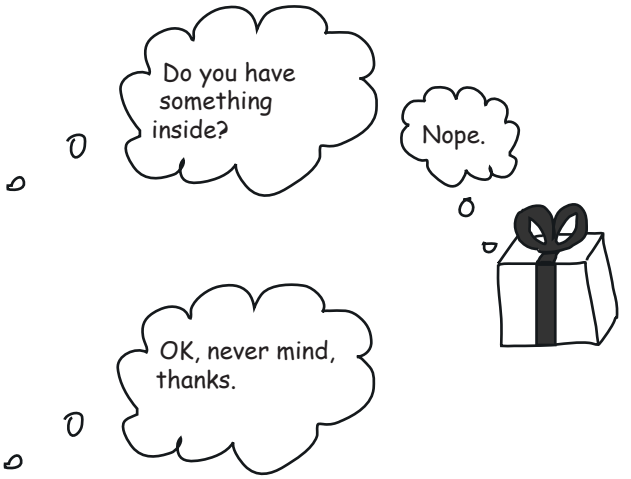
You've just introduced two new steps for me to get my ice cream!

Yes, but now we have a way to ask if we have a result

Optional gives us a way to find out about, and deal with, the times when you don't get an ice cream.


```
Optional<IceCream> optional =
  getIceCream("Strawberry");
```




```
if (optional.isPresent()) {
```

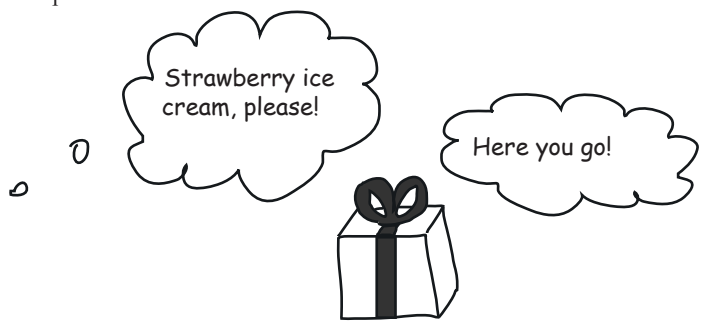


```
} else {
  System.out.println("No ice
cream for you!");
}
```

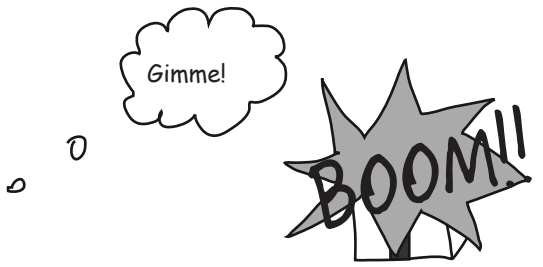
In the past, methods might have thrown Exceptions for this case, or return "null", or a special type of "Not Found" ice-cream instance. Returning an *Optional* from a method makes it really clear that anything calling the method ***needs*** to check if there's a result first, and ***then*** make their own decision about what to do if there isn't one.

Don't forget to talk to the Optional wrapper

The important thing about Optional results is that **they can be empty**. If you don't check first to see if there's a value present and the result is empty, you will get an exception.

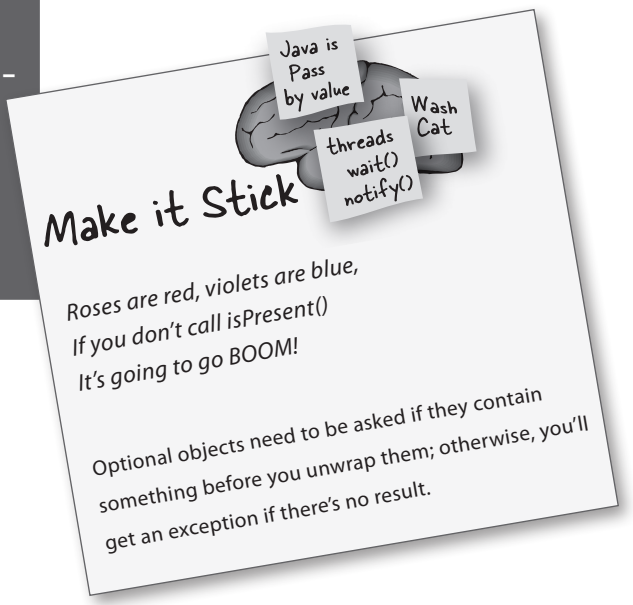



```
Optional<IceCream> optional =
  getIceCream("Strawberry");
```




```
IceCream ice = optional.get();
```

```
File  Edit  Window  Help  Boom
%java OptionalExamples

Exception in thread "main" java.util.No-
SuchElementException: No value present
      at java.base/java.util.Optional.
get(Optional.java:148)
      at ch10c.OptionalExamples.
main(OptionalExamples.java:11)
```



The Unexpected Coffee

Alex was programming her mega-ultra-clever (Java-powered) coffee machine to give her the types of coffee that suited her best at different times of day.

Five-Minute Mystery

In the afternoons, Alex wanted the machine to give her the weakest coffee it had available (she had enough to keep her up at night; she didn't need caffeine adding to her problems!). As an experienced software developer, she knew the Streams API would give her the best stream of coffee at the right time.

The coffees would automatically be sorted from the weakest to the strongest using natural ordering, so she gave the coffee machine these instructions:

```
Optional<String> afternoonCoffee = coffees.stream()
                                          .map(Coffee::getName)
                                          .sorted()
                                          .findFirst();
```

The very next day, she asked for an afternoon coffee. To her horror, the machine presented her with an Americano, not the Decaf Cappuccino she was expecting.

"I can't drink that!! I'll be up all night worrying about my latest software project!"

What happened? Why did the coffee machine give Alex an Americano?

→ Answers on page 419.

Pool Puzzle

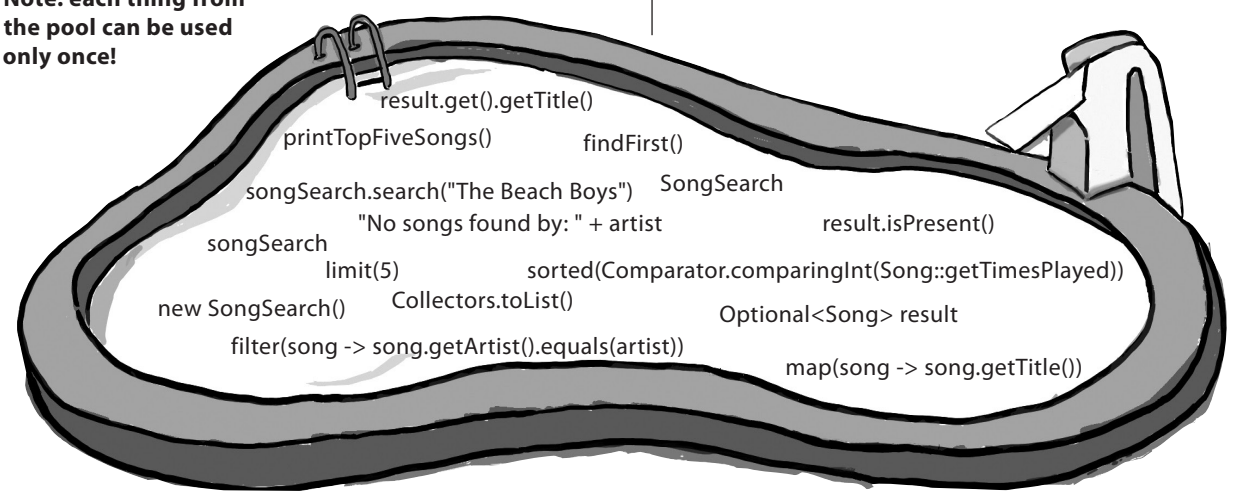
Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a class that will compile and run and produce the output listed.

```
public class StreamPuzzle {
  public static void main(String[] args) {
    SongSearch songSearch = _____________;
    songSearch._____________;
    ____________.search("The Beatles");
    ____________________________________;
  }
}
class _____________ {
  private final List<Song> songs =
      new JukeboxData.Songs().getSongs();

  void printTopFiveSongs() {
    List<String> topFive = songs.stream()
                              ._____________
                              ._____________
                              ._____________
                              .collect(_____________);
    System.out.println(topFive);
  }
  void search(String artist) {
    _________ = songs.stream()
                     ._________________
                     ._________________;
    if (_________________) {
      System.out.println(_________________);
    } else {
      System.out.println(_________________);
    }
  }
}
```

Output

```
File  Edit  Window  Help  DiveIn
%java StreamPuzzle
[Immigrant Song, With a Little Help from My Friends, Hallucinate, Pasos de cero, Cassidy]
With a Little Help from My Friends
No songs found by: The Beach Boys
```

Note: each thing from the pool can be used only once!



Answers on page 420.

Mixed Messages

(from page 372)

Candidates:

```
for (int i = 1; i < nums.size(); i++)
    output += nums.get(i) + " ";
```

```
for (Integer num : nums)
    output += nums + " ";
```

```
for (int i = 0; i <= nums.length; i++)
    output += nums.get(i) + " ";
```

```
for (int i = 0; i <= nums.size(); i++)
    output += nums.get(i) + " ";
```

Possible output:

```
1 2 3 4 5
```

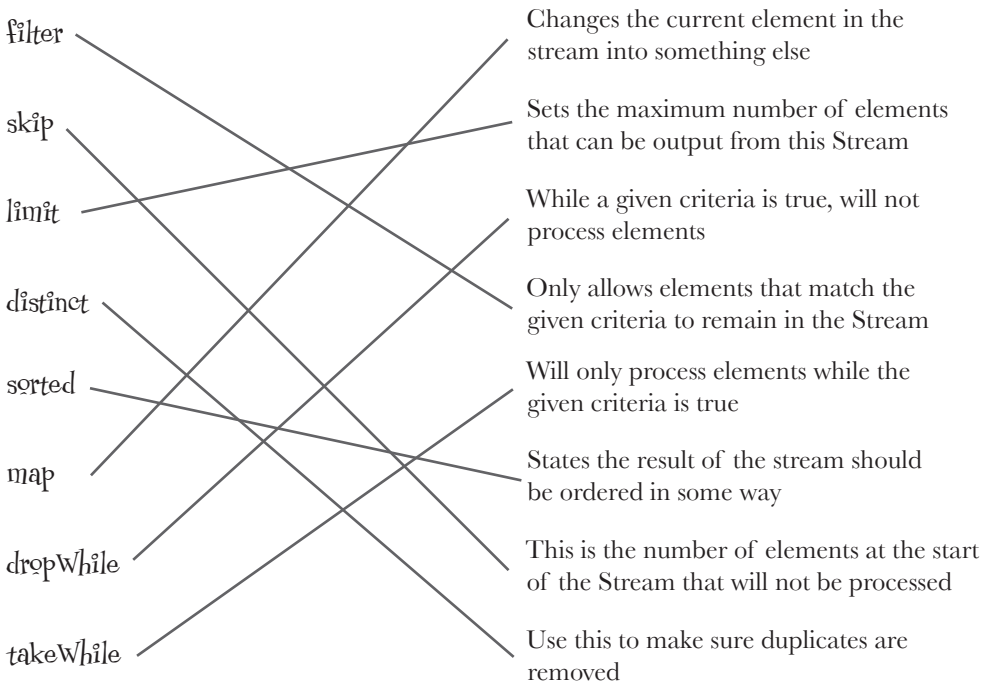
```
Compiler error
```

```
2 3 4 5
```

```
Exception thrown
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

(from page 374)



Code Magnets (from page 386)

What would happen if the stream operations were in a different order? Does it matter?

```
import java.util.*;
import java.util.stream.*;

public class CoffeeOrder {
  public static void main(String[] args) {
    List<String> coffees = List.of("Cappuccino",
            "Americano", "Espresso", "Cortado", "Mocha",
            "Cappuccino", "Flat White", "Latte");

    List<String> coffeesEndingInO = coffees.stream()
                                           .filter(s -> s.endsWith("o"))
                                           .sorted()
                                           .distinct()
                                           .collect(Collectors.toList());
    System.out.println(coffeesEndingInO);
  }
}
```

```
File Edit Window Help Cafelito
%java CoffeeOrder

[Americano, Cappuccino,
Cortado, Espresso]
```

BE the Compiler (from page 395)

- [x] `Runnable r = () -> System.out.println("Hi!");`
- [x] `Consumer<String> c = s -> System.out.println(s);`
- [ ] `Supplier<String> s = () -> System.out.println("Some string");` — Should return a String but doesn't
- [ ] `Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2);` — Should take only one parameter but has two
- [ ] `Runnable r = (String str) -> System.out.println(str);` — Should not have parameters
- [x] `Function<String, Integer> f = s -> s.length();`
- [x] `Supplier<String> s = () -> "Some string";`
- [ ] `Consumer<String> c = s -> "String" + s;` — This single-line lambda effectively returns a String when a consumer method should return nothing. Even though there's no "return," this calculated String value is assumed to be the returned value.
- [ ] `Function<String, Integer> f = (int i) -> "i = " + i;` — Should have a String parameter and return an int, but instead it has an int param and returns a String
- [ ] `Supplier<String> s = s -> "Some string: " + s;` — Should not have any parameters
- [ ] `Function<String, Integer> f = () -> System.out.println("Some string");` — Should take a String parameter. Should return an int, but actually returns nothing.

Sharpen your pencil
(from page 397)

BiPredicate

Modifier and Type	Method
default BiPredicate<T,U>	and(BiPredicate<? super T,? super U> other)
default BiPredicate<T,U>	negate()
default BiPredicate<T,U>	or(BiPredicate<? super T,? super U> other)
boolean	test(T t, U u)

Has a Single Abstract Method, test(). The others are all default methods.

ActionListener

Modifier and Type	Method
void	actionPerformed(ActionEvent e)

Has a Single Abstract Method

Iterator

Modifier and Type	Method
default void	forEachRemaining(Consumer<? super E> action)
boolean	hasNext()
E	next()
default void	remove()

Has TWO abstract methods, hasNext() and next()

Function

Modifier and Type	Method
default <V> Function<T,V>	andThen(Function<? super R,? extends V> after)
R	apply(T t)
default <V> Function<V,R>	compose(Function<? super V,? extends T> before)
static <T> Function<T,T>	identity()

Has a Single Abstract Method, apply(). The others are default and static methods.

SocketOption

Modifier and Type	Method
String	name()
Class<T>	type()

Has two abstract methods

Five-Minute Mystery (from page 415)

Alex didn't pay attention to the order of the stream operations. She first mapped the coffee objects to a stream of Strings, and then ordered that. Strings are naturally ordered alphabetically, so when the coffee machine got the "first" of these results for Alex's afternoon coffee, it was brewing a fresh "Americano."

If Alex wanted to order the coffees by strength, with the weakest (1 out of 5) first, she needed to order the stream of coffees first, before mapping it to a String name,

