater than the `Song` passed to the `compareTo()` method. Returning zero means the `Songs` are equal (at least for the purpose of sorting... it doesn't necessarily mean they're the same object). You might, for example, have two `Songs` with the same title.

(Which brings up a whole different can of worms we'll look at later...)

**The big question is: what makes one song less than, equal to, or greater than another song?**

**You can't implement the Comparable interface until you make that decision.**



### Sharpen your pencil

Write in your idea and pseudo code (or better, REAL code) for implementing the `compareTo()` method in a way that will sort() the `Song` objects by title.

Hint: if you're on the right track, it should take less than 3 lines of code!

## the Comparable interface

# The new, improved, comparable Song class

We decided we want to sort by title, so we implement the `compareTo()` method to compare the title of the `Song` passed to the method against the title of the song on which the `compareTo()` method was invoked. In other words, the song running the method has to decide how its title compares to the title of the method parameter.

Hmmm... we know that the `String` class must know about alphabetical order, because the `sort()` method worked on a list of `Strings`. We know `String` has a `compareTo()` method, so why not just call it? That way, we can simply let one title `String` compare itself to another, and we don't have to write the comparing/alphabetizing algorithm!

```
class Song implements Comparable<Song> {
    String title;
    String artist;
    String rating;
    String bpm;

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

Usually these match...we're specifying the type that the implementing class can be compared against.

This means that `Song` objects can be compared to other `Song` objects, for the purpose of sorting.

The `sort()` method sends a `Song` to `compareTo()` to see how that `Song` compares to the `Song` on which the method was invoked.

← Simple! We just pass the work on to the `title` `String` objects, since we know `Strings` have a `compareTo()` method.

This time it worked. It prints the list, then calls `sort` which puts the songs in alphabetical order by title.

```
File Edit Window Help Ambient
%java Jukebox3
[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

## We can sort the list, but...

There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

But when you make a collection element comparable (by having it implement Comparable), you get only one chance to implement the compareTo() method. So what can you do?

The horrible way would be to use a flag variable in the Song class, and then do an *if* test in compareTo() and give a different result depending on whether the flag is set to use title or artist for the comparison.

But that's an awful and brittle solution, and there's something much better. Something built into the API for just this purpose—when you want to sort the same thing in more than one way.

**Look at the Collections class API again. There's a second sort() method—and it takes a Comparator.**

That's not good enough.  
Sometimes I want it to sort  
by artist instead of title.



The sort() method is overloaded to take something called a Comparator.

Note to self: figure out how to get/make a Comparator that can compare and order the songs by artist instead of title...

Collections (Java 2 Platform SE 5.0)	
file:///Users/kathy/Public/docs/api/index.html	Google
Jellyvision, Inc. Collections ...form SE 5.0 Caffeinated ...d Brain Day Brand Noise Diva Marketing	>
<pre>static &lt;K, V&gt; Map&lt;K, V&gt; singletonMap(K key, V value)     Returns an immutable map, mapping only the specified key to the specified value. </pre>	
<pre>static &lt;T&gt; void sort(List&lt;T&gt; list)     Sorts the specified list into ascending order, according to the <i>natural ordering</i> of its elements. </pre>	
<pre>static &lt;T&gt; void sort(List&lt;T&gt; list, Comparator&lt;? super T&gt; c)     Sorts the specified list according to the order induced by the specified comparator. </pre>	

## the Comparator interface

# Using a custom Comparator

An element in a list can compare *itself* to another of its own type in only one way, using its `compareTo()` method. But a Comparator is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an `ArtistComparator`. Sort by beats per minute? Make a `BPMComparator`.

Then all you need to do is call the overloaded `sort()` method that takes the List and the Comparator that will help the `sort()` method put things in order.

The `sort()` method that takes a Comparator will use the Comparator instead of the element's own `compareTo()` method, when it puts the elements in order. In other words, if your `sort()` method gets a Comparator, it won't even *call* the `compareTo()` method of the elements in the list. The `sort()` method will instead invoke the `compare()` method on the Comparator.

So, the rules are:

- ▶ Invoking the one-argument `sort(List o)` method means the list element's `compareTo()` method determines the order. So the elements in the list **MUST** implement the Comparable interface.
  
- ▶ Invoking `sort(List o, Comparator c)` means the list element's `compareTo()` method will NOT be called, and the Comparator's `compare()` method will be used instead. That means the elements in the list do NOT need to implement the Comparable interface.

---

**Q:** So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

**A:** That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

## java.util.Comparator

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

If you pass a Comparator to the `sort()` method, the sort order is determined by the Comparator rather than the element's own `compareTo()` method.

**Q:** But why doesn't every class implement Comparable?

**A:** Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And you aren't taking a huge risk by not implementing Comparable, since a programmer can compare anything in any way that he chooses using his own custom Comparator.

## Updating the Jukebox to use a Comparator

We did three new things in this code:

- 1) Created an inner class that implements Comparator (and thus the `compare()` method that does the work previously done by `compareTo()`).
- 2) Made an instance of the Comparator inner class.
- 3) Called the overloaded sort() method, giving it both the song list and the instance of the Comparator inner class.

Note: we also updated the Song class `toString()` method to print both the song title and the artist. (It prints `title: artist` regardless of how the list is sorted.)

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

public class Jukebox5 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox5().go();
    }

    class ArtistCompare implements Comparator<Song> {
        public int compare(Song one, Song two) {
            return one.getArtist().compareTo(two.getArtist());
        }
    } This becomes a String (the artist)
}

public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
}

ArtistCompare artistCompare = new ArtistCompare();
Collections.sort(songList, artistCompare);

System.out.println(songList);
}

void getSongs() {
    // I/O code here
}

void addSong(String lineToParse) {
    // parse line and add to song list
}
```

*Create a new inner class that implements Comparator (note that its type parameter matches the type we're going to compare—in this case Song objects.)*

*We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.*

*Make an instance of the Comparator inner class.*

*Invoke sort(), passing it the list and a reference to the new custom Comparator object.*

**Note:** we've made sort-by-title the default sort, by keeping the `compareTo()` method in `Song` use the titles. But another way to design this would be to implement both the title sorting and artist sorting as inner Comparator classes, and not have `Song` implement Comparable at all. That means we'd always use the two-arg version of `Collections.sort()`.

### collections exercise

```
import _____;
public class SortMountains {
    LinkedList_____ mtn = new LinkedList_____( );
    class NameCompare _____ {
        public int compare(Mountain one, Mountain two) {
            return _____;
        }
    }
    class HeightCompare _____ {
        public int compare(Mountain one, Mountain two) {
            return (_____);
        }
    }
    public static void main(String [] args) {
        new SortMountains().go();
    }
    public void go() {
        mtn.add(new Mountain("Longs", 14255));
        mtn.add(new Mountain("Elbert", 14433));
        mtn.add(new Mountain("Maroon", 14156));
        mtn.add(new Mountain("Castle", 14265));
        System.out.println("as entered:\n" + mtn);
        NameCompare nc = new NameCompare();
        _____;
        System.out.println("by name:\n" + mtn);
        HeightCompare hc = new HeightCompare();
        _____;
        System.out.println("by height:\n" + mtn);
    }
}
class Mountain {
    _____;
    _____;
    _____ {
        _____;
        _____;
        _____;
    }
    _____ {
        _____;
    }
}
```



### Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the program will create the output shown.

Note: answers are at the end of the chapter.

#### Output:

```
File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```



## Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the “possible answers” list, to correctly answer the question. Answers are at the end of the chapter.

### Possible Answers:

**Comparator,**  
**Comparable,**  
**compareTo( ),**  
**compare( ),**  
**yes,**  
**no**

Given the following compilable statement:

```
Collections.sort(myArrayList) ;
```

1. What must the class of the objects stored in myArrayList implement? \_\_\_\_\_
2. What method must the class of the objects stored in myArrayList implement? \_\_\_\_\_
3. Can the class of the objects stored in myArrayList implement both Comparator AND Comparable? \_\_\_\_\_

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare) ;
```

4. Can the class of the objects stored in myArrayList implement Comparable? \_\_\_\_\_
5. Can the class of the objects stored in myArrayList implement Comparator? \_\_\_\_\_
6. Must the class of the objects stored in myArrayList implement Comparable? \_\_\_\_\_
7. Must the class of the objects stored in myArrayList implement Comparator? \_\_\_\_\_
8. What must the class of the myCompare object implement? \_\_\_\_\_
9. What method must the class of the myCompare object implement? \_\_\_\_\_

## dealing with duplicates

# Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great, now we know how to sort on both *title* (using the Song object's `compareTo()` method) and *artist* (using the Comparator's `compare()` method). But there's a new problem we didn't notice with a test sample of the jukebox text file—*the sorted list contains duplicates*.

It appears that the diner jukebox just keeps writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The `SongListMore.txt` jukebox text file is a complete record of every song that was played, and might contain the same song multiple times.

```
File Edit Window Help TooManyNotes
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT] ← Before sorting.

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7] ← After sorting using
                     the Song's own
                     compareTo() method
                     (sort by title).

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7] ← After sorting using
                     the ArtistCompare
                     Comparator (sort by
                     artist name).
```

`SongListMore.txt`

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

The `SongListMore` text file now has duplicates in it, because the jukebox machine is writing every song three times in a row, followed by "Circles", a song that had been played earlier.

We can't change the way the text file is written because sometimes we're going to need all that information. We have to change the java code.

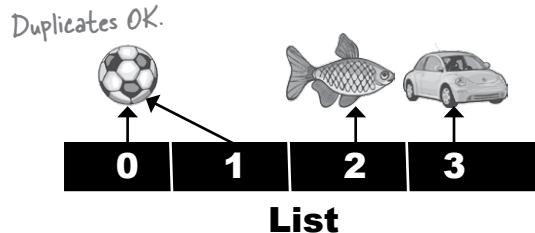
## We need a Set instead of a List

From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**. **ArrayList** is a **List**, but it looks like **Set** is exactly what we need.

### ► LIST - when sequence matters

Collections that know about *index position*.

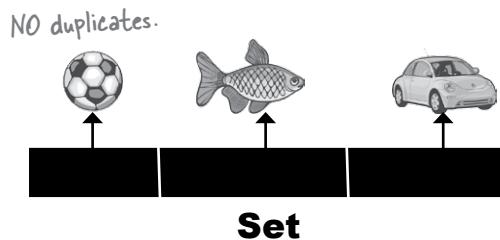
Lists know where something is in the list. You can have more than one element referencing the same object.



### ► SET - when uniqueness matters

Collections that *do not allow duplicates*.

Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).

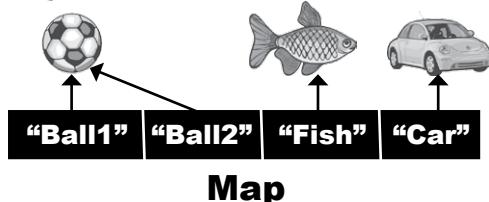


### ► MAP - when finding something by key matters

Collections that use *key-value pairs*.

Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. Although keys are typically String names (so that you can make name/value property lists, for example), a key can be any object.

Duplicate values OK, but NO duplicate keys.

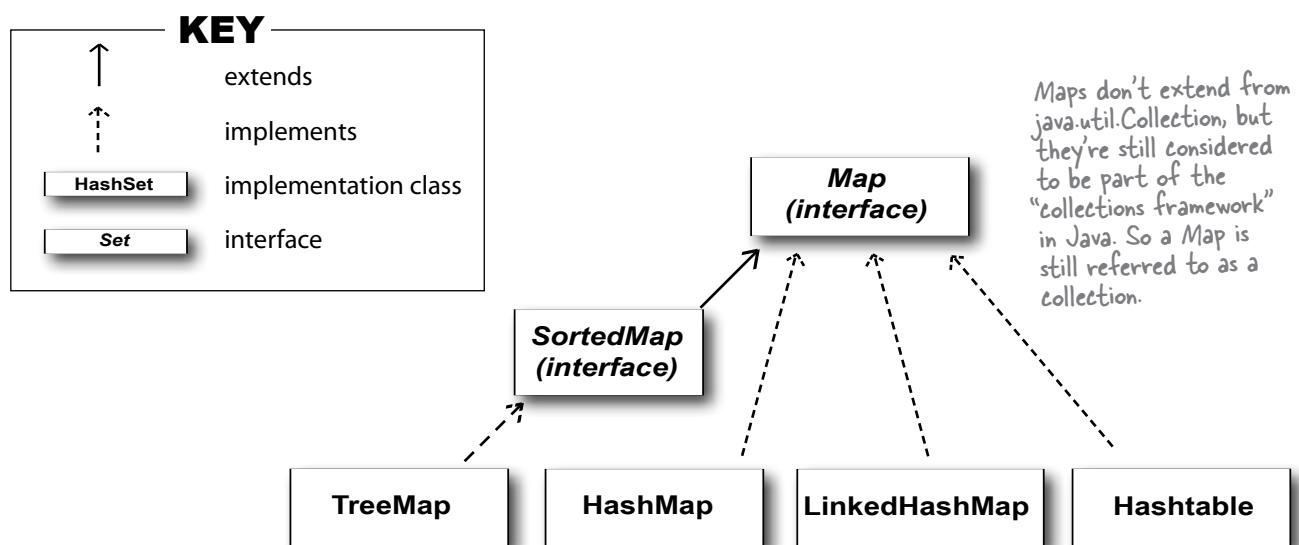
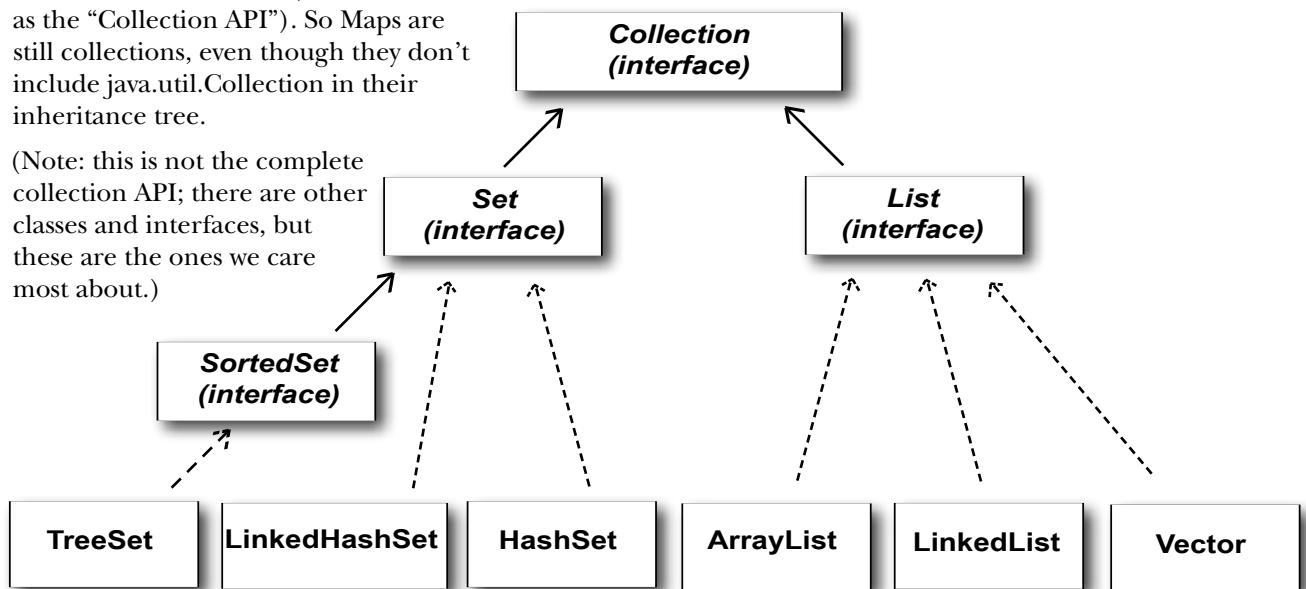


## the collections API

# The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework" (also known as the "Collection API"). So Maps are still collections, even though they don't include `java.util.Collection` in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)



## Using a HashSet instead of ArrayList

We added on to the Jukebox to put the songs in a HashSet. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions. And to make it easier to read the output, we went back to the earlier version of the Song's `toString()` method, so that it prints only the title instead of title *and* artist.)

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

public class Jukebox6 {
    ArrayList<Song> songList = new ArrayList<Song>(); ↗
    // main method etc.

    public void go() {
        getSongs(); ← We didn't change getSong(), so it still puts the songs in an ArrayList
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);

        HashSet<Song> songSet = new HashSet<Song>();
        songSet.addAll(songList); ← HashSet has a simple addAll() method that can
        System.out.println(songSet); ← take another collection and use it to populate
        }                                ← the HashSet. It's the same as if we added each
        // getSong() and addSong() methods
    }
}
```

```
File Edit Window Help GetBetterMusic
%java Jukebox6
[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]
[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]
[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

The Set didn't help!!  
We still have all the duplicates!

(And it lost its sort order  
when we put the list into a  
HashSet, but we'll worry about  
that one later...)

## object equality

# What makes two objects equal?

First, we have to ask—what makes two Song references duplicates? They must be considered *equal*. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

This brings up a key issue: *reference* equality vs. *object* equality.

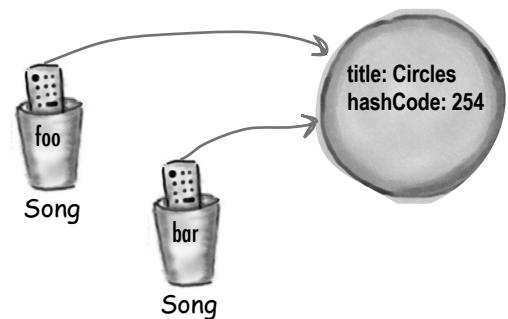
### ► Reference equality

#### Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the `hashCode()` method on both references, you'll get the same result. If you don't override the `hashCode()` method, the default behavior (remember, you inherited this from class `Object`) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the `==` operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.

If two objects `foo` and `bar` are equal, `foo.equals(bar)` must be `true`, and both `foo` and `bar` must return the same value from `hashCode()`. For a Set to treat two objects as duplicates, you must override the `hashCode()` and `equals()` methods inherited from class `Object`, so that you can make two different objects be viewed as equal.



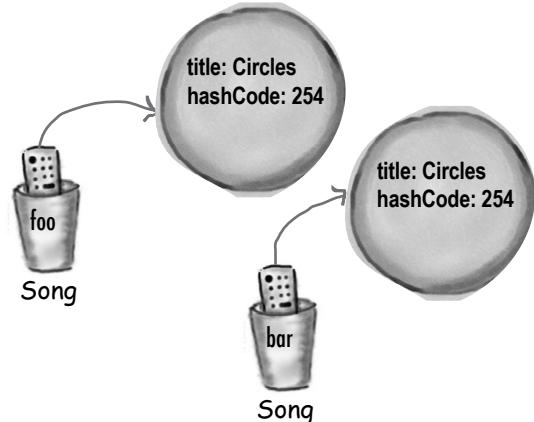
```
if (foo == bar) {  
    // both references are referring  
    // to the same object on the heap  
}
```

### ► Object equality

#### Two references, two objects on the heap, but the objects are considered *meaningfully equivalent*.

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the `hashCode()` and `equals()` methods inherited from class `Object`.

As we said above, if you *don't* override `hashCode()`, the default behavior (from `Object`) is to give each object a unique hashcode value. So you must override `hashCode()` to be sure that two equivalent objects return the same hashcode. But you must also override `equals()` so that if you call it on *either* object, passing in the other object, always returns `true`.



```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {  
    // both references are referring to either a  
    // a single object, or to two objects that are equal  
}
```

## How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a HashSet, it uses the object's hashCode value to determine where to put the object in the Set. But it also compares the object's hashCode to the hashCode of all the other objects in the HashSet, and if there's no matching hashCode, the HashSet assumes that this new object is not a duplicate.

In other words, if the hashCodes are different, the HashSet assumes there's no way the objects can be equal!

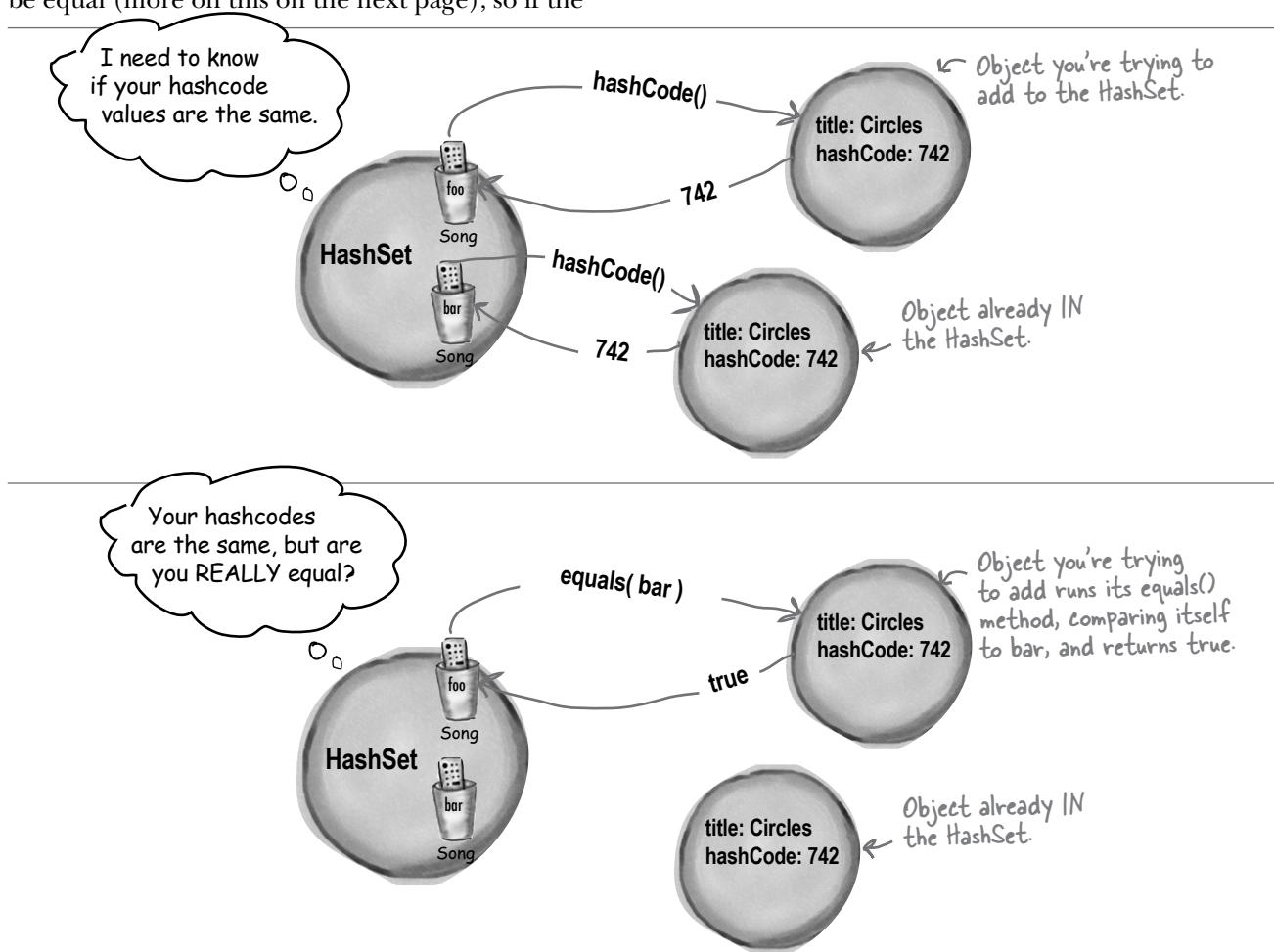
So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hashCode() might *not* be equal (more on this on the next page), so if the

HashSet finds a matching hashCode for two objects—one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hashCode-matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.



## overriding hashCode() and equals()

### The Song class with overridden hashCode() and equals()

```
class Song implements Comparable<Song>{
    String title;
    String artist;
    String rating;
    String bpm;

    public boolean equals(Object aSong) {
        Song s = (Song) aSong;
        return getTitle().equals(s.getTitle()); ←
    }

    public int hashCode() {
        return title.hashCode(); ←
    }

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

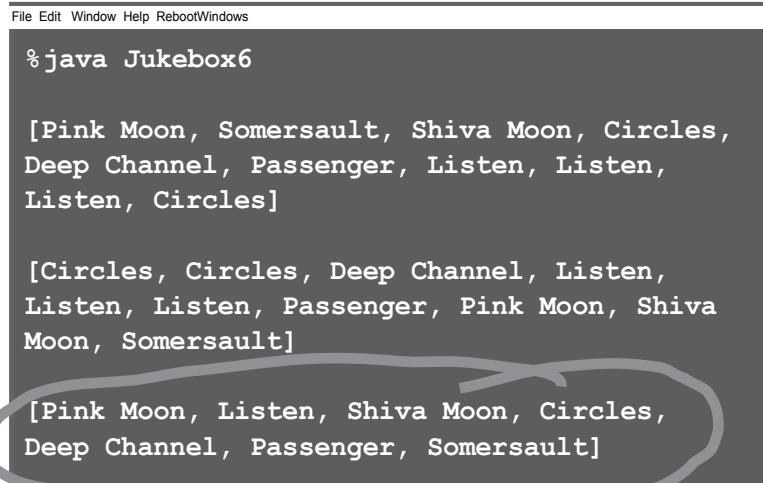
    public String toString() {
        return title;
    }
}
```

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here... the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable.

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.



## Java Object Law For hashCode() and equals()

The API docs for class Object state the rules you **MUST** follow:

- ▶ If two objects are equal, they **MUST** have matching hashcodes.
- ▶ If two objects are equal, calling `equals()` on either object **MUST** return true. In other words, if `(a.equals(b))` then `(b.equals(a))`.
- ▶ If two objects have the same hashCode value, they are **NOT** required to be equal. But if they're equal, they **MUST** have the same hashCode value.
- ▶ So, if you override `equals()`, you **MUST** override `hashCode()`.
- ▶ The default behavior of `hashCode()` is to generate a unique integer for each object on the heap. So if you don't override `hashCode()` in a class, no two objects of that type can EVER be considered equal.
- ▶ The default behavior of `equals()` is to do an `==` comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override `equals()` in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.

`a.equals(b)` must also mean that  
`a.hashCode() == b.hashCode()`

But `a.hashCode() == b.hashCode()` does NOT have to mean `a.equals(b)`

there are no  
Dumb Questions

**Q:** How come hashcodes can be the same even if objects aren't equal?

**A:** HashSets use hashcodes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hashCode as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one..." the HashSet gets the hashCode value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hashCode #742 is stored...", and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use HashSets effectively. In reality, developing a good hashCode algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that hashcodes can be the same without necessarily guaranteeing that the objects are equal, because the "hashing algorithm" used in the `hashCode()` method might happen to return the same value for multiple objects. And yes, that means that multiple objects would all land in the same bucket in the HashSet (because each bucket represents a single hashCode value), but that's not the end of the world. It might mean that the HashSet is just a little less efficient (or that it's filled with an extremely large number of elements), but if the HashSet finds more than one object in the same hashCode bucket, the HashSet will simply use the `equals()` method to see if there's a perfect match. In other words, hashCode values are sometimes used to narrow down the search, but to find the one exact match, the HashSet still has to take all the objects in that one bucket (the bucket for all objects with the same hashCode) and then call `equals()` on them to see if the object it's looking for is in that bucket.

## TreeSets and sorting

# And if we want the set to stay sorted, we've got TreeSet

TreeSet is similar to HashSet in that it prevents duplicates. But it also *keeps* the list sorted. It works just like the sort() method in that if you make a TreeSet using the set's no-arg constructor, the TreeSet uses each object's compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead. The downside to TreeSet is that if you don't *need* sorting, you're still paying for it with a small performance hit. But you'll probably find that the hit is almost impossible to notice for most apps.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
    ArrayList<Song> songList = new ArrayList<Song>();
    int val;

    public static void main(String[] args) {
        new Jukebox8().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
        TreeSet<Song> songSet = new TreeSet<Song>();
        songSet.addAll(songList); ←
        System.out.println(songSet);
    }

    void getSongs() {
        try {
            File file = new File("SongListMore.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Instantiate a TreeSet instead of HashSet.  
Calling the no-arg TreeSet constructor  
means the set will use the Song object's  
compareTo() method for the sort.  
(We could have passed in a Comparator.)

We can add all the songs from the HashSet  
using addAll(). (Or we could have added the  
songs individually using songSet.add()  
just the way we added songs to the ArrayList.)

## What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it.*



Look at this code.  
Read it carefully, then  
answer the questions  
below. (Note: there  
are no syntax errors in  
this code.)

```
import java.util.*;

public class TestTree {
    public static void main (String[] args) {
        new TestTree().go();
    }

    public void go() {
        Book b1 = new Book("How Cats Work");
        Book b2 = new Book("Remix your Body");
        Book b3 = new Book("Finding Emo");

        TreeSet<Book> tree = new TreeSet<Book>();
        tree.add(b1);
        tree.add(b2);
        tree.add(b3);
        System.out.println(tree);
    }
}

class Book {
    String title;
    public Book(String t) {
        title = t;
    }
}
```

- 1). What is the result when you compile this code?
- 

- 2). If it compiles, what is the result when you run the TestTree class?
- 

- 3). If there is a problem (either compile-time or runtime) with this code, how would you fix it?
-

how TreeSets sort

## TreeSet elements **MUST** be comparable

TreeSet can't read the programmer's mind to figure out how the objects should be sorted. You have to tell the TreeSet *how*.

### To use a TreeSet, one of these things must be true:

- ▶ The elements in the list must be of a type that implements Comparable

The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type. The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>() the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and... can't.

```
class Book implements Comparable {  
    String title;  
    public Book(String t) {  
        title = t;  
    }  
    public int compareTo(Object b) {  
        Book book = (Book) b;  
        return (title.compareTo(book.title));  
    }  
}
```

## OR

- ▶ You use the TreeSet's overloaded constructor that takes a Comparator

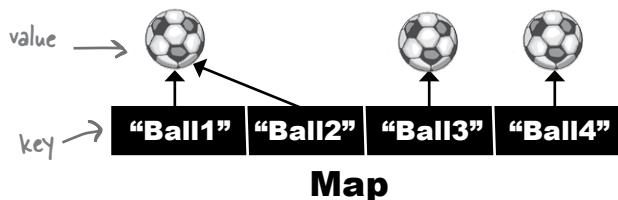
TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

```
public class BookCompare implements Comparator<Book> {  
    public int compare(Book one, Book two) {  
        return (one.title.compareTo(two.title));  
    }  
  
class Test {  
    public void go() {  
        Book b1 = new Book("How Cats Work");  
        Book b2 = new Book("Remix your Body");  
        Book b3 = new Book("Finding Emo");  
        BookCompare bCompare = new BookCompare();  
        TreeSet<Book> tree = new TreeSet<Book>(bCompare);  
        tree.add(new Book("How Cats Work"));  
        tree.add(new Book("Finding Emo"));  
        tree.add(new Book("Remix your Body"));  
        System.out.println(tree);  
    }  
}
```

## We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Although keys will often be Strings, they can be any Java object (or, through autoboxing, a primitive).



**Each element in a Map is actually TWO objects—a key and a value. You can have duplicate values, but NOT duplicate keys.**

### Map example

```
import java.util.*;
public class TestMap {
    public static void main(String[] args) {
        HashMap<String, Integer> scores = new HashMap<String, Integer>();
        scores.put("Kathy", 42);
        scores.put("Bert", 343);
        scores.put("Skyler", 420);
        System.out.println(scores);
        System.out.println(scores.get("Bert"));
    }
}
```

HashMap needs TWO type parameters—one for the key and one for the value.

↓                    ↓

Use put() instead of add(), and now of course it takes two arguments (key, value).

The get() method takes a key, and returns the value (in this case, an Integer).