```
afternoonCoffee = coffees.stream()
                         .sorted()
                         .map(Coffee::getName)
                         .findFirst();
```

Then the coffee machine will brew her a decaf instead of an Americano.

Pool Puzzle (from page 416)

```
public class StreamPuzzle {
  public static void main(String[] args) {
    SongSearch songSearch = new SongSearch();
    songSearch.printTopFiveSongs();
    songSearch.search("The Beatles");
    songSearch.search("The Beach Boys");
  }
}
class SongSearch {
  private final List<Song> songs =
      new JukeboxData.Songs().getSongs();

  void printTopFiveSongs() {
    List<String> topFive = songs.stream()
                          .sorted(Comparator.comparingInt(Song::getTimesPlayed))
                          .map(song -> song.getTitle())
                          .limit(5)
                          .collect(Collectors.toList());
    System.out.println(topFive);
  }
  void search(String artist) {
    Optional<Song> result = songs.stream()
                        .filter(song -> song.getArtist().equals(artist))
                        .findFirst();
    if (result.isPresent()) {
      System.out.println(result.get().getTitle());
    } else {
      System.out.println("No songs found by: " + artist);
    }
  }
}
```

Risky Behavior

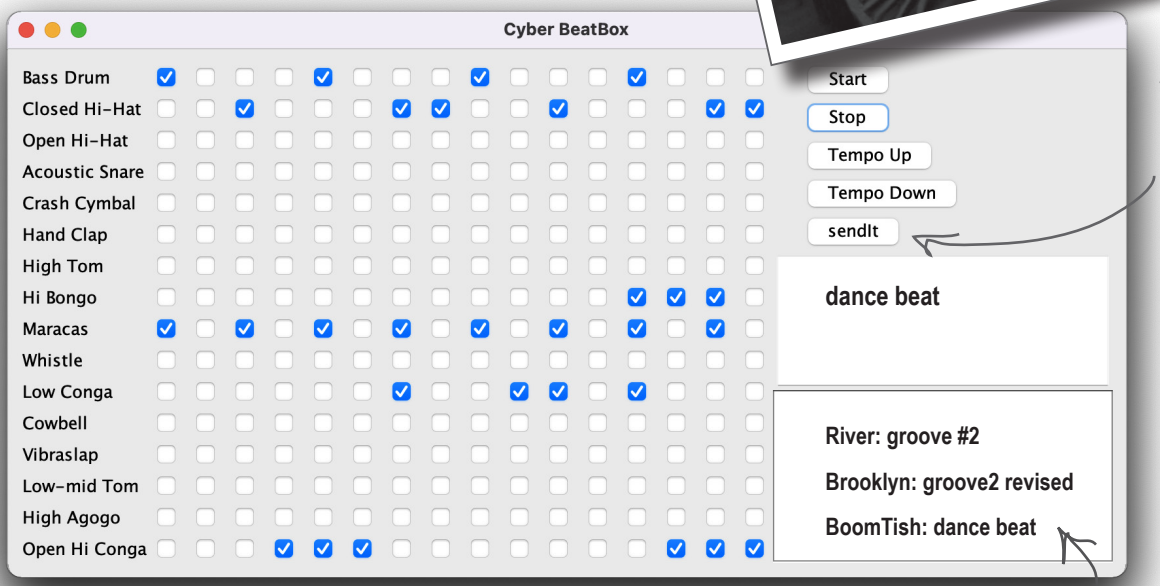
Stuff happens. The file isn't there. The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very* wrong. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? And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

Let's make a Music Machine

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

The finished BeatBox looks something like this:

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



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.

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

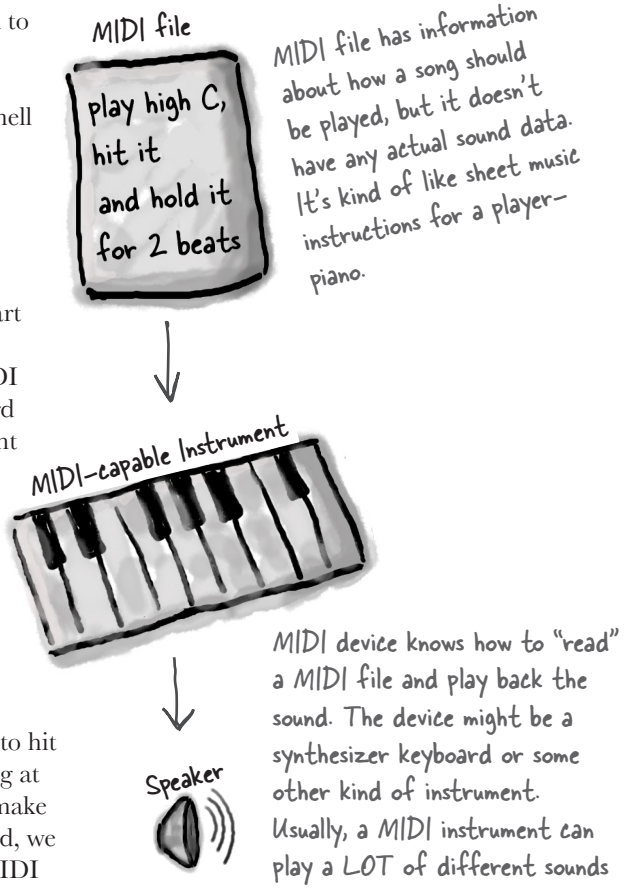
We'll start with the basics

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

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

The JavaSound API

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



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

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

First we need a Sequencer

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

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

public class MusicTest1 {
  public void play() {
    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      System.out.println("Successfully got a sequencer");
    }
  }

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

import the javax.sound.midi package

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

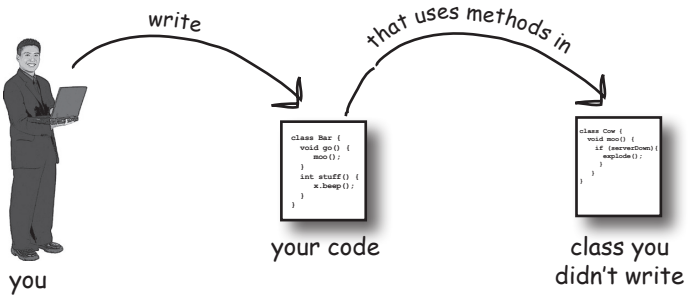
Something's wrong!

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

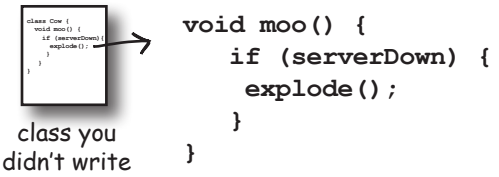
```
File Edit Window Help SayWhat?

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

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

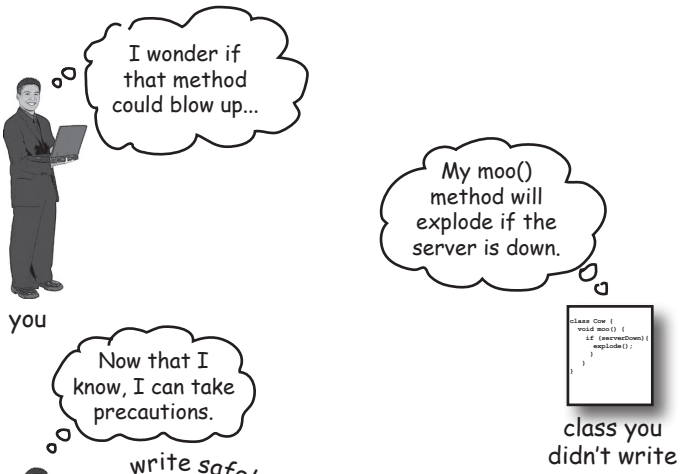
1. **Let's say you want to call a method in a class that you didn't write.**



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



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



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

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

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

So, how does a method tell you it might throw an exception? You find a **`throws`** clause in the risky method's declaration.

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

getSequencer

```
public static Sequencer getSequencer()
                              throws MidiUnavailableException
```

Obtains the default `Sequencer`, connected to a default device. The returned `Sequencer` instance is connected to the default `Synthesizer`, as returned by `getSynthesizer()`. If there is no `Synthesizer` available, or the default `Synthesizer` cannot be opened, the `sequencer` is connected to the default `Receiver`, as returned by `getReceiver()`. The connection is made by retrieving a `Transmitter` instance from the `Sequencer` and setting its `Receiver`. Closing and re-opening the sequencer will restore the connection to the default device.

This method is equivalent to calling `getSequencer(true)`.

If the system property `javax.sound.midi.Sequencer` is defined or it is defined in the file "sound.properties", it is used to identify the default sequencer. For details, refer to the `class description`.

Returns:

the default sequencer, connected to a default Receiver

Throws:

`MidiUnavailableException` - if the sequencer is not available due to resource restrictions, or there is no `Receiver` available by any installed `MidiDevice`, or no sequencer is installed in the system

See Also:

`getSequencer(boolean)`, `getSynthesizer()`, `getReceiver()`

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

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

Risky methods that could fail at runtime declare the exceptions that might happen using "throws SomeKindOfException" on their method declaration.

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

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

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

Dear Compiler,

I know I'm taking a risk here, but don't you think it's worth it? What should I do?

Signed, Geeky in Waikiki

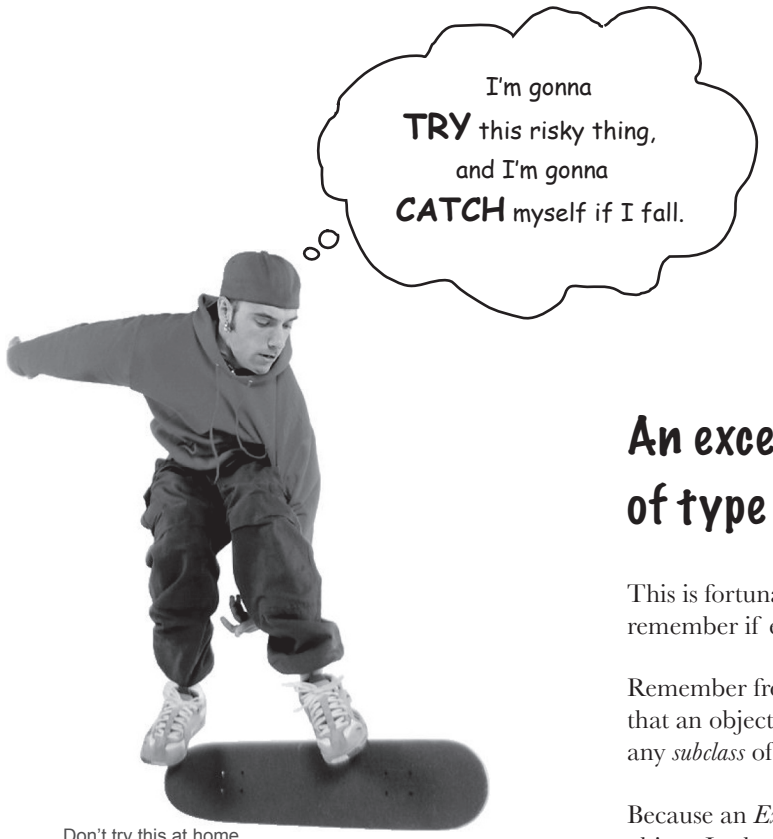
Dear Geeky,

Life is short (especially on the heap). Take the risk. Try it. But just in case things don't work out, be sure to catch any problems before all hell breaks loose.

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

public class MusicTest1 {

  public void play() {
    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      System.out.println("Successfully got a sequencer");
    } catch(MidiUnavailableException e) {
      System.out.println("Bummer");
    }
  }

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

Put the risky thing in a "try" block. It's the "risky" getSequencer method that might throw an exception.

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



Don't try this at home.

An exception is an object... of type Exception

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

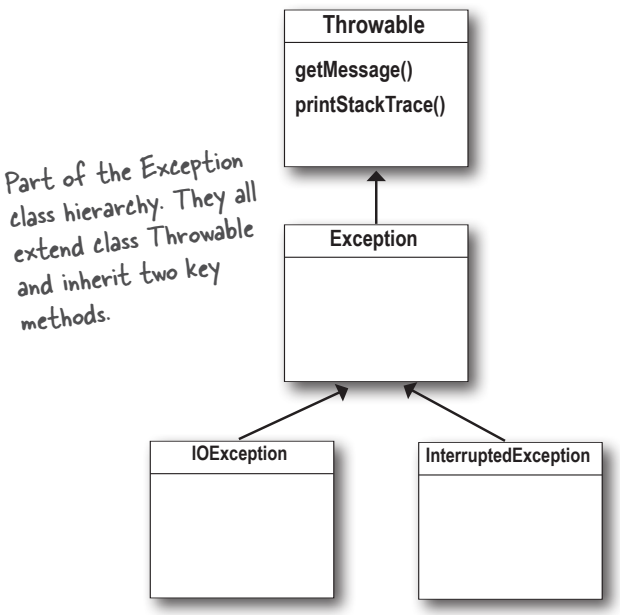
Remember from the polymorphism chapters (7 and 8) that an object of type Exception *can* be an instance of any *subclass* of Exception.

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

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

It's just like declaring a method argument.

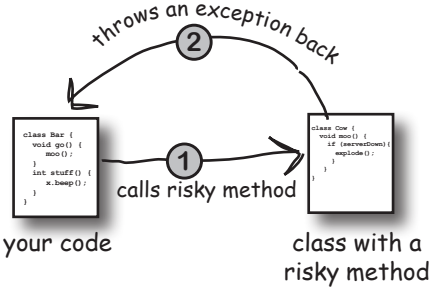
This code runs only if an Exception is thrown.



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

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

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



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

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

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

① Risky, exception-throwing code:

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

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

Create a new Exception object and throw it.

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

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

② Your code that *calls* the risky method:

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

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

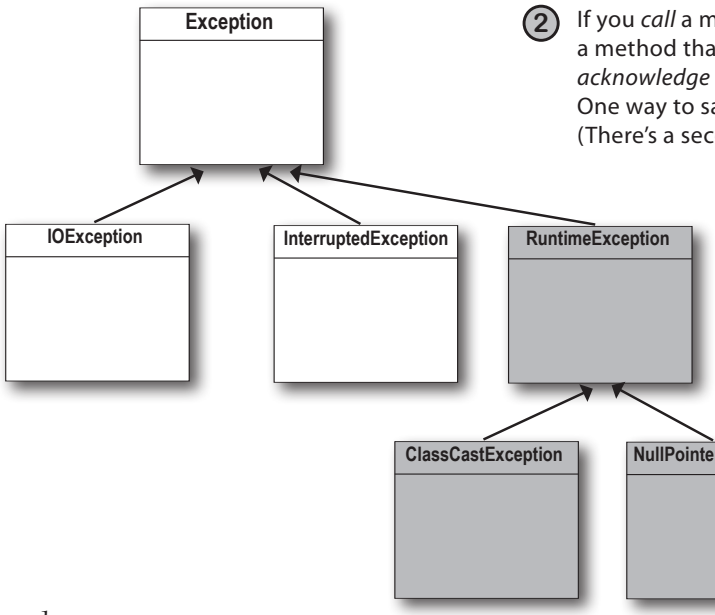
The compiler checks for everything except RuntimeExceptions.

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

The compiler guarantees:

1. If you *throw* an exception in your code, you *must* declare it using the *throws* keyword in your method declaration.

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



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

there are no Dumb Questions

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

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

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

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

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

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

BULLET POINTS

- A method can throw an exception when something fails at runtime.
- An exception is always an object of type Exception. (This, as you remember from the polymorphism chapters (7 and 8), means the object is from a class that has Exception somewhere up its inheritance tree.)
- The compiler does NOT pay attention to exceptions that are of type **RuntimeException**. A RuntimeException does not have to be declared or wrapped in a try/catch (although you're free to do either or both of those things).
- All Exceptions the compiler cares about are called "checked exceptions," which really means *compiler*-checked exceptions. Only RuntimeExceptions are excluded from compiler checking. All other exceptions must be acknowledged in your code.
- A method throws an exception with the keyword **`throw`**, followed by a new exception object:

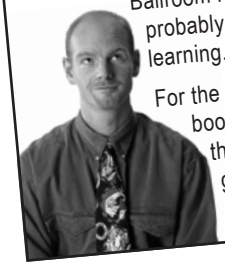
```
throw new NoCaffeineException();
```

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

Metacognitive tip

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

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



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

Sharpen your pencil

Which of these do you think might throw an exception that the compiler should care about? These are things that you CAN'T control in your code. We did the first one.

(Because it was the easiest.)

→ Yours to solve.

Things you want to do	What might go wrong
✓ Connect to a remote server	The server is down
__ Access an array beyond its length	
__ Display a window on the screen	
__ Retrieve data from a database	
__ See if a text file is where you *think* it is	
__ Create a new file	
__ Read a character from the command line	

Flow control in try/catch blocks

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

If the try succeeds

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

```
try {
    Foo f = x.doRiskyThing();
    int b = f.getNum();

} catch (Exception e) {
    System.out.println("failed");
}
System.out.println("We made it!");
```

First the try block runs, then the code below the catch runs.

The code in the catch block never runs.

```
File Edit Window Help RiskAll
%java Tester
We made it!
```

If the try fails

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

```
try {
    Foo f = x.doRiskyThing();
    int b = f.getNum();

} catch (Exception e) {
    System.out.println("failed");
}
System.out.println("We made it!");
```

The try block runs, but the call to doRiskyThing() throws an exception, so the rest of the try block doesn't run.

The catch block runs, then the method continues on.

The rest of the try block never runs, which is a Good Thing because the rest of the try depends on the success of the call to doRiskyThing().

```
File Edit Window Help RiskAll
%java Tester
failed
We made it!
```

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 e) {
  e.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 e) {
  e.printStackTrace();
  turnOvenOff();
}
```

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

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

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

Sharpen your pencil

→ Yours to solve.

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");
  }
}

class ScaryException extends Exception {
}
```

Output when `test = "no"`

Output when `test = "yes"`

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

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

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

Catching multiple exceptions

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

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

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

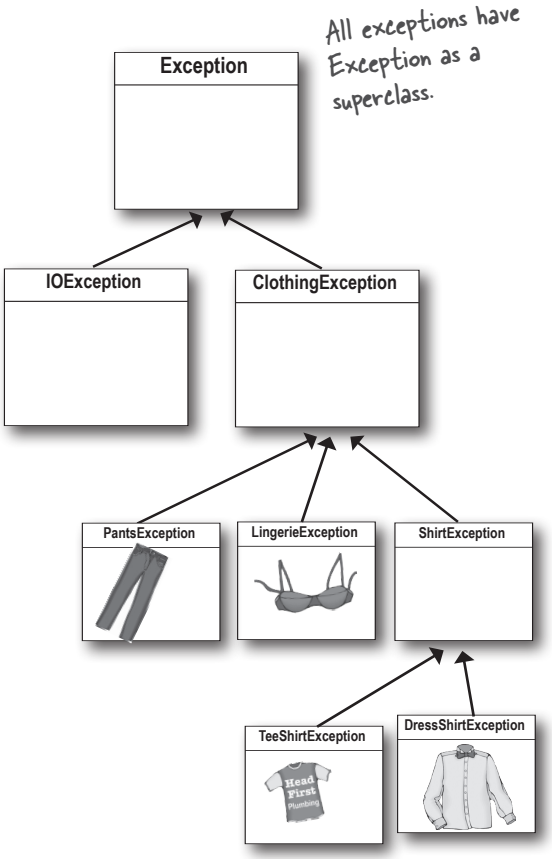
```
public class WashingMachine {
  public void go() {
    Laundry laundry = new Laundry();
    try {
      laundry.doLaundry();
    } catch (PantsException pex) {
      // recovery code

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

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

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

Exceptions are polymorphic

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



① You can DECLARE exceptions using a superclass 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 superclass of the exception thrown.

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

Can catch any ClothingException subclass

```
try {
  laundry.doLaundry();
} catch(ShirtException shex) {
  // 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 superclass Exception in the catch clause, so that you'll be able to catch *any* exception that might be thrown.

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

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

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

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

```
try {
  laundry.doLaundry();

} catch (TeeShirtException tex) {
  // recovery from TeeShirtException

} catch (LingerieException lex) {
  // recovery from LingerieException

} catch (ClothingException cex) {
  // recovery from all others
}
```

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

All other ClothingExceptions are caught here.

Multiple catch blocks must be ordered from smallest to biggest

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

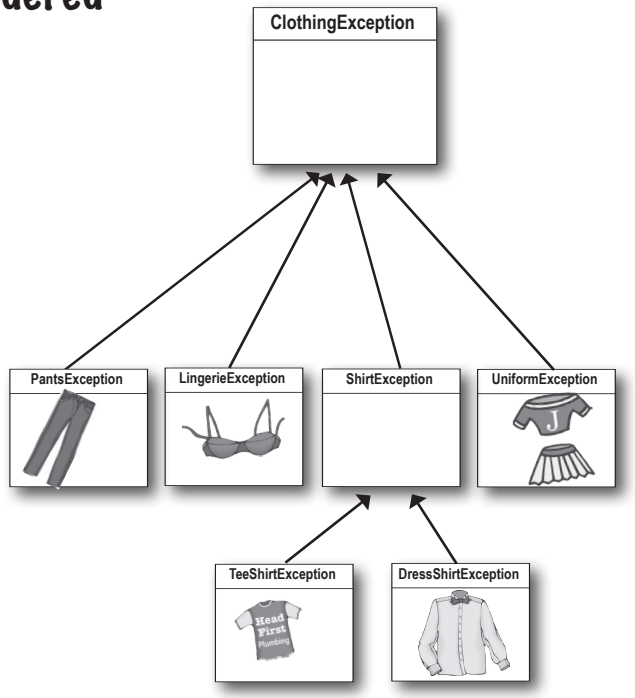
```
catch(TeeShirtException tex)
```

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

```
catch(ShirtException sex)
```

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

```
catch(ClothingException cex)
```



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

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

You can't put bigger baskets above smaller baskets

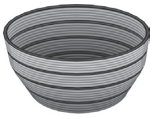
Well, you *can*, but it won't compile. Catch blocks are not like overloaded methods where the best match is picked. With catch blocks, the JVM simply starts at the first one and works its way down until it finds a catch that's broad enough (in other words, high enough on the inheritance tree) to handle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

Don't do this!

```
try {
  laundry.doLaundry();
```


```
}  catch(ClothingException cex) {
  // recovery from ClothingException
```

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



```
}  catch(ShirtException shex) {
  // 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 (exceptions at the same level of the hierarchy tree, like PantsException and LingerieException) can be in any order, because they can't catch one another's exceptions.

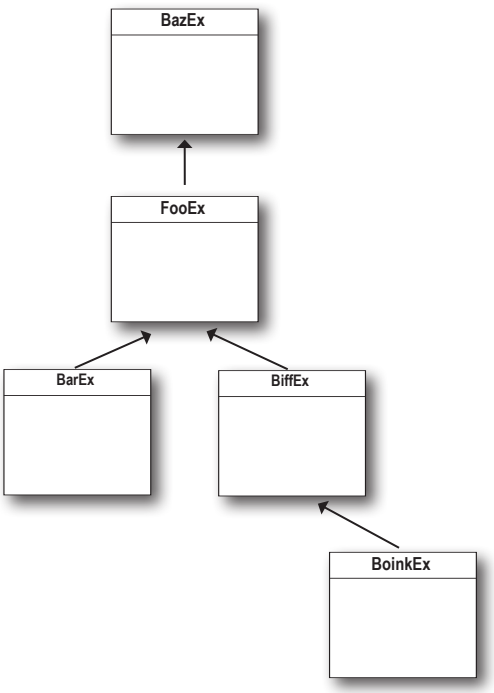
You could put ShirtException above LingerieException, and nobody would mind. Because even though ShirtException is a bigger (broader) type because it can catch other classes (its own subclasses), ShirtException can't catch a LingerieException, so there's no problem.

Sharpen your pencil

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
}
```

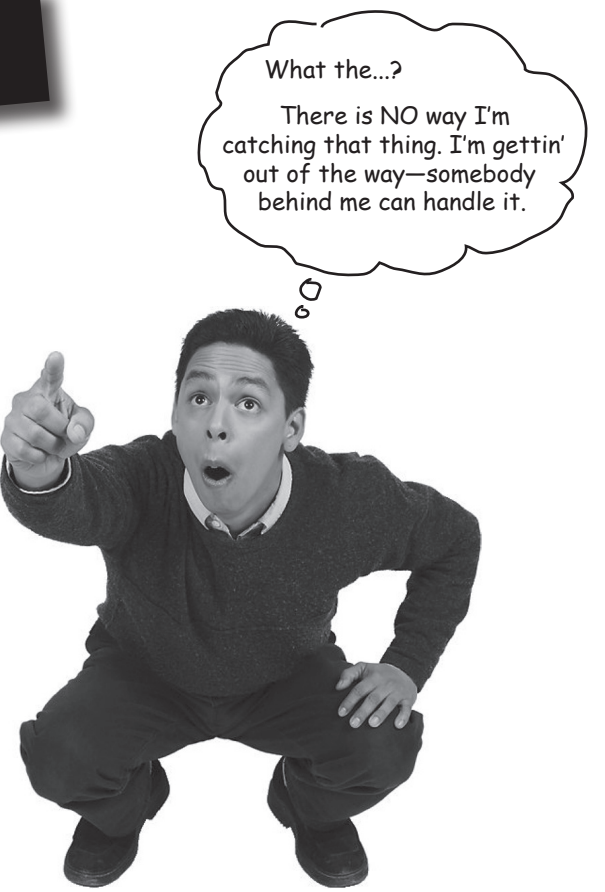
Your task is to create two different *legal* try/catch structures (similar to the one above left) to accurately represent the class diagram shown on the left. Assume ALL of these exceptions might be thrown by the method with the try block.



Both answers on page 457.

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

just duck it



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

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

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

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

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

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

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

Ducking (by declaring) only delays the inevitable

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

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

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

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

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

1 doLaundry() throws a ClothingException

main() calls foo()

foo() calls doLaundry()

doLaundry() is running and throws a ClothingException

2 foo() ducks the exception

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

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

3 main() ducks the exception

foo() pops off the stack, 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 T-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();
  }
}
```

TROUBLE!!

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

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

Getting back to our music code...

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

```
public void play() {
  try {
    Sequencer sequencer = MidiSystem.getSequencer();
    System.out.println("Successfully got a sequencer");

  } catch(MidiUnavailableException e) {
    System.out.println("Bummer");
  }
}
```

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 fnf), 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?

2 You cannot put code between the try and the catch.

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

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

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

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

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

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

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

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

Code Kitchen

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

The rest of this chapter is optional; you can use Ready-Bake Code for all the music apps.

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

Making actual sound

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

You need FOUR things:

1 The thing that plays the music

2 The music to be played...a song.

3 The part of the Sequence that holds the actual information

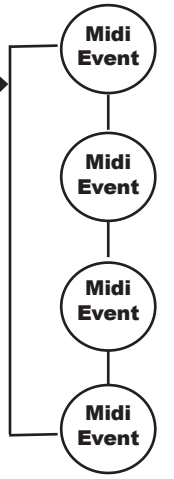
4 The actual music information: notes to play, how long, etc.

Sequencer plays **Sequence** has a **Track** holds **Midi Event**

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

The Sequence is **the song**, the single piece of music that the Sequencer will play.

For this book, all we want to do will fit on a single track, so for us one song needs only one track. This Track is where all the song data (MIDI information) lives.



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

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

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

And you need FIVE steps:

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

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

2. Make a new **Sequence**

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

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

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

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

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

Version 1: Your very first sound player app

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

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

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

  public void play() {
    try {
      Sequencer player = MidiSystem.getSequencer();
      player.open();

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

      Track track = seq.createTrack();

      ShortMessage msg1 = new ShortMessage();
      msg1.setMessage(NOTE_ON, 1, 44, 100);
      MidiEvent noteOn = new MidiEvent(msg1, 1);
      track.add(noteOn);

      ShortMessage msg2 = new ShortMessage();
      msg2.setMessage(NOTE_OFF, 1, 44, 100);
      MidiEvent noteOff = new MidiEvent(msg2, 16);
      track.add(noteOff);

      player.setSequence(seq);

      player.start();

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

Don't forget to import the midi package.

We're using a static import here so we can use the constants in the ShortMessage class.

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

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

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

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

Give the Sequence to the Sequencer (like selecting the song to play).

start() the Sequencer (play the song).

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 msg = new ShortMessage();
```

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

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

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

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

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

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

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

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

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

MIDI message: the heart of a MidiEvent

A MIDI message holds the part of the event that says *what* to do. It's 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.