```
File Edit Window Help WhereAmI
%java TestMap
{Skyler=420, Bert=343, Kathy=42}
343
```

When you print a Map, it gives you the key=value, in braces {} instead of the brackets [] you see when you print lists and sets.

## generic types

# Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be... *weird*. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story.

We'll start with a reminder on how *array* arguments work, polymorphically, and then look at doing the same thing with generic lists. The code below compiles and runs without errors:

### Here's how it works with regular arrays:

```
import java.util.*;
```

```
public class TestGenerics1 {
    public static void main(String[] args) {
        new TestGenerics1().go();
    }
```

```
public void go() {
    Animal[] animals = {new Dog(), new Cat(), new Dog()};
    Dog[] dogs = {new Dog(), new Dog(), new Dog()};
    takeAnimals(animals);
    takeAnimals(dogs);
```

Call `takeAnimals()`, using both array types as arguments...

✓ Declare and create an `Animal` array, that holds both dogs and cats.

← Declare and create a `Dog` array, that holds only Dogs (the compiler won't let you put a Cat in).

```
    } }
```

The crucial point is that the `takeAnimals()` method can take an `Animal[]` or a `Dog[]`, since Dog IS-A Animal. Polymorphism in action.

Remember, we can call ONLY the methods declared in type animal, since the animals parameter is of type `Animal` array, and we didn't do any casting. (What would we cast it to? That array might hold both Dogs and Cats.)

```
abstract class Animal {
    void eat() {
        System.out.println("animal eating");
    }
}
class Dog extends Animal {
    void bark() { }
}
class Cat extends Animal {
    void meow() { }
}
```

The simplified Animal class hierarchy.

## Using polymorphic arguments and generics

So we saw how the whole thing worked with arrays, but will it work the same way when we switch from an array to an ArrayList? Sounds reasonable, doesn't it?

First, let's try it with only the Animal ArrayList. We made just a few changes to the go() method:

### Passing in just ArrayList<Animal>

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat()); ← We have to add one at a time since there's no
    animals.add(new Dog()); shortcut syntax like there is for array creation.

    takeAnimals(animals); ← This is the same code, except now the "animals"
}                                variable refers to an ArrayList instead of array.
```

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

A simple change from Animal[] to ArrayList<Animal>.

The method now takes an ArrayList instead of an array, but everything else is the same. Remember, that for loop syntax works for both arrays and collections.

### Compiles and runs just fine

```
File Edit Window Help CatFoodIsBetter
%java TestGenerics2

animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```

## polymorphism and generics

# But will it work with ArrayList<Dog> ?

Because of polymorphism, the compiler let us pass a Dog array to a method with an Animal array argument. No problem. And an ArrayList<Animal> can be passed to a method with an ArrayList<Animal> argument. So the big question is, will the ArrayList<Animal> argument accept an ArrayList<Dog>? If it works with arrays, shouldn't it work here too?

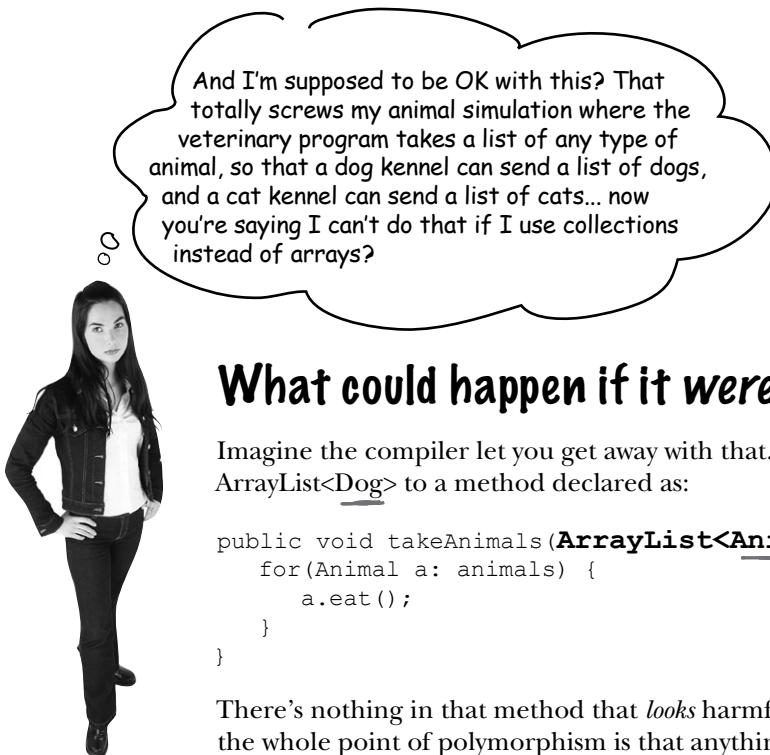
## Passing in just ArrayList<Dog>

```
public void go() {  
    ArrayList<Animal> animals = new ArrayList<Animal>();  
    animals.add(new Dog());  
    animals.add(new Cat());  
    animals.add(new Dog());  
    takeAnimals(animals); ← We know this line worked fine.  
  
    ArrayList<Dog> dogs = new ArrayList<Dog>();  
    dogs.add(new Dog());           Make a Dog ArrayList and put a couple dogs in.  
    dogs.add(new Dog());  
    takeAnimals(dogs); ← Will this work now that we changed  
    }                           from an array to an ArrayList?  
  
    public void takeAnimals(ArrayList<Animal> animals) {  
        for(Animal a: animals) {  
            a.eat();  
        }  
    }
```

## When we compile it:

```
File Edit Window Help CatsAreSmarter  
%java TestGenerics3  
  
TestGenerics3.java:21: takeAnimals(java.util.  
ArrayList<Animal>) in TestGenerics3 cannot be applied to  
(java.util.ArrayList<Dog>)  
    takeAnimals(dogs);  
    ^  
1 error
```

It looked so right,  
but went so wrong...



## What could happen if it were allowed...

Imagine the compiler let you get away with that. It let you pass an `ArrayList<Dog>` to a method declared as:

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the `eat()` method), a Dog can do as well. So what's the problem with having the method call `eat()` on each of the Dog references?

*Nothing.* Nothing at all.

There's nothing wrong with *that* code. But imagine *this* code instead:

```
public void takeAnimals(ArrayList<Animal> animals) {
    animals.add(new Cat());
} ← Yikes!! We just stuck a Cat in what might be a Dogs-only ArrayList.
```

So that's the problem. There's certainly nothing wrong with adding a Cat to an `ArrayList<Animal>`, and that's the whole point of having an `ArrayList` of a supertype like `Animal`—so that you can put all types of animals in a single `Animal` `ArrayList`.

But if you passed a Dog `ArrayList`—one meant to hold ONLY Dogs—to this method that takes an `Animal` `ArrayList`, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog `ArrayList` into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

*If you declare a method to take `ArrayList<Animal>` it can take ONLY an `ArrayList<Animal>`, not `ArrayList<Dog>` or `ArrayList<Cat>`.*

## arrays vs. ArrayLists



Wait a minute... if this is why they won't let you pass a Dog ArrayList into a method that takes an Animal ArrayList—to stop you from possibly putting a Cat in what was actually a Dog list, then why does it work with arrays? Don't you have the same problem with arrays? Can't you still add a Cat object to a Dog[]?

### **Array types are checked again at runtime, but collection type checks happen only when you compile**

Let's say you *do* add a Cat to an array declared as Dog[] (an array that was passed into a method argument declared as Animal[], which is a perfectly legal assignment for arrays).

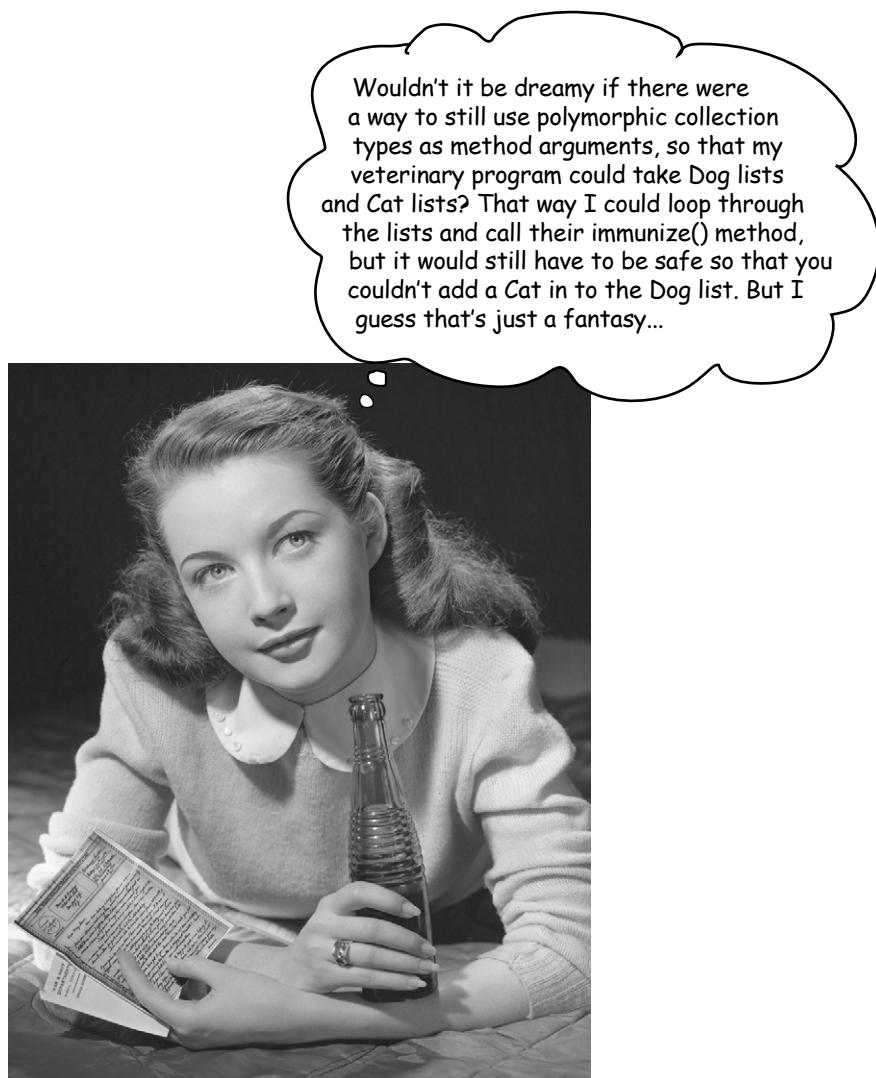
```
public void go() {  
    Dog[] dogs = {new Dog(), new Dog(), new Dog()};  
    takeAnimals(dogs);  
}  
  
public void takeAnimals(Animal[] animals) {  
    animals[0] = new Cat();  
}
```

We put a new Cat into a Dog array. The compiler allowed it, because it knows that you might have passed a Cat array or Animal array to the method, so to the compiler it was possible that this was OK.

### **It compiles, but when we run it:**

```
File Edit Window Help CatsAreSmarter  
%java TestGenerics1  
Exception in thread "main" java.lang.ArrayStoreException:  
Cat  
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)  
        at TestGenerics1.go(TestGenerics1.java:12)  
        at TestGenerics1.main(TestGenerics1.java:5)
```

Whew! At least the JVM stopped it.



## generic wildcards

# Wildcards to the rescue

It looks unusual, but there *is* a way to create a method argument that can accept an ArrayList of any Animal subtype. The simplest way is to use a **wildcard**—added to the Java language explicitly for this reason.

```
public void takeAnimals(ArrayList<? extends Animal> animals) {  
    for(Animal a: animals) {  
        a.eat();  
    }  
}
```

So now you're wondering, "What's the *difference*? Don't you have the same problem as before? The method above isn't doing anything dangerous—calling a method any Animal subtype is guaranteed to have—but can't someone still change this to add a Cat to the *animals* list, even though it's really an ArrayList<Dog>? And since it's not checked again at runtime, how is this any different from declaring it without the wildcard?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!



Remember, the keyword "extends" here means either extends OR implements depending on the type. So if you want to take an ArrayList of types that implement the Pet interface, you'd declare it as:

ArrayList<? extends Pet>

**When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.**

**You can still invoke methods on the elements in the list, but you cannot add elements to the list.**

**In other words, you can do things *with* the list elements, but you can't put *new* things in the list. So you're safe at runtime, because the compiler won't let you do anything that might be horrible at runtime.**

**So, this is OK inside takeAnimals():**

```
for(Animal a: animals) {  
    a.eat();  
}
```

**But THIS would not compile:**

```
animals.add(new Cat());
```

## Alternate syntax for doing the same thing

You probably remember that when we looked at the `sort()` method, it used a generic type, but with an unusual format where the type parameter was declared before the return type. It's just a different way of declaring the type parameter, but the results are the same:

**This:**

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

**Does the same thing as this:**

```
public void takeThing(ArrayList<? extends Animal> list)
```

Dumb Questions<sup>there are no</sup>

**Q:** If they both do the same thing, why would you use one over the other?

**A:** It all depends on whether you want to use "T" somewhere else. For example, what if you want the method to have two arguments—both of which are lists of a type that extend `Animal`? In that case, it's more efficient to just declare the type parameter once:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

Instead of typing:

```
public void takeThing(ArrayList<? extends Animal> one,
                      ArrayList<? extends Animal> two)
```

**be the compiler exercise**



## BE the compiler, advanced



Your job is to play compiler and determine which of these statements would compile. But some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations. In some cases, you might have to guess, but the point is to come up with a reasonable answer based on what you know so far.

(Note: assume that this code is within a legal class and method.)

### Compiles?

- `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- `List<Animal> list = new ArrayList<Animal>();`
- `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- `ArrayList<Animal> animals = dogs;`
- `List<Dog> dogList = dogs;`
- `ArrayList<Object> objects = new ArrayList<Object>();`
- `List<Object> objList = objects;`
- `ArrayList<Object> objs = new ArrayList<Dog>();`

## collections with generics

```

import java.util.*;
public class SortMountains {
    LinkedList<Mountain> mtn = new LinkedList<Mountain>();
    class NameCompare implements Comparator <Mountain> {
        public int compare(Mountain one, Mountain two) {
            return one.name.compareTo(two.name);
        }
    }
    class HeightCompare implements Comparator <Mountain> {
        public int compare(Mountain one, Mountain two) {
            return (two.height - one.height);
        }
    }
    public static void main(String [] args) {
        new SortMountains().go();
    }
    public void go() {
        mtn.add(new Mountain("Longs", 14255));
        mtn.add(new Mountain("Elbert", 14433));
        mtn.add(new Mountain("Maroon", 14156));
        mtn.add(new Mountain("Castle", 14265));
        System.out.println("as entered:\n" + mtn);
        NameCompare nc = new NameCompare();
        Collections.sort(mtn, nc);
        System.out.println("by name:\n" + mtn);
        HeightCompare hc = new HeightCompare();
        Collections.sort(mtn, hc);
        System.out.println("by height:\n" + mtn);
    }
}
class Mountain {
    String name;
    int height;
    Mountain(String n, int h) {
        name = n;
        height = h;
    }
    public String toString() {
        return name + " " + height;
    }
}

```

## Solution to the “Reverse Engineer” sharpen exercise

Did you notice that the height list is  
in DESCENDING sequence? :)

### Output:

```

File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]

```

**fill-in-the-blank solution**

**Exercise Solution**

**Possible Answers:**

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in myArrayList implement?

**Comparable**

2. What method must the class of the objects stored in myArrayList implement?

**compareTo( )**

3. Can the class of the objects stored in myArrayList implement both  
Comparator AND Comparable?

**yes**

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in myArrayList implement Comparable?

**yes**

5. Can the class of the objects stored in myArrayList implement Comparator?

**yes**

6. Must the class of the objects stored in myArrayList implement Comparable?

**no**

7. Must the class of the objects stored in myArrayList implement Comparator?

**no**

8. What must the class of the myCompare object implement?

**Comparator**

9. What method must the class of the myCompare object implement?

**compare( )**



## BE the compiler solution

Compiles?

- `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- `List<Animal> list = new ArrayList<Animal>();`
- `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- `ArrayList<Animal> animals = dogs;`
- `List<Dog> dogList = dogs;`
- `ArrayList<Object> objects = new ArrayList<Object>();`
- `List<Object> objList = objects;`
- `ArrayList<Object> objs = new ArrayList<Dog>();`



## 17 package, jars and deployment

# Release Your Code



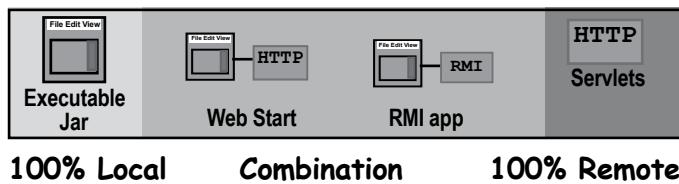
**It's time to let go.** You wrote your code. You tested your code. You refined your code.

You told everyone you know that if you never saw a line of code again, that'd be fine. But in the end, you've created a work of art. The thing actually runs! But now what? *How do you give it to end users?* *What exactly do you give to end users?* What if you don't even know who your end users are? In these final two chapters, we'll explore how to organize, package, and deploy your Java code. We'll look at local, semi-local, and remote deployment options including executable jars, Java Web Start, RMI, and Servlets. In this chapter, we'll spend most of our time on organizing and packaging your code—things you'll need to know regardless of your ultimate deployment choice. In the final chapter, we'll finish with one of the coolest things you can do in Java. Relax. Releasing your code is not saying goodbye. There's always maintenance...

## Deploying your application

What exactly *is* a Java application? In other words, once you're done with development, what is it that you deliver? Chances are, your end-users don't have a system identical to yours. More importantly, they don't have your application. So now it's time to get your program in shape for deployment into The Outside World. In this chapter, we'll look at local deployments, including Executable Jars and the part-local/part-remote technology called Java Web Start. In the next chapter, we'll look at the more remote deployment options, including RMI and Servlets.

### Deployment options



A Java program is a bunch of classes. That's the output of your development.

The real question is what to do with those classes when you're done.

#### ① Local

The entire application runs on the end-user's computer, as a stand-alone, probably GUI, program, deployed as an executable JAR (we'll look at JAR in a few pages.)

#### ② Combination of local and remote

The application is distributed with a client portion running on the user's local system, connected to a server where other parts of the application are running.

#### ③ Remote

The entire Java application runs on a server system, with the client accessing the system through some non-Java means, probably a web browser.

But before we really get into the whole deployment thing, let's take a step back and look at what happens when you've finished programming your app and you simply want to pull out the class files to give them to an end-user. What's really *in* that working directory?



What are the advantages and disadvantages of delivering your Java program as a local, stand-alone application running on the end-user's computer?

What are the advantages and disadvantages of delivering your Java program as web-based system where the user interacts with a web browser, and the Java code runs as servlets on the server?



## Imagine this scenario...

Bob's happily at work on the final pieces of his cool new Java program. After weeks of being in the "I'm-just-one-compile-away" mode, this time he's really done. The program is a fairly sophisticated GUI app, but since the bulk of it is Swing code, he's made only nine classes of his own.

At last, it's time to deliver the program to the client. He figures all he has to do is copy the nine class files, since the client already has the Java API installed. He starts by doing an **ls** on the directory where all his files are...



Whoa! Something strange has happened. Instead of 18 files (nine source code files and nine compiled class files), he sees 31 files, many of which have very strange names like:

Account\$FileListener.class

Chart\$SaveListener.class

and on it goes. He had completely forgotten that the compiler has to generate class files for all those inner class GUI event listeners he made, and that's what all the strangely-named classes are.

Now he has to carefully extract all the class files he needs. If he leaves even one of them out, his program won't work. But it's tricky since he doesn't want to accidentally send the client one of his *source* code files, yet everything is in the same directory in one big mess.

## organizing your classes

# Separate source code and class files

A single directory with a pile of source code and class files is a mess. It turns out, Bob should have been organizing his files from the beginning, keeping the source code and compiled code separate. In other words, making sure his compiled class files didn't land in the same directory as his source code.

*The key is a combination of directory structure organization and the `-d` compiler option.*

There are dozens of ways you can organize your files, and your company might have a specific way they want you to do it. We recommend an organizational scheme that's become almost standard, though.

With this scheme, you create a project directory, and inside that you create a directory called **source** and a directory called **classes**. You start by saving your source code (.java files) into the **source** directory. Then the trick is to compile your code in such a way that the output (the .class files) ends up in the **classes** directory.

And there's a nice compiler flag, `-d`, that lets you do that.

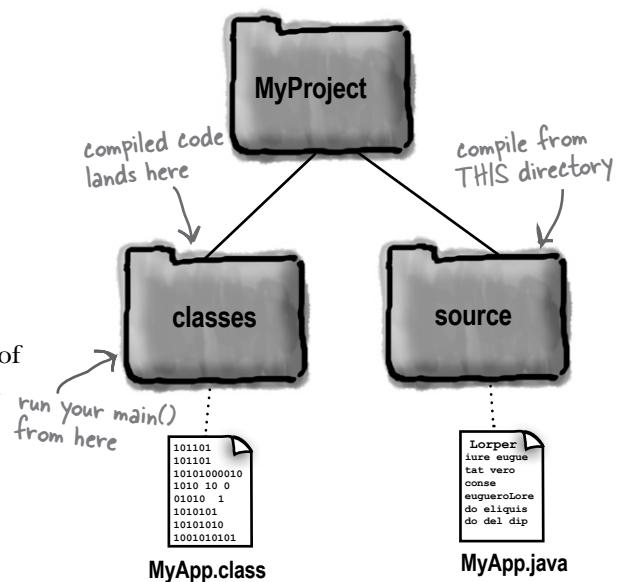


## Compiling with the `-d` (directory) flag

```
%cd MyProject/source  
%javac -d .../classes MyApp.java
```

tells the compiler to put the compiled code (class files) into the "classes" directory that's one directory up and back down again from the current working directory.

the last thing is still the name of the java file to compile



By using the `-d` flag, you get to decide which *directory* the compiled code lands in, rather than accepting the default of class files landing in the same directory as the source code. To compile all the .java files in the source directory, use:

```
%javac -d .../classes *.java
```

\*.java compiles ALL source files in the current directory

## Running your code

```
%cd MyProject/classes  
%java MyApp
```

run your program from the classes' directory.

(troubleshooting note: everything in this chapter assumes that the current working directory (i.e. the ".") is in your classpath. If you have explicitly set a classpath environment variable, be certain that it contains the ".")

## Put your Java in a JAR



A JAR file is a Java ARchive. It's based on the pkzip file format, and it lets you bundle all your classes so that instead of presenting your client with 28 class files, you hand over just a single JAR file. If you're familiar with the tar command on UNIX, you'll recognize the jar tool commands. (Note: when we say JAR in all caps, we're referring to the archive *file*. When we use lowercase, we're referring to the *jar tool* you use to create JAR files.)

The question is, what does the client *do* with the JAR? How do you get it to *run*?

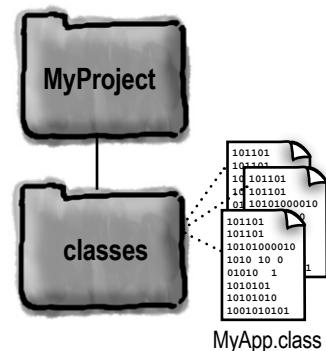
You make the JAR *executable*.

An executable JAR means the end-user doesn't have to pull the class files out before running the program. The user can run the app while the class files are still in the JAR. The trick is to create a *manifest* file, that goes in the JAR and holds information about the files in the JAR. To make a JAR executable, the manifest must tell the JVM *which class has the main() method!*

### Making an executable JAR

#### ① Make sure all of your class files are in the classes directory

We're going to refine this in a few pages, but for now, keep all your class files sitting in the directory named 'classes'.

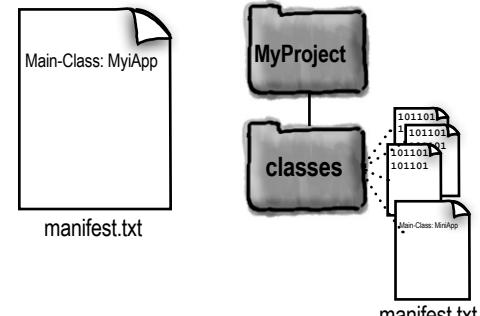


#### ② Create a manifest.txt file that states which class has the main() method

Make a text file named manifest.txt that has one line:

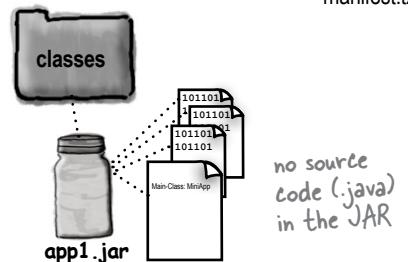
Main-Class: MyApp ← *don't put the .class  
on the end*

Press the return key after typing the Main-Class line, or your manifest may not work correctly. Put the manifest file into the "classes" directory.

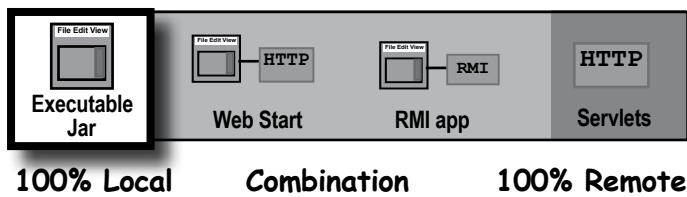


#### ③ Run the jar tool to create a JAR file that contains everything in the classes directory, plus the manifest.

```
%cd MyProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MyApp.class
```



## executable JAR

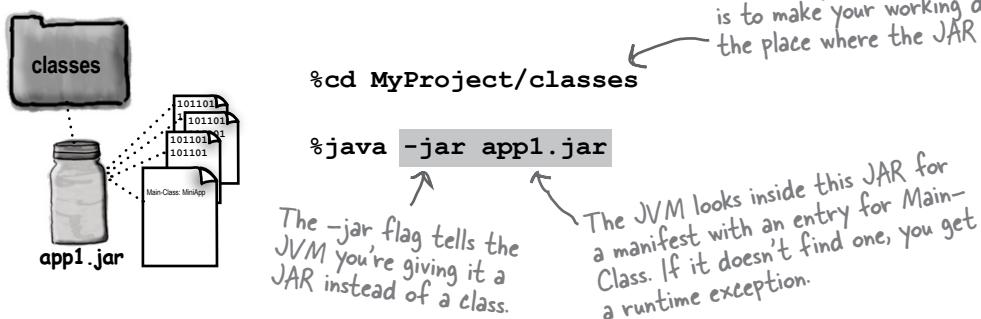


Most 100% local Java apps are deployed as executable JAR files.

## Running (executing) the JAR

Java (the JVM) is capable of loading a class from a JAR, and calling the main() method of that class. In fact, the entire application can stay in the JAR. Once the ball is rolling (i.e., the main() method starts running), the JVM doesn't care *where* your classes come from, as long as it can find them. And one of the places the JVM looks is within any JAR files in the classpath. If it can *see* a JAR, the JVM will *look* in that JAR when it needs to find and load a class.

The JVM has to 'see' the JAR, so it must be in your classpath. The easiest way to make the JAR visible is to make your working directory the place where the JAR is.



Depending on how your operating system is configured, you might even be able to simply double-click the JAR file to launch it. This works on most flavors of Windows, and Mac OS X. You can usually make this happen by selecting the JAR and telling the OS to “Open with...” (or whatever the equivalent is on your operating system).

there are no  
Dumb Questions

**Q:** Why can't I just JAR up an entire directory?

**A:** The JVM looks inside the JAR and expects to find what it needs *right there*. It won't go digging into other directories, unless the class is part of a package, and even *then* the JVM looks only in the directories that match the package statement.

**Q:** What did you just say?

**A:** You can't put your class files into some arbitrary directory and JAR them up that way. But if your classes belong to packages, you can JAR up the entire package directory structure. In fact, you *must*. We'll explain all this on the next page, so you can relax.

## Put your classes in packages!

So you've written some nicely reusable class files, and you've posted them in your internal development library for other programmers to use. While basking in the glow of having just delivered some of the (in your humble opinion) best examples of OO ever conceived, you get a phone call. A frantic one. Two of your classes have the same name as the classes Fred just delivered to the library. And all hell is breaking loose out there, as naming collisions and ambiguities bring development to its knees.

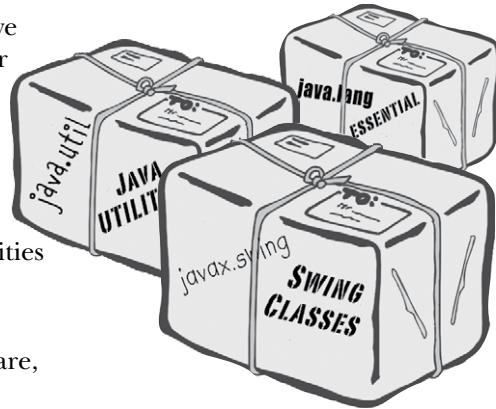
And all because you didn't use packages! Well, you did use packages, in the sense of using classes in the Java API that are, of course, in packages. But you didn't put your own classes into packages, and in the Real World, that's Really Bad.

We're going to modify the organizational structure from the previous pages, just a little, to put classes into a package, and to JAR the entire package. Pay very close attention to the subtle and picky details. Even the tiniest deviation can stop your code from compiling and/or running.

## Packages prevent class name conflicts

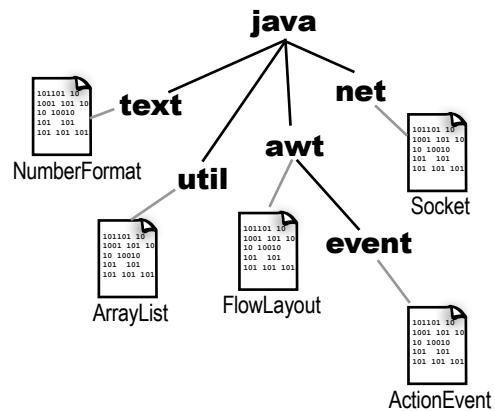
Although packages aren't just for preventing name collisions, that's a key feature. You might write a class named *Customer* and a class named *Account* and a class named *ShoppingCart*. And what do you know, half of all developers working in enterprise e-commerce have probably written classes with those names. In an OO world, that's just dangerous. If part of the point of OO is to write reusable components, developers need to be able to piece together components from a variety of sources, and build something new out of them. Your components have to be able to 'play well with others', including those you didn't write or even know about.

Remember way back in chapter 6 when we discussed how a package name is like the full name of a class, technically known as the *fully-qualified name*. Class *ArrayList* is really *java.util.ArrayList*, *JButton* is really *javax.swing.JButton*, and *Socket* is really *java.net.Socket*. Notice that two of those classes, *ArrayList* and *Socket*, both have *java* as their "first name". In other words, the first part of their fully-qualified names is "java". Think of a hierarchy when you think of package structures, and organize your classes accordingly.



Package structure of the Java API for:

- `java.text.NumberFormat`
- `java.util.ArrayList`
- `java.awt.FlowLayout`
- `java.awt.event.ActionEvent`
- `java.net.Socket`



What does this picture look like to you? Doesn't it look a whole lot like a directory hierarchy?

## package naming



## Preventing package name conflicts

Putting your class in a package reduces the chances of naming conflicts with other classes, but what's to stop two programmers from coming up with identical *package* names? In other words, what's to stop two programmers, each with a class named Account, from putting the class in a package named shopping.customers? Both classes, in that case, would *still* have the same name:

*shopping.customers.Account*

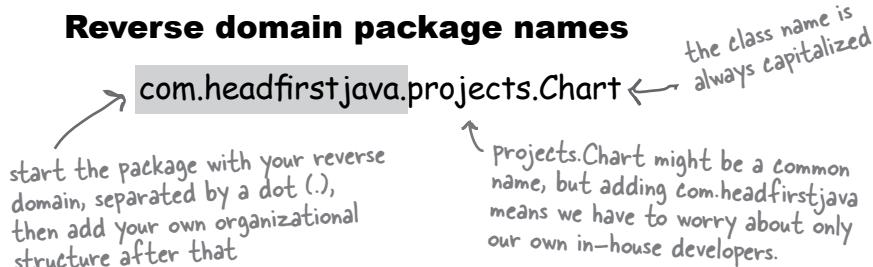
Sun strongly suggests a package naming convention that greatly reduces that risk—prepend every class with your reverse domain name. Remember, domain names are guaranteed to be unique. Two different guys can be named Bartholomew Simpson, but two different domains cannot be named doh.com.

Packages can prevent name conflicts, but only if you choose a package name that's guaranteed to be unique. The best way to do that is to preface your packages with your reverse domain name.

*com.headfirstbooks.Book*

← package name

↑ class name



## To put your class in a package:

### ① Choose a package name

We're using `com.headfirstjava` as our example. The class name is `PackageExercise`, so the fully-qualified name of the class is now: `com.headfirstjava.PackageExercise`.

### ② Put a package statement in your class

It must be the first statement in the source code file, above any import statements. There can be only one package statement per source code file, so **all classes in a source file must be in the same package**. That includes inner classes, of course.