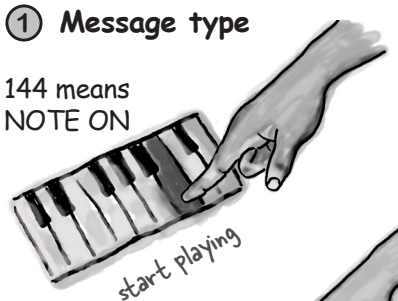
```
msg.setMessage(144,   1,   44,   100);
```

message type
channel
note to play
velocity

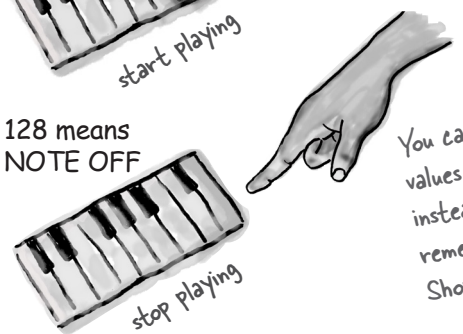
The last 3 args vary depending on the message type. This is a NOTE ON message, so the other args are for things the Sequencer needs to know in order to play a note.

1 Message type

144 means NOTE ON



128 means NOTE OFF



You can use the constant values in ShortMessage instead of having to remember the numbers, e.g., ShortMessage.NOTE_ON.

The Message says what to do; the MidiEvent says when to do it.

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.

```
msg.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.

```
msg.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

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.*;
import static javax.sound.midi.ShortMessage.*;

public class MiniMusicCmdLine {
  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);
    }
  }

  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();

      ShortMessage msg1 = new ShortMessage();
      msg1.setMessage(PROGRAM_CHANGE, 1, instrument, 0);
      MidiEvent changeInstrument = new MidiEvent(msg1, 1);
      track.add(changeInstrument);

      ShortMessage msg2 = new ShortMessage();
      msg2.setMessage(NOTE_ON, 1, note, 100);
      MidiEvent noteOn = new MidiEvent(msg2, 1);
      track.add(noteOn);

      ShortMessage msg3 = new ShortMessage();
      msg3.setMessage(NOTE_OFF, 1, note, 100);
      MidiEvent noteOff = new MidiEvent(msg3, 16);
      track.add(noteOff);

      player.setSequence(seq);
      player.start();

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

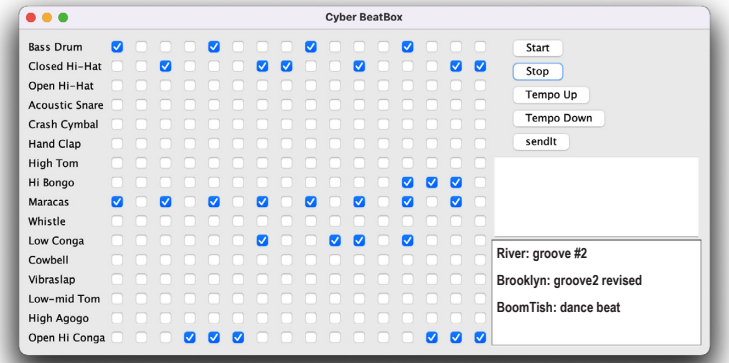
Run it with two int args from 0 to 127. Try these for starters:

```
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

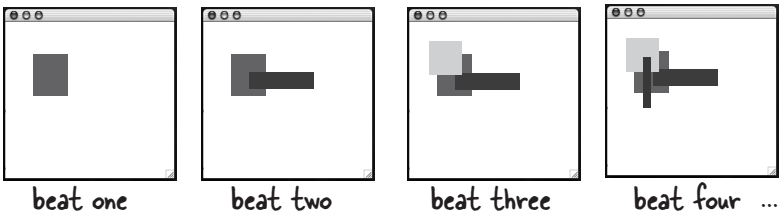
Chapter 17: the goal

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 (14, 15, and 16) will get us there.



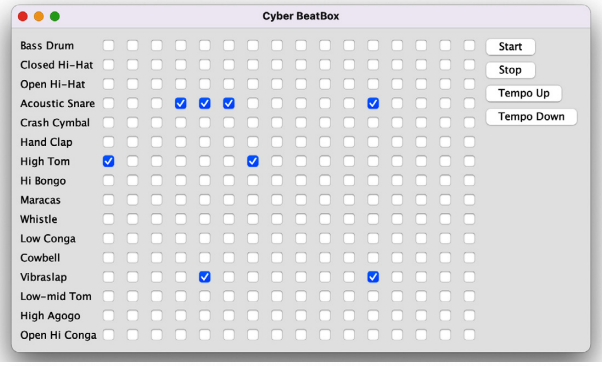
Chapter 14: MIDI events

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).



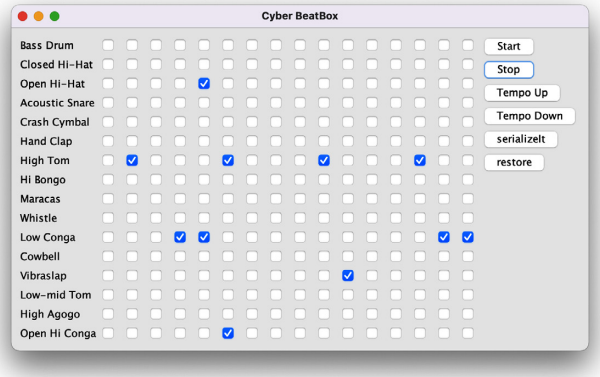
Chapter 15: Standalone BeatBox

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 16: 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.



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 throw only 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 a checked exception.

17. Runtime exceptions must be handled or declared.

Answers on page 457.

Code Magnets

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

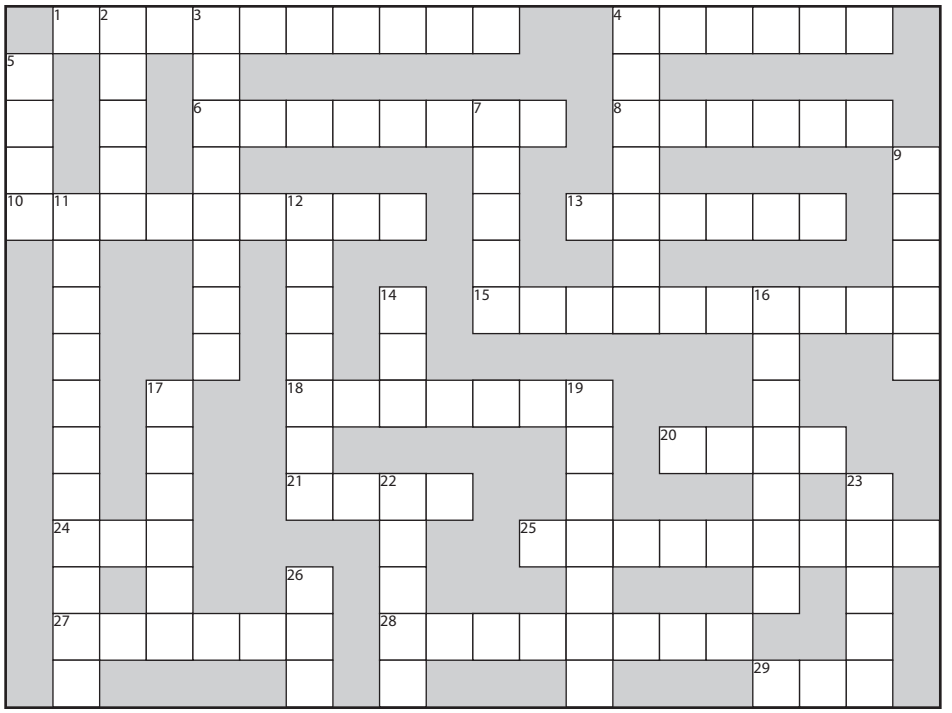
```
File  Edit  Window  Help  ThrowUp
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

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

Answers on page 458.

JavaCross

Answers on page 459.



You know what to do!

Across

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

Down

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

More Hints:

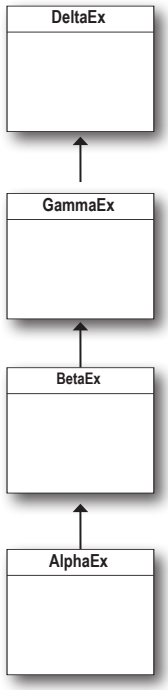
Across
6. A Java child
8. Start a method
13. Instead of declare
20. Also a type of collection
21. Quack
27. Starts a problem
28. Not Abstract

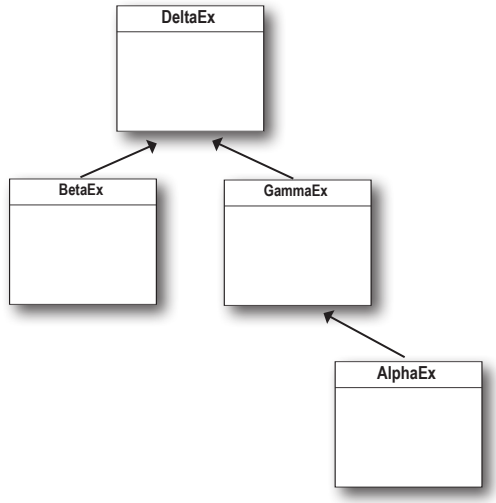
Down
2. Or a mouthwash
3. For _____ (not example)
5. Numbers . . .
9. Only public or default
16. _____ the family fortune
17. Not a 'getter'

Solution

Sharpen your pencil

(from page 440)





Exercise Solution

True or False (from page 454)

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.

Exercise Solutions

Code Magnets (from page 455)

```
class MyEx extends Exception { }

public class ExTestDrive {
  public static void main(String[] args) {
    String test = args[0];
    try {
      System.out.print("t");
      doRisky(test);
      System.out.print("o");
    } catch (MyEx e) {
      System.out.print("a");
    } finally {
      System.out.print("w");
    }
    System.out.println("s");
  }

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

    if ("yes".equals(t)) {
      throw new MyEx();
    }

    System.out.print("r");
  }
}
```

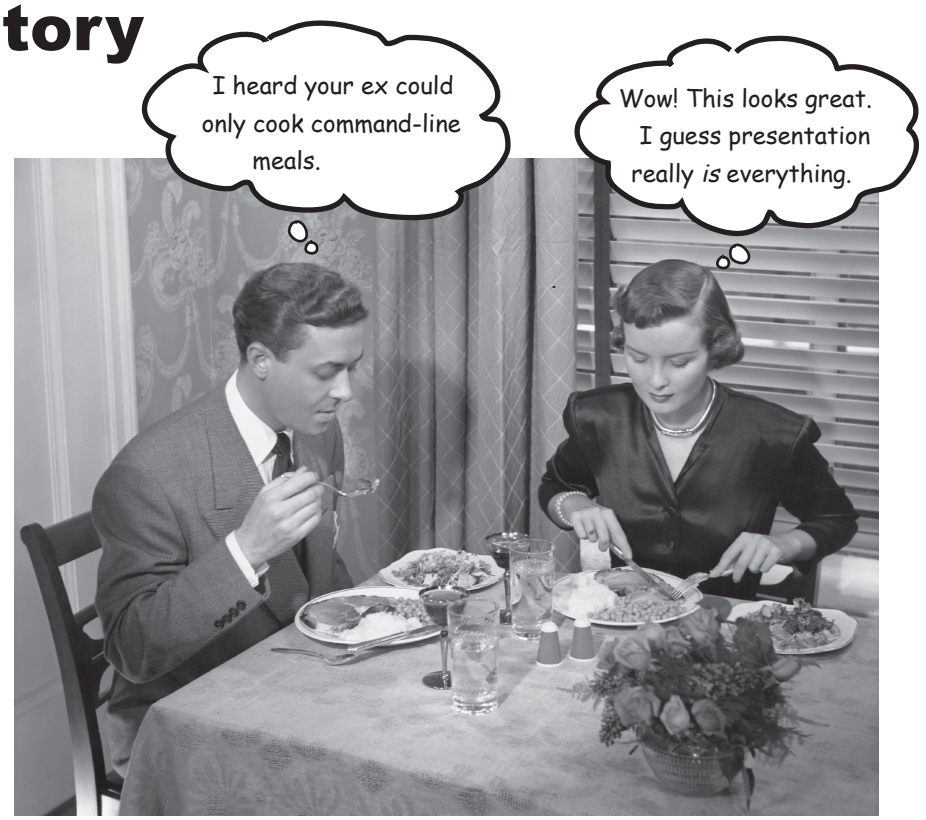
File Edit Window Help Chill

```
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

JavaCross (from page 456)

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

A Very Graphic Story



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

It all starts with a window

A JFrame is the object that represents a window on the screen. It's where you put all the interface things like buttons, check boxes, 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 an old Mac OS X:

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

Two issues!
1. Swing? This looks like Swing code. You're really gonna teach us Swing?
2. That window looks really old-fashioned.

She's asked a couple of really good questions. In a few pages we'll address these questions with an extra-special "No Dumb Questions."

Put widgets in the window

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

Making a GUI is easy:

1. Make a frame (a JFrame)

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

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

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

3. Add the widget to the frame

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

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

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

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

Your first GUI: a button on a frame

```
import javax.swing.*;

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

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

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(button);

    frame.setSize(300, 300);

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

don't forget to import this swing package

make a frame and a button

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

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

add the button to the frame's content pane

give the frame a size, in pixels

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

Let's see what happens when we run it:

```
%java SimpleGui1
```

Whoa! That's a Really Big Button.

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

But nothing happens when I click it...

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

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

We need two things:

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

When the user clicks, we want to know.