```
package com.headfirstjava;

import javax.swing.*;

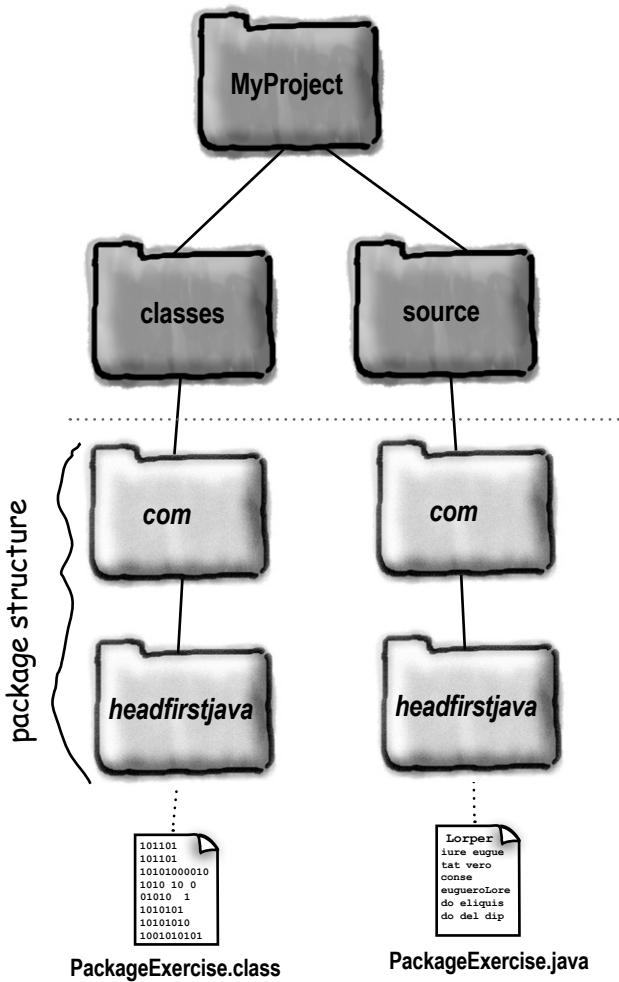
public class PackageExercise {
    // life-altering code here
}
```

### ③ Set up a matching directory structure

It's not enough to *say* your class is in a package, by merely putting a package statement in the code. Your class isn't *truly* in a package until you put the class in a matching directory structure. So, if the fully-qualified class name is `com.headfirstjava.PackageExercise`, you **must** put the `PackageExercise` source code in a directory named `headfirstjava`, which **must** be in a directory named `com`.

It is *possible* to compile without doing that, but trust us—it's not worth the other problems you'll have. Keep your source code in a directory structure that matches the package structure, and you'll avoid a ton of painful headaches down the road.

**You must put a class into a directory structure that matches the package hierarchy.**



Set up a matching directory structure for both the source and classes trees.

compile and run with packages

## Compiling and running with packages

When your class is in a package, it's a little trickier to compile and run. The main issue is that both the compiler and JVM have to be capable of finding your class and all of the other classes it uses.

For the classes in the core API, that's never a problem. Java always knows where its own stuff is. But for your classes, the solution of compiling from the same directory where the source files are simply won't work (or at least not *reliably*). We guarantee, though, that if you follow the structure we describe on this page, you'll be successful. There are other ways to do it, but this is the one we've found the most reliable and the easiest to stick to.

### Compiling with the `-d (directory)` flag

`%cd MyProject/source` ← stay in the source directory! Do NOT cd down into the directory where the .java file is!

`%javac -d ../classes com/headfirstjava/PackageExercise.java`

tells the compiler to put the compiled code (class files) into the classes directory, within the right package structure!! Yes, it knows.

Now you have to specify the PATH to get to the actual source file.

To compile all the .java files in the com.headfirstjava package, use:

`%javac -d ../classes com/headfirstjava/*.java`

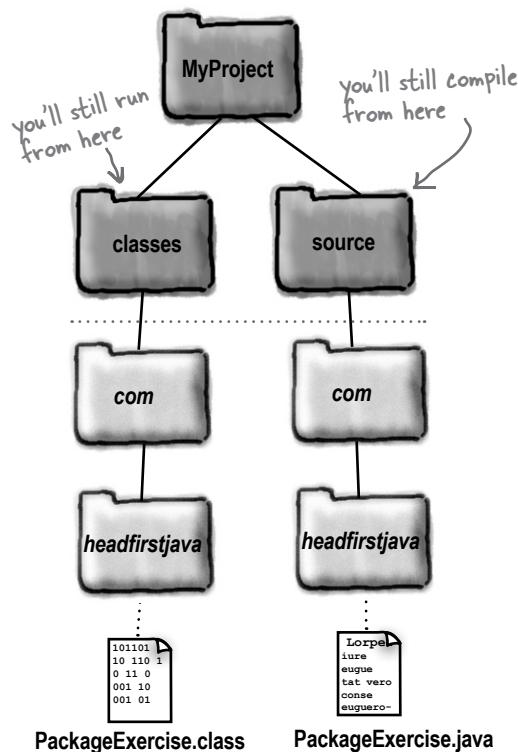
compiles every source (.java) file in this directory

### Running your code

`%cd MyProject/classes` run your program from the classes' directory.

`%java com.headfirstjava.PackageExercise`

You **MUST** give the fully-qualified class name! The JVM will see that, and immediately look inside its current directory (classes) and expect to find a directory named com, where it expects to find a directory named headfirstjava, and in there it expects to find the class. If the class is in the "com" directory, or even in "classes", it won't work!



## The `-d` flag is even cooler than we said

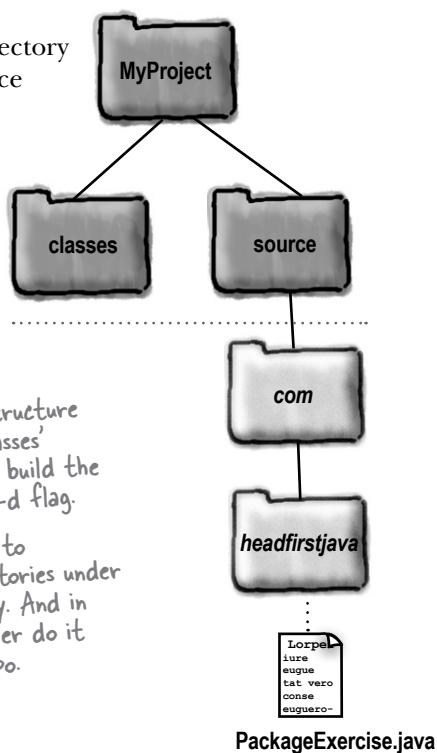
Compiling with the `-d` flag is wonderful because not only does it let you send your compiled class files into a directory other than the one where the source file is, but it also knows to put the class into the correct directory structure for the package the class is in.

But it gets even better!

Let's say that you have a nice directory structure all set up for your source code. But you haven't set up a matching directory structure for your classes directory. Not a problem! Compiling with `-d` tells the compiler to not just *put* your classes into the correct directory tree, but to *build* the directories if they don't exist.

If the package directory structure doesn't exist under the 'classes' directory, the compiler will build the directories if you use the `-d` flag.

So you don't actually have to physically create the directories under the 'classes' root directory. And in fact, if you let the compiler do it there's no chance of a typo.



**The `-d` flag tells the compiler, “Put the class into its package directory structure, using the class specified after the `-d` as the root directory. But... if the directories aren't there, create them first and then put the class in the right place!”**

there are no  
**Dumb Questions**

**Q:** I tried to cd into the directory where my main class was, but now the JVM says it can't find my class! But it's right THERE in the current directory!

**A:** Once your class is in a package, you can't call it by its 'short' name. You MUST specify, at the command-line, the fully-qualified name of the class whose `main()` method you want to run. But since the fully-qualified name includes the *package* structure, Java insists that the class be in a matching *directory* structure. So if at the command-line you say:

`%java com.foo.Book`  
the JVM will look in its current directory (and the rest of its classpath), for a directory named "com". **It will not look for a class named Book, until it has found a directory named "com" with a directory inside named "foo".**

Only then will the JVM accept that it's found the correct `Book` class. If it finds a `Book` class anywhere else, it assumes the class isn't in the right structure, even if it is! The JVM won't for example, look back up the directory tree to say, "Oh, I can see that above us is a directory named `com`, so this must be the right package..."

## JARs and packages

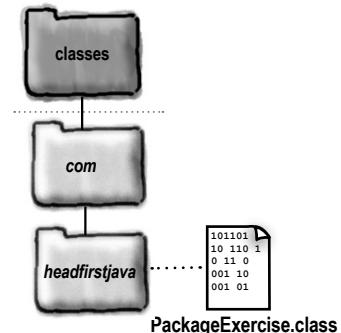
# Making an executable JAR with packages



When your class is in a package, the package directory structure must be inside the JAR! You can't just pop your classes in the JAR the way we did pre-packages. And you must be sure that you don't include any other directories above your package. The first directory of your package (usually com) must be the first directory within the JAR! If you were to accidentally include the directory *above* the package (e.g. the "classes" directory), the JAR wouldn't work correctly.

## Making an executable JAR

- ① Make sure all of your class files are within the correct package structure, under the classes directory.

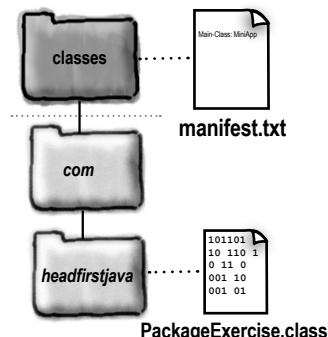


- ② Create a manifest.txt file that states which class has the main() method, and be sure to use the fully-qualified class name!

Make a text file named manifest.txt that has a single line:

```
Main-Class: com.headfirstjava.PackageExercise
```

Put the manifest file into the classes directory



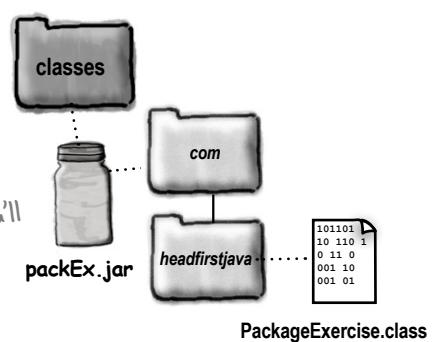
- ③ Run the jar tool to create a JAR file that contains the package directories plus the manifest

The only thing you need to include is the 'com' directory, and the entire package (and all classes) will go into the JAR.

```
%cd MyProject/classes
```

```
%jar -cvmf manifest.txt packEx.jar com
```

All you specify is the com directory! And you'll get everything in it!



## So where did the manifest file go?

Why don't we look inside the JAR and find out? From the command-line, the jar tool can do more than just create and run a JAR. You can extract the contents of a JAR (just like 'unzipping' or 'untarring').

Imagine you've put the packEx.jar into a directory named Skyler.

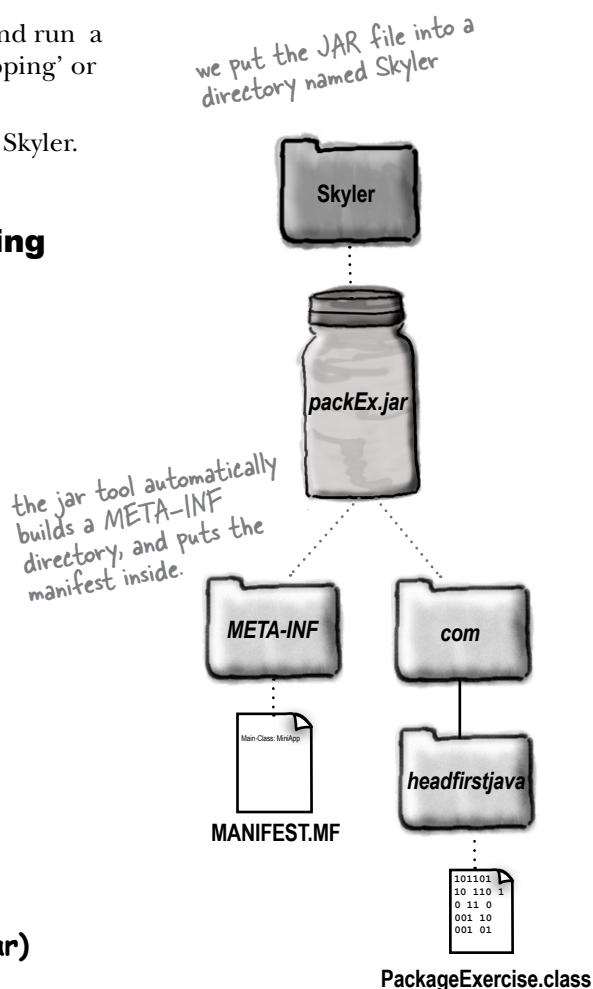
### jar commands for listing and extracting

#### ① List the contents of a JAR

```
% jar -tf packEx.jar
```

↑ "-tf stands for 'Table File' as in  
"show me a table of the JAR file"

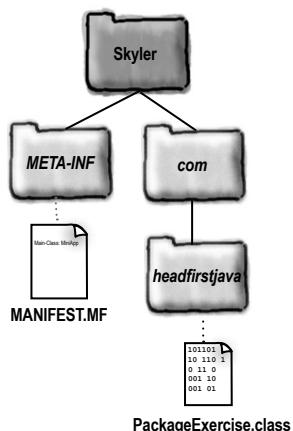
```
File Edit Window Help Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/
PackageExercise.class
```



#### ② Extract the contents of a JAR (i.e. unjar)

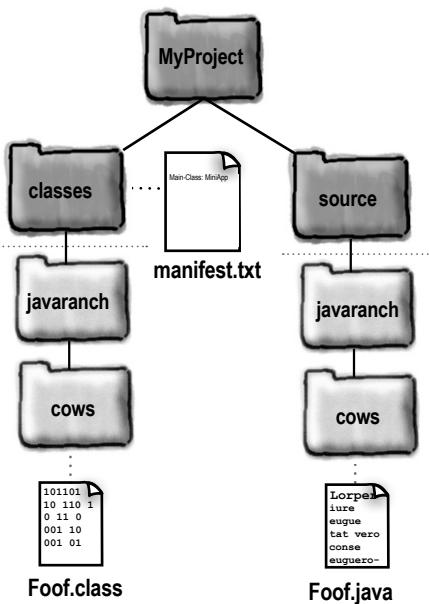
```
% cd Skyler
% jar -xf packEx.jar
```

↑  
"-xf stands for 'Extract File' and it works just like unzipping or untarring. If you extract the packEx.jar, you'll see the META-INF directory and the com directory in your current directory



META-INF stands for 'meta information'. The jar tool creates the META-INF directory as well as the MANIFEST.MF file. It also takes the contents of your manifest file, and puts it into the MANIFEST.MF file. So, your manifest file doesn't go into the JAR, but the contents of it are put into the 'real' manifest (MANIFEST.MF).

## organizing your classes



Given the package/directory structure in this picture, figure out what you should type at the command-line to compile, run, create a JAR, and execute a JAR. Assume we're using the standard where the package directory structure starts just below *source* and *classes*. In other words, the *source* and *classes* directories are not part of the package.

### Compile:

```
%cd source  
%javac _____
```

### Run:

```
%cd _____  
%java _____
```

### Create a JAR

```
%cd _____  
% _____
```

### Execute a JAR

```
%cd _____  
% _____
```

Bonus question: What's wrong with the package name?

there are no  
Dumb Questions

**Q:** What happens if you try to run an executable JAR, and the end-user doesn't have java installed?

**A:** Nothing will run, since without a JVM, Java code can't run. The end-user must have Java installed.

**Q:** How can I get Java installed on the end-user's machine?

Ideally, you can create a custom installer and distribute it along with your application. Several companies offer installer programs ranging from simple to extremely powerful. An installer program could, for example, detect whether or not the end-user has an appropriate version of Java installed, and if not, install and configure Java before installing your application. Installshield, InstallAnywhere, and DeployDirector all offer Java installer solutions.

Another cool thing about some of the installer programs is that you can even make a deployment CD-ROM that includes installers for all major Java platforms, so... one CD to rule them all. If the user's running on Solaris, for example, the Solaris version of Java is installed. On Windows, the Windows, version, etc. If you have the budget, this is by far the easiest way for your end-users to get the right version of Java installed and configured.

### BULLET POINTS

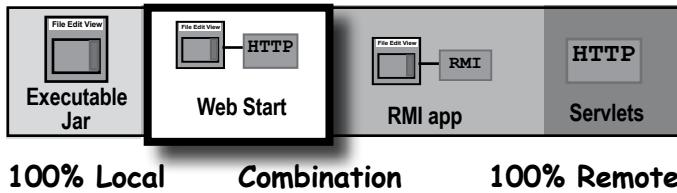


- Organize your project so that your source code and class files are not in the same directory.
- A standard organization structure is to create a *project* directory, and then put a *source* directory and a *classes* directory inside the project directory.
- Organizing your classes into packages prevents naming collisions with other classes, if you prepend your reverse domain name on to the front of a class name.
- To put a class in a package, put a package statement at the top of the source code file, before any import statements:  
`package com.wickedlysmart;`
- To be in a package, a class must be in a *directory structure that exactly matches the package structure*. For a class, `com.wickedlysmart.Foo`, the Foo class must be in a directory named `wickedlysmart`, which is in a directory named `com`.
- To make your compiled class land in the correct package directory structure under the *classes* directory, use the `-d` compiler flag:  
`% cd source`  
`% javac -d ../classes com/wickedlysmart/Foo.java`
- To run your code, cd to the *classes* directory, and give the fully-qualified name of your class:  
`% cd classes`  
`% java com.wickedlysmart.Foo`
- You can bundle your classes into JAR (Java ARchive) files. JAR is based on the pkzip format.
- You can make an executable JAR file by putting a manifest into the JAR that states which class has the `main()` method. To create a manifest file, make a text file with an entry like the following (for example):  
`Main-Class: com.wickedlysmart.Foo`
- Be sure you hit the return key after typing the `Main-Class` line, or your manifest file may not work.
- To create a JAR file, type:  
`jar -cvfm manifest.txt MyJar.jar com`
- The entire package directory structure (and *only* the directories matching the package) must be immediately inside the JAR file.
- To run an executable JAR file, type:  
`java -jar MyJar.jar`

wouldn't it be dreamy...

Executable JAR files are nice, but wouldn't it be dreamy if there were a way to make a rich, stand-alone client GUI that could be distributed over the Web? So that you wouldn't have to press and distribute all those CD-ROMs. And wouldn't it be just wonderful if the program could automatically update itself, replacing just the pieces that changed?





## Java Web Start

With Java Web Start (JWS), your application is launched for the first time from a Web browser (get it? *Web Start?*) but it runs as a stand-alone application (well, *almost*), without the constraints of the browser. And once it's downloaded to the end-user's machine (which happens the first time the user accesses the browser link that starts the download), it *stays* there.

Java Web Start is, among other things, a small Java program that lives on the client machine and works much like a browser plug-in (the way, say, Adobe Acrobat Reader opens when your browser gets a .pdf file). This Java program is called the **Java Web Start 'helper app'**, and its key purpose is to manage the downloading, updating, and launching (executing) of *your* JWS apps.

When JWS downloads your application (an executable JAR), it invokes the main() method for your app. After that, the end-user can launch your application directory from the JWS helper app *without* having to go back through the Web page link.

But that's not the best part. The amazing thing about JWS is its ability to detect when even a small part of application (say, a single class file) has changed on the server, and—with any end-user intervention—download and integrate the updated code.

There's still an issue, of course, like how does the end-user *get* Java and Java Web Start? They need both—Java to run the app, and Java Web Start (a small Java application itself) to handle retrieving and launching the app. But even *that* has been solved. You can set things up so that if your end-users don't have JWS, they can download it from Sun. And if they *do* have JWS, but their version of Java is out-of-date (because you've specified in your JWS app that you need a specific version of Java), the Java 2 Standard Edition can be downloaded to the end-user machine.

Best of all, it's simple to use. You can serve up a JWS app much like any other type of Web resource such as a plain old HTML page or a JPEG image. You set up a Web (HTML) page with a link to your JWS application, and you're in business.

In the end, your JWS application isn't much more than an executable JAR that end-users can download from the Web.

End-users launch a Java Web Start app by clicking on a link in a Web page. But once the app downloads, it runs outside the browser, just like any other stand-alone Java application. In fact, a Java Web Start app is just an executable JAR that's distributed over the Web.

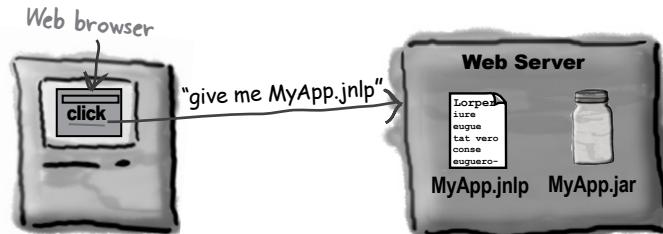
## Java Web Start

### How Java Web Start works

- ① The client clicks on a Web page link to your JWS application (a .jnlp file).

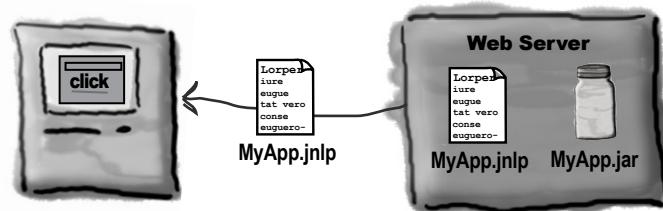
The Web page link

```
<a href="MyApp.jnlp">Click</a>
```



- ② The Web server (HTTP) gets the request and sends back a .jnlp file (this is NOT the JAR).

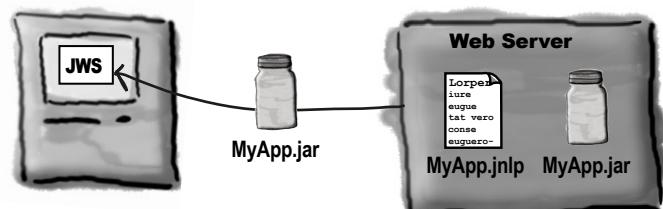
The .jnlp file is an XML document that states the name of the application's executable JAR file.



- ③ Java Web Start (a small 'helper app' on the client) is started up by the browser. The JWS helper app reads the .jnlp file, and asks the server for the MyApp.jar file.

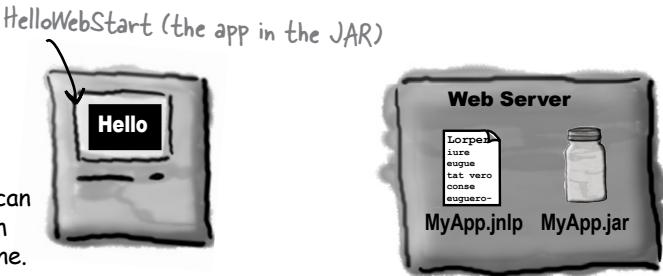


- ④ The Web server 'serves' up the requested .jar file.



- ⑤ Java Web Start gets the JAR and starts the application by calling the specified main() method (just like an executable JAR).

Next time the user wants to run this app, he can open the Java Web Start application and from there launch your app, without even being online.



## The .jnlp file

To make a Java Web Start app, you need to create a .jnlp (Java Network Launch Protocol) file that describes your application. This is the file the JWS app reads and uses to find your JAR and launch the app (by calling the JAR's main() method). A .jnlp file is a simple XML document that has several different things you can put in, but as a minimum, it should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="0.2 1.0"
      codebase="http://127.0.0.1/~kathy"
      href="MyApp.jnlp">
```

The 'codebase' tag is where you specify the 'root' of where your web start stuff is on the server. We're testing this on our localhost, so we're using the local loopback address "127.0.0.1". For web start apps on our internet web server, this would say, "http://www.wickedlysmart.com"

This is the location of the .jnlp file relative to the codebase. This example shows that MyApp.jnlp is available in the root directory of the web server, not nested in some other directory.

```
<information>
  <title>kathy App</title>
  <vendor>Wickedly Smart</vendor>
  <homepage href="index.html"/>
  <description>Head First WebStart demo</description>
  <icon href="kathys.gif"/>
  <offline-allowed/>
</information>
```

Be sure to include all of these tags, or your app might not work correctly! The 'information' tags are used by the JWS helper app, mostly for displaying when the user wants to relaunch a previously-downloaded application.

This means the user can run your program without being connected to the internet. If the user is offline, it means the automatic-updating feature won't work.

```
<resources>
  <j2se version="1.3+/">
  <jar href="MyApp.jar"/>
</resources>
```

This says that your app needs version 1.3 of Java, or greater.

The name of your executable JAR! You might have other JAR files as well, that hold other classes or even sounds and images used by your app.

```
<application-desc main-class="HelloWebStart"/>
</jnlp>
```

This is like the manifest Main-Class entry... it says which class in the JAR has the main() method.

deploying with JWS

## Steps for making and deploying a Java Web Start app

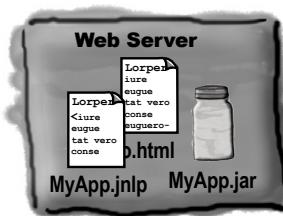
- ① Make an executable JAR for your application.



- ② Write a .jnlp file.



- ③ Place your JAR and .jnlp files on your Web server.



- ④ Add a new mime type to your Web server.

application/x-java-jnlp-file

This causes the server to send the .jnlp file with the correct header, so that when the browser receives the .jnlp file it knows what it is and knows to start the JWS helper app.

Web Server  
configure  
mime type

- ⑤ Create a Web page with a link to your .jnlp file

```
<HTML>
  <BODY>
    <a href="MyApp2.jnlp">Launch My Application</a>
  </BODY>
</HTML>
```

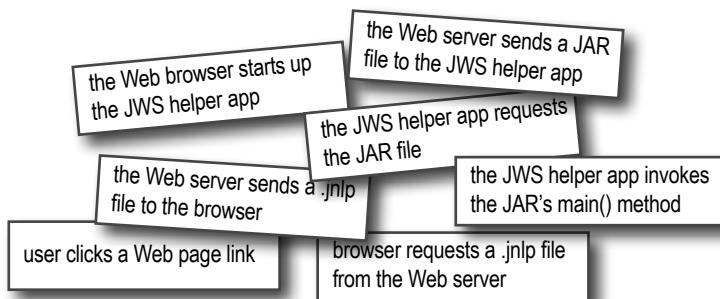




## What's First?



Look at the sequence of events below, and place them in the order in which they occur in a JWS application.



### package, jars and deployment

1.

2.

3.

4.

5.

6.

7.

*there are no*  
**Dumb Questions**

#### Q: How is Java Web Start different from an applet?

**A:** Applets can't live outside of a Web browser. An applet is downloaded from the Web as part of a Web page rather than simply from a Web page. In other words, to the browser, the applet is just like a JPEG or any other resource. The browser uses either a Java plug-in or the browser's own built-in Java (far less common today) to run the applet. Applets don't have the same level of functionality for things such as automatic updating, and they must always be launched from the browser. With JWS applications, once they're downloaded from the Web, the user doesn't even have to be using a browser to relaunch the application locally. Instead, the user can start up the JWS helper app, and use it to launch the already-downloaded application again.

#### Q: What are the security restrictions of JWS?

**A:** JWS apps have several limitations including being restricted from reading and writing to the user's hard drive. But... JWS has its own API with a special open and save dialog box so that, with the user's permission, your app can save and read its own files in a special, restricted area of the user's drive.

#### BULLET POINTS

- Java Web Start technology lets you deploy a stand-alone client application from the Web.
- Java Web Start includes a 'helper app' that must be installed on the client (along with Java).
- A Java Web Start (JWS) app has two pieces: an executable JAR and a .jnlp file.
- A .jnlp file is a simple XML document that describes your JWS application. It includes tags for specifying the name and location of the JAR, and the name of the class with the main() method.
- When a browser gets a .jnlp file from the server (because the user clicked on a link to the .jnlp file), the browser starts up the JWS helper app.
- The JWS helper app reads the .jnlp file and requests the executable JAR from the Web server.
- When the JWS gets the JAR, it invokes the main() method (specified in the .jnlp file).

**exercise:** True or False



We explored packaging, deployment, and JWS in this chapter. Your job is to decide whether each of the following statements is true or false.

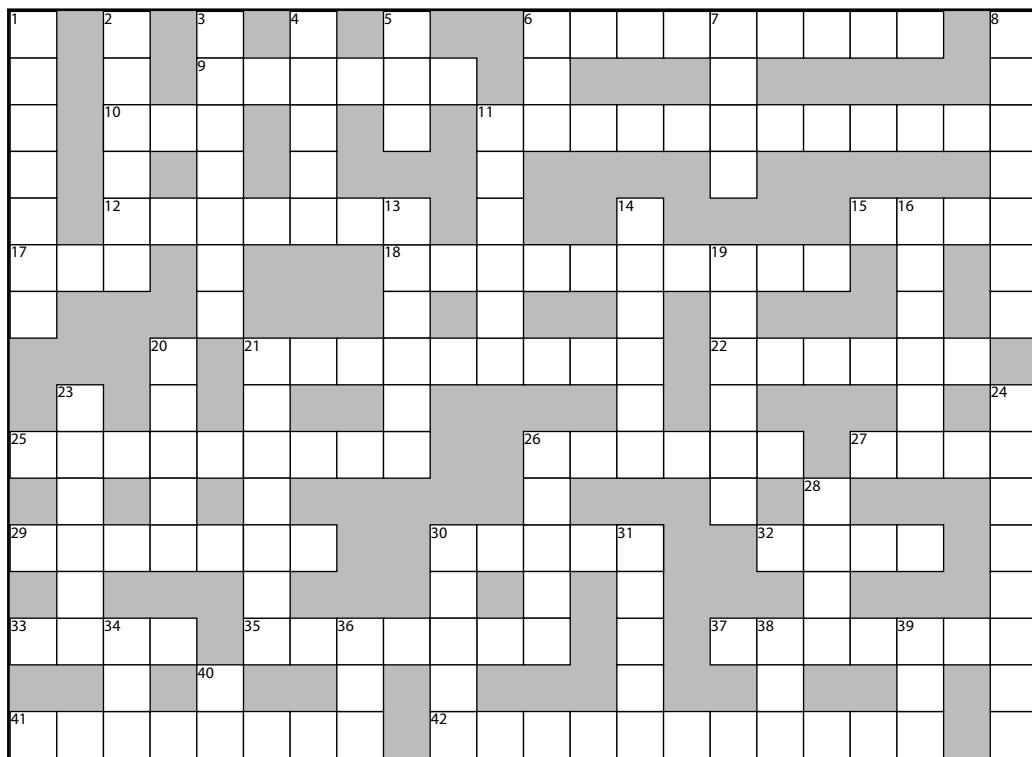
**TRUE OR FALSE**

1. The Java compiler has a flag, -d, that lets you decide where your .class files should go.
2. A JAR is a standard directory where your .class files should reside.
3. When creating a Java Archive you must create a file called jar.mf.
4. The supporting file in a Java Archive declares which class has the main() method.
5. JAR files must be unzipped before the JVM can use the classes inside.
6. At the command line, Java Archives are invoked using the -arch flag.
7. Package structures are meaningfully represented using hierarchies.
8. Using your company's domain name is not recommended when naming packages.
9. Different classes within a source file can belong to different packages.
10. When compiling classes in a package, the -p flag is highly recommended.
11. When compiling classes in a package, the full name must mirror the directory tree.
12. Judicious use of the -d flag can help to assure that there are no typos in your class tree.
13. Extracting a JAR with packages will create a directory called meta-inf.
14. Extracting a JAR with packages will create a file called manifest.mf.
15. The JWS helper app always runs in conjunction with a browser.
16. JWS applications require a .nlp (Network Launch Protocol) file to work properly.
17. A JWS's main method is specified in its JAR file.



package, jars and deployment

## Summary-Cross 7.0



Anything in the book  
is fair game for this  
one!