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

there are no Dumb Questions

Q: **I heard that nobody uses Swing anymore.**

A: There are other options, like JavaFX. But there are no clear winners in the endless and ongoing "Which approach should I use to make GUIs in Java?" debate. The good news is that if you learn a little Swing, that knowledge will help you whichever way you end up going. For example, if you want to do Android development, your Swing knowledge will make learning to code Android apps easier.

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 screens in this book use a number of "look and feels," including the default system look and feel (for macOS), the OS X ***Aqua*** look and feel, or the ***Metal*** (cross platform) look and feel.

Q: **Isn't Aqua really old?**

A: Yes, but we like it.

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):

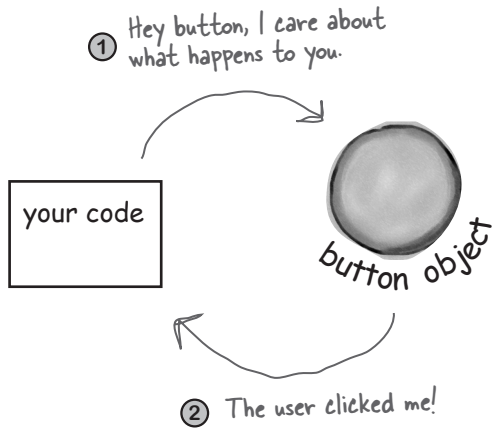
```
public void changeIt() {
  button.setText("I've been clicked!");
}
```

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 the Tempo" button, the user wants the slow-the-music-tempo action to occur. If it's a Send button on a chat client, the user wants the send-my-message action to happen. So the most straightforward event is when the user clicked the button, indicating they want an action to occur.

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

First, the button needs to know that we care.



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

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

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

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

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

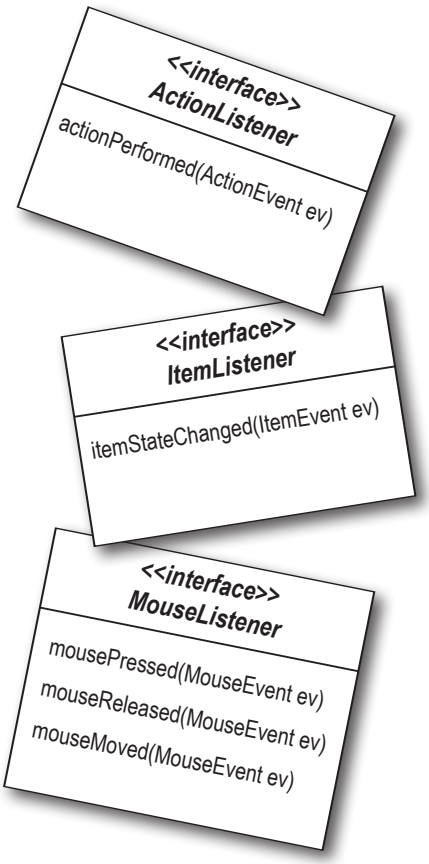
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through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

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

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

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

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



How the listener and source communicate:

The Listener

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

The Event Source

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

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

Getting a button's ActionEvent

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

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

public class SimpleGui2 implements ActionListener {
  private JButton button;

  public static void main(String[] args) {
    SimpleGui2 gui = new SimpleGui2();
    gui.go();
  }

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

    button.addActionListener(this);

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

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

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

1 Implement the interface. This says, "an instance of SimpleGui2 IS-A ActionListener."

(The button will give events only to ActionListener implementers.)

NOTE: You wouldn't usually make your main GUI class implement ActionListener like this; this is just the simplest way to get started. We'll see better ways of creating ActionListeners as we go through this chapter.

2 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!

3

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 here. Knowing the event happened is enough info for us.

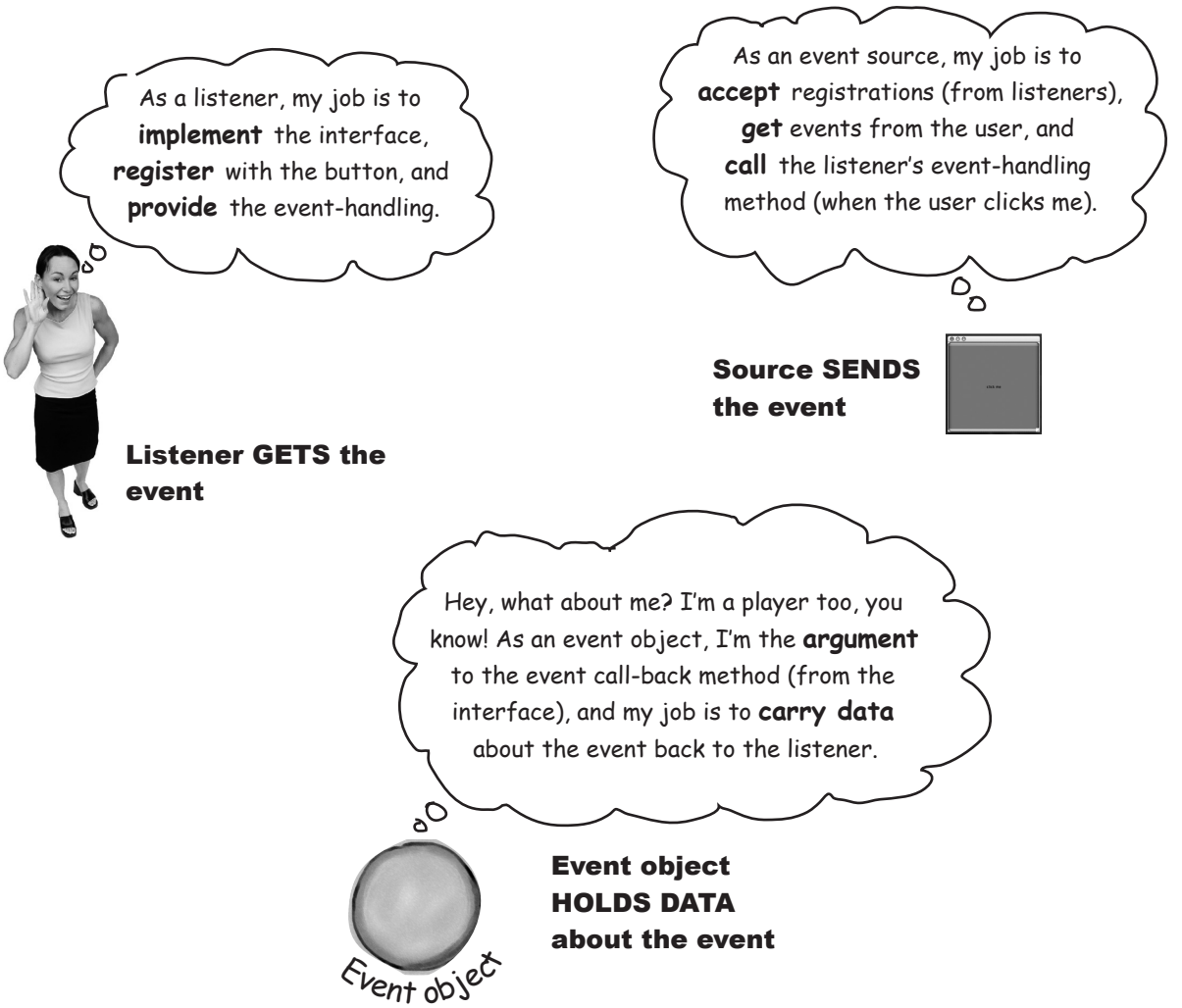
Listeners, Sources, and Events

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

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

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

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



there are no Dumb Questions

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

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

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

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

Or sometimes you might want to register the *same* listener with *multiple* objects. An on-screen 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) is the source of one or more events. Match the widgets with the events they might cause. Some widgets might be a source of more than one event, and some events can be generated by more than one widget.

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

→ Yours to solve.

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.

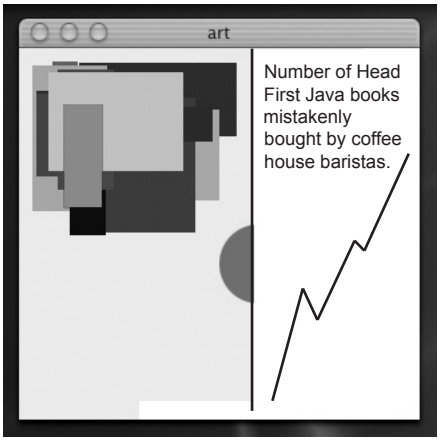
Three ways to put things on your GUI:

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.

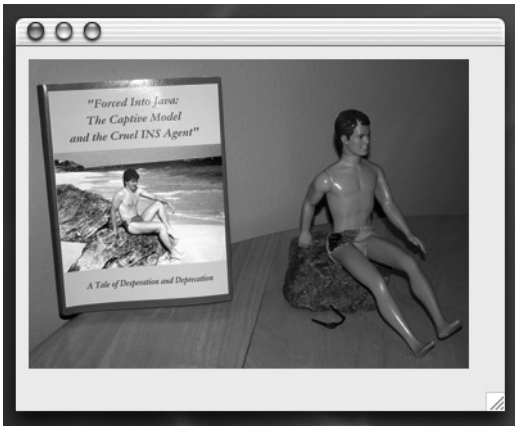


② Draw 2D graphics on a widget

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.

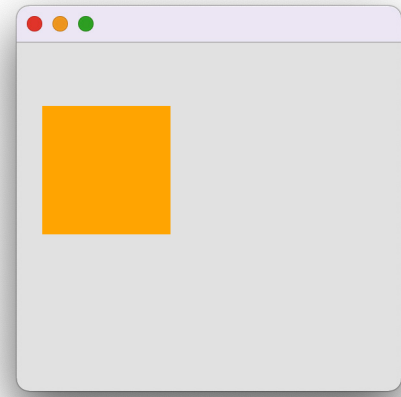
```
graphics.drawImage(myPic,10,10,this);
```

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.

A Swing frame with a custom drawing panel



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

class MyDrawPanel extends JPanel {

  public void paintComponent(Graphics g) {
    g.setColor(Color.orange);

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

You need both of these.

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

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

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

Fun things to do in paintComponent()

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

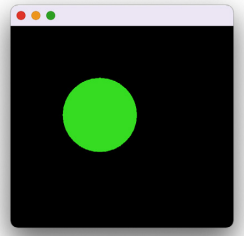
Display a JPEG

```
public void paintComponent(Graphics g) {

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

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

}
```

Your filename 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());

  Random random = new Random();
  int red = random.nextInt(256);
  int green = random.nextInt(256);
  int blue = random.nextInt(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)."

Earlier, we used Math.random, but now we know how to use the Java libraries we can use java.util.Random. It has a nextInt method that takes a max value and returns a number between 0 (inclusive) and this max value (not inclusive). In this case 0–256.

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

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

Behind every good Graphics reference is a Graphics2D object

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

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

So the parameter "g" IS-A Graphics object. This 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 paintComponent parameter ("g") straight from the method. But you can *cast* it with a new Graphics2D variable:

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

Methods you can call on a Graphics reference:

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

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

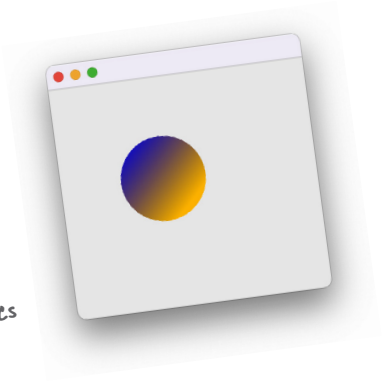
```
Graphics2D g2d = (Graphics2D) g;
```

Methods you can call on a Graphics2D reference:

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

(These are not complete method lists; check the API for more)

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



```
public void paintComponent(Graphics g) {

    Graphics2D g2d = (Graphics2D) g;

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

    g2d.setPaint(gradient);

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

}
```

It's really a Graphics2D object masquerading as a mere Graphics object.

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

Starting point

Starting color

Ending point

Ending color

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

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

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

    Random random = new Random();
    int red = random.nextInt(256);
    int green = random.nextInt(256);
    int blue = random.nextInt(256);
    Color startColor = new Color(red, green, blue);

    red = random.nextInt(256);
    green = random.nextInt(256);
    blue = random.nextInt(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 to 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, check box, 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.

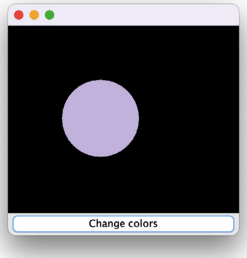
GRAPHICS

- 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;
  ```

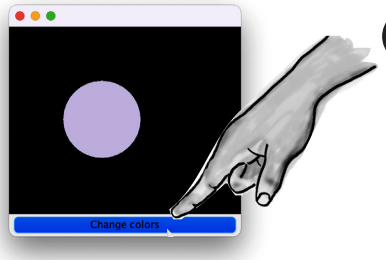
We can get an event.
We can paint graphics.
But can we paint graphics *when* we get an event?

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

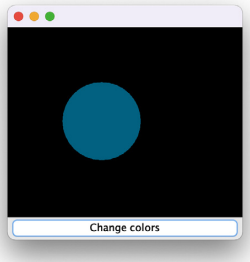


1 The frame is built with the two widgets (your drawing panel and a button). A listener is created and registered with the button. Then the frame is displayed, and it just waits for the user to click.



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

GUI layouts: putting more than one widget on a frame

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

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

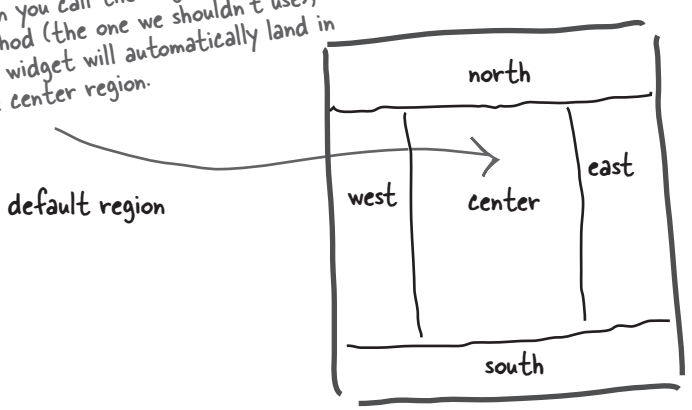
This isn't really the way you're supposed to do it (the one-arg add method).

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

```
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.

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

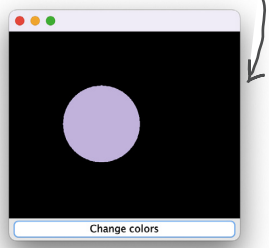


Sharpen your pencil

Given the pictures on page 477, write the code that adds the button and the panel to the frame.

→ Yours to solve.

The circle changes color each time you click the button.



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

Button is in the SOUTH region of the frame.

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

public class SimpleGui3 implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    SimpleGui3 gui = new SimpleGui3();
    gui.go();
  }

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

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

    MyDrawPanel drawPanel = new MyDrawPanel();

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

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

Add the listener (this) to the button.

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

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

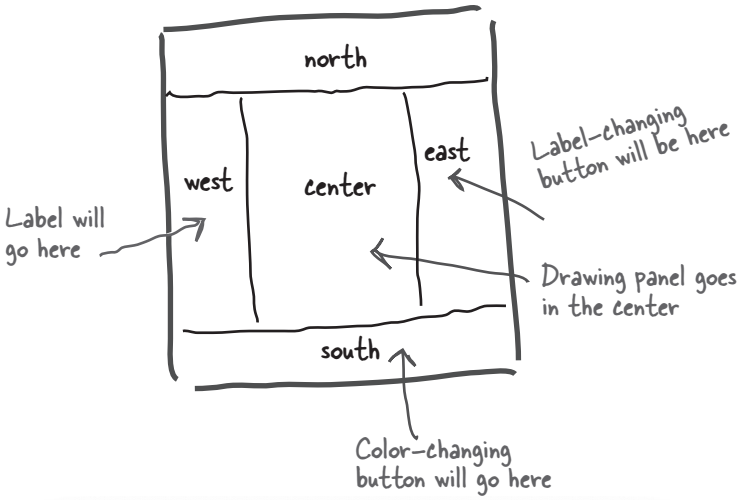
```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
     // Code to fill the oval with a random color
     // See page 365 for the code
   }

}
```

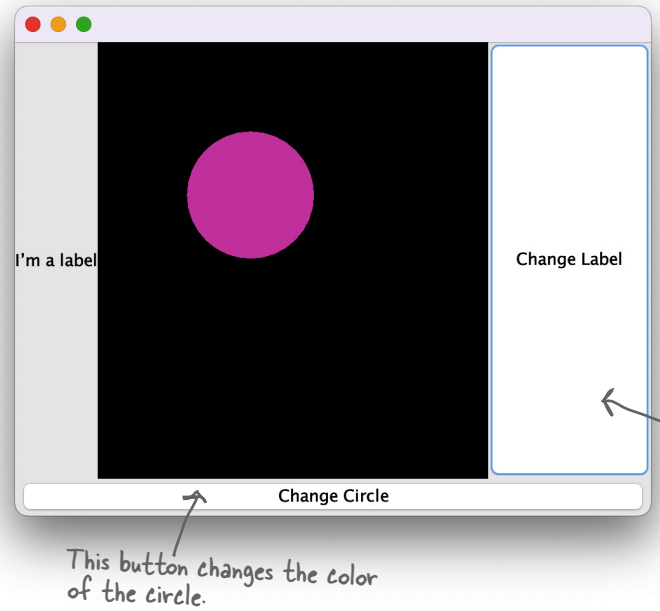
The drawing panel's paintComponent() method is called every time the user clicks.

Let's try it with TWO buttons

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

So now we need FOUR widgets





And we need to get TWO events

Uh-oh.

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

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

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);
    labelButton.addActionListener(this);
    // more gui code here ...
  }

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

Register the same listener with both buttons.

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

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

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

③ *Option Three*
Create two separate ActionListener classes

```
class MyGui {
  private JFrame frame;
  private JLabel label;

  void gui() {
    // code to instantiate the two listeners and register one
    // with the color button and the other with the label button
  }
}
```

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

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

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

Problem! This class has no reference to the variable "label."

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

There has got to be a better way!

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

Inner class to the rescue!

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

Simple inner class:

```
class MyOuterClass  {

   class MyInnerClass {
      void go() {
       }
    }

}
```

Inner class is fully enclosed by outer class

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

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

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

Inner class using an outer class variable

```
class MyOuterClass  {

  private int x;

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

} // close outer class
```

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

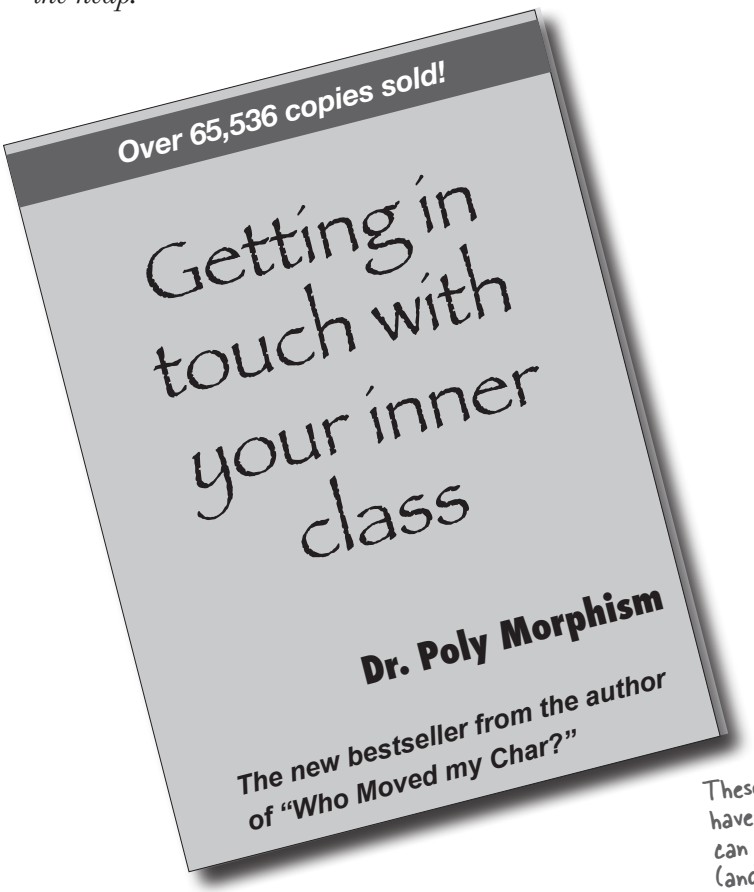
An inner class instance must be tied to an outer class instance*

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

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?

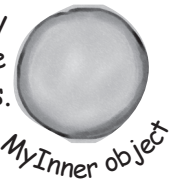
Can *any* arbitrary instance of the inner class access the methods and variables of *any* instance of the outer class? **No!**

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

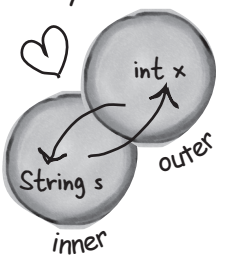


1. Make an instance of the outer class

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



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



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

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

How to make an instance of an inner class

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

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

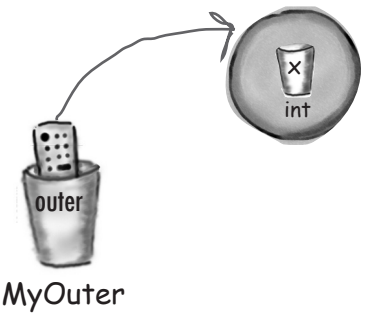
```
class MyOuter  {

  private int x;

  MyInner inner = new MyInner();

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

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

} // close outer class
```

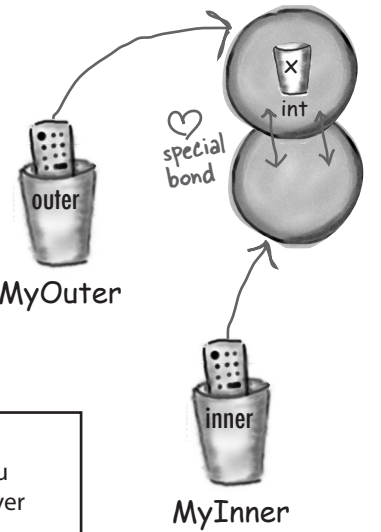
The outer class has a private instance variable "x."

Make an instance of the inner class.

Call a method on the inner class.

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



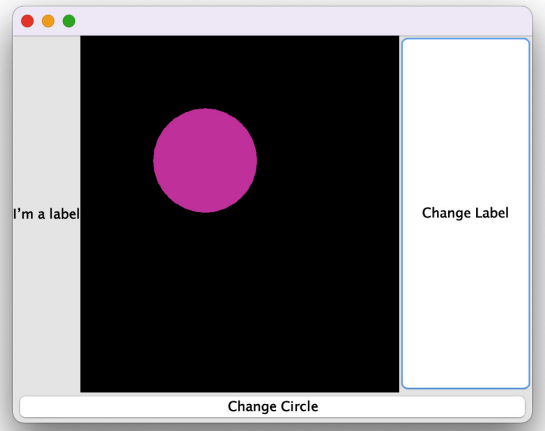


Sidebar

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();
  }
}
```

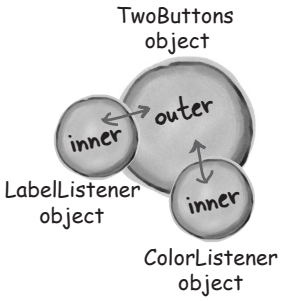
Now we can get the two-button code working




```
public class TwoButtons {
  private JFrame frame;
  private 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(500, 400);
    frame.setVisible(true);
  }

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

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

Much better: the main GUI class doesn't implement ActionListener now.

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



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

Inner class knows about "label."

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

Java Exposed

This week's interview:
Instance of an Inner Class

HeadFirst: What makes inner classes important?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

HeadFirst: Oh, well you just...hmmm. I think I'm getting it. You can always *implement* more than one interface, but you can *extend* only *one* class. You can be only 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 you become instantly bonded to the outer object, is that right?

Inner object: Yes, that's right.

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?

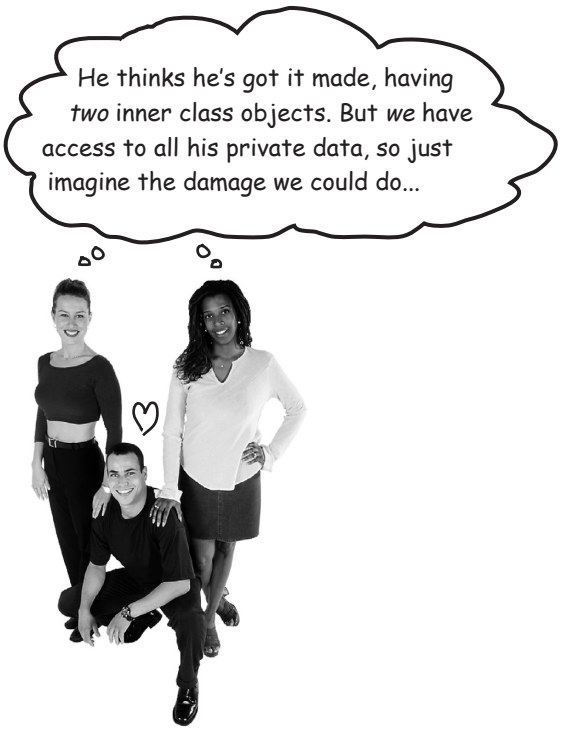
Inner object: All at the same time. There. Satisfied?

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.

Inner objects: You got it!

HeadFirst: One more question. I've heard that when lambdas came along, you were almost put out of a job?

Inner objects: Ouch, that really hurts! Okay, full disclosure, there are many cases for which a lambda is an easier to read, more concise way to do what I do. But inner classes have been around for a long time, and you're sure to encounter us in older code. Plus, those pesky lambdas aren't better at everything..



Can we take another look at that inner class code from a few pages back? It looks kind of clunky and hard to read.

Lambdas to the rescue! (again)

He's not wrong! One way to interpret the two highlighted lines of code would be:

"When the `labelButton` ActionListener gets an event, `setText("Ouch");`"

Not only are those two ideas separated from each other in the code, the inner class takes FIVE lines of code to invoke the setText method. And of course, everything we've said about the labelButton code is also true about the colorButton code.

Remember a few pages back we said that in order to implement the ActionListener interface you had provide code for its actionPerformed method? Hmmm...does that ring any bells?

```
...
  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();

    // code to add widgets, here
    frame.setSize(500, 400);
    frame.setVisible(true);
  }

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

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

ActionListener is a Functional Interface

Remember that a lambda provides an implementation for a functional interface's one and only *abstract* method.

Since ActionListener is a functional interface, you can replace the inner classes we saw on the previous page with lambda expressions.

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

    JButton labelButton = new JButton("Change Label");
    labelButton.addActionListener(event -> label.setText("Ouch!"));

    JButton colorButton = new JButton("Change Circle");
    colorButton.addActionListener(event -> frame.repaint());

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

    // code to add widgets, here
    frame.setSize(500, 400);
    frame.setVisible(true);
  }
```

These two pieces of highlighted code are the lambdas that replace the inner classes.

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

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

All of the inner class code is gone! Not needed! Bye bye.

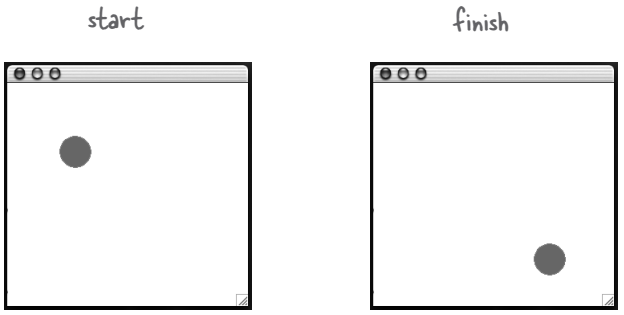
Lambdas, clearer and more concise

Well, maybe not quite yet, but once you get used to reading lambdas, we're pretty sure you'll agree that they make your code clearer.

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.)