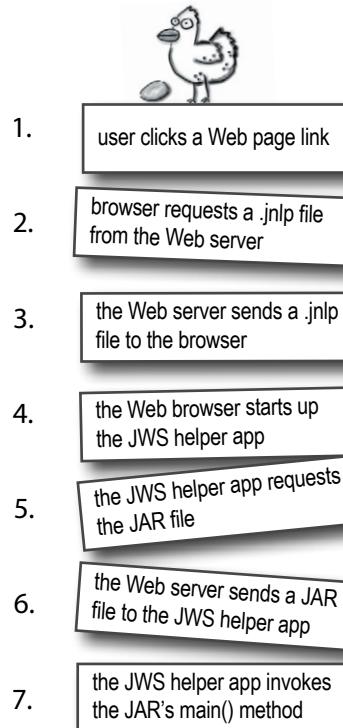
### Across

- 6. Won't travel
- 9. Don't split me
- 10. Release-able
- 11. Got the key
- 12. I/O gang
- 15. Flatten
- 17. Encapsulated returner
- 18. Ship this one
- 21. Make it so
- 22. I/O sieve
- 25. Disk leaf
- 26. Mine is unique
- 27. GUI's target
- 29. Java team
- 30. Factory
- 32. For a while
- 33. Atomic \* 8
- 35. Good as new
- 37. Pairs event
- 41. Where do I start
- 42. A little firewall

### Down

- 1. Pushy widgets
- 2. \_\_\_\_ of my desire
- 3. 'Abandoned' moniker
- 4. A chunk
- 5. Math not trig
- 6. Be brave
- 7. Arrange well
- 8. Swing slang
- 11. I/O canals
- 13. Organized release
- 14. Not for an instance
- 16. Who's allowed
- 19. Efficiency expert
- 20. Early exit
- 21. Common wrapper
- 23. Yes or no
- 24. Java jackets
- 26. Not behavior
- 28. Socket's suite
- 30. I/O cleanup
- 31. Milli-nap
- 34. Trig method
- 36. Encaps method
- 38. JNLP format
- 39. VB's final
- 40. Java branch

## exercise solutions



- True** 1. The Java compiler has a flag, -d, that lets you decide where your .class files should go.
- False** 2. A JAR is a standard directory where your .class files should reside.
- False** 3. When creating a Java Archive you must create a file called jar.mf.
- True** 4. The supporting file in a Java Archive declares which class has the main() method.
- False** 5. JAR files must be unzipped before the JVM can use the classes inside.
- False** 6. At the command line, Java Archives are invoked using the -arch flag.
- True** 7. Package structures are meaningfully represented using hierarchies.
- False** 8. Using your company's domain name is not recommended when naming packages.
- False** 9. Different classes within a source file can belong to different packages.
- False** 10. When compiling classes in a package, the -p flag is highly recommended.
- True** 11. When compiling classes in a package, the full name must mirror the directory tree.
- True** 12. Judicious use of the -d flag can help to assure that there are no typos in your tree.
- True** 13. Extracting a JAR with packages will create a directory called meta-inf.
- True** 14. Extracting a JAR with packages will create a file called manifest.mf.
- False** 15. The JWS helper app always runs in conjunction with a browser.
- False** 16. JWS applications require a .nlp (Network Launch Protocol) file to work properly.
- False** 17. A JWS's main method is specified in its JAR file.



## Summary-Cross 7.0

D	O	G	T	M	TRAN	SIENT	W
I	B	A	T	O	M	I	C
A	J	R	K	N	S	Y	NCHRONIZED
L	E	B	E		T	T	G
O	C	H	A	I	N	E	SAVE
GET	G			EXECUTABLE		C	T
S	E		P	A	A	U	C
B	R	I	M	P	L	E	FILTER
DIRECTORY	N	O	E	M	E	T	S
NAME	O	IMPLEMENT		NT	SOCKET	USER	W
PACKAGE	R	I	F	E	SOCK	R	A
RE	E	M	T	E	E	P	P
BYTERESTORE	RESTORE	L	T	L	LOOP	R	P
BYTE	RESTORE	E	E	E	E	M	X
MANIFEST	AI	E	S	E	EXTREME	X	R
					ENCAPSULATE	S	



## 18 remote deployment with RMI

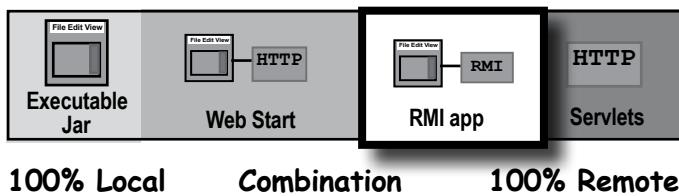
# Distributed Computing



Everyone says long-distance relationships are hard, but with RMI, it's easy. No matter how far apart we *really* are, RMI makes it *seem* like we're together.

**Being remote doesn't have to be a bad thing.** Sure, things *are* easier when all the parts of your application are in one place, in one heap, with one JVM to rule them all. But that's not always possible. Or desirable. What if your application handles powerful computations, but the end-users are on a wimpy little Java-enabled device? What if your app needs data from a database, but for security reasons, only code on your server can access the database? Imagine a big e-commerce back-end, that has to run within a transaction-management system? Sometimes, part of your app must run on a server, while another part (usually a client) must run on a *different* machine. In this chapter, we'll learn to use Java's amazingly simple Remote Method Invocation (RMI) technology. We'll also take a quick peek at Servlets, Enterprise Java Beans (EJB), and Jini, and look at the ways in which EJB and Jini *depend* on RMI. We'll end the book by writing one of the coolest things you can make in Java, a *universal service browser*.

## how many heaps?

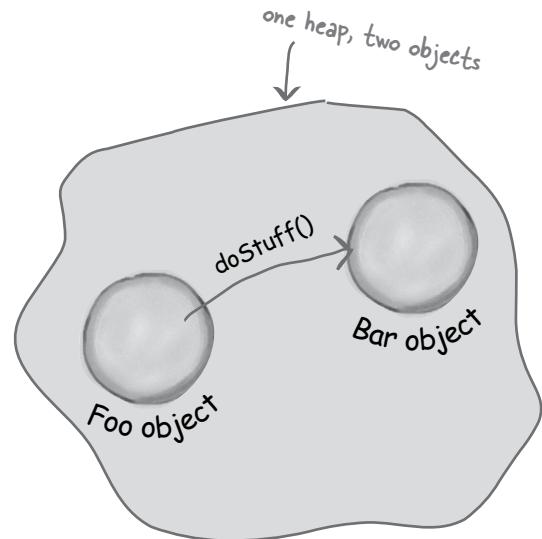


**Method calls are always between two objects on the same heap.**

So far in this book, every method we've invoked has been on an object running in the same virtual machine as the caller. In other words, the calling object and the callee (the object we're invoking the method on) live on the same heap.

```
class Foo {  
    void go() {  
        Bar b = new Bar();  
        b.doStuff();  
    }  
  
    public static void main (String[] args) {  
        Foo f = new Foo();  
        f.go();  
    }  
}
```

In the code above, we know that the Foo instance referenced by *f* and the Bar object referenced by *b* are both on the same heap, run by the same JVM. Remember, the JVM is responsible for stuffing bits into the reference variable that represent *how to get to an object on the heap*. The JVM always knows where each object is, and how to get to it. But the JVM can know about references on only its *own* heap! You can't, for example, have a JVM running on one machine knowing about the heap space of a JVM running on a *different* machine. In fact, a JVM running on one machine can't know anything about a different JVM running on the *same* machine. It makes no difference if the JVMs are on the same or different physical machines; it matters only that the two JVMs are, well, two different invocations of the JVM.



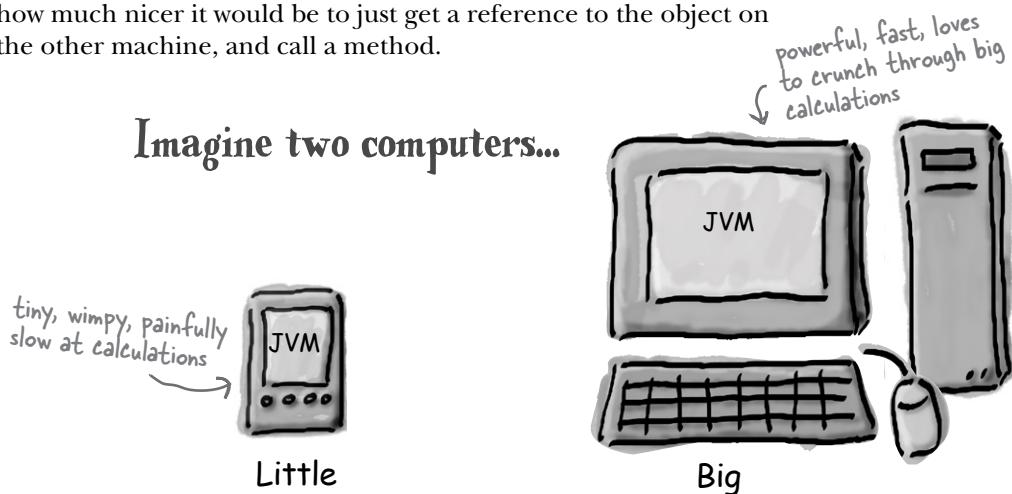
In most applications, when one object calls a method on another, both objects are on the same heap. In other words, both are running within the same JVM.

## What if you want to invoke a method on an object running on another machine?

We know how to get information from one machine to another—with Sockets and I/O. We open a Socket connection to another machine, and get an OutputStream and write some data to it.

But what if we actually want to *call a method* on something running in another machine... another JVM? Of course we could always build our own protocol, and when you send data to a ServerSocket the server could parse it, figure out what you meant, do the work, and send back the result on another stream. What a pain, though. Think how much nicer it would be to just get a reference to the object on the other machine, and call a method.

Imagine two computers...



Big has something Little wants.

Compute power.

Little wants to send some data to Big, so that Big can do the heavy computing.

Little wants simply to call a method...

```
double doCalcUsingDatabase(CalcNumbers numbers)
```

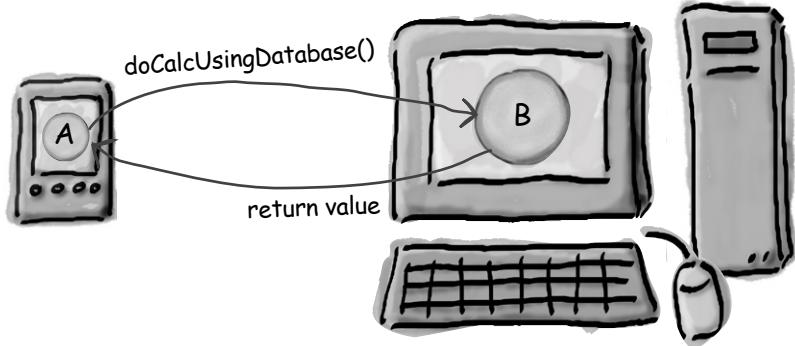
and get back the result.

But how can Little get a reference to an object on Big?

**two objects, two heaps**

## Object A, running on Little, wants to call a method on Object B, running on Big.

The question is, how do we get an object on one machine (which means a different heap/JVM) to call a method on another machine?



## But you can't do that.

Well, not directly anyway. You can't get a reference to something on another heap. If you say:

Dog d = ???

Whatever *d* is referencing must be in the same heap space as the code running the statement.

But imagine you want to design something that will use Sockets and I/O to communicate your intention (a method invocation on an object running on another machine), yet still *feel* as though you were making a local method call.

In other words, you want to cause a method invocation on a *remote* object (i.e., an object in a heap somewhere else), but with code that lets you *pretend* that you're invoking a method on a local object. The ease of a plain old everyday method call, but the power of remote method invocation. That's our goal.

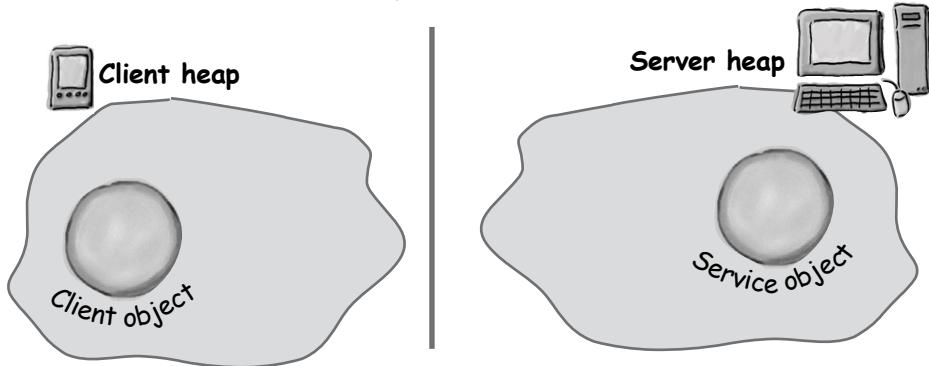
That's what RMI (Remote Method Invocation) gives you!

But let's step back and imagine how you would design RMI if you were doing it yourself. Understanding what you'd have to build yourself will help you learn how RMI works.

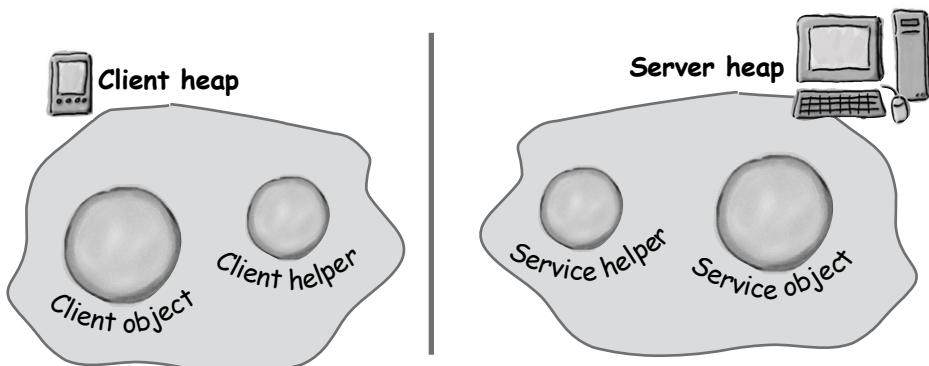
## A design for remote method calls

**Create four things: server, client, server helper, client helper**

- ① Create client and server apps. The server app is the **remote service** that has an object with the method that the client wants to invoke.



- ② Create client and server 'helpers'. They'll handle all the low-level networking and I/O details so your client and service can pretend like they're in the same heap.



## client and server helpers

### The role of the ‘helpers’

The ‘helpers’ are the objects that actually do the communicating. They make it possible for the client to *act* as though it’s calling a method on a local object. In fact, it *is*. The client calls a method on the client helper, *as if the client helper were the actual service*. *The client helper is a proxy for the Real Thing.*

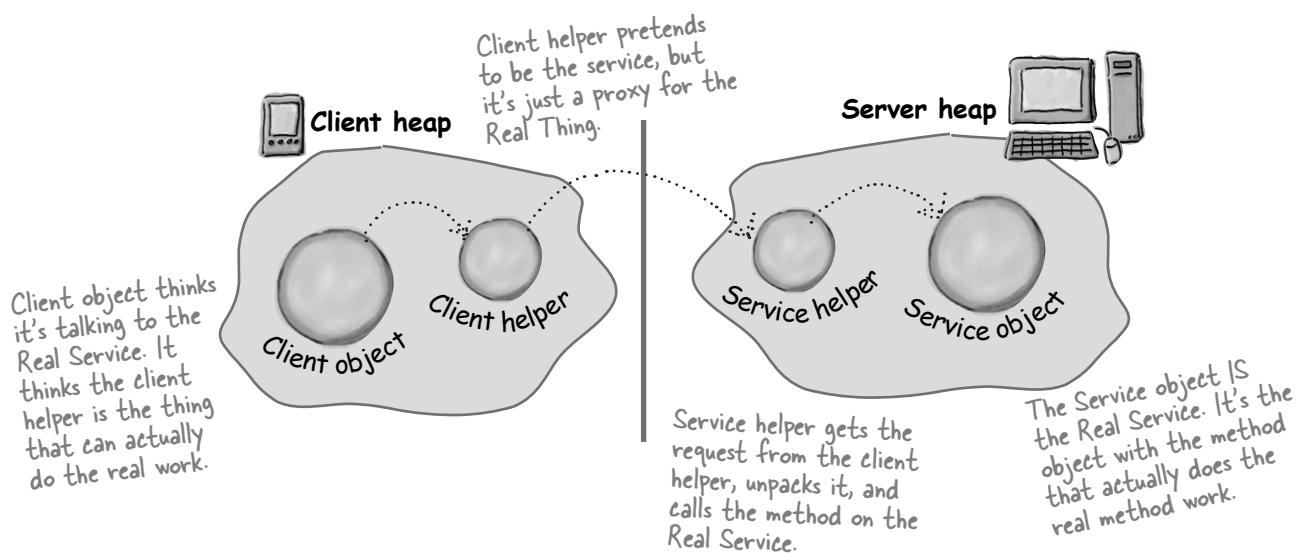
In other words, the client object *thinks* it’s calling a method on the remote service, because the client helper is *pretending* to be the service object. *Pretending to be the thing with the method the client wants to call!*

But the client helper isn’t really the remote service. Although the client helper *acts* like it (because it has the same method that the service is advertising), the client helper doesn’t have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the *real* method on the *real* service object. So to the service object, the call is local. It’s coming from the service helper, not a remote client.

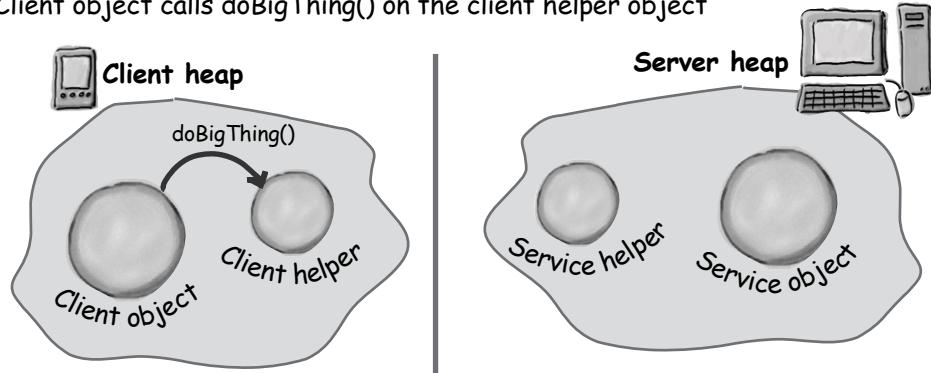
The service helper gets the return value from the service, packs it up, and ships it back (over a Socket’s output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

Your client object gets to act like it’s making remote method calls. But what it’s really doing is calling methods on a heap-local ‘proxy’ object that handles all the low-level details of Sockets and streams.

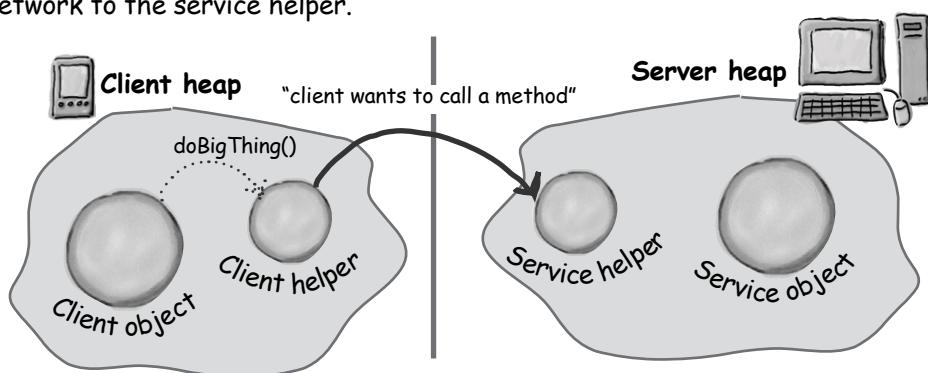


## How the method call happens

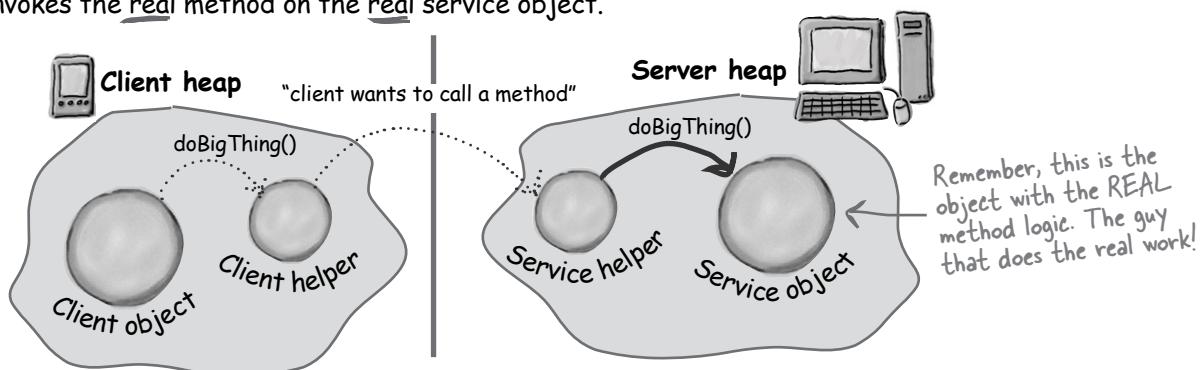
- ① Client object calls doBigThing() on the client helper object



- ② Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



- ③ Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



## RMI helper objects

# Java RMI gives you the client and service helper objects!

In Java, RMI builds the client and service helper objects for you, and it even knows how to make the client helper look like the Real Service. In other words, RMI knows how to give the client helper object the same methods you want to call on the remote service.

Plus, RMI provides all the runtime infrastructure to make it work, including a lookup service so that the client can find and get the client helper (the proxy for the Real Service).

With RMI, you don't write *any* of the networking or I/O code yourself. The client gets to call remote methods (i.e. the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

Almost.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky!

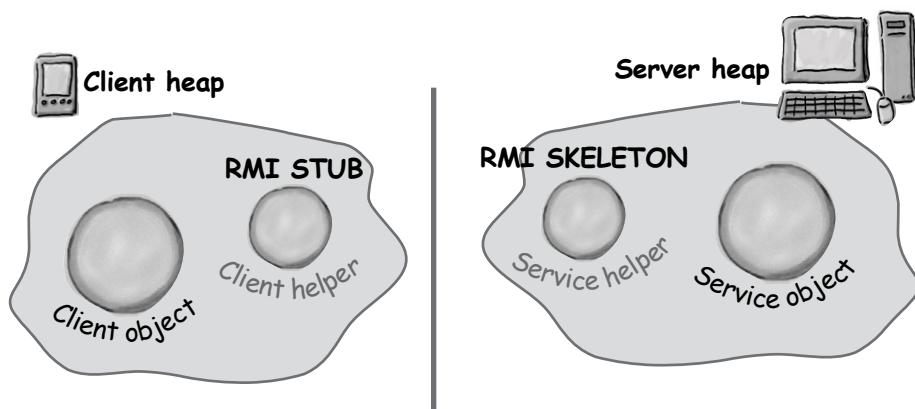
They throw exceptions all over the place.

So, the client does have to acknowledge the risk. The client has to acknowledge that when it calls a remote method, even though to the client it's just a local call to the proxy/helper object, the call *ultimately* involves Sockets and streams. The client's original call is *local*, but the proxy turns it into a *remote* call. A remote call just means a method that's invoked on an object on another JVM. *How* the information about that call gets transferred from one JVM to another depends on the protocol used by the helper objects.

With RMI, you have a choice of protocols: JRMP or IIOP. JRMP is RMI's 'native' protocol, the one made just for Java-to-Java remote calls. IIOP, on the other hand, is the protocol for CORBA (Common Object Request Broker Architecture), and lets you make remote calls on things which aren't necessarily Java objects. CORBA is usually *much* more painful than RMI, because if you don't have Java on both ends, there's an awful lot of translation and conversion that has to happen.

But thankfully, all we care about is Java-to-Java, so we're sticking with plain old, remarkably easy RMI.

## In RMI, the client helper is a 'stub' and the server helper is a 'skeleton'.



# Making the Remote Service

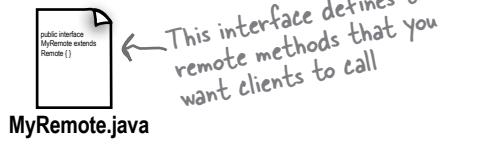
This is an **overview** of the five steps for making the remote service (that runs on the server). Don't worry, each step is explained in detail over the next few pages.



## Step one:

### Make a Remote Interface

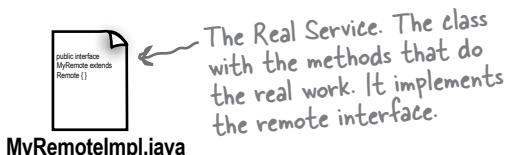
The remote interface defines the methods that a client can call remotely. It's what the client will use as the polymorphic class type for your service. Both the Stub and actual service will implement this!



## Step two:

### Make a Remote Implementation

This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client wants to call methods on.



## Step three:

### Generate the stubs and skeletons using rmic

These are the client and server 'helpers'. You don't have to create these classes or ever look at the source code that generates them. It's all handled automatically when you run the rmic tool that ships with your Java development kit.

Running rmic against the actual service implementation class...

spits out two new classes for the helper objects

```
File Edit Window Help Eat
% rmic MyRemoteImpl
```

MyRemoteImpl\_Stub.class

MyRemoteImpl\_Skel.class

## Step four:

### Start the RMI registry (rmiregistry)

The *rmiregistry* is like the white pages of a phone book. It's where the user goes to get the proxy (the client stub/helper object).

```
File Edit Window Help Drink
% rmiregistry
```

run this in a separate terminal

## Step five:

### Start the remote service

You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.

```
File Edit Window Help BeMerry
% java MyRemoteImpl
```

a remote interface

## Step one: Make a Remote Interface



### ① Extend java.rmi.Remote

Remote is a ‘marker’ interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say ‘extends’ here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```

Your interface has to announce that it's for remote method calls. An interface can't implement anything, but it can extend other interfaces.

### ② Declare that all methods throw a RemoteException

The remote interface is the one the client uses as the polymorphic type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of Bad Things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*; ← the Remote interface is in java.rmi
```

```
public interface MyRemote extends Remote {  
    public String sayHello() throws RemoteException;  
}
```

Every remote method call is considered ‘risky’. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

### ③ Be sure arguments and return values are primitives or Serializable

Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that’s done through Serialization. Same thing with return values. If you use primitives, Strings, and the majority of types in the API (including arrays and collections), you’ll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

```
public String sayHello() throws RemoteException;
```

↑  
This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

## Step two: Make a Remote Implementation

### ① Implement the Remote interface

Your service has to implement the remote interface—the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() { ←
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.



MyRemoteImpl.java

### ② Extend UnicastRemoteObject

In order to work as a remote service object, your object needs some functionality related to ‘being remote’. The simplest way is to extend UnicastRemoteObject (from the `java.rmi.server` package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

### ③ Write a no-arg constructor that declares a RemoteException

Your new superclass, `UnicastRemoteObject`, has one little problem—its constructor throws a `RemoteException`. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the `RemoteException`. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException {}
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

### ④ Register the service with the RMI registry

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static `rebind()` method of the `java.rmi.Naming` class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

## stubs and skeletons

# Step three: generate stubs and skeletons

### ① Run rmic on the remote implementation class (not the remote interface)

The rmic tool, that comes with the Java software development kit, takes a service implementation and creates two new classes, the stub and the skeleton. It uses a naming convention that is the name of your remote implementation, with either \_Stub or \_Skel added to the end. There are other options with rmic, including not generating skeletons, seeing what the source code for these classes looked like, and even using IIOP as the protocol. The way we're doing it here is the way you'll usually do it. The classes will land in the current directory (i.e. whatever you did a cd to). Remember, *rmic* must be able to see your implementation class, so you'll probably run rmic from the directory where your remote implementation is. (We're deliberately not using packages here, to make it simpler. In the Real World, you'll need to account for package directory structures and fully-qualified names).

Notice that you don't say ".class" on the end. Just the class name.

```
File Edit Window Help Whuffie  
%rmic MyRemoteImpl
```

spits out two new classes for the helper objects

```
101101  
10 110 1  
0 11 0  
001 10  
001 01
```

MyRemoteImpl\_Stub.class

```
101101  
10 110 1  
0 11 0  
001 10  
001 01
```

MyRemoteImpl\_Skel.class

# Step four: run rmiregistry

### ① Bring up a terminal and start the rmiregistry.

Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your 'classes' directory.

```
File Edit Window Help Huh?  
%rmiregistry
```

# Step five: start the service

### ① Bring up another terminal and start your service

This might be from a main() method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

```
File Edit Window Help Huh?  
%java MyRemoteImpl
```

## Complete code for the server side



### The Remote interface:

```

import java.rmi.*;
          ↗ RemoteException and Remote
          ↗ interface are in java.rmi package

public interface MyRemote extends Remote {
          ↗ Your interface MUST extend
          ↗ java.rmi.Remote

    public String sayHello() throws RemoteException;
          ↗ All of your remote methods must
          ↗ declare a RemoteException
}

```

### The Remote service (the implementation):

```

import java.rmi.*;
import java.rmi.server.*;
          ↗ UnicastRemoteObject is in the
          ↗ java.rmi.server package
          ↗ extending UnicastRemoteObject is the
          ↗ easiest way to make a remote object

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

    public String sayHello() {           ↗ You have to implement all the
        return "Server says, 'Hey'";
    }                                     ↗ interface methods, of course. But
                                         ↗ notice that you do NOT have to
                                         ↗ declare the RemoteException.

    public MyRemoteImpl() throws RemoteException { }           ↗ You MUST implement your
                                                               ↗ remote interface!!

    public static void main (String[] args) {
        try {
            MyRemote service = new MyRemoteImpl();
            Naming.rebind("Remote Hello", service);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

          ↗ your superclass constructor (for
          ↗ UnicastRemoteObject) declares an exception, so
          ↗ YOU must write a constructor, because it means
          ↗ that your constructor is calling risky code (its
          ↗ super constructor)

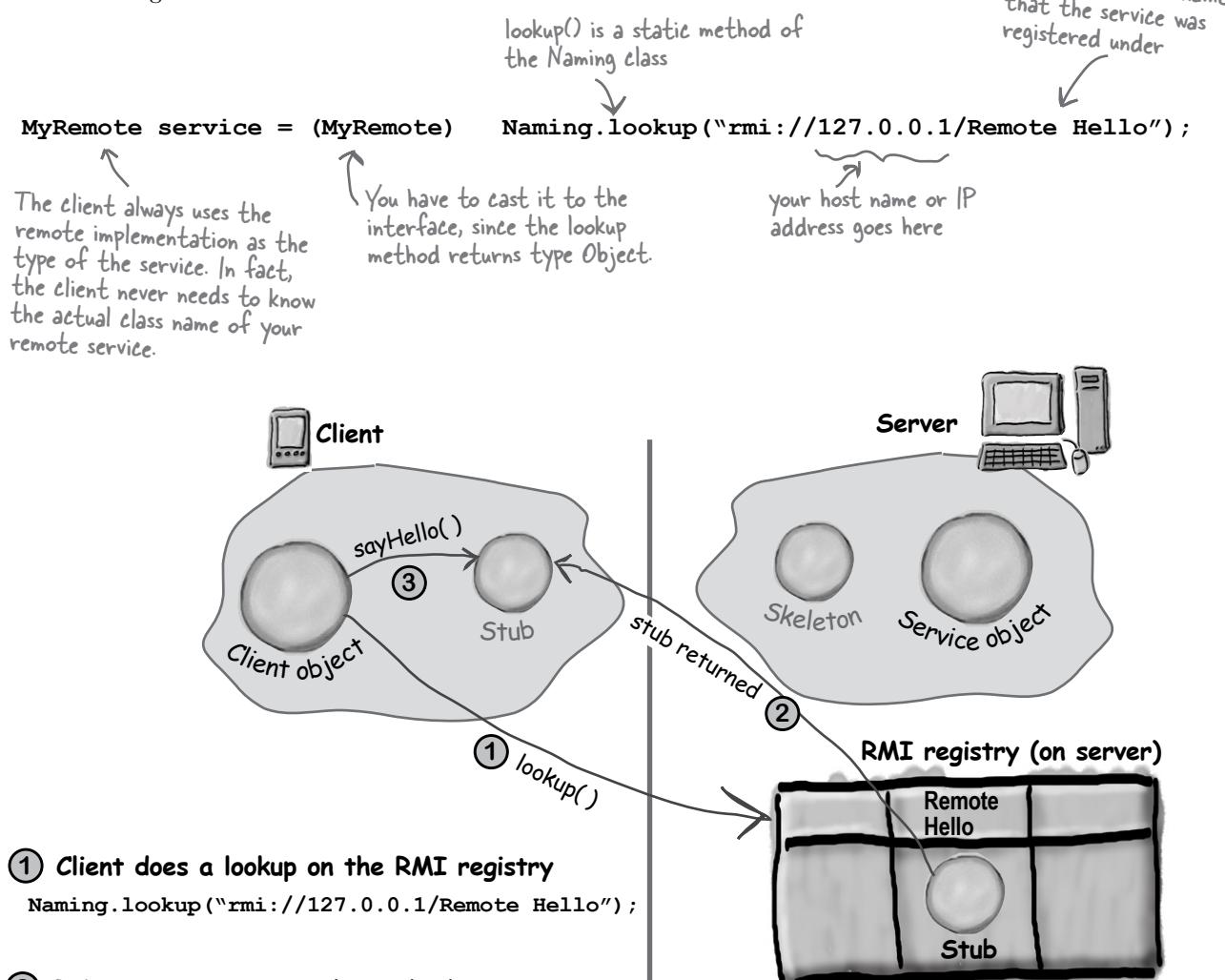
          ↗ Make the remote object, then 'bind' it to the
          ↗ rmiregistry using the static Naming.rebind(). The
          ↗ need to look it up in the rmi registry.

```

## getting the stub

# How does the client get the stub object?

The client has to get the stub object, since that's the thing the client will call methods on. And that's where the RMI registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."



## How does the client get the stub class?

Now we get to the interesting question. Somehow, someway, the client must have the stub class (that you generated earlier using rmic) at the time the client does the lookup, or else the stub won't be serialized on the client and the whole thing blows up. In a simple system, you can simply hand-deliver the stub class to the client.

There's a much cooler way, though, although it's beyond the scope of this book. But just in case you're interested, the cooler way is called "dynamic class downloading". With dynamic class downloading, a stub object (or really any Serialized object) is 'stamped' with a URL that tells the RMI system on the client where to find the class file for that object. Then, in the process of deserializing an object, if RMI can't find the class locally, it uses that URL to do an HTTP Get to retrieve the class file. So you'd need a simple Web server to serve up class files, and you'd also need to change some security parameters on the client. There are a few other tricky issues with dynamic class downloading, but that's the overview.

## Complete client code

```

import java.rmi.*;
import java.util.*;

public class MyRemoteClient {
    public static void main (String[] args) {
        new MyRemoteClient().go();
    }

    public void go() {
        try {
            MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");
            String s = service.sayHello();
            System.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

The Naming class (for doing the remiregistry lookup) is in the java.rmi package

It comes out of the registry as type Object, so don't forget the cast

You need the IP address or hostname

and the name used to bind/rebind the service

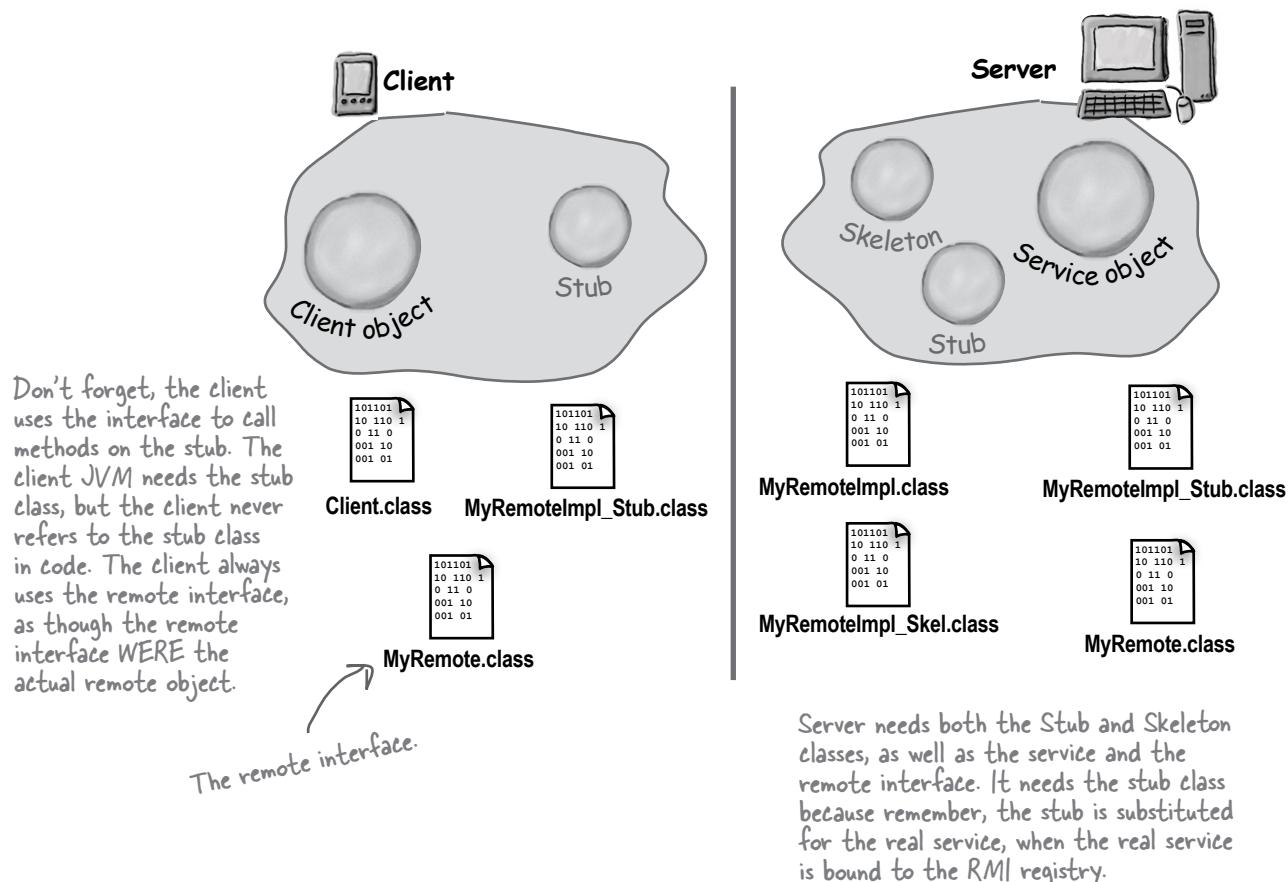
It looks just like a regular old method call! (Except it must acknowledge the RemoteException)

## RMI class files

# Be sure each machine has the class files it needs.

The top three things programmers do wrong with RMI are:

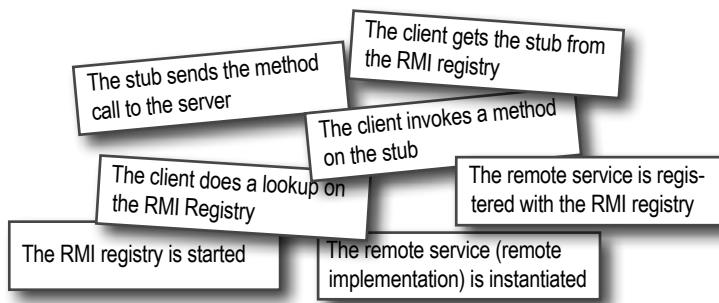
- 1) Forget to start rmiregistry before starting remote service (when you register the service using Naming.rebind(), the rmiregistry must be running!)
- 2) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)
- 3) Forget to give the stub class to the client.





## What's First?

Look at the sequence of events below, and place them in the order in which they occur in a Java RMI application.



- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.



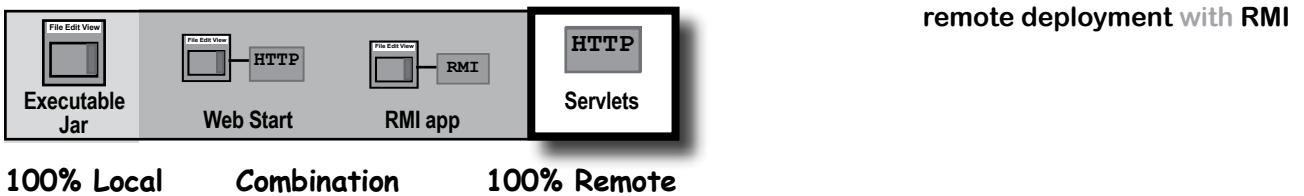
### BULLET POINTS

- An object on one heap cannot get a normal Java reference to an object on a different heap (which means running on a different JVM)
- Java Remote Method Invocation (RMI) makes it seem like you're calling a method on a remote object (i.e. an object in a different JVM), but you aren't.
- When a client calls a method on a remote object, the client is really calling a method on a *proxy* of the remote object. The proxy is called a 'stub'.
- A stub is a client helper object that takes care of the low-level networking details (sockets, streams, serialization, etc.) by packaging and sending method calls to the server.
- To build a remote service (in other words, an object that a remote client can ultimately call methods on), you must start with a remote interface.
- A remote interface must extend the `java.rmi.Remote` interface, and all methods must declare `RemoteException`.
- Your remote service implements your remote interface.
- Your remote service should extend `UnicastRemoteObject`. (Technically there are other ways to create a remote object, but extending `UnicastRemoteObject` is the simplest).
- Your remote service class must have a constructor, and the constructor must declare a `RemoteException` (because the superclass constructor declares one).
- Your remote service must be instantiated, and the object registered with the RMI registry.
- To register a remote service, use the static `Naming.rebind("Service Name", serviceInstance);`
- The RMI registry must be running on the same machine as the remote service, before you try to register a remote object with the RMI registry.
- The client looks up your remote service using the static `Naming.lookup("rmi://MyHostName/ServiceName");`
- Almost everything related to RMI can throw a `RemoteException` (checked by the compiler). This includes registering or looking up a service in the registry, and *all* remote method calls from the client to the stub.

uses for RMI

## Yeah, but who really uses RMI?





remote deployment with RMI

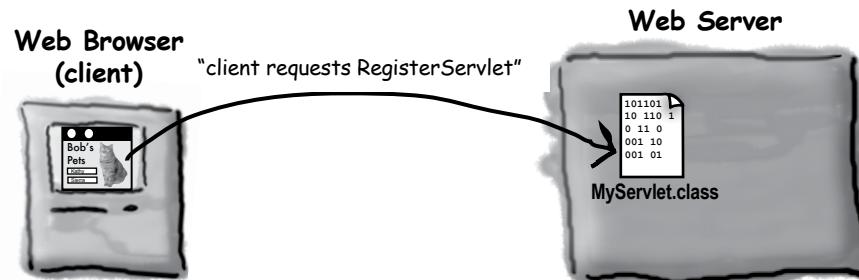
## What about Servlets?

Servlets are Java programs that run on (and with) an HTTP web server. When a client uses a web browser to interact with a web page, a request is sent back to the web server. If the request needs the help of a Java servlet, the web server runs (or calls, if the servlet is already running) the servlet code. Servlet code is simply code that runs on the server, to do work as a result of whatever the client requests (for example, save information to a text file or database on the server). If you're familiar with CGI scripts written in Perl, you know exactly what we're talking about. Web developers use CGI scripts or servlets to do everything from sending user-submitted info to a database, to running a web-site's discussion board.

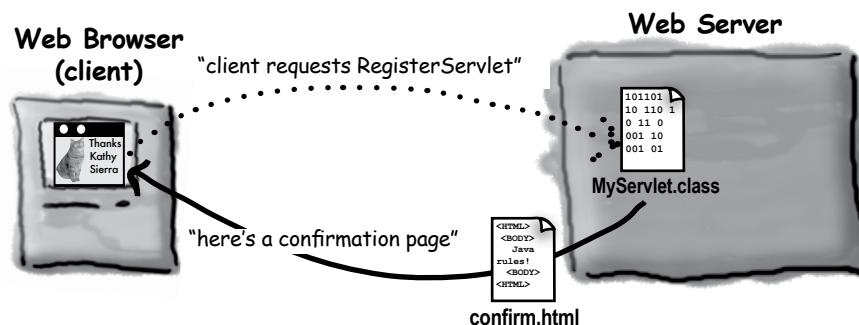
*And even servlets can use RMI!*

By far, the most common use of J2EE technology is to mix servlets and EJBs together, where servlets are the client of the EJB. And in that case, *the servlet is using RMI to talk to the EJBs.* (Although the way you use RMI with EJB is a *little* different from the process we just looked at.)

- ① Client fills out a registration form and clicks 'submit'.  
The HTTP server (i.e. web server) gets the request, sees that it's for a servlet, and sends the request to the servlet.



- ② Servlet (Java code) runs, adds data to the database, composes a web page (with custom info) and sends it back to the client where it displays in the browser.

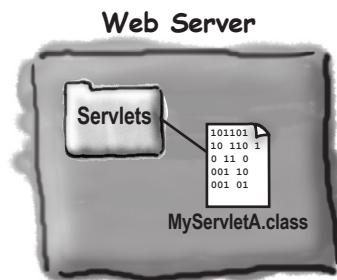


## very simple servlet

# Steps for making and running a servlet

### ① Find out where your servlets need to be placed.

For these examples, we'll assume that you already have a web server up and running, and that it's already configured to support servlets. The most important thing is to find out exactly where your servlet class files have to be placed in order for your server to 'see' them. If you have a web site hosted by an ISP, the hosting service can tell you where to put your servlets, just as they'll tell you where to place your CGI scripts.



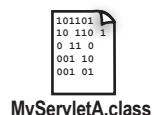
### ② Get the servlets.jar and add it to your classpath

Servlets aren't part of the standard Java libraries; you need the servlets classes packaged into the servlets.jar file. You can download the servlets classes from java.sun.com, or you can get them from your Java-enabled web server (like Apache Tomcat, at the apache.org site). Without these classes, you won't be able to compile your servlets.



### ③ Write a servlet class by extending HttpServlet

A servlet is just a Java class that extends HttpServlet (from the javax.servlet.http package). There are other types of servlets you can make, but most of the time we care only about HttpServlet.



```
public class MyServletA extends HttpServlet { ... }
```

### ④ Write an HTML page that invokes your servlet

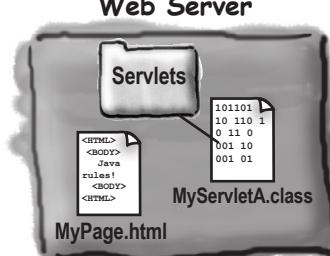
When the user clicks a link that references your servlet, the web server will find the servlet and invoke the appropriate method depending on the HTTP command (GET, POST, etc.)



```
<a href="servlets/MyServletA">This is the most amazing servlet.</a>
```

### ⑤ Make your servlet and HTML page available to your server

This is completely dependent on your web server (and more specifically, on which *version* of Java Servlets that you're using). Your ISP may simply tell you to drop it into a "Servlets" directory on your web site. But if you're using, say, the latest version of Tomcat, you'll have a lot more work to do to get the servlet (and web page) into the right location. (We just happen to have a book on this too.)



## A very simple Servlet

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

Besides io, we need to import two of the servlet packages.
Remember, these two packages are NOT part of the Java
standard libraries -- you have to download them separately

public class MyServletA extends HttpServlet {
    Most 'normal' servlets will extend
    HttpServlet, then override one or
    more methods.

    Override the doGet for simple
    HTTP GET messages.

    public void doGet (HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
        This tells the server (and browser) what kind of
        'thing' is coming back from the server as a result of
        this servlet running.

        PrintWriter out = response.getWriter();
        The response object gives us an output stream to
        'write' information back out to the server.

        String message = "If you're reading this, it worked!";

        out.println("<HTML><BODY>");
        out.println("<H1>" + message + "</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
  
```

The web server calls this method, handing you the client's request (you can get data out of it) and a 'response' object that you'll use to send back a response (a page).

What we 'write' is an HTML page! The page gets delivered through the server back to the browser, just like any other HTML page, even though this is a page that never existed until now. In other words, there's no .html file somewhere with this stuff in it.

What the web page looks like:

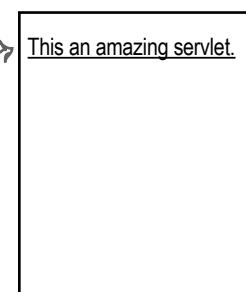
## HTML page with a link to this servlet

```

<HTML>
  <BODY>
    <a href="servlets/MyServletA">This is an amazing servlet.</a>
  </BODY>
</HTML>

```

click the link →  
to trigger the  
servlet



**BULLET POINTS**

- Servlets are Java classes that run entirely on (and/or within) an HTTP (web) server.
- Servlets are useful for running code on the server as a result of client interaction with a web page. For example, if a client submits information in a web page form, the servlet can process the information, add it to a database, and send back a customized, confirmation response page.
- To compile a servlet, you need the servlet packages which are in the `servlets.jar` file. The servlet classes are not part of the Java standard libraries, so you need to download the `servlets.jar` from [java.sun.com](http://java.sun.com) or get them from a servlet-capable web server. (Note: the Servlet library is included with the Java 2 Enterprise Edition (J2EE))
- To run a servlet, you must have a web server capable of running servlets, such as the Tomcat server from [apache.org](http://apache.org).
- Your servlet must be placed in a location that's specific to your particular web server, so you'll need to find that out before you try to run your servlets. If you have a web site hosted by an ISP that supports servlets, the ISP will tell you which directory to place your servlets in.
- A typical servlet extends `HttpServlet` and overrides one or more servlet methods, such as `doGet()` or `doPost()`.
- The web server starts the servlet and calls the appropriate method (`doGet()`, etc.) based on the client's request.
- The servlet can send back a response by getting a `PrintWriter` output stream from the `response` parameter of the `doGet()` method.
- The servlet 'writes' out an HTML page, complete with tags.

*there are no  
Dumb Questions*

**Q:** What's a JSP, and how does it relate to servlets?

**A:** JSP stands for Java Server Pages. In the end, the web server turns a JSP into a servlet, but the difference between a servlet and a JSP is what YOU (the developer) actually create. With a servlet, you write a Java *class* that contains *HTML* in the output statements (if you're sending back an HTML page to the client). But with a JSP, it's the opposite—you write an *HTML* page that contains *Java* code!

This gives you the ability to have dynamic web pages where you write the page as a normal HTML page, except you embed Java code (and other tags that "trigger" Java code at runtime) that gets processed at runtime. In other words, part of the page is customized at runtime when the Java code runs.

The main benefit of JSP over regular servlets is that it's just a lot easier to write the HTML part of a servlet as a JSP page than to write HTML in the torturous print out statements in the servlet's response. Imagine a reasonably complex HTML page, and now imagine formatting it within `println` statements. Yikes!

But for many applications, it isn't necessary to use JSPs because the servlet doesn't need to send a dynamic response, or the HTML is simple enough not to be such a big pain. And, there are still many web servers out there that support servlets but do not support JSPs, so you're stuck.

Another benefit of JSPs is that you can separate the work by having the Java developers write the servlets and the web page developers write the JSPs. That's the promised benefit, anyway. In reality, there's still a Java learning curve (and a tag learning curve) for anyone writing a JSP, so to think that an HTML web page designer can bang out JSPs is not realistic. Well, not without tools. But that's the good news—authoring tools are starting to appear, that help web page designers create JSPs without writing the code from scratch.

**Q:** Is this all you're gonna say about servlets? After such a huge thing on RMI?

**A:** Yes. RMI is part of the Java language, and all the classes for RMI are in the standard libraries. Servlets and JSPs are *not* part of the Java language; they're considered *standard extensions*. You can run RMI on any modern JVM, but Servlets and JSPs require a properly configured web server with a servlet "container". This is our way of saying, "it's beyond the scope of this book." But you can read much more in the lovely *Head First Servlets & JSP*.

## Just for fun, let's make the Phrase-O-Matic work as a servlet

Now that we told you that we won't say any more about servlets, we can't resist servletizing (yes, we *can* verbify it) the Phrase-O-Matic from chapter 1. A servlet is still just Java. And Java code can call Java code from other classes. So a servlet is free to call a method on the Phrase-O-Matic. All you have to do is drop the Phrase-O-Matic class into the same directory as your servlet, and you're in business. (The Phrase-O-Matic code is on the next page).



```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
        String title = "PhraseOMatic has generated the following phrase./";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOMatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>" + PhraseOMatic.makePhrase());
        out.println("<P><a href=\"KathyServlet\">make another phrase</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}

```

See? Your servlet can call methods on another class. In this case, we're calling the static `makePhrase()` method of the `PhraseOMatic` class (on the next page).

## Phrase-O-Matic code

# Phrase-O-Matic code, servlet-friendly

This is a slightly different version from the code in chapter one. In the original, we ran the entire thing in a main() method, and we had to rerun the program each time to generate a new phrase at the command-line. In this version, the code simply returns a String (with the phrase) when you invoke the static makePhrase() method. That way, you can call the method from any other code and get back a String with the randomly-composed phrase.

Please note that these long String[] array assignments are a victim of word-processing here—don't type in the hyphens! Just keep on typing and let your code editor do the wrapping. And whatever you do, don't hit the return key in the middle of a String (i.e. something between double quotes).

```
public class PhraseOMatic {
    public static String makePhrase() {

        // make three sets of words to choose from
        String[] wordListOne = {"24/7", "multi-Tier", "30,000 foot", "B-to-B", "win-win", "front-
end", "web-based", "pervasive", "smart", "six-sigma", "critical-path", "dynamic"};

        String[] wordListTwo = {"empowered", "sticky", "valued-added", "oriented", "centric",
"distributed", "clustered", "branded", "outside-the-box", "positioned", "networked", "fo-
cused", "leveraged", "aligned", "targeted", "shared", "cooperative", "accelerated"};

        String[] wordListThree = {"process", "tipping point", "solution", "architecture",
"core competency", "strategy", "mindshare", "portal", "space", "vision", "paradigm", "mis-
sion"}};

        // find out how many words are in each list
        int oneLength = wordListOne.length;
        int twoLength = wordListTwo.length;
        int threeLength = wordListThree.length;

        // generate three random numbers, to pull random words from each list
        int rand1 = (int) (Math.random() * oneLength);
        int rand2 = (int) (Math.random() * twoLength);
        int rand3 = (int) (Math.random() * threeLength);

        // now build a phrase
        String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " +
wordListThree[rand3];