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);
  }
}
```

Remember! The system invokes the paintComponent method; you don't have to.

Each time paintComponent() is called, the oval gets painted at a different location.

Sharpen your pencil

→ Answers on page 494.

But where do we get the new x and y coordinates?

And who calls repaint()?

See if you can **design a simple solution** to get the ball to animate from the top left of the drawing panel down to the bottom right. Our answer is on the next page, so don't turn this page until you're done!

Big Huge Hint: make the drawing panel an inner class.

Another Hint: don't put any kind of repeat loop in the paintComponent() method.

Write your ideas (or the code) here:

The complete simple animation code

```
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.TimeUnit;

public class SimpleAnimation {
  private int xPos = 70;
  private int yPos = 70;

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

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

    MyDrawPanel drawPanel = new MyDrawPanel();

    frame.getContentPane().add(drawPanel);
    frame.setSize(300, 300);
    frame.setVisible(true);

    for (int i = 0; i < 130; i++) {
      xPos++;
      yPos++;

      drawPanel.repaint();

      try {
        TimeUnit.MILLISECONDS.sleep(50);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

  class MyDrawPanel extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.green);
      g.fillOval(xPos, yPos, 40, 40);
    }
  }
}
```

Make two instance variables in the main GUI class, for the x and y coordinates of the circle.

Nothing new here. Make the widgets and put them in the frame.

This is where the action is!

Repeat this 130 times.

increment the x and y coordinates

Tell the panel to repaint itself (so we can see the circle in the new location).

Pause between repaints (otherwise it will move so quickly you won't SEE it move). Don't worry, you weren't supposed to already know this. We'll look at this in Chapter 17.

Now it's an inner class.

Use the continually updated x and y coordinates of the outer class.

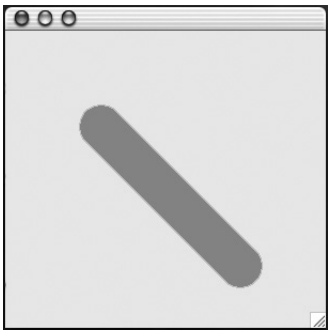
Did it work?

You might not have got the smooth animation that you expected.

What did we do wrong?

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

We need to erase what was already there! Or we might get trails.



Uh–oh. It didn't move...it smeared.

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.

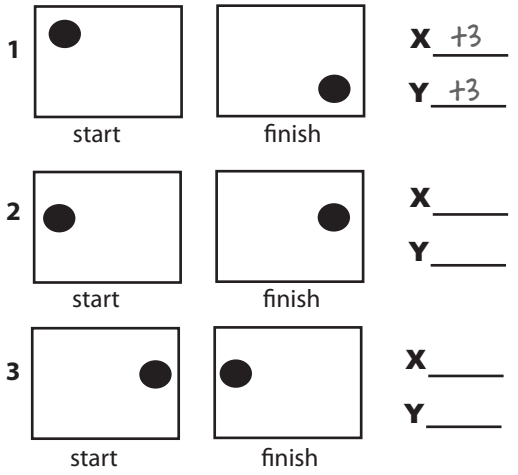
```
public void paintComponent(Graphics g) {
  g.setColor(Color.white);
  g.fillRect(0, 0, this.getWidth(), this.getHeight());

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

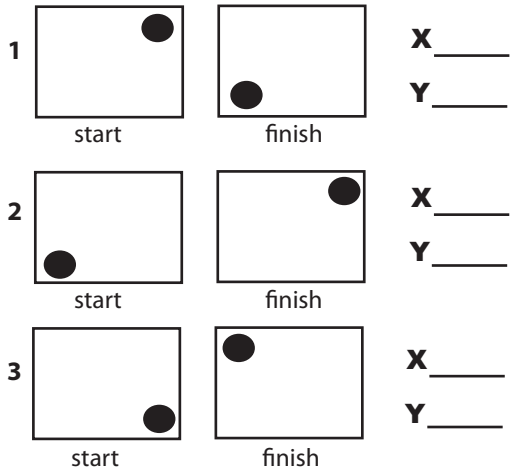
getWidth() and getHeight() are methods inherited from JPanel.

Sharpen your pencil (optional, just for fun)

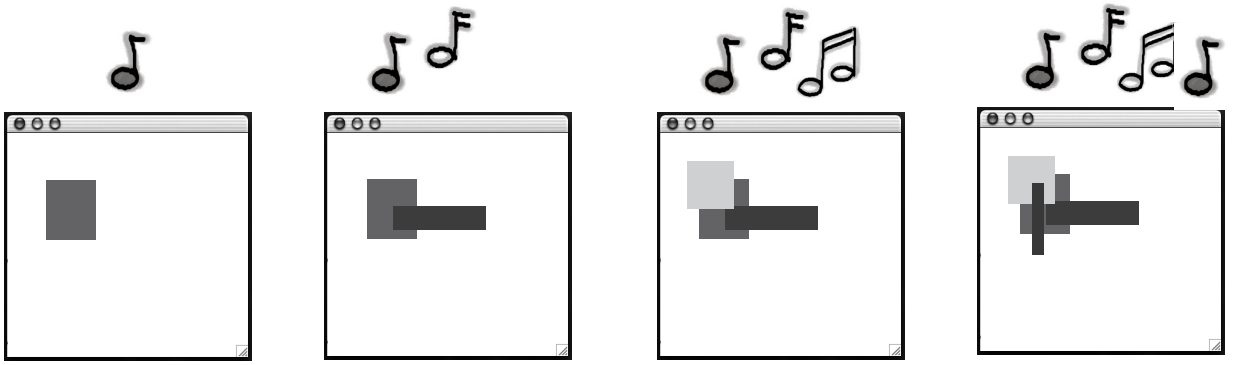
→ Yours to solve.

What changes would you make to the x and y coordinates to produce the animations below? (Assume the first example moves in 3-pixel increments.)





Code Kitchen



Let's make a music video. We'll use Java-generated random graphics that keep time with the music beats.

Along the way we'll register (and listen for) a new kind of non-GUI event, triggered by the music itself.

Remember, this part is all optional. But we think it's good for you. And you'll like it. And you can use it to impress people.

(OK, sure, it might work only on people who are really easy to impress, but still....)

Listening for a non-GUI event

OK, maybe not a music video, but we *will* make a program that draws random graphics on the screen with the beat of the music. In a nutshell, the program listens for the beat of the music and draws a random graphic rectangle with each beat.

That brings up some new issues for us. So far, we've listened for only GUI events, but now we need to listen for a particular kind of MIDI event. Turns out, listening for a non-GUI event is just like listening for GUI events: you implement a listener interface, register the listener with an event source, and 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 workaround. 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.

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 the previous 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 msg1 = new ShortMessage();
msg1.setMessage(NOTE_ON, 1, 44, 100);
MidiEvent noteOn = new MidiEvent(msg1, 1);
track.add(noteOn);

ShortMessage msg2 = new ShortMessage();
msg2.setMessage(NOTE_OFF, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(msg2, 16);
track.add(noteOff);
```

Things that have to happen for each event:

1. Make a message instance

```
ShortMessage msg = new ShortMessage();
```

2. Call setMessage() with the instructions

```
msg.setMessage(NOTE_ON, 1, instrument, 0);
```

3. Make a MidiEvent instance for the message

```
MidiEvent noteOn = new MidiEvent(msg, 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 command, int channel, int one, int two, int tick) {
  MidiEvent event = null;
  try {
    ShortMessage msg = new ShortMessage();
    msg.setMessage(command, channel, one, two);
    event = new MidiEvent(msg, tick);
  } catch (Exception e) {
    e.printStackTrace();
  }
  return event;
}
```

The four arguments for the message

The event 'tick' for WHEN this message should happen

Whoo! A method with five parameters.

Make the message and the event, using the method parameters.

Return the event (a MidiEvent all loaded up with the message).

Version One: using 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.*;
import static javax.sound.midi.ShortMessage.*;

public class MiniMusicPlayer1 {
  public static void main(String[] args) {
    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();

      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      for (int i = 5; i < 61; i += 4) {
        track.add(makeEvent(NOTE_ON, 1, i, 100, i));
        track.add(makeEvent(NOTE_OFF, 1, i, 100, i + 2));
      }

      sequencer.setSequence(seq);
      sequencer.setTempoInBPM(220);
      sequencer.start();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static MidiEvent makeEvent(int cmd, int chnl, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage msg = new ShortMessage();
      msg.setMessage(cmd, chnl, one, two);
      event = new MidiEvent(msg, tick);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return event;
  }
}
```

Don't forget the imports.

Make (and open) a sequencer.

Make a sequence and a track.

Make a bunch of events to make the notes keep going up (from piano note 5 to piano note 61).

Call our new makeEvent() method to make the message and event; then add the result (the MidiEvent returned from makeEvent()) to the track. These are NOTE ON and NOTE OFF pairs.

Start it running

Version Two: registering and getting ControllerEvents

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

public class MiniMusicPlayer2 {
  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(event -> System.out.println("la"), eventsIWant);

      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      for (int i = 5; i < 60; i += 4) {
        track.add(makeEvent(NOTE_ON, 1, i, 100, i));

        track.add(makeEvent(CONTROL_CHANGE, 1, 127, 0, i));

        track.add(makeEvent(NOTE_OFF, 1, i, 100, i + 2));
      }

      sequencer.setSequence(seq);
      sequencer.setTempoInBPM(220);
      sequencer.start();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static MidiEvent makeEvent(int cmd, int chnl, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage msg = new ShortMessage();
      msg.setMessage(cmd, chnl, one, two);
      event = new MidiEvent(msg, tick);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return event;
  }
}
```

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 care about one event, #127.

Each time we get the event, we'll print "la" to the command line. We're using a lambda expression here to handle this ControllerEvent.

Here's how we pick up the beat—we insert our OWN ControllerEvent (CONTROL_CHANGE) with an argument for event number #127. This event will do NOTHING! We put it in JUST so that we can get an event each time a note is played. In other words, its sole purpose is so that something will fire that WE can listen for (we can't listen for NOTE ON/OFF events). 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.

Code that's different from the previous version is highlighted in gray (and we've moved the code out of the main() method into its own go() method).

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 and 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:

The drawing panel is a listener.

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

  private boolean msg = false;

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

  public void paintComponent(Graphics g) {
    if (msg) {
      int r = random.nextInt(250);
      int gr = random.nextInt(250);
      int b = random.nextInt(250);

      g.setColor(new Color(r, gr, b));

      int height = random.nextInt(120) + 10;
      int width = random.nextInt(120) + 10;

      int xPos = random.nextInt(40) + 10;
      int yPos = random.nextInt(40) + 10;

      g.fillRect(xPos, yPos, width, height);
      msg = false;
    }
  }
}
```

We set a flag to false, and we'll set it to true only when we get an event.

The event handler method (from the ControllerEvent listener interface). We're not using a lambda expression this time, because we want the Panel to listen to ControllerEvents.

We got an event, so we set the flag to true and call repaint()

We have to use a flag because OTHER things might trigger a repaint(), and we want to paint ONLY when there's a ControllerEvent.

The rest is code to generate a random color and paint a semirandom rectangle.

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 javax.swing.*;
import java.awt.*;
import java.util.Random;

import static javax.sound.midi.ShortMessage.*;

public class MiniMusicPlayer3 {
  private MyDrawPanel panel;
  private Random random = new Random();

  public static void main(String[] args) {
    MiniMusicPlayer3 mini = new MiniMusicPlayer3();
    mini.go();
  }

  public void setUpGui() {
    JFrame frame = new JFrame("My First Music Video");
    panel = new MyDrawPanel();
    frame.setContentPane(panel);
    frame.setBounds(30, 30, 300, 300);
    frame.setVisible(true);
  }

  public void go() {
    setUpGui();

    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();
      sequencer.addControllerEventListener(panel, new int[]{127});
      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      int note;
      for (int i = 0; i < 60; i += 4) {
        note = random.nextInt(50) + 1;
        track.add(makeEvent(NOTE_ON, 1, note, 100, i));
        track.add(makeEvent(CONTROL_CHANGE, 1, 127, 0, i));
        track.add(makeEvent(NOTE_OFF, 1, note, 100, i + 2));
      }

      sequencer.setSequence(seq);
      sequencer.start();
      sequencer.setTempoInBPM(120);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

```
  public static MidiEvent makeEvent(int cmd, int chnl, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage msg = new ShortMessage();
      msg.setMessage(cmd, chnl, one, two);
      event = new MidiEvent(msg, tick);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return event;
  }

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

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

    public void paintComponent(Graphics g) {
      if (msg) {
        int r = random.nextInt(250);
        int gr = random.nextInt(250);
        int b = random.nextInt(250);

        g.setColor(new Color(r, gr, b));

        int height = random.nextInt(120) + 10;
        int width = random.nextInt(120) + 10;

        int xPos = random.nextInt(40) + 10;
        int yPos = random.nextInt(40) + 10;

        g.fillRect(xPos, yPos, width, height);
        msg = false;
      }
    }
  }

}
```

Who Am I?

A bunch of Java hotshots, 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. _____

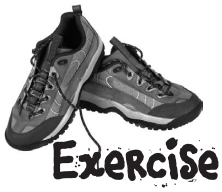
I'm typically bound to an instance. _____

The "g" in (Graphics g) is really of this class. _____

The method that gets paintComponent( ) rolling. _____

The package where most of the Swingers reside. _____

Answers on page 507.



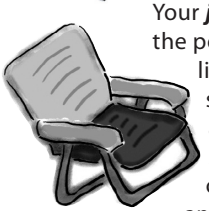
BE the Compiler

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

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

class InnerButton {
  private JButton button;

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

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

    button = new JButton("A");
    button.addActionListener();

    frame.getContentPane().add(
            BorderLayout.SOUTH, button);
    frame.setSize(200, 100);
    frame.setVisible(true);
  }

  class ButtonListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (button.getText().equals("A")) {
        button.setText("B");
      } else {
        button.setText("A");
      }
    }
  }
}
```

Answers on page 507.

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.*;
import java.util.concurrent.TimeUnit;
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 {
        TimeUnit.MILLISECONDS.sleep(50);
       } catch(Exception ex) { }
       }
   }
   class MyDrawP extends JPanel {
       public void paintComponent (Graphics
                                   _________) {
          ____________________________________;
          ____________________________________;
          ____________________________________;
          ____________________________________;
       }
   }
}
```

g.fillRect(x,y,x-500,y-250)
g.fillRect(x,y,500-x*2,250-y*2)
g.fillRect(500-x*2,250-y*2,x,y)
x++ g.fillRect(0,0,250,500)
y++ g.fillRect(0,0,500,250)

i++
i++, y++
i++, y++, x++

Animate frame = new Animate()
MyDrawP drawP = new MyDrawP()
ContentPane drawP = new ContentPane()

drawP.setSize(500,270)
frame.setSize(500,270)
panel.setSize(500,270)

g.setColor(blue)
g.setColor(white)
g.setColor(Color.blue)
g.setColor(Color.white)

g
draw
frame
panel

drawP.paint()
draw.repaint()
drawP.repaint()

Answers on page 508.