        // now return it
        return ("What we need is a " + phrase);
    }
}
```

## Enterprise JavaBeans: RMI on steroids

RMI is great for writing and running remote services. But you wouldn't run something like an Amazon or eBay on RMI alone. For a large, deadly serious, enterprise application, you need something more. You need something that can handle transactions, heavy concurrency issues (like a gazillion people are hitting your server at once to buy those organic dog kibbles), security (not just anyone should hit your payroll database), and data management. For that, you need an *enterprise application server*.

In Java, that means a Java 2 Enterprise Edition (J2EE) server. A J2EE server includes both a web server and an Enterprise JavaBeans (EJB) server, so that you can deploy an application that includes both servlets and EJBs. Like servlets, EJB is way beyond the scope of this book, and there's no way to show "just a little" EJB example with code, but we *will* take a quick look at how it works. (For a much more detailed treatment of EJB, we can recommend the lively Head First EJB certification study guide.)

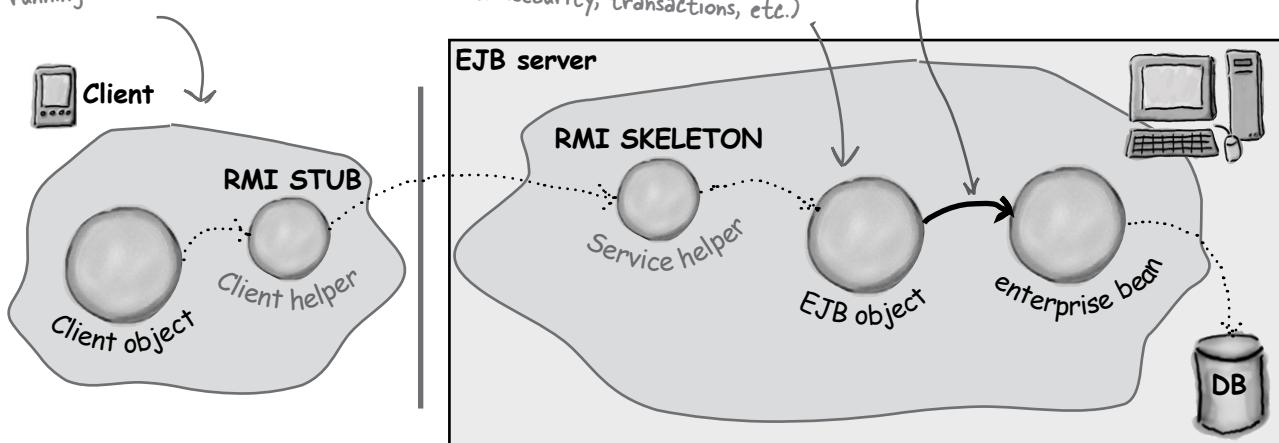
An EJB server adds a bunch of services that you don't get with straight RMI. Things like transactions, security, concurrency, database management, and networking.

An EJB server steps into the middle of an RMI call and layers in all of the services.

This client could be ANYTHING, but typically an EJB client is a servlet running in the same J2EE server.

Here's where the EJB server gets involved! The EJB object intercepts the calls to the bean (the bean holds the real business logic) and layers in all the services provided by the EJB server (security, transactions, etc.)

The bean object is protected from direct client access! Only the server can actually talk to the bean. This lets the server do things like say, "Whoa! This client doesn't have the security clearance to call this method..." Almost everything you pay for in an EJB server happens right here, where the server steps in!



This is only a small part of the EJB picture!

## a little Jini

# For our final trick... a little Jini

We love Jini. We think Jini is pretty much the best thing in Java. If EJB is RMI on steroids (with a bunch of managers), Jini is RMI with *wings*. Pure Java *bliss*. Like the EJB material, we can't get into any of the Jini details here, but if you know RMI, you're three-quarters of the way there. In terms of technology, anyway. In terms of *mindset*, it's time to make a big leap. No, it's time to *fly*.

Jini uses RMI (although other protocols can be involved), but gives you a few key features including:

*Adaptive discovery*

*Self-healing networks*

With RMI, remember, the client has to know the name and location of the remote service. The client code for the lookup includes the IP address or hostname of the remote service (because that's where the RMI registry is running) *and* the logical name the service was registered under.

But with Jini, the client has to know only one thing: *the interface implemented by the service!* That's it.

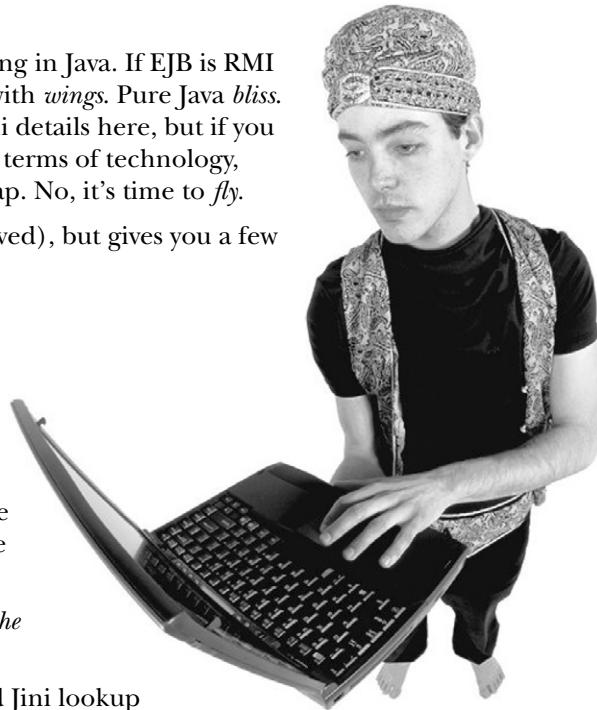
So how do you find things? The trick revolves around Jini lookup services. Jini lookup services are far more powerful and flexible than the RMI registry. For one thing, Jini lookup services announce themselves to the network, *automatically*. When a lookup service comes online, it sends a message (using IP multicast) out to the network saying, "I'm here, if anyone's interested."

But that's not all. Let's say you (a client) come online *after* the lookup service has already announced itself, *you* can send a message to the entire network saying, "Are there any lookup services out there?"

Except that you're not really interested in the lookup service *itself*—you're interested in the services that are *registered* with the lookup service. Things like RMI remote services, other serializable Java objects, and even devices such as printers, cameras, and coffee-makers.

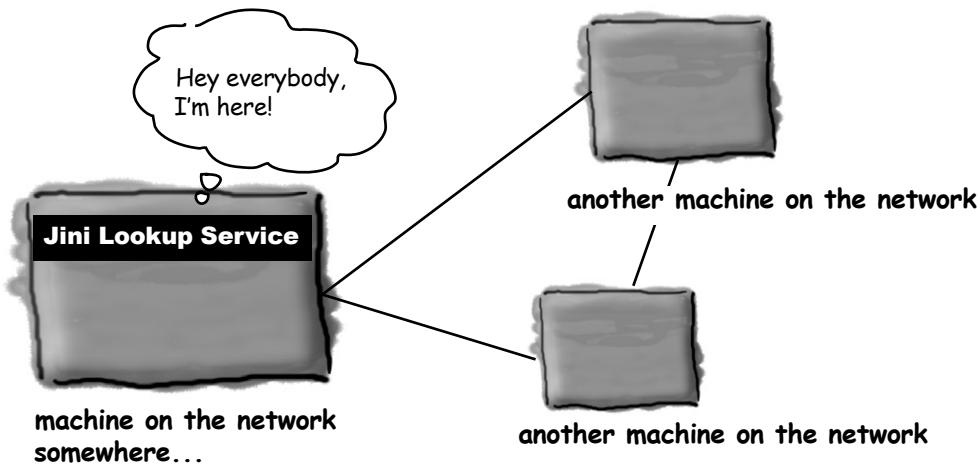
And here's where it gets even more fun: when a service comes online, it will dynamically discover (and *register* itself with) any Jini lookup services on the network. When the service registers with the lookup service, the service sends a serialized object to be placed in the lookup service. That serialized object can be a stub to an RMI remote service, a driver for a networked device, or even the whole service itself that (once you get it from the lookup service) runs locally on your machine. And instead of registering by *name*, the service registers by the *interface* it implements.

Once you (the client) have a reference to a lookup service, you can say to that lookup service, "Hey, do you have anything that implements ScientificCalculator?" At that point, the lookup service will check its list of registered interfaces, and assuming it finds a match, says back to you, "Yes I *do* have something that implements that interface. Here's the serialized object the ScientificCalculator service registered with me."

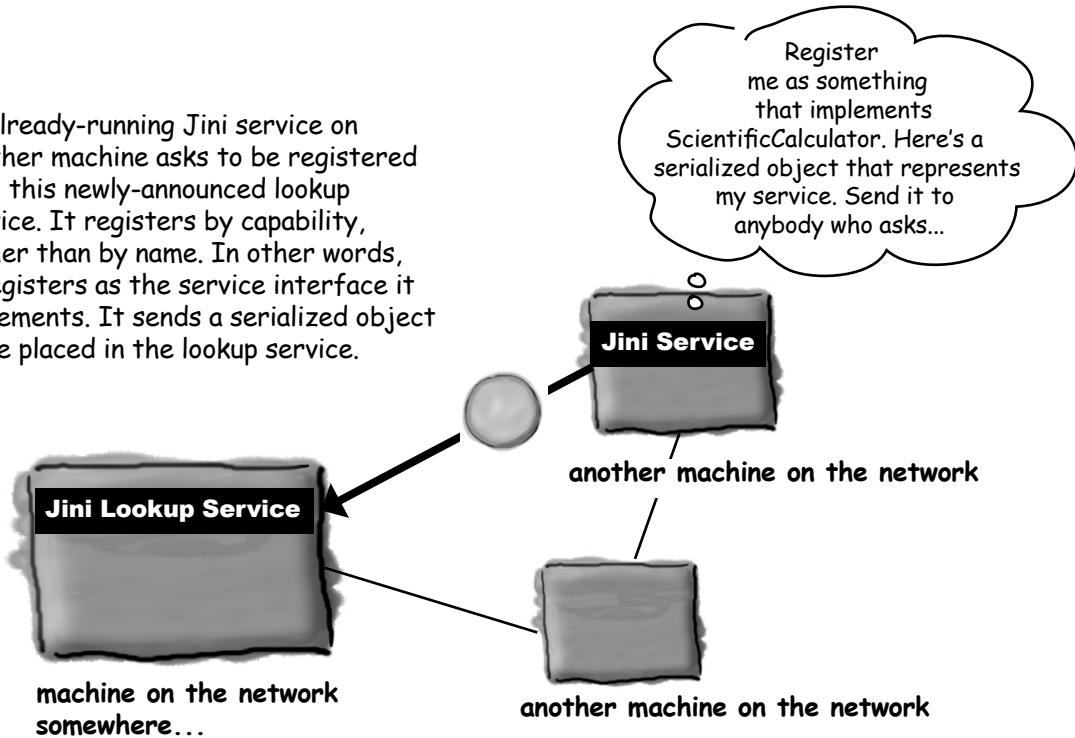


## Adaptive discovery in action

- ① Jini lookup service is launched somewhere on the network, and announces itself using IP multicast.



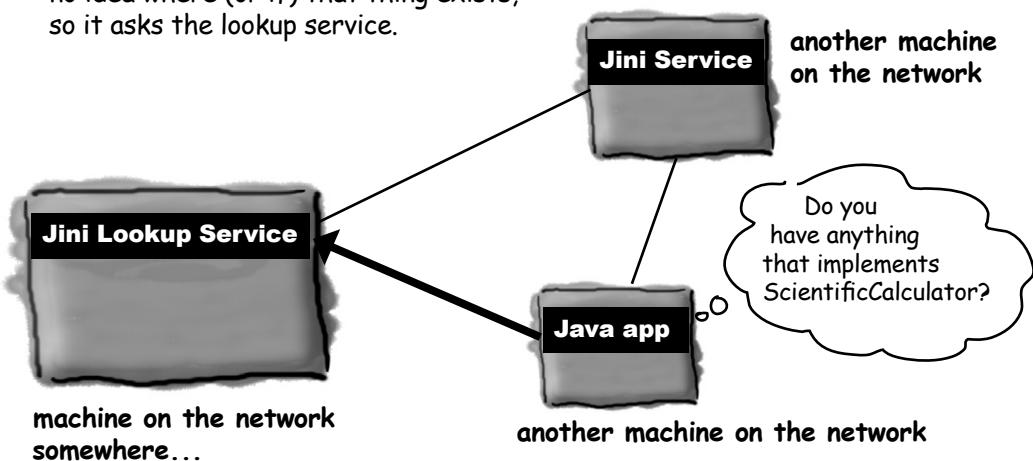
- ② An already-running Jini service on another machine asks to be registered with this newly-announced lookup service. It registers by capability, rather than by name. In other words, it registers as the service interface it implements. It sends a serialized object to be placed in the lookup service.



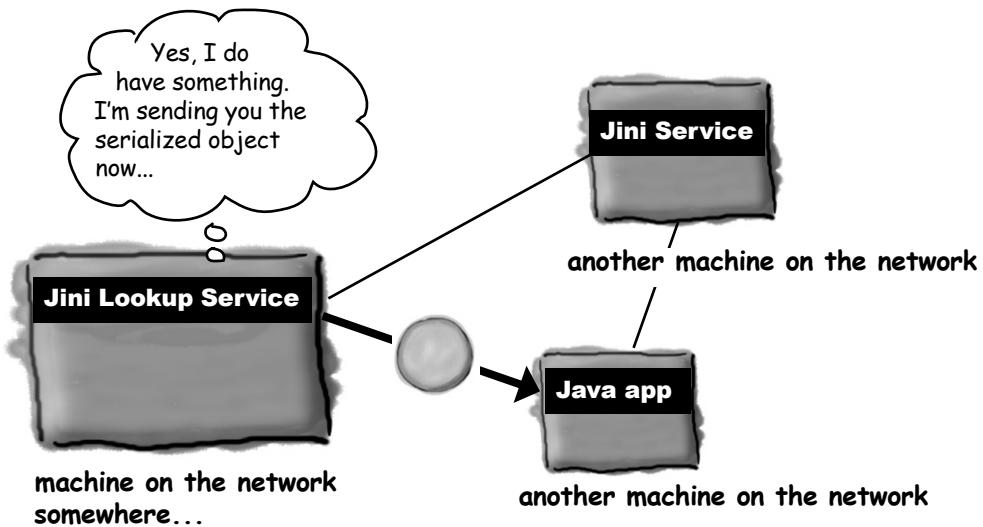
adaptive discovery in Jini

## Adaptive discovery in action, continued...

- ③ A client on the network wants something that implements the `ScientificCalculator` interface. It has no idea where (or if) that thing exists, so it asks the lookup service.

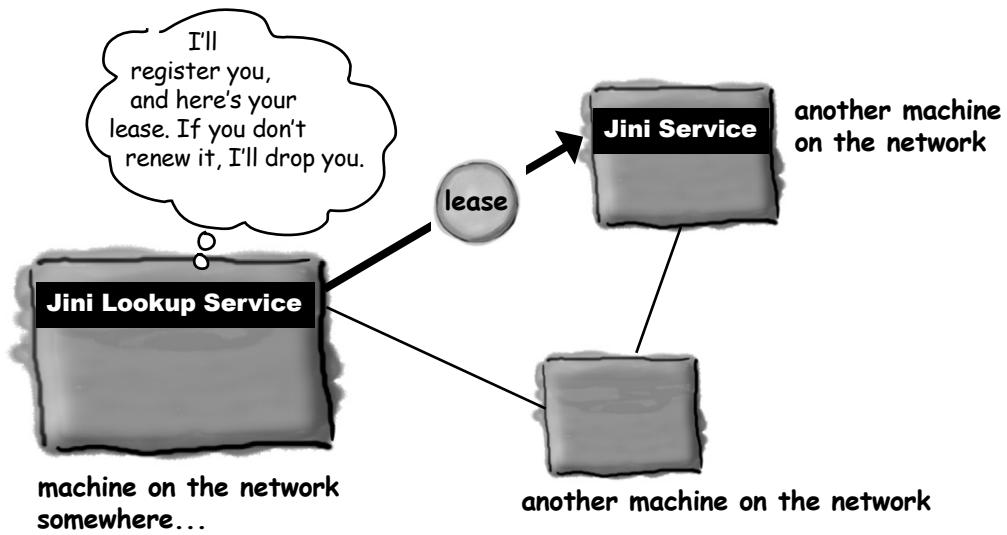


- ④ The lookup service responds, since it does have something registered as a `ScientificCalculator` interface.

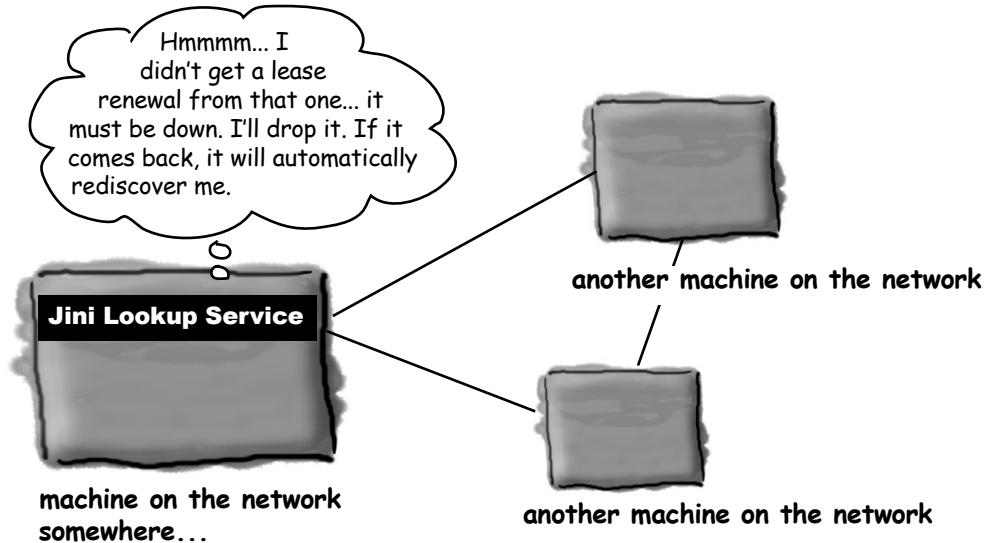


## Self-healing network in action

- ① A Jini Service has asked to register with the lookup service. The lookup service responds with a "lease". The newly-registered service must keep renewing the lease, or the lookup service assumes the service has gone offline. The lookup service wants always to present an accurate picture to the rest of the network about which services are available.



- ② The service goes offline (somebody shuts it down), so it fails to renew its lease with the lookup service. The lookup service drops it.



## universal service project

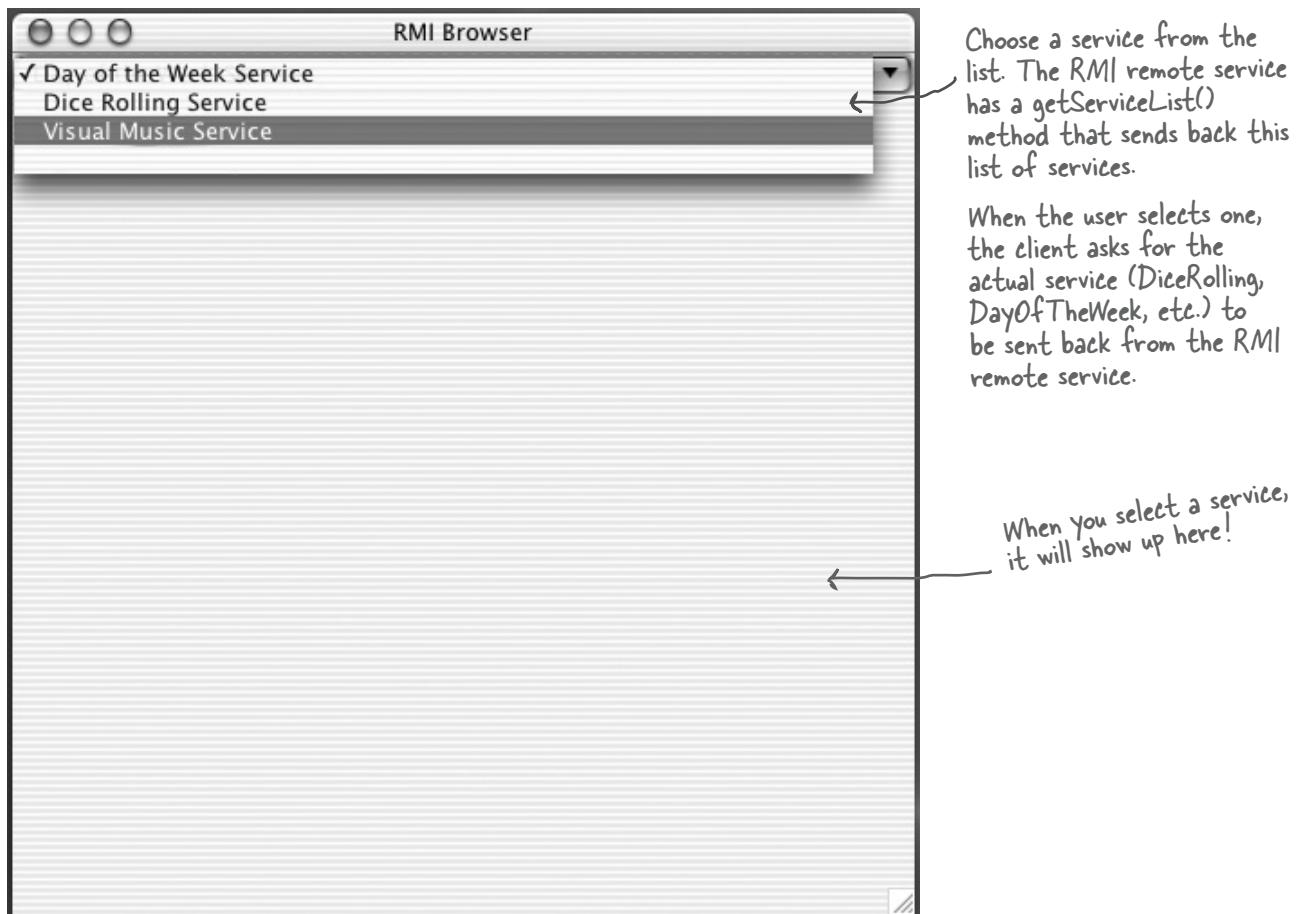
### Final Project: the Universal Service browser

We're going to make something that isn't Jini-enabled, but quite easily could be. It will give you the flavor and feeling of Jini, but using straight RMI. In fact the main difference between our application and a Jini application is how the service is discovered. Instead of the Jini lookup service, which automatically announces itself and lives anywhere on the network, we're using the RMI registry which must be on the same machine as the remote service, and which does not announce itself automatically.

And instead of our service registering itself automatically with the lookup service, *we* have to register it in the RMI registry (using `Naming.rebind()`).

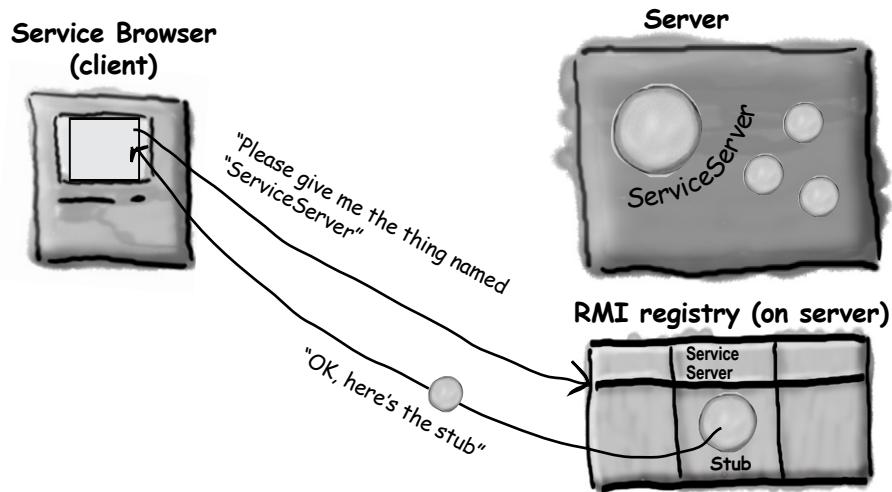
But once the client has found the service in the RMI registry, the rest of the application is almost identical to the way we'd do it in Jini. (The main thing missing is the *lease* that would let us have a self-healing network if any of the services go down.)

The universal service browser is like a specialized web browser, except instead of HTML pages, the service browser downloads and displays interactive Java GUIs that we're calling *universal services*.

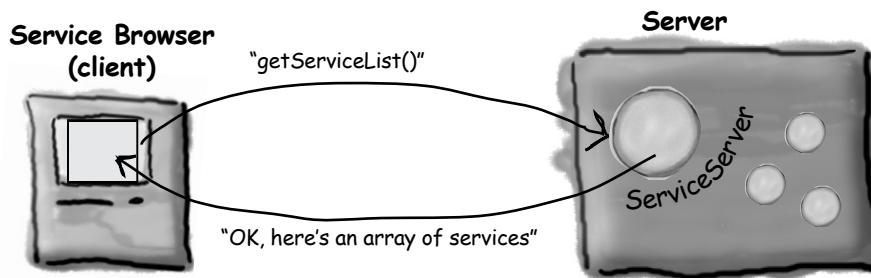


## How it works:

- ① Client starts up and does a lookup on the RMI registry for the service called "ServiceServer", and gets back the stub.



- ② Client calls `getServiceList()` on the stub. The ServiceServer returns an array of services



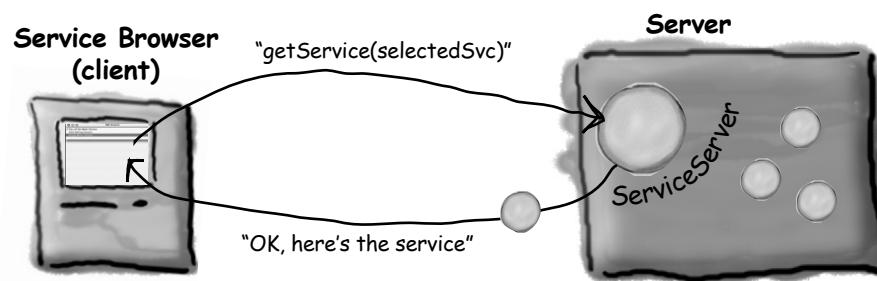
- ③ Client displays the list of services in a GUI



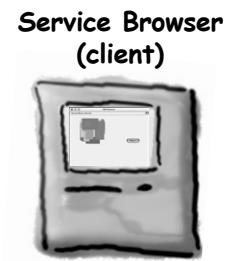
universal service browser

## How it works, continued...

- ④ User selects from the list, so client calls the `getService()` method on the remote service. The remote service returns a serialized object that is an actual service that will run inside the client browser.



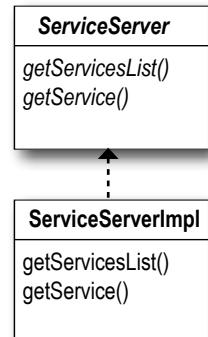
- ⑤ Client calls the `getGuiPanel()` on the serialized service object it just got from the remote service. The GUI for that service is displayed inside the browser, and the user can interact with it locally. At this point, we don't need the remote service unless/until the user decides to select another service.



## The classes and interfaces:

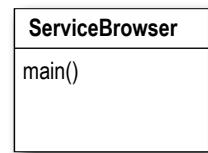
### ① interface ServiceServer implements Remote

A regular old RMI remote interface for the remote service (the remote service has the method for getting the service list and returning a selected service).



### ② class ServiceServerImpl implements ServiceServer

The actual RMI remote service (extends UnicastRemoteObject). Its job is to instantiate and store all the services (the things that will be shipped to the client), and register the server itself (ServiceServerImpl) with the RMI registry.



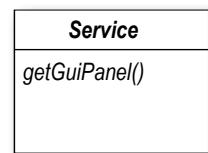
### ③ class ServiceBrowser

The client. It builds a very simple GUI, does a lookup in the RMI registry to get the ServiceServer stub, then calls a remote method on it to get the list of services to display in the GUI list.

### ④ interface Service

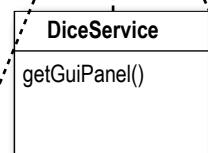
This is the key to everything. This very simple interface has just one method, getGuiPanel(). Every service that gets shipped over to the client must implement this interface. This is what makes the whole thing UNIVERSAL! By implementing this interface, a service can come over even though the client has no idea what the actual class (or classes) are that make up that service. All the client knows is that whatever comes over, it implements the Service interface, so it MUST have a getGuiPanel() method.

The client gets a serialized object as a result of calling getService(selectedSvc) on the ServiceServer stub, and all the client says to that object is, "I don't know who or what you are, but I DO know that you implement the Service interface, so I know I can call getGuiPanel() on you. And since getGuiPanel() returns a JPanel, I'll just slap it into the browser GUI and start interacting with it!"



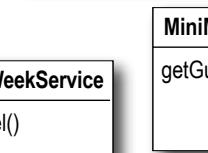
### ⑤ class DiceService implements Service

Got dice? If not, but you need some, use this service to roll anywhere from 1 to 6 virtual dice for you.



### ⑥ class MiniMusicService implements Service

Remember that fabulous little 'music video' program from the first GUI Code Kitchen? We've turned it into a service, and you can play it over and over and over until your roommates finally leave.



### ⑦ class DayOfTheWeekService implements Service

Were you born on a Friday? Type in your birthday and find out.



universal service code

### interface ServiceServer (the remote interface)

```
import java.rmi.*;
public interface ServiceServer extends Remote {
    Object[] getServiceList() throws RemoteException;
    Service getService(Object serviceKey) throws RemoteException;
}
```

A normal RMI remote interface, defines the two methods the remote service will have.

### interface Service (what the GUI services implement)

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

public interface Service extends Serializable {
    public JPanel getGuiPanel();
}
```

A plain old (i.e. non-remote) interface, that defines the one method that any universal service must have—getGuiPanel(). The interface extends Serializable, so that any class implementing the Service interface will automatically be Serializable.

That's a must, because the services get shipped over the wire from the server, as a result of the client calling getService() on the remote ServiceServer.

**class ServiceServerImpl (the remote implementation)**

```

import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class ServiceServerImpl extends UnicastRemoteObject implements ServiceServer {
    HashMap serviceList; A normal RMI implementation

    public ServiceServerImpl() throws RemoteException {
        setUpServices();
    }

    private void setUpServices() {
        serviceList = new HashMap();
        serviceList.put("Dice Rolling Service", new DiceService());
        serviceList.put("Day of the Week Service", new DayOfTheWeekService());
        serviceList.put("Visual Music Service", new MiniMusicService());
    }

    public Object[] getServiceList() {
        System.out.println("in remote");
        return serviceList.keySet().toArray();
    }

    public Service getService(Object serviceKey) throws RemoteException {
        Service theService = (Service) serviceList.get(serviceKey);
        return theService;
    }

    public static void main (String[] args) {
        try {
            Naming.rebind("ServiceServer", new ServiceServerImpl());
        } catch(Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("Remote service is running");
    }
}

```

*The services will be stored in a HashMap collection. Instead of putting ONE value object (whatever you want), you put TWO -- a key object (like a String) and a value object (like a DiceService). (see appendix B for more on HashMap)*

*When the constructor is called, initialize the actual universal services (DiceService, MiniMusicService, etc.)*

*Make the services (the actual service objects) and put them into the HashMap, with a String name (for the 'key').*

*Client calls this in order to get a list of services to display in the browser (so the user can select one). We send an array of type Object (even though it has Strings inside) by making an array of just the KEYS that are in the HashMap. We won't send an actual Service object unless the client asks for it by calling getService().*

*Client calls this method after the user selects a service from the displayed list of services (that it got from the method above). This code uses the key (the same key originally sent to the client) to get the corresponding service out of the HashMap.*

## ServiceBrowser code

### class ServiceBrowser (the client)

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

public class ServiceBrowser {

    JPanel mainPanel;
    JComboBox serviceList;
    ServiceServer server;

    public void buildGUI() {
        JFrame frame = new JFrame("RMI Browser");
        mainPanel = new JPanel();
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);

        Object[] services = getServicesList(); ← this method does the RMI registry lookup,
        serviceList = new JComboBox(services); ← gets the stub, and calls getServiceList().
                                                (The actual method is on the next page).
                                                Add the services (an array of Objects) to the
                                                JComboBox (the list). The JComboBox knows how to
                                                make displayable Strings out of each thing in the array.

        frame.getContentPane().add(BorderLayout.NORTH, serviceList);
        serviceList.addActionListener(new myListListener());

        frame.setSize(500,500);
        frame.setVisible(true);
    }

    void loadService(Object serviceSelection) {
        try {
            Service svc = server.getService(serviceSelection);

            mainPanel.removeAll();
            mainPanel.add(svc.getGuiPanel());
            mainPanel.validate();
            mainPanel.repaint();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Here's where we add the actual service to the GUI, after the user has selected one. (This method is called by the event listener on the JComboBox). We call getService() on the remote server (the stub for ServiceServer) and pass it the String that was displayed in the list (which is the SAME String we originally got from the server when we called getServiceList()). The server returns the actual service (serialized), which is automatically deserialized (thanks to RMI) and we simply call the getGuiPanel() on the service and add the result (a JPanel) to the browser's mainPanel.

```

Object[] getServicesList() {
    Object obj = null;
    Object[] services = null;

    try {
        obj = Naming.lookup("rmi://127.0.0.1/ServiceServer");
    }

    catch (Exception ex) {
        ex.printStackTrace();
    }
    server = (ServiceServer) obj;
}

try {
    services = server.getServiceList();
}

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

class MyListListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

        Object selection = serviceList.getSelectedItem();
        loadService(selection);
    }
}

public static void main(String[] args) {
    new ServiceBrowser().buildGUI();
}
}

```

*Do the RMI lookup, and get the stub*

*Cast the stub to the remote interface type, so that we can call getServiceList() on it*

*getServiceList() gives us the array of Objects, that we display in the JComboBox for the user to select from.*

*If we're here, it means the user made a selection from the JComboBox list. So, take the selection they made and load the appropriate service. (see the loadService method on the previous page, that asks the server for the service that corresponds with this selection)*

## DiceService code

### class DiceService (a universal service, implements Service)

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

public class DiceService implements Service {

    JLabel label;
    JComboBox numOfDice;

    public JPanel getGuiPanel() {
        JPanel panel = new JPanel();
        JButton button = new JButton("Roll 'em!");
        String[] choices = {"1", "2", "3", "4", "5"};
        numOfDice = new JComboBox(choices);
        label = new JLabel("dice values here");
        button.addActionListener(new RollEmListener());
        panel.add(numOfDice);
        panel.add(button);
        panel.add(label);
        return panel;
    }

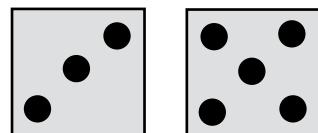
    public class RollEmListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            // roll the dice
            String diceOutput = "";
            String selection = (String) numOfDice.getSelectedItem();
            int numOfDiceToRoll = Integer.parseInt(selection);
            for (int i = 0; i < numOfDiceToRoll; i++) {
                int r = (int) ((Math.random() * 6) + 1);
                diceOutput += (" " + r);
            }
            label.setText(diceOutput);
        }
    }
}
```



Here's the one important method! The method of the Service interface-- the one the client's gonna call when this service is selected and loaded. You can do whatever you want in the getGuiPanel() method, as long as you return a JPanel, so it builds the actual dice-rolling GUI.

## Sharpen your pencil

Think about ways to improve the DiceService. One suggestion: using what you learned in the GUI chapters, make the dice graphical. Use a rectangle, and draw the appropriate number of circles on each one, corresponding to the roll for that particular die.



**class MiniMusicService (a universal service, implements Service)**

```

import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusicService implements Service {
    MyDrawPanel myPanel;

    public JPanel getGuiPanel() {
        JPanel mainPanel = new JPanel();
        myPanel = new MyDrawPanel();
        JButton playItButton = new JButton("Play it");
        playItButton.addActionListener(new PlayItListener());
        mainPanel.add(myPanel);
        mainPanel.add(playItButton);
        return mainPanel;
    }

    public class PlayItListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            try {
                Sequencer sequencer = MidiSystem.getSequencer();
                sequencer.open();

                sequencer.addControllerEventListener(myPanel, new int[] {127});
                Sequence seq = new Sequence(Sequence.PPQ, 4);
                Track track = seq.createTrack();

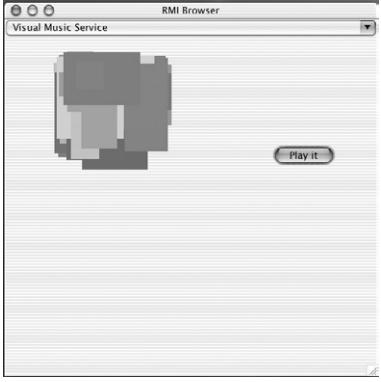
                for (int i = 0; i < 100; i += 4) {

                    int rNum = (int) ((Math.random() * 50) + 1);
                    if (rNum < 38) { // so now only do it if num < 38 (75% of the time)
                        track.add(makeEvent(144, 1, rNum, 100, i));
                        track.add(makeEvent(176, 1, 127, 0, i));
                        track.add(makeEvent(128, 1, rNum, 100, i + 2));
                    }
                } // end loop

                sequencer.setSequence(seq);
                sequencer.start();
                sequencer.setTempoInBPM(220);
            } catch (Exception ex) {ex.printStackTrace();}
        }
    } // close actionPerformed
} // close inner class

```

The service method! All it does is display a button and the drawing service (where the rectangles will eventually be painted).



This is all the music stuff from the Code Kitchen in chapter 12, so we won't annotate it again here.

## MiniMusicService code

### class MiniMusicService, continued...

```
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;
}
```

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

    // only if we got an event do we want to paint
    boolean msg = false;

    public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
    }

    public Dimension getPreferredSize() {
        return new Dimension(300,300);
    }

    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
} // close class
```

Nothing new on this entire page. You've seen it all in the graphics CodeKitchen. If you want another exercise, try annotating this code yourself, then compare it with the CodeKitchen in the "A very graphic story" chapter.

**class DayOfTheWeekService (a universal service, implements Service)**

```

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

public class DayOfTheWeekService implements Service {

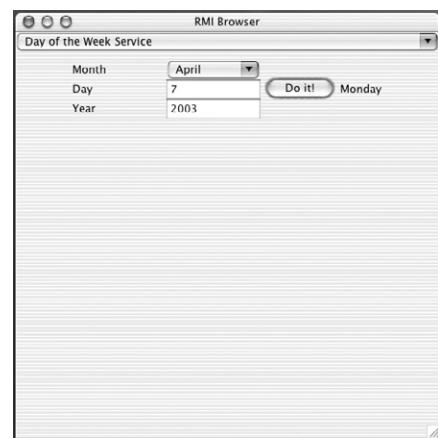
    JLabel outputLabel;
    JComboBox month;
    JTextField day;
    JTextField year;

    public JPanel getGuiPanel() {
        JPanel panel = new JPanel();
        JButton button = new JButton("Do it!");
        button.addActionListener(new DoItListener());
        outputLabel = new JLabel("date appears here");
        DateFormatSymbols dateStuff = new DateFormatSymbols();
        month = new JComboBox(dateStuff.getMonths());
        day = new JTextField(8);
        year = new JTextField(8);
        JPanel inputPanel = new JPanel(new GridLayout(3,2));
        inputPanel.add(new JLabel("Month"));
        inputPanel.add(month);
        inputPanel.add(new JLabel("Day"));
        inputPanel.add(day);
        inputPanel.add(new JLabel("Year"));
        inputPanel.add(year);
        panel.add(inputPanel);
        panel.add(button);
        panel.add(outputLabel);
        return panel;
    }

    public class DoItListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            int monthNum = month.getSelectedIndex();
            int dayNum = Integer.parseInt(day.getText());
            int yearNum = Integer.parseInt(year.getText());
            Calendar c = Calendar.getInstance();
            c.set(Calendar.MONTH, monthNum);
            c.set(Calendar.DAY_OF_MONTH, dayNum);
            c.set(Calendar.YEAR, yearNum);
            Date date = c.getTime();
            String dayOfWeek = (new SimpleDateFormat("EEEE")).format(date);
            outputLabel.setText(dayOfWeek);
        }
    }
}

```

The Service interface method  
that builds the GUI



Refer to chapter 10 if you need a reminder of how number and date formatting works. This code is slightly different, however, because it uses the Calendar class. Also, the SimpleDateFormat lets us specify a pattern for how the date should print out.

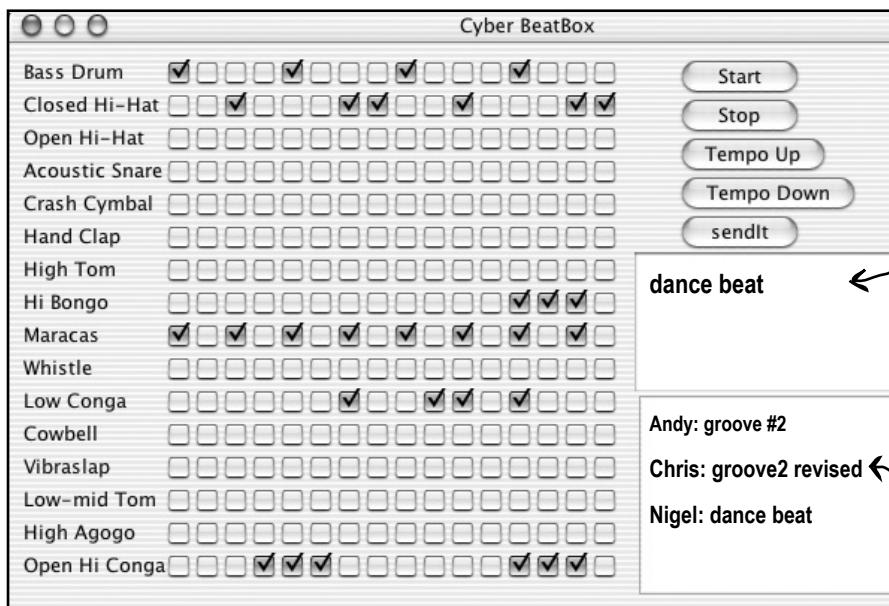
the end... sort of



**Congratulations!**  
**You made it to the end.**

**Of course, there's still the two appendices.  
And the index.  
And then there's the web site...  
There's no escape, really.**

## Appendix A: Final Code Kitchen



Your message gets sent to the other players, along with your current beat pattern, when you hit "sendit".

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

**Finally, the complete version of the BeatBox!**

**It connects to a simple MusicServer so that you can send and receive beat patterns with other clients.**

**final BeatBox code**

## Final BeatBox client program

Most of this code is the same as the code from the CodeKitchens in the previous chapters, so we don't annotate the whole thing again. The new parts include:

GUI - two new components are added for the text area that displays incoming messages (actually a scrolling list) and the text field.

NETWORKING - just like the SimpleChatClient in this chapter, the BeatBox now connects to the server and gets an input and output stream.

THREADS - again, just like the SimpleChatClient, we start a 'reader' class that keeps looking for incoming messages from the server. But instead of just text, the messages coming in include TWO objects: the String message and the serialized ArrayList (the thing that holds the state of all the checkboxes.)

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

public class BeatBoxFinal {

    JFrame theFrame;
    JPanel mainPanel;
    JList incomingList;
    JTextField userMessage;
    ArrayList<JCheckBox> checkboxList;
    int nextNum;
    Vector<String> listVector = new Vector<String>();
    String userName;
    ObjectOutputStream out;
    ObjectInputStream in;
    HashMap<String, boolean[]> otherSeqsMap = new HashMap<String, boolean[]>();

    Sequencer sequencer;
    Sequence sequence;
    Sequence mySequence = null;
    Track track;

    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};

}
```

## appendix A Final Code Kitchen

```

public static void main (String[] args) {
    new BeatBoxFinal().startUp(args[0]); // args[0] is your user ID/screen name
}

public void startUp(String name) {
    userName = name;
    // open connection to the server
    try {
        Socket sock = new Socket("127.0.0.1", 4242);
        out = new ObjectOutputStream(sock.getOutputStream());
        in = new ObjectInputStream(sock.getInputStream());
        Thread remote = new Thread(new RemoteReader());
        remote.start();
    } catch(Exception ex) {
        System.out.println("couldn't connect - you'll have to play alone.");
    }
    setUpMidi();
    buildGUI();
} // close startUp

public void buildGUI() {                                     GUI code, nothing new here

    theFrame = new JFrame("Cyber BeatBox");
    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");
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    JButton sendIt = new JButton("sendIt");
    sendIt.addActionListener(new MySendListener());
    buttonBox.add(sendIt);

    userMessage = new JTextField();
}

```

Add a command-line argument for your screen name.  
Example: % java BeatBoxFinal theFlash

Nothing new... set up the networking, I/O, and make (and start) the reader thread.

## final BeatBox code

```
buttonBox.add(userMessage);

incomingList = new JList();
incomingList.addListSelectionListener(new MyListSelectionListener());
incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane theList = new JScrollPane(incomingList);
buttonBox.add(theList);
incomingList.setListData(listVector); // no data to start with
```

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

theFrame.setBounds(50,50,300,300);
theFrame.pack();
theFrame.setVisible(true);
} // close buildGUI

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();}
}

Nothing else on this page is new

Get the Sequencer, make a Sequence, and make a Track

JList is a component we haven't used before. This is where the incoming messages are displayed. Only instead of a normal chat where you just LOOK at the messages, in this app you can SELECT a message from the list to load and play the attached beat pattern.



```

public void buildTrackAndStart() {
    ArrayList<Integer> trackList = null; // this will hold the instruments for each
    sequence.deleteTrack(track);
    track = sequence.createTrack();
    for (int i = 0; i < 16; i++) {
        trackList = new ArrayList<Integer>();
        for (int j = 0; j < 16; j++) {
            JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
            if (jc.isSelected()) {
                int key = instruments[i];
                trackList.add(new Integer(key));
            } else {
                trackList.add(null); // because this slot should be empty in the track
            }
        } // close inner loop
        makeTracks(trackList);
    } // close outer loop
    track.add(makeEvent(192, 9, 1, 0, 15)); // - so we always go to full 16 beats
    try {
        sequencer.setSequence(sequence);
        sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
        sequencer.start();
        sequencer.setTempoInBPM(120);
    } catch (Exception e) {e.printStackTrace();}
} // close method

public class MyStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        buildTrackAndStart();
    } // close actionPerformed
} // close inner class

public class MyStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequencer.stop();
    } // close actionPerformed
} // close inner class

public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    } // close actionPerformed
} // close inner class

```

Build a track by walking through the checkboxes to get their state, and mapping that to an instrument (and making the MidiEvent for it). This is pretty complex, but it is EXACTLY as it was in the previous chapters, so refer to previous CodeKitchens to get the full explanation again.

The GUI listeners.  
Exactly the same as the previous chapter's version.

## final BeatBox code

```
public class MyDownTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
}

public class MySendListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        // make an arraylist of just the STATE of the checkboxes
        boolean[] checkboxState = new boolean[256];
        for (int i = 0; i < 256; i++) {
            JCheckBox check = (JCheckBox) checkboxList.get(i);
            if (check.isSelected()) {
                checkboxState[i] = true;
            }
        } // close loop
        String messageToSend = null;
        try {
            out.writeObject(userName + nextNum++ + ":" + userMessage.getText());
            out.writeObject(checkboxState);
        } catch (Exception ex) {
            System.out.println("Sorry dude. Could not send it to the server.");
        }
        userMessage.setText("");
    } // close actionPerformed
} // close inner class

public class MyListSelectionListener implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent le) {
        if (!le.getValueIsAdjusting()) {
            String selected = (String) incomingList.getSelectedValue();
            if (selected != null) {
                // now go to the map, and change the sequence
                boolean[] selectedState = (boolean[]) otherSeqsMap.get(selected);
                changeSequence(selectedState);
                sequencer.stop();
                buildTrackAndStart();
            }
        }
    }
} // close valueChanged
} // close inner class
```

This is new... it's a lot like the SimpleChatClient, except instead of sending a String message, we serialize two objects (the String message and the beat pattern) and write those two objects to the socket output stream (to the server).

This is also new -- a ListSelectionListener that tells us when the user made a selection on the list of messages. When the user selects a message, we IMMEDIATELY load the associated beat pattern (it's in the HashMap called otherSeqsMap) and start playing it. There's some if tests because of little quirky things about getting ListSelectionEvents.

## appendix A Final Code Kitchen

```

public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;
    public void run() {
        try {
            while((obj=in.readObject()) != null) {
                System.out.println("got an object from server");
                System.out.println(obj.getClass());
                String nameToShow = (String) obj;
                checkboxState = (boolean[]) in.readObject();
                otherSeqsMap.put(nameToShow, checkboxState);
                listVector.add(nameToShow);
                incomingList.setListData(listVector);
            } // close while
        } catch(Exception ex) {ex.printStackTrace();}
    } // close run
} // close inner class

```

This is the thread job -- read in data from the server. In this code, 'data' will always be two serialized objects: the String message and the beat pattern (an ArrayList of checkbox state values)

```

public class MyPlayMineListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        if (mySequence != null) {
            sequence = mySequence; // restore to my original
        }
    } // close actionPerformed
} // close inner class

```

When a message comes in, we read (deserialize) the two objects (the message and the ArrayList of Boolean checkbox state values) and add it to the JList component. Adding to a JList is a two-step thing: you keep a Vector of the lists data (Vector is an old-fashioned ArrayList), and then tell the JList to use that Vector as its source for what to display in the list.

```

public void changeSequence(boolean[] checkboxState) {
    for (int i = 0; i < 256; i++) {
        JCheckBox check = (JCheckBox) checkboxList.get(i);
        if (checkboxState[i]) {
            check.setSelected(true);
        } else {
            check.setSelected(false);
        }
    } // close loop
} // close changeSequence

```

This method is called when the user selects something from the list. We IMMEDIATELY change the pattern to the one they selected.

```

public void makeTracks(ArrayList list) {
    Iterator it = list.iterator();
    for (int i = 0; i < 16; i++) {
        Integer num = (Integer) it.next();
        if (num != null) {
            int numKey = num.intValue();
            track.add(makeEvent(144,9,numKey, 100, i));
            track.add(makeEvent(128,9,numKey,100, i + 1));
        }
    } // close loop
} // close makeTracks()

```

All the MIDI stuff is exactly the same as it was in the previous version.

## final BeatBox code

```
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 makeEvent  
  
} // close class
```

Nothing new. Just like the last version.



What are some of the ways you can improve this program?

Here are a few ideas to get you started:

- 1) Once you select a pattern, whatever current pattern was playing is blown away. If that was a new pattern you were working on (or a modification of another one), you're out of luck. You might want to pop up a dialog box that asks the user if he'd like to save the current pattern.
- 2) If you fail to type in a command-line argument, you just get an exception when you run it! Put something in the main method that checks to see if you've passed in a command-line argument. If the user doesn't supply one, either pick a default or print out a message that says they need to run it again, but this time with an argument for their screen name.
- 3) It might be nice to have a feature where you can click a button and it will generate a random pattern for you. You might hit on one you really like. Better yet, have another feature that lets you load in existing 'foundation' patterns, like one for jazz, rock, reggae, etc. that the user can add to.

You can find existing patterns on the Head First Java web start.

## Final BeatBox server program

Most of this code is identical to the SimpleChatServer we made in the Networking and Threads chapter. The only difference, in fact, is that this server receives, and then re-sends, two serialized objects instead of a plain String (although one of the serialized objects happens to be a String).

```

import java.io.*;
import java.net.*;
import java.util.*;

public class MusicServer {

    ArrayList<ObjectOutputStream> clientOutputStreams;

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

    public class ClientHandler implements Runnable {

        ObjectInputStream in;
        Socket clientSocket;

        public ClientHandler(Socket socket) {
            try {
                clientSocket = socket;
                in = new ObjectInputStream(clientSocket.getInputStream());
            } catch (Exception ex) {ex.printStackTrace();}
        } // close constructor

        public void run() {
            Object o2 = null;
            Object o1 = null;
            try {
                while ((o1 = in.readObject()) != null) {
                    o2 = in.readObject();

                    System.out.println("read two objects");
                    tellEveryone(o1, o2);
                } // close while
            } catch (Exception ex) {ex.printStackTrace();}
        } // close run
    } // close inner class
}

```

**final BeatBox code**

```
public void go() {
    clientOutputStreams = new ArrayList<ObjectOutputStream>();

    try {
        ServerSocket serverSock = new ServerSocket(4242);

        while(true) {
            Socket clientSocket = serverSock.accept();
            ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
            clientOutputStreams.add(out);

            Thread t = new Thread(new ClientHandler(clientSocket));
            t.start();

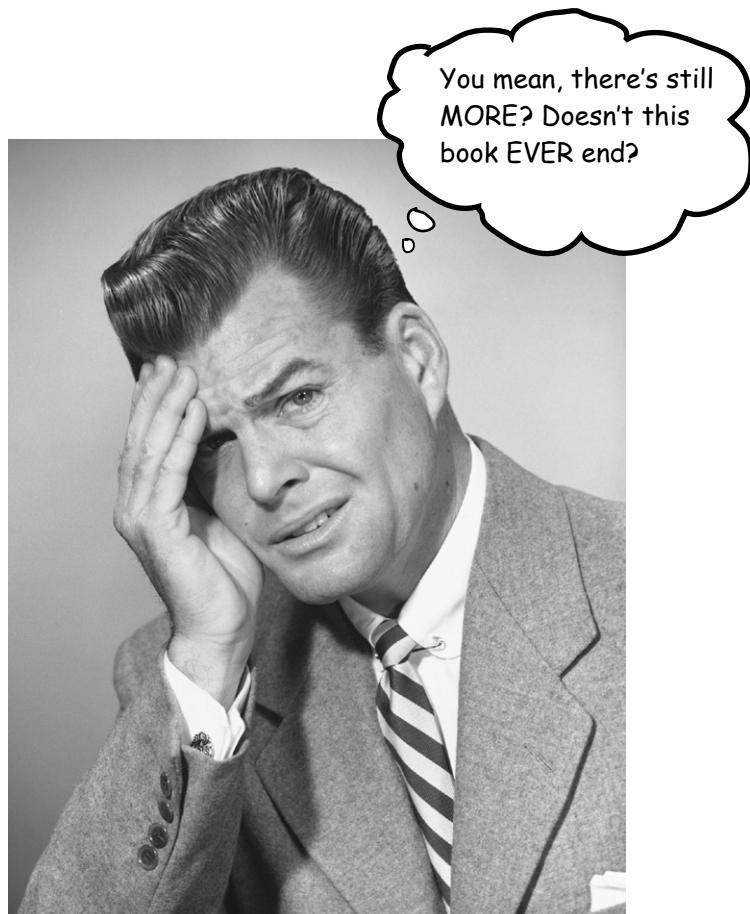
            System.out.println("got a connection");
        }
    } catch(Exception ex) {
        ex.printStackTrace();
    }
} // close go

public void tellEveryone(Object one, Object two) {
    Iterator it = clientOutputStreams.iterator();
    while(it.hasNext()) {
        try {
            ObjectOutputStream out = (ObjectOutputStream) it.next();
            out.writeObject(one);
            out.writeObject(two);
        } catch(Exception ex) {ex.printStackTrace();}
    }
} // close tellEveryone

} // close class
```

# Appendix B

## The Top Ten Topics that almost made it into the Real Book...



We covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into JavaLand without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small appendix. Actually, we *did* originally include everything you need to know about Java (not already covered by the other chapters), by reducing the type point size to .00003. It all fit, but nobody could read it. So, we threw most of it away, but kept the best bits for this Top Ten appendix.

This really *is* the end of the book. Except for the index (a must-read!).

## bit manipulation

# #10 Bit Manipulation

## Why do you care?

We've talked about the fact that there are 8 bits in a byte, 16 bits in a short, and so on. You might have occasion to turn individual bits on or off. For instance you might find yourself writing code for your new Java enabled toaster, and realize that due to severe memory limitations, certain toaster settings are controlled at the bit level. For easier reading, we're showing only the last 8 bits in the comments rather than the full 32 for an int).

### Bitwise NOT Operator: ~

This operator 'flips all the bits' of a primitive.

```
int x = 10; // bits are 00001010
x = ~x; // bits are now 11110101
```

The next three operators compare two primitives on a bit by bit basis, and return a result based on comparing these bits. We'll use the following example for the next three operators:

```
int x = 10; // bits are 00001010
int y = 6; // bits are 00000110
```

### Bitwise AND Operator: &

This operator returns a value whose bits are turned on only if *both* original bits are turned on:

```
int a = x & y; // bits are 00000010
```

### Bitwise OR Operator: |

This operator returns a value whose bits are turned on only if *either* of the original bits are turned on:

```
int a = x | y; // bits are 00001110
```

### Bitwise XOR (exclusive OR) Operator: ^

This operator returns a value whose bits are turned on only if *exactly one* of the original bits are turned on:

```
int a = x ^ y; // bits are 00001100
```

## The Shift Operators

These operators take a single integer primitive and shift (or slide) all of its bits in one direction or another. If you want to dust off your binary math skills, you might realize that shifting bits *left* effectively *multiplies* a number by a power of two, and shifting bits *right* effectively *divides* a number by a power of two.

We'll use the following example for the next three operators:

```
int x = -11; // bits are 11110101
```

Ok, ok, we've been putting it off, here is the world's shortest explanation of storing negative numbers, and *two's complement*. Remember, the leftmost bit of an integer number is called the *sign bit*. A negative integer number in Java *always* has its sign bit turned *on* (i.e. set to 1). A positive integer number always has its sign bit turned *off* (0). Java uses the *two's complement* formula to store negative numbers. To change a number's sign using two's complement, flip all the bits, then add 1 (with a byte, for example, that would mean adding 00000001 to the flipped value).

### Right Shift Operator: >>

This operator shifts all of a number's bits right by a certain number, and fills all of the bits on the left side with whatever the original leftmost bit was. **The sign bit does *not* change**:

```
int y = x >> 2; // bits are 11111101
```

### Unsigned Right Shift Operator: >>>

Just like the right shift operator BUT it ALWAYS fills the leftmost bits with zeros. **The sign bit *might* change**:

```
int y = x >>> 2; // bits are 00111101
```

### Left Shift Operator: <<

Just like the unsigned right shift operator, but in the other direction; the rightmost bits are filled with zeros. **The sign bit *might* change**.

```
int y = x << 2; // bits are 11010100
```

## #9 Immutability

### Why do you care that Strings are immutable?

When your Java programs start to get big, you'll inevitably end up with lots and lots of String objects. For security purposes, and for the sake of conserving memory (remember your Java programs can run on teeny Java-enabled cell phones), Strings in Java are immutable. What this means is that when you say:

```
String s = "0";
for (int x = 1; x < 10; x++) {
    s = s + x;
}
```

What's actually happening is that you're creating ten String objects (with values "0", "01", "012", through "0123456789"). In the end *s* is referring to the String with the value "0123456789", but at this point there are *ten* Strings in existence!

Whenever you make a new String, the JVM puts it into a special part of memory called the 'String Pool' (sounds refreshing doesn't it?). If there is already a String in the String Pool with the same value, the JVM doesn't create a duplicate, it simply refers your reference variable to the existing entry. The JVM can get away with this because Strings are immutable; one reference variable can't change a String's value out from under another reference variable referring to the same String.

The other issue with the String pool is that the Garbage Collector *doesn't go there*. So in our example, unless by coincidence you later happen to make a String called "01234", for instance, the first nine Strings created in our *for* loop will just sit around wasting memory.

### How does this save memory?

Well, if you're not careful, *it doesn't!* But if you understand how String immutability works, than you can sometimes take advantage of it to save memory. If you have to do a lot of String manipulations (like concatenations, etc.), however, there is another class StringBuilder, better suited for that purpose. We'll talk more about StringBuilder in a few pages.

### Why do you care that Wrappers are immutable?

In the Math chapter we talked about the two main uses of the wrapper classes:

- Wrapping a primitive so it can pretend to be an object.
- Using the static utility methods (for example, Integer.parseInt()).

It's important to remember that when you create a wrapper object like:

```
Integer iWrap = new Integer(42);
```

That's it for that wrapper object. Its value will *always* be 42. *There is no setter method for a wrapper object.* You can, of course, refer *iWrap* to a *different* wrapper object, but then you'll have *two* objects. Once you create a wrapper object, there's no way to change the *value* of that object!



## assertions

# #8 Assertions

We haven't talked much about how to debug your Java program while you're developing it. We believe that you should learn Java at the command line, as we've been doing throughout the book. Once you're a Java pro, if you decide to use an IDE\*, you might have other debugging tools to use. In the old days, when a Java programmer wanted to debug her code, she'd stick a bunch of `System.out.println()` statements throughout the program, printing current variable values, and "I got here" messages, to see if the flow control was working properly. (The ready-bake code in chapter 6 left some debugging 'print' statements in the code.) Then, once the program was working correctly, she'd go through and take all those `System.out.println()` statements back out again. It was tedious and error prone. But as of Java 1.4 (and 5.0), debugging got a whole lot easier. The answer?

## Assertions

Assertions are like `System.out.println()` statements on steroids. Add them to your code as you would add `println` statements. The Java 5.0 compiler assumes you'll be compiling source files that are 5.0 compatible, so as of Java 5.0, compiling with assertions is enabled by default.

At runtime, if you do nothing, the assert statements you added to your code will be ignored by the JVM, and won't slow down your program. But if you tell the JVM to *enable* your assertions, they will help you do your debugging, without changing a line of code!

Some folks have complained about having to leave assert statements in their production code, but leaving them in can be really valuable when your code is already deployed in the field. If your client is having trouble, you can instruct the client to run the program with assertions enabled, and have the client send you the output. If the assertions were stripped out of your deployed code, you'd never have that option. And there is almost no downside; when assertions are not enabled, they are completely ignored by the JVM, so there's no performance hit to worry about.

## How to make Assertions work

Add assertion statements to your code wherever you believe that something *must be true*. For instance:

```
assert (height > 0);  
  
// if true, program continues normally  
// if false, throw an AssertionError
```

You can add a little more information to the stack trace by saying:

```
assert (height > 0) : "height = " +  
height + " weight = " + weight;
```

The expression after the colon can be any legal Java expression *that resolves to a non-null value*. But whatever you do, *don't create assertions that change an object's state!* If you do, enabling assertions at runtime might change how your program performs.

## Compiling and running with Assertions

To *compile* with assertions:

```
javac TestDriveGame.java
```

(Notice that no command line options were necessary.)

To *run* with assertions:

```
java -ea TestDriveGame
```

\* IDE stands for Integrated Development Environment and includes tools such as Eclipse, Borland's JBuilder, or the open source NetBeans ([netbeans.org](http://netbeans.org)).

## #7 Block Scope

In chapter 9, we talked about how local variables live only as long as the method in which they're declared stays on the stack. But some variables can have even *shorter* lifespans. Inside of methods, we often create *blocks* of code. We've been doing this all along, but we haven't explicitly *talked* in terms of *blocks*. Typically, blocks of code occur within methods, and are bounded by curly braces {}. Some common examples of code blocks that you'll recognize include loops (*for*, *while*) and conditional expressions (like *if* statements).

Let's look at an example:

```
void doStuff() { ← start of the method block
    int x = 0; ← local variable scoped to the entire method
    for(int y = 0; y < 5; y++) { ← beginning of a for loop block, and y is
        x = x + y; ← No problem, x and y are both in scope
    } ← end of the for loop block
    x = x * y; ← Aack! Won't compile! y is out of scope here! (this is not
} ← end of the method block, now x is also out of scope
      the way it works in some other languages, so beware!
```

In the previous example, *y* was a block variable, declared inside a block, and *y* went out of scope as soon as the for loop ended. Your Java programs will be more debuggable and expandable if you use local variables instead of instance variables, and block variables instead of local variables, whenever possible. The compiler will make sure that you don't try to use a variable that's gone out of scope, so you don't have to worry about runtime meltdowns.

## linked invocations

# #6 Linked Invocations

While you did see a little of this in this book, we tried to keep our syntax as clean and readable as possible. There are, however, many legal shortcuts in Java, that you'll no doubt be exposed to, especially if you have to read a lot code you didn't write. One of the more common constructs you will encounter is known as *linked invocations*. For example:

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2,"umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// result is sb = summer
```

What in the world is happening in the second line of code? Admittedly, this is a contrived example, but you need to learn how to decipher these.

1 - Work from left to right.

2 - Find the result of the leftmost method call, in this case `sb.delete(3, 6)`. If you look up `StringBuffer` in the API docs, you'll see that the `delete()` method returns a `StringBuffer` object. The result of running the `delete()` method is a `StringBuffer` object with the value "spr".

3 - The next leftmost method (`insert()`) is called on the newly created `StringBuffer` object "spr". The result of that method call (the `insert()` method), is *also* a `StringBuffer` object (although it doesn't have to be the same type as the previous method return), and so it goes, the returned object is used to call the next method to the right. In theory, you can link as many methods as you want in a single statement (although it's rare to see more than three linked methods in a single statement). Without linking, the second line of code from above would be more readable, and look something like this:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"umme");
sb = sb.deleteCharAt(1);
```

But here's a more common, and useful example, that you saw us using, but we thought we'd point it out again here. This is for when your `main()` method needs to invoke an instance method of the main class, but you don't need to keep a *reference* to the instance of the class. In other words, the `main()` needs to create the instance *only* so that `main()` can invoke one of the instance's *methods*.

```
class Foo {
    public static void main(String [] args) [
        new Foo().go(); ← we want to call go(), but we don't care about
    }                               the Foo instance, so we don't bother assigning
    void go() {                     the new Foo object to a reference.
        // here's what we REALLY want...
    }
}
```

## #5 Anonymous and Static Nested Classes

### Nested classes come in many flavors

In the GUI event-handling section of the book, we started using inner (nested) classes as a solution for implementing listener interfaces. That's the most common, practical, and readable form of an inner class—where the class is simply nested within the curly braces of another enclosing class. And remember, it means you need an instance of the outer class in order to get an instance of the inner class, because the inner class is a *member* of the outer/enclosing class.

But there are other kinds of inner classes including *static* and *anonymous*. We're not going into the details here, but we don't want you to be thrown by strange syntax when you see it in someone's code. Because out of virtually anything you can do with the Java language, perhaps nothing produces more bizarre-looking code than anonymous inner classes. But we'll start with something simpler—static nested classes.

#### Static nested classes

You already know what static means—something tied to the class, not a particular instance. A static nested class looks just like the non-static classes we used for event listeners, except they're marked with the keyword *static*.

```
public class FooOuter {  
    static class BarInner {  
        void sayIt() {  
            System.out.println("method of a static inner class");  
        }  
    }  
}  
  
class Test {  
    public static void main (String[] args) {  
        FooOuter.BarInner foo = new FooOuter.BarInner();  
        foo.sayIt();  
    }  
}
```

A static nested class is just that—a class enclosed within another, and marked with the static modifier.

Because a static nested class is...static, you don't use an instance of the outer class. You just use the name of the class, the same way you invoke static methods or access static variables.

Static nested classes are more like regular non-nested classes in that they don't enjoy a special relationship with an enclosing outer object. But because static nested classes are still considered a *member* of the enclosing/outer class, they still get access to any private members of the outer class... but *only the ones that are also static*. Since the static nested class isn't connected to an instance of the outer class, it doesn't have any special way to access the non-static (instance) variables and methods.

when arrays aren't enough

## #5 Anonymous and Static Nested Classes, continued

### The difference between *nested* and *inner*

Any Java class that's defined within the scope of another class is known as a *nested* class. It doesn't matter if it's anonymous, static, normal, whatever. If it's inside another class, it's technically considered a *nested* class. But *non-static* nested classes are often referred to as *inner* classes, which is what we called them earlier in the book. The bottom line: all inner classes are nested classes, but not all nested classes are inner classes.

### Anonymous inner classes

Imagine you're writing some GUI code, and suddenly realize that you need an instance of a class that implements ActionListener. But you realize you don't *have* an instance of an ActionListener. Then you realize that you also never wrote a *class* for that listener. You have two choices at that point:

1) Write an inner class in your code, the way we did in our GUI code, and then instantiate it and pass that instance into the button's event registration (addActionListener()) method.

OR

2) Create an *anonymous* inner class and instantiate it, right there, just-in-time. *Literally right where you are at the point you need the listener object*. That's right, you create the class and the instance in the place where you'd normally be supplying just the instance. Think about that for a moment—it means you pass the entire *class* where you'd normally pass only an *instance* into a method argument!

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
    public static void main (String[] args) {
        JFrame frame = new JFrame();
        JButton button = new JButton("click");
        frame.getContentPane().add(button);
        // button.addActionListener(quitListener);
    }
}
```

This statement:  
button.addActionListener (**new ActionListener()** {  
 public void actionPerformed(ActionEvent ev) {  
 System.exit(0);  
 }  
});  
ends down here!  
}  
}

We made a frame and added a button, and now we need to register an action listener with the button. Except we never made a class that implements the ActionListener interface...

Normally we'd do something like this—passing in a reference to an instance of an inner class... an inner class that implements ActionListener (and the actionPerformed() method).

But now instead of passing in an object reference, we pass in... the whole new class definition!! In other words, we write the class that implements ActionListener RIGHT HERE WHERE WE NEED IT. The syntax also creates an instance of the class automatically.

Notice that we say "new ActionListener()" even though ActionListener is an interface and so you can't MAKE an instance of it! But this syntax really means, "create a new class (with no name) that implements the ActionListener interface, and by the way, here's the implementation of the interface methods actionPerformed()."

## #4 Access Levels and Access Modifiers (Who Sees What)

Java has *four access levels* and *three access modifiers*. There are only *three* modifiers because the *default* (what you get when you don't use any access modifier) is one of the four access levels.

**Access Levels** (in order of how restrictive they are, from least to most restrictive)

- public* ← public means any code anywhere can access the public thing (by 'thing' we mean class, variable, method, constructor, etc.).
- protected* ← protected works just like default (code in the same package has access), EXCEPT it also allows subclasses outside the package to inherit the protected thing.
- default* ← default access means that only code within the same package as the class with the default thing can access the default thing.
- private* ← private means that only code within the same class can access the private thing. Keep in mind it means private to the class, not private to the object. One Dog can see another Dog object's private stuff, but a Cat can't see a Dog's privates.

### Access modifiers

public  
protected  
private

Most of the time you'll use only public and private access levels.

#### **public**

Use public for classes, constants (static final variables), and methods that you're exposing to other code (for example getters and setters) and most constructors.

#### **private**

Use private for virtually all instance variables, and for methods that you don't want outside code to call (in other words, methods *used* by the public methods of your class).

But although you might not use the other two (protected and default), you still need to know what they do because you'll see them in other code.

**when arrays aren't enough**

## #4 Access Levels and Access Modifiers, cont.

### **default and protected**

#### **default**

Both protected and default access levels are tied to packages. Default access is simple—it means that only code *within the same package* can access code with default access. So a default class, for example (which means a class that isn't explicitly declared as *public*) can be accessed by only classes within the same package as the default class.

But what does it really mean to *access* a class? Code that does not have access to a class is not allowed to even *think* about the class. And by think, we mean *use* the class in code. For example, if you don't have access to a class, because of access restriction, you aren't allowed to instantiate the class or even declare it as a type for a variable, argument, or return value. You simply can't type it into your code at all! If you do, the compiler will complain.

Think about the implications—a default class with public methods means the public methods aren't really public at all. You can't access a method if you can't *see* the class.

Why would anyone want to restrict access to code within the same package? Typically, packages are designed as a group of classes that work together as a related set. So it might make sense that classes within the same package need to access one another's code, while as a package, only a small number of classes and methods are exposed to the outside world (i.e. code outside that package).

OK, that's default. It's simple—if something has default access (which, remember, means no explicit access modifier!), only code within the same package as the default *thing* (class, variable, method, inner class) can access that *thing*.

Then what's *protected* for?

#### **protected**

Protected access is almost identical to default access, with one exception: it allows subclasses to *inherit* the protected thing, *even if those subclasses are outside the package of the superclass they extend*. That's it. That's *all* protected buys you—the ability to let your subclasses be outside your superclass package, yet still *inherit* pieces of the class, including methods and constructors.

Many developers find very little reason to use protected, but it is used in some designs, and some day you might find it to be exactly what you need. One of the interesting things about protected is that—unlike the other access levels—protected access applies only to *inheritance*. If a subclass-outside-the-package has a *reference* to an instance of the superclass (the superclass that has, say, a protected method), the subclass can't access the protected method using that superclass reference! The only way the subclass can access that method is by *inheriting* it. In other words, the subclass-outside-the-package doesn't have *access* to the protected method, it just *has* the method, through inheritance.

## #3 String and StringBuffer/StringBuilder Methods

Two of the most commonly used classes in the Java API are String and StringBuffer (remember from #9 a few pages back, Strings are immutable, so a StringBuffer/StringBuilder can be a lot more efficient if you're manipulating a String). As of Java 5.0 you should use the **StringBuilder** class instead of **StringBuffer**, unless your String manipulations need to be thread-safe, which is not common. Here's a brief overview of the **key** methods in these classes:

**Both String and StringBuffer/StringBuilder classes have:**

```
char charAt(int index);           // what char is at a certain position
int length();                   // how long is this
String substring(int start, int end); // get a part of this
String toString();              // what's the String value of this
```

**To concatenate Strings:**

```
String concat(string);          // for the String class
String append(String);          // for StringBuffer & StringBuilder
```

**The String class has:**

```
String replace(char old, char new); // replace all occurrences of a char
String substring(int begin, int end); // get a portion of a String
char [] toCharArray();             // convert to an array of chars
String toLowerCase();              // convert all characters to lower case
String toUpperCase();              // convert all characters to upper case
String trim();                   // remove whitespace from the ends
String valueOf(char [])
String valueOf(int i)            // make a String out of a primitive
                                // other primitives are supported as well
```

**The StringBuffer & StringBuilder classes have:**

```
StringBxxxx delete(int start, int end);           // delete a portion
StringBxxxx insert(int offset, any primitive or a char []); // insert something
StringBxxxx replace(int start, int end, String s);    // replace this part with this String
StringBxxxx reverse();                            // reverse the SB from front to back
void setCharAt(int index, char ch);               // replace a given character
```

Note: StringBxxxx refers to either **StringBuffer** or **StringBuilder**, as appropriate.

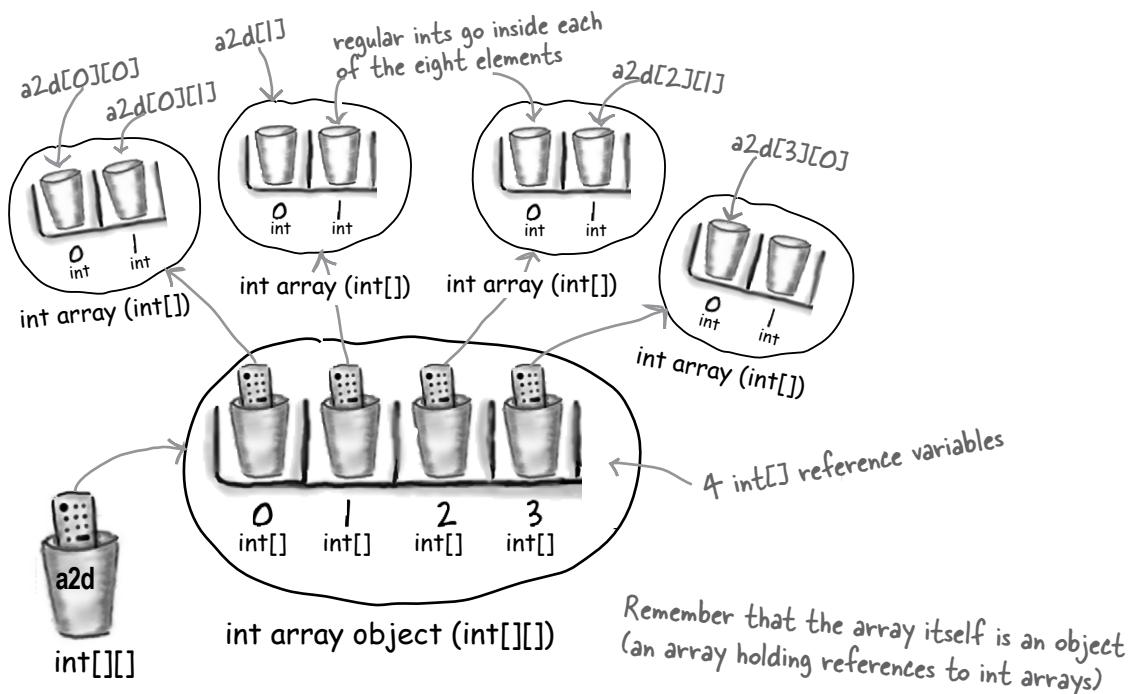
when arrays aren't enough

## #2 Multidimensional Arrays

In most languages, if you create, say, a  $4 \times 2$  two-dimensional array, you would visualize a rectangle, 4 elements by 2 elements, with a total of 8 elements. But in Java, such an array would actually be 5 arrays linked together! In Java, a two dimensional array is simply *an array of arrays*. (A three dimensional array is an array of arrays of arrays, but we'll leave that for you to play with.) Here's how it works

```
int[][] a2d = new int [4][2];
```

The JVM creates an array with 4 elements. *Each* of these four elements is actually a reference variable referring to a (newly created), int array with 2 elements.



### Working with multidimensional arrays

- To access the second element in the third array: `int x = a2d[2][1]; // remember, 0 based!`
- To make a one-dimensional reference to one of the sub-arrays: `int[] copy = a2d[1];`
- Short-cut initialization of a  $2 \times 3$  array: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- To make a 2d array with irregular dimensions:

```
int[][] y = new int [2][]; // makes only the first array, with a length of 2
y[0] = new int [3]; // makes the first sub-array 3 elements in length
y[1] = new int [5]; // makes the second sub-array 5 elements in length
```

## And the number one topic that didn't quite make it in...

### #1 Enumerations (also called Enumerated Types or Enums)

We've talked about constants that are defined in the API, for instance, `JFrame.EXIT_ON_CLOSE`. You can also create your own constants by marking a variable `static final`. But sometimes you'll want to create a set of constant values to represent the *only* valid values for a variable. This set of valid values is commonly referred to as an *enumeration*. Before Java 5.0 you could only do a half-baked job of creating an enumeration in Java. As of Java 5.0 you can create full fledged enumerations that will be the envy of all your pre-Java 5.0-using friends.

#### Who's in the band?

Let's say that you're creating a website for your favorite band, and you want to make sure that all of the comments are directed to a particular band member.

#### The old way to fake an "enum":

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;

// later in the code
if (selectedBandMember == JERRY) {
    // do JERRY related stuff
}
```

*We're hoping that by the time we got here "selectedBandMember" has a valid value!*

The good news about this technique is that it DOES make the code easier to read. The other good news is that you can't ever change the value of the fake enums you've created; JERRY will always be 1. The bad news is that there's no easy or good way to make sure that the value of `selectedBandMember` will always be 1, 2, or 3. If some hard to find piece of code sets `selectedBandMember` equal to 812, it's pretty likely your code will break...

when arrays aren't enough

## #1 Enumerations, cont.

The same situation using a genuine Java 5.0 enum. While this is a very basic enumeration, most enumerations usually *are* this simple.

A new, official "enum":

```
public enum Members { JERRY, BOBBY, PHIL };
public Members selectedBandMember;

// later in the code

if (selectedBandMember == Members.JERRY) {
    // do JERRY related stuff
}
```

No need to worry about this variable's value!

This kind of looks like a simple class definition doesn't it? It turns out that enums ARE a special kind of class. Here we've created a new enumerated type called "Members".

The "selectedBandMember" variable is of type "Members", and can ONLY have a value of "JERRY", "BOBBY", or "PHIL".

The syntax to refer to an enum "instance".

### Your enum extends java.lang.Enum

When you create an enum, you're creating a new class, and *you're implicitly extending java.lang.Enum*. You can declare an enum as its own standalone class, in its own source file, or as a member of another class.

### Using "if" and "switch" with Enums

Using the enum we just created, we can perform branches in our code using either the if or switch statement. Also notice that we can compare enum instances using either == or the .equals() method. Usually == is considered better style.

```
Members n = Members.BOBBY; ← Assigning an enum value to a variable.

if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");
if (n == Members.BOBBY) System.out.println("Rat Dog"); ← Both of these work fine!
                                                               "Rat Dog" is printed.

Members ifName = Members.PHIL;
switch (ifName) {
    case JERRY: System.out.print("make it sing ");
    case PHIL: System.out.print("go deep ");
    case BOBBY: System.out.println("Cassidy! "); ← Pop Quiz! What's the output?
}

Answer: Cassidy! go deep
```

## #1 Enumerations, completed

### A really tricked-out version of a similar enum

You can add a bunch of things to your enum like a constructor, methods, variables, and something called a constant-specific class body. They're not common, but you might run into them:

```
public class HfjEnum {
    enum Names {
        JERRY("lead guitar") { public String sings() {
            return "plaintively"; }
        },
        BOBBY("rhythm guitar") { public String sings() {
            return "hoarsely"; }
        },
        PHIL("bass");
    }

    private String instrument;

    Names(String instrument) {
        this.instrument = instrument;
    }
    public String getInstrument() {
        return this.instrument;
    }
    public String sings() {
        return "occasionally";
    }
}

public static void main(String [] args) {
    for (Names n : Names.values()) {
        System.out.print(n);
        System.out.print(", instrument: " + n.getInstrument());
        System.out.println(", sings: " + n.sings());
    }
}
}
```

Annotations on the code:

- An annotation pointing to the constructor of the enum: "This is an argument passed in to the constructor declared below."
- An annotation pointing to the constant-specific class bodies: "These are the so-called 'constant-specific class bodies'. Think of them as overriding the basic enum method (in this case the "sing()" method), if sing() is called on a variable with an enum value of JERRY or BOBBY."
- An annotation pointing to the enum's constructor: "This is the enum's constructor. It runs once for each declared enum value (in this case it runs three times)."
- An annotation pointing to the methods: "You'll see these methods being called from "main()"."
- An annotation pointing to the built-in values() method: "Every enum comes with a built-in "values()" method which is typically used in a "for" loop as shown."

```
File Edit Window Help Bootleg
%java HfjEnum
JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

Notice that the basic "sing()" method is only called when the enum value has no constant-specific class body.

**when arrays aren't enough**



## Five-Minute Mystery

### A Long Trip Home

Captain Byte of the Flatland starship “Traverser” had received an urgent, Top Secret transmission from headquarters. The message contained 30 heavily encrypted navigational codes that the Traverser would need to successfully plot a course home through enemy sectors. The enemy Hackarians, from a neighboring galaxy, had devised a devilish code-scrambling ray that was capable of creating bogus objects on the heap of the Traverser’s only navigational computer. In addition, the alien ray could alter valid reference variables so that they referred to these bogus objects. The only defense the Traverser crew had against this evil Hackarian ray was to run an inline virus checker which could be imbedded into the Traverser’s state of the art Java 6 code.



Captain Byte gave Ensign Smith the following programming instructions to process the critical navigational codes:

“Put the first five codes in an array of type ParsecKey. Put the last 25 codes in a five by five, two dimensional array of type QuadrantKey. Pass these two arrays into the plotCourse() method of the public final class ShipNavigation. Once the course object is returned run the inline virus checker against all the programs reference variables and then run the NavSim program and bring me the results.”

A few minutes later Ensign Smith returned with the NavSim output. “NavSim output ready for review, sir”, declared Ensign Smith. “Fine”, replied the Captain, “Please review your work”. “Yes sir!”, responded the Ensign, “First I declared and constructed an array of type ParsecKey with the following code; ParsecKey [] p = new ParsecKey[5]; , next I declared and constructed an array of type QuadrantKey with the following code: QuadrantKey [] [] q = new QuadrantKey [5] [5]; . Next, I loaded the first 5 codes into the ParsecKey array using a ‘for’ loop, and then I loaded the last 25 codes into the QuadrantKey array using nested ‘for’ loops. Next, I ran the virus checker against all 32 reference variables, 1 for the ParsecKey array, and 5 for its elements, 1 for the QuadrantKey array, and 25 for its elements. Once the virus check returned with no viruses detected, I ran the NavSim program and re-ran the virus checker, just to be safe... Sir ! “

Captain Byte gave the Ensign a cool, long stare and said calmly, “Ensign, you are confined to quarters for endangering the safety of this ship, I don’t want to see your face on this bridge again until you have properly learned your Java! Lieutenant Boolean, take over for the Ensign and do this job correctly!”

Why did the captain confine the Ensign to his quarters?



## Five-Minute Mystery Solution



### A Long Trip Home

Captain Byte knew that in Java, multidimensional arrays are actually arrays of arrays. The five by five QuadrantKey array 'q', would actually need a total of 31 reference variables to be able to access all of its components:

- 1 - reference variable for 'q'
- 5 - reference variables for  $q[0] - q[4]$
- 25 - reference variables for  $q[0][0] - q[4][4]$

The ensign had forgotten the reference variables for the five one dimensional arrays embedded in the 'q' array. Any of those five reference variables could have been corrupted by the Hackarian ray, and the ensign's test would never reveal the problem.





# Index

## Symbols

&, &&, !, || (boolean operators) 151, 660  
 &, <<, >>, >>>, ^, |, ~ (bitwise operators) 660  
 ++ – (increment/decrement) 105, 115  
 + (String concatenation operator) 17  
 . (dot operator) 36  
     reference 54  
 <, <=, ==, !=, >, >= (comparison operators) 86, 114, 151  
 <, <=, ==, >, >= (comparison operators) 11

## A

abandoned objects. *See* garbage collection  
 abstract  
     class 200–210  
     class modifier 200  
 abstract methods  
     declaring 203  
 access  
     and inheritance 180  
     class modifiers 667  
     method modifiers 81, 667  
     variable modifiers 81, 667  
 accessors and mutators. *See* getters and setters  
 ActionListener interface 358, 358–361  
 addActionListener() 359–361  
 advice guy 480, 484  
 Aeron™ 28  
 animation 382–385  
 API 154–155, 158–160  
     ArrayList 532  
     collections 558

appendix A 649–658  
 beat box final client 650  
 beat box final server 657  
 appendix B  
     access levels and modifiers 667  
     assertions 662  
     bit manipulation 660  
     block scope 663  
     immutability 661  
     linked invocations 664  
     multidimensional arrays 670  
     String and StringBuffer methods 669

apples and oranges 137  
 arguments  
     method 74, 76, 78  
     polymorphic 187  
 ArrayList 132, 133–138, 156, 208, 558  
     API 532  
     ArrayList<Object> 211–213  
     autoboxing 288–289  
     casting 229

arrays  
     about 17, 59, 135  
     assigning 59  
     compared to ArrayList 134–137  
     creation 60  
     declaring 59  
     length attribute 17  
     multidimensional 670  
     objects, of 60, 83  
     primitives, of 59

assertions  
     assertions 662  
 assignments, primitive 52  
 assignments, reference variables 55, 57, 83  
 atomic code blocks 510–512. *See also* threads

## the index

audio. *See* midi  
autoboxing 288–291  
  and operators 291  
  assignments 291

## B

bark different 73  
bathtub 177  
beat box 316, 347, 472. *See also* appendix A  
beer 14  
behavior 73  
Bela Fleck 30  
bitwise operators 660  
bit shifting 660  
block scope 663  
boolean 51  
boolean expressions 11, 114  
  logical 151  
BorderLayout manager 370–371, 401, 407  
BoxLayout manager 411  
brain barbell 33, 167, 188  
break statement 105  
BufferedReader 454, 478  
BufferedWriter 453  
buffers 453, 454  
byte 51  
bytecode 2

## C

Calendar 303–305  
  methods 305  
casting  
  explicit primitive 117  
  explicit reference 216  
  implicit primitive 117  
catching exceptions 326

catch 338  
catching multiple exceptions 329, 330, 332  
try 321  
catch blocks 326, 338  
  catching multiple exceptions 329, 330, 332  
chair wars 28, 166  
char 51  
chat client 486  
  with threads 518  
chat server (simple) 520  
checked exceptions  
  runtime vs. 324  
checking account. *See* Ryan and Monica  
check box (JCheckBox) 416  
class  
  abstract 200–210  
  concrete 200–210  
  designing 34, 41, 79  
  final 283  
  fully qualified names 154–155, 157  
client/server 473  
code kitchen  
  beat box save and restore 462  
  final beat box. *See* appendix A  
  making the GUI 418  
  music with graphics 386  
  playing sound 339  
coffee cups 51  
collections 137, 533  
  API 558  
  ArrayList 137  
  ArrayList<Object> 211–213  
  Collections.sort() 534, 539  
  HashMap 533  
  HashSet 533  
  LinkedHashMap 533  
  LinkedList 533  
  List 557  
  Map 557, 567  
  parameterized types 137

Set 557  
 TreeSet 533  
 Collections.sort() 534, 539  
   Comparator 551  
   compare() 553  
 Comparable 547, 566  
   and TreeSet 566  
   compareTo() method 549  
 Comparator 551, 566  
   and TreeSet 566  
 compare() 553  
 compareTo() 549  
 comparing with == 86  
 compiler 2  
   about 18  
   java -d 590  
 concatenate 17  
 concrete classes 200–210  
 conditional expressions 10, 11, 13  
 constants 282  
 constructors  
   about 240  
   chaining 250–256  
   overloaded 256  
   superclass 250–256  
 contracts 190–191, 218  
 cups 51  
 curly braces 10

**D**

daily advice client 480  
 daily advice server 484  
 dancing girl 316  
 dates  
   Calendar 303  
   methods 305  
   formatting 301  
 GregorianCalendar 303

java.util.Date 303  
 deadlock 516  
 deadly diamond of death 223  
 declarations  
   about 50  
   exceptions 335–336  
   instance variables 50  
 default access 668  
 default value 84  
 deployment options 582, 608  
 deserialized objects 441. *See also* serialization  
 directory structures  
   packages 589  
   servlets 626  
 doctor 169  
 dot operator  
   reference 54  
 double 51  
 duck 277  
   construct 242  
   garbage collect 261  
 ducking exceptions 335

**E**

EJB 631  
 encapsulation  
   about 79–82  
   benefits 80  
 end of book 648  
 enumerations 671–672  
 enums 671–672  
 equality 560  
   and hashCode() 561  
 equals() 561  
 equals( )  
   about 209  
   Object class 209

## the index

- event handling 357–361
  - event object 361
  - listener interface 358–361
  - using inner classes 379
- event source 359–361
- exceptions
  - about 320, 325, 338
  - catch 321, 338
  - catching multiple exceptions 329, 332
  - checked vs. runtime 324
  - declaring 335–336
  - ducking 335–336
  - finally 327
  - flow control 326
  - handle or declare law 337
  - propagating 335–336
  - remote exceptions 616
  - throwing 323–326
  - try 321, 338
- executable JAR 585–586, 586
  - with packages 592, 592–593
- exercises
  - be the... 88, 118, 266, 310, 395
  - code magnets 20, 43, 64, 119, 312, 349, 467, 524–525
  - honeypot 267
  - true or false 311, 348, 466, 602
  - what's the declaration 231
  - what's the picture 230
  - which layout manager? 424
  - who am I 45, 89, 394
- Extreme Programming 101
- F**
  - File 452
  - FileInputStream 441. *See also* I/O
  - FileOutputStream 432
  - FileReader 454. *See also* I/O
  - files
    - File class 452
- garbage collection
  - about 40
  - eligible objects 260–263
  - heap 57, 58
  - nulling references 58
  - reassigning references 58
  - generics 540, 542, 568–574
  - methods 544
- FileWriter 447
- File class 452
- final
  - class 189, 283
  - methods 189, 283
  - static variables 282
  - variables 282, 283
- finally block 327
- fireside chats
  - about 18
- five minute mystery. *See* puzzles
- float 51
- FlowLayout 403, 408–410
- flow control
  - exceptions 326
- font 406
- formatting
  - dates 301–302
  - format specifiers 295–296
  - argument 300
  - numbers 294–295
  - printf() 294
  - String.format() 294
- for loops 105
- fully qualified name 154, 157
  - packages 587

## G

- garbage collection
  - about 40
  - eligible objects 260–263
  - heap 57, 58
  - nulling references 58
  - reassigning references 58
  - generics 540, 542, 568–574
  - methods 544

- wildcards 574
- getters and setters 79
- ghost town 109
- giraffe 50
- girl dreaming
  - inner classes 375
  - Java Web Start 596
- girl in a tub 177
- girl who isn't getting it 182–188
- graphics 364–366. *See also GUI*
  - Graphics2D class 366
  - Graphics object 364
- GregorianCalendar 303
- guessing game 38
- GUI 406
  - about 354, 400
  - animation 382–385
  - BorderLayout 370–371, 401, 407
  - box layout 403, 411
  - buttons 405
  - components 354, 363–368, 400
  - event handling 357–361, 379
  - flow layout 403, 408
  - frames 400
  - graphics 363–367
  - ImageIcon class 365
  - JButton 400
  - JLabel 400
  - JPanel 400, 401
  - JTextArea 414
  - JTextField 413
  - layout managers 401–412
  - listener interface 358–361
  - scrolling (JScrollPane) 414
  - Swing 354
- GUI Constants
  - ScrollPaneConstants.HORIZONTAL\_SCROLLBAR\_NEVER 415
  - ScrollPaneConstants.VERTICAL\_SCROLLBAR\_ALWAYS 415
- GUI methods
  - drawImage() 365
  - fillOval() 365
  - fillRect() 364
  - gradientPaint(). *See also GUI*
  - paintComponent() 364
  - setColor() 364
  - setFont() 406
- GUI Widgets 354
  - JButton 354, 405
  - JCheckBox 416
  - JFrame 354, 400, 401
  - JList 417
  - JPanel 400, 401
  - JScrollPane 414, 417
  - JTextArea 414
  - JTextField 413

## H

- HAS-A 177–181
- hashCode() 561
- HashMap 533, 558
- HashSet 533, 558
- Hashtable 558
- heap
  - about 40, 57, 236–238
  - garbage collection 40, 57, 58

## I

- I/O
  - BufferedReader 454, 478
  - BufferedWriter 453
  - buffers 453
  - deserialization 441
  - FileInputStream 441
  - FileOutputStream 432
  - FileWriter 447
  - InputStreamReader 478
  - ObjectInputStream 441
  - ObjectOutputStream 432, 437
  - serialization 432, 434–439, 437, 446, 460

## the index

streams 433, 437  
with sockets 478  
if -else 13  
if statement 13  
immutability, Strings  
    immutability 661  
implements 224  
imports  
    static imports 307  
import statement 155, 157  
increment 105  
inheritance  
    about 31, 166–192  
    and abstract classes 201  
    animals 170–175  
    IS-A 214, 251  
    super 228  
initializing  
    instance variables 84  
    primitives 84  
    static variables 281  
inner classes  
    about 376–386  
    events 379  
inner class threesome 381  
InputStreamReader 478  
instance variables  
    about 34, 73  
    declaring 84  
    default values 84  
    initializing 84  
    life and scope 258–263  
    local variables vs. 236–238, 239  
    static vs. 277  
instantiation. *See* objects  
int 50  
    primitive 51  
Integer. *See* wrapper  
interfaces

about 219–227  
for serialization 437  
implementing 224, 437  
implementing multiple 226  
java.io.Serializable 437  
IP address. *See* networking  
IS-A 177–181, 251

## J

J2EE 631

### JAR files

basic commands 593  
executable 585–586, 592  
manifest 585  
running executable 586, 592  
tool 593  
with Java Web Start 598

Java, about 5, 6

javac. *See* compiler

Java in a Nutshell 158–159

java sound 317, 340

Java Web Start 597–601

    jnlp file 598, 599

Jini 632–635

JNLP 598

    jnlp file 599

JPEG 365

JVM

    about 2, 18

JWS. *See* Java Web Start

## K

keywords 53

## L

l 264

layout managers 401–412

    BorderLayout 370–371, 403, 407

BoxLayout 403, 411  
 FlowLayout 403, 408–410  
 lingerie, exceptions 329  
 LinkedHashMap 533, 558  
 LinkedHashSet 558  
 LinkedList 533, 558  
 linked invocations 664  
 List 557  
 listeners  
   listener interface 358–361  
 literals, assigning values  
   primitive 52  
 local  
   variables 85, 236, 236–238, 258–263  
 locks  
   object 509  
   threads 509  
 long 51  
 loops  
   about 10  
   break 105  
   for 105  
   while 115  
 lost update problem. *See* threads

## M

main() 9, 38  
 make it stick 53, 87, 157, 179, 227, 278  
 manifest file 585  
 Map 557, 567  
 Math class  
   methods 274–278, 286  
   random() 111  
 memory  
   garbage collection 260–263  
 metacognitive tip 33, 108, 325  
 methods  
   about 34, 78

abstract 203  
 arguments 74, 76, 78  
 final 283  
 generic arguments 544  
 on the stack 237  
 overloading 191  
 overriding 32, 167–192  
 return 75, 78  
 static 274–278  
 midi 317, 340–346, 387–390  
 midi sequencer 340–346  
 MINI Cooper 504  
 modifiers  
   class 200  
   method 203  
 multidimensional arrays 670  
 multiple inheritance 223  
 multiple threads. *See* threads  
 music. *See* midi  
 mystery. *See* puzzles

## N

naming 53. *See also* RMI  
   classes and interfaces 154–155, 157  
   collisions 587  
   packages 587  
 networking  
   about 473  
   ports 475  
   sockets 475  
 new 55  
 null  
   reference 262  
 numbers  
   formatting 294–295

## O

ObjectOutputStream 432, 437

## the index

- objects
  - about 55
  - arrays 59, 60, 83
  - comparing 209
  - creation 55, 240–256
  - eligible for garbage collection 260–263
  - equality 560
  - equals() 209, 561
  - life 258–263
  - locks 509
- Object class
  - about 208–216
  - equals() 561
  - hashCode() 561
  - overriding methods 563
- object graph 436, 438
- object references 54, 56
  - assignment 55, 262
  - casting 216
  - comparing 86
  - equality 560
  - nulling 262
  - polymorphism 185–186
- OO
  - contracts 190–191, 218
  - deadly diamond of death 223
  - design 34, 41, 79, 166–191
  - HAS-A 177–181
  - inheritance 166–192
  - interfaces 219–227
  - IS-A 177–181, 251
  - overload 191
  - override 167–192
  - polymorphism 183, 183–191, 206–217
  - superclass 251–256
- operators
  - and autoboxing 291
  - bitwise 660
  - comparison 151
  - conditional 11
  - decrement 115
  - increment 105, 115
  - logical 151
  - shift 660
- overload 191
  - constructors 256
- override
  - about 32, 167–192
  - polymorphism. *See* polymorphism

## P

- packages 154–155, 157, 587–593
  - directory structure 589
  - organizing code 589
- paintComponent() 364–368
- parameter. *See* arguments
- parameterized types 137
- parsing an int. *See* wrapper
- parsing text with String.split() 458
- pass-by-copy. *See* pass-by-value
- pass-by-value 77
- phrase-o-matic 16
- polymorphism 183–191
  - abstract classes 206–217
  - and exceptions 330
  - arguments and return types 187
  - references of type Object 211–213
- pool puzzle. *See* puzzles
- ports 475
- prep code 99–102
- primitives 53
  - == operator 86
  - autoboxing 288–289
  - boolean 51
  - byte 51
  - char 51
  - double 51
  - float 51
  - int 51

ranges 51  
 short 51  
 type 51  
 primitive casting  
     explicit primitive 117  
`printf()` 294  
`PrintWriter` 479  
 private  
     access modifier 81  
 protected 668  
 public  
     access modifier 81, 668  
 puzzles

    five minute mystery 92, 527, 674  
 Java cross 22, 120, 162, 350, 426, 603  
 pool puzzle 24, 44, 65, 91, 194, 232, 396

## Q

quiz card builder 448, 448–451

## R

rabbit 50  
`random()` 111  
 ready-bake code 112, 152–153, 520  
 reference variables. *See* object references  
     casting 216  
 registry, RMI 615, 617, 620  
 remote control 54, 57  
 remote interface. *See* RMI  
 reserved words 53  
 return types  
     about 75  
     polymorphic 187  
     values 78  
 risky code 319–336  
 RMI  
     about 614–622  
     client 620, 622

compiler 618  
*Jini*. *See also* Jini  
`Naming.lookup()` 620  
`Naming.rebind()`. *See also* RMI  
 registry 615, 617, 620  
 remote exceptions 616  
 remote implementation 615, 617  
 remote interface 615, 616  
`rmic` 618  
 skeleton 618  
 stub 618  
`UnicastRemoteObject` 617  
 universal service browser 636–648  
`rmic`. *See* RMI  
`run()`  
     overriding in Runnable interface 494  
 Runnable interface 492  
     about 493  
     `run()` 493, 494  
     threads 493  
 runnable thread state 495  
 Ryan and Monica 505–506  
     introduction 505–506

## S

scary objects 200  
 scheduling threads  
     scheduling 496–498  
 scope  
     variables 236–238, 258–263  
 scrolling (JScrollPane) 414  
 serialization 434–439, 446  
     deserialization 460  
     interface 437  
     `ObjectInputStream`. *See* I/O  
     `objectOutputStream` 432  
     objects 460  
     object graph 436  
     reading. *See* I/O  
     restoring 460. *See also* I/O

## the index

saving 432  
serialVersionUID 461  
transient 439  
versioning 460, 461  
writing 432

server  
    socket 483. *See also* socket

servlet 625–627

Set 557  
    importance of equals() 561  
    importance of hashCode() 561

short 51

short circuit logical operators 151

sink a dot com 96–112, 139–150

skeleton. *See* RMI

sleep() 501–503

sleeping threads 501–503

snowboard 214

socket  
    about 475  
    addresses 475  
    creating 478  
    I/O 478  
    ports 475  
    reading from 478  
    server 483  
    TCP/IP 475  
    writing to 479

sorting  
    Collections.sort() 534, 539, 547  
    Comparable interface 547, 549  
    Comparator 551, 553  
    TreeSet 564–566

source files  
    structure of 7

specifiers  
    format specifiers 295, 298  
    argument specifier 300

stack  
    heap vs. 236  
    methods on 237  
    scope 236  
    threads 490  
    trace 323

static  
    enumerated types 671  
    initializer 282  
    Math class methods 274–278  
    methods 274–278  
    static imports 307  
    variables 282

streams 433. *See also* I/O

String  
    arrays 17  
    concatenating 17  
    methods 669  
    parsing 458  
    String.format() 294–297  
    String.split() 458

StringBuffer/StringBuilder  
    methods 669

stub. *See* RMI

subclass  
    about 31, 166–192

super 228  
    about 31

superclass  
    about 166–192, 214–217, 228

super constructor 250–256

Swing. *See* GUI

synchronized  
    methods 510. *See also* threads

syntax  
    about 10, 12

System.out.print() 13

System.out.println() 13

# T

talking head 203

TCP ports 475

Telluride 30

testing

- extreme programming 101

text

- parsing with `String.split()` 458 458

- read from a file. *See also I/O*

- write to a file 447

text area (`JTextArea`) 414

text field (`JTextField`) 413

`Thread.sleep()` 501–503

threads

- about 489–515

- deadlock 516

- locks 509

- lost update problem 512–514

- `run()` 493, 494

- `Runnable` 492, 493, 494

- Ryan and Monica problem 505–507

- scheduling 496, 496–498

- `sleep()` 501–503

- stack 490–491

- `start()` 492

- starting 492

- states 495, 496

- summary 500, 517

- synchronized 510–512

- unpredictability 498–499

throw

- exceptions 323–326

- `throws` 323–326

transient 439

`TreeMap` 558

`TreeSet` 533, 558, 564–566, 566

try

- blocks 321, 326

type 50

- parameter 137, 542, 544

type-safety 540

- and generics 540

# U

universal service browser 636–648

# V

variables

- assigning 52, 262

- declaring 50, 54, 84, 236–238

- local 85, 236–238

- nulling 262

- primitive 51, 52

- references 54, 55, 56, 185–186

- scope 236–238

- static. *See static*

variable declarations 50

- instance 84

- primitive 51

- reference 54

virtual method invocation 175

# W

web start. *See Java Web Start*

while loops 11, 115

wildcard 574

wine 202

wrapper 287

- autoboxing 288–289

- conversion utilities 292

- `Integer.parseInt()` 104, 106, 117

writing. *See I/O*

Don't you know about the web site?  
We've got answers to some of the  
Sharpens, examples, the Code Kitchens,  
Ready-bake Code, and daily updates  
from the Head First author blogs!

## This isn't goodbye

**Bring your brain over to  
[wickedlysmart.com](http://wickedlysmart.com)**