Exercise Solutions

Who Am I? (from page 504)

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

BE the Compiler (from page 505)

Once this code is fixed, it will create a GUI with a button that toggles between A and B when you click it.

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

class InnerButton {
  private JButton button;

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

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

    button = new JButton("A");
    button.addActionListener(new ButtonListener());

    frame.getContentPane().add(
            BorderLayout.SOUTH, button);
    frame.setSize(200, 100);
    frame.setVisible(true);
  }

  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (button.getText().equals("A")) {
        button.setText("B");
      } else {
        button.setText("A");
      }
    }
  }
}
```

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

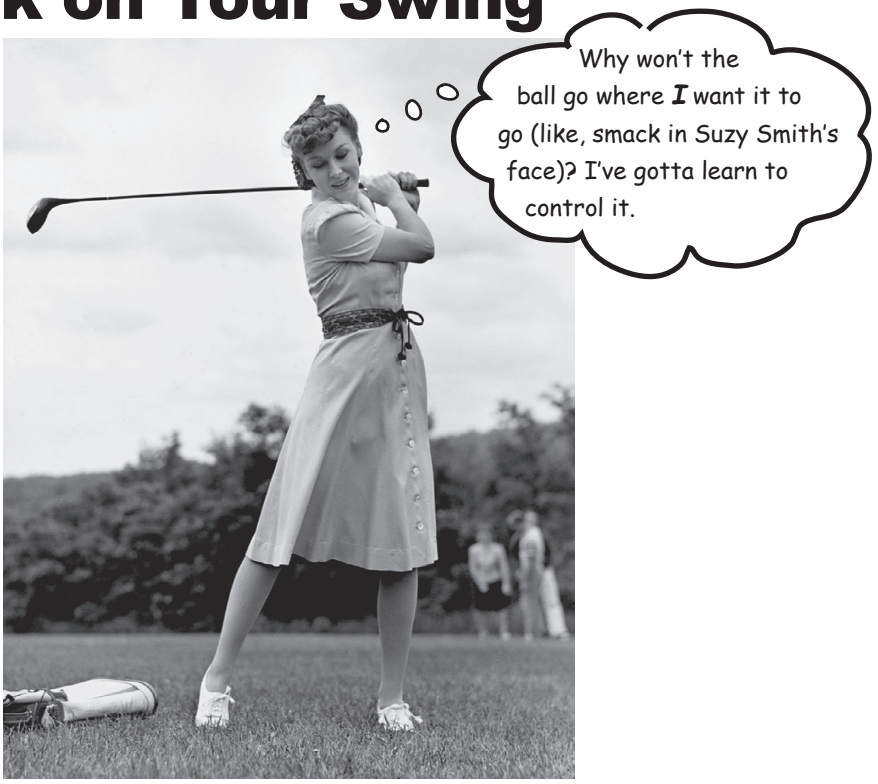
ActionListener is an interface; interfaces are implemented, not extended.

Pool Puzzle (from page 506)

```
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.TimeUnit;

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 {
        TimeUnit.MILLISECONDS.sleep(50);
      } catch(Exception ex) { }
    }
  }
  class MyDrawP extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.white);
      g.fillRect(0, 0, 500, 250);
      g.setColor(Color.blue);
      g.fillRect(x, y, 500-x*2, 250-y*2);
    }
  }
}
```

The Amazing, Shrinking, Blue Rectangle.

Work on Your Swing



Swing is easy. Unless you actually *care* where things end up on the screen. Swing code *looks* easy, but then you compile it, run it, look at it, and think, "hey, *that's* not supposed to go *there*." The thing that makes it *easy* to *code* is the thing that makes it *hard* to *control*—the **Layout Manager**. Layout Manager objects control the size and location of the widgets in a Java GUI. They do a ton of work on your behalf, but you won't always like the results. You want two buttons to be the same size, but they aren't. You want the text field to be three inches long, but it's nine. Or one. And *under* the label instead of *next* to it. But with a little work, you can get layout managers to submit to your will. Learning a little Swing will give you a head start for most GUI programming you'll ever do. Wanna write an Android app? Working through this chapter will give you a head start.

Swing components

Component is the more correct term for what we've been calling a *widget*. The *things* you put in a GUI. *The things a user sees and interacts with*. Text fields, buttons, scrollable lists, radio buttons, etc., are all components. In fact, they all extend **`javax.swing.JComponent`**.

A widget is technically a Swing Component. Almost everything you can stick in a GUI extends from javax.swing.JComponent.

Components can be nested

In Swing, virtually *all* components are capable of holding other components. In other words, *you can stick just about anything into anything else*. But most of the time, you'll add *user interactive* components such as buttons and lists into *background* components (often called containers) such as frames and panels. Although it's *possible* to put, say, a panel inside a button, that's pretty weird and won't win you any usability awards.

With the exception of JFrame, though, the distinction between *interactive* components and *background* components is artificial. A JPanel, for example, is usually used as a background for grouping other components, but even a JPanel can be interactive. Just as with other components, you can register for the JPanel's events including mouse clicks and keystrokes.

Four steps to making a GUI (review)

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);
```

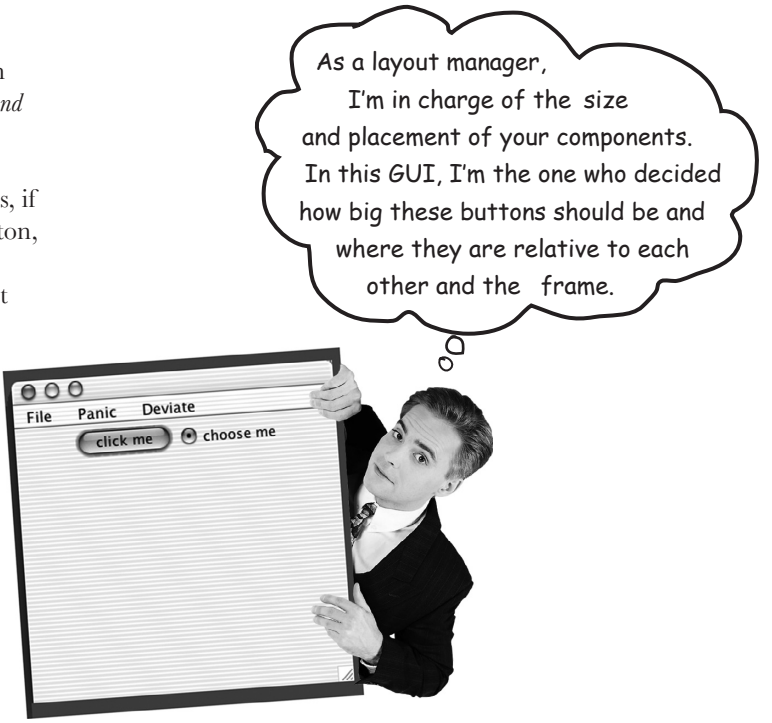
4. Display it (give it a size and make it visible)
```
frame.setSize(300,300);
frame.setVisible(true);
```

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.

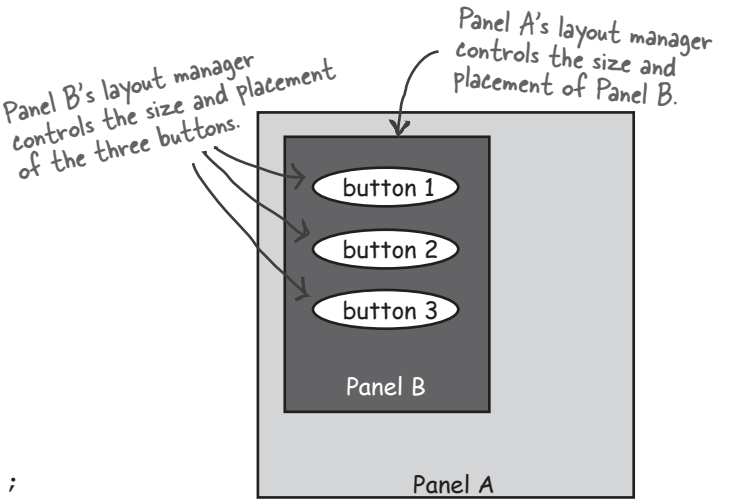


If a panel holds five things, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things (e.g., if any of those five things are panels or other containers that hold other things), then those *other* things are placed according to the layout manager of the thing holding them.

When we say *hold*, we really mean *add* as in, a panel *holds* a button because the button was *added* to the panel using something like:

```
myPanel.add(button);
```

Layout managers come in several flavors, and each background component can have its own layout manager. Layout managers have their own policies to follow when building a layout. For example, one layout manager might insist that all components in a panel must be the same size, arranged in a grid, while another layout manager might let each component choose its own size but stack them vertically. Here's an example of nested layouts:

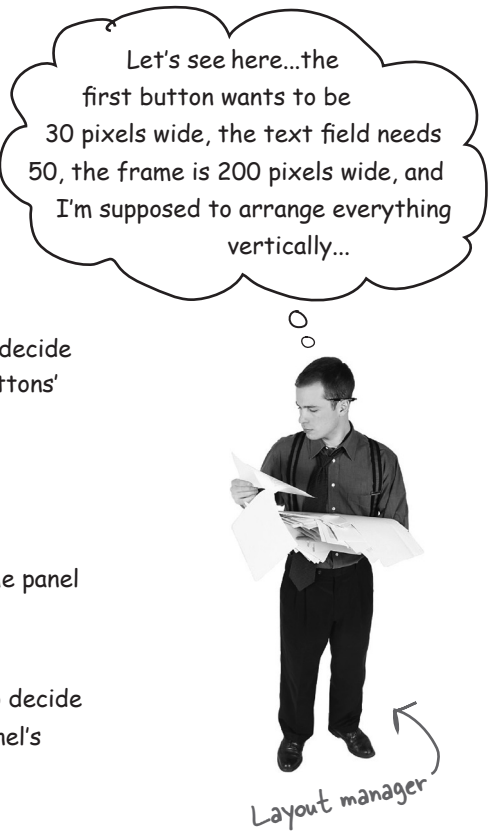



```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```

Panel A's layout manager has NOTHING to say about the three buttons. The hierarchy of control is only one level—Panel A's layout manager controls only the things added directly to Panel A, and does not control anything nested within those added components.

How does the layout manager decide?

Different layout managers have different policies for arranging components (like, arrange in a grid, make them all the same size, stack them vertically, etc.), but the components being 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.

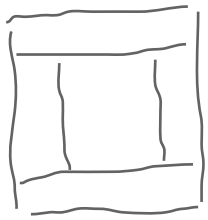
Different layout managers have different policies

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!**

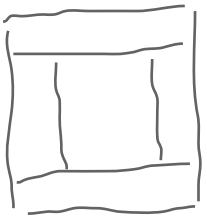


FlowLayout

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 arrange them horizontally, but usually we're just concerned with vertically). It's like a FlowLayout but instead of having automatic "component wrapping," you can insert a sort of "component return key" and **force** the components to start a new line.



BorderLayout cares about five regions: east, west, north, south, and center

Let's add a button to the east region:

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

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);
  }
}
```

BorderLayout is in the java.awt package.

Specify the region.

Brain Barbell

How did the BorderLayout manager come up with this size for the button?

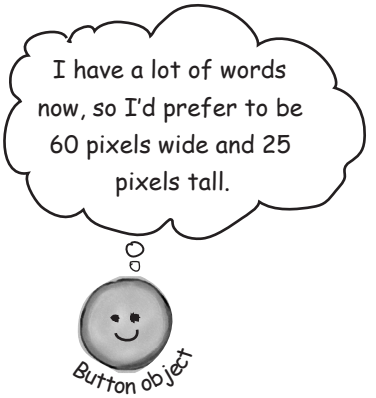
What are the factors the layout manager has to consider?

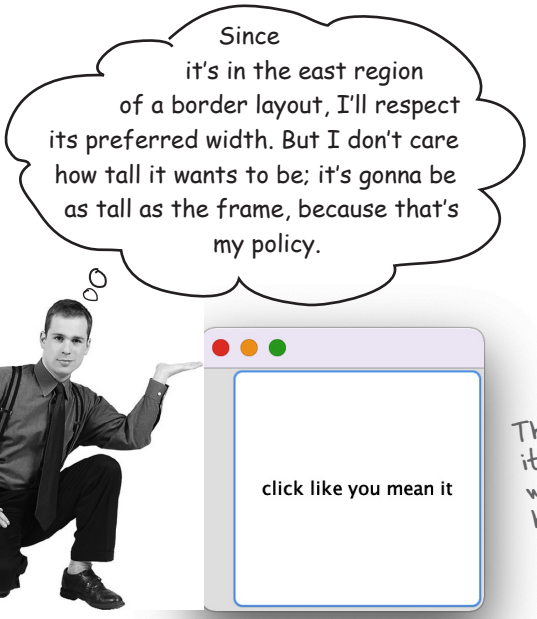
Why isn't it wider or taller?

Watch what happens when we give the button more characters...

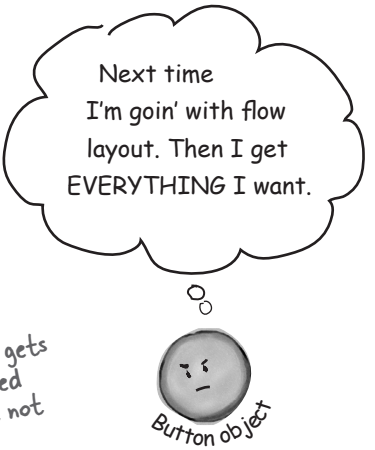
We changed only the text on the button.

```
public void go() {
  JFrame frame = new JFrame();
  JButton button = new JButton("click like you mean it");
  frame.getContentPane().add(BorderLayout.EAST, button);
  frame.setSize(200, 200);
  frame.setVisible(true);
}
```



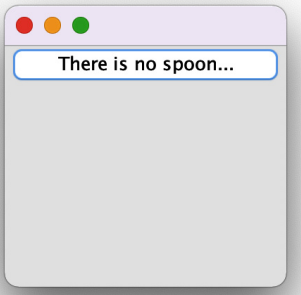


The button gets its preferred width, but not height.



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.

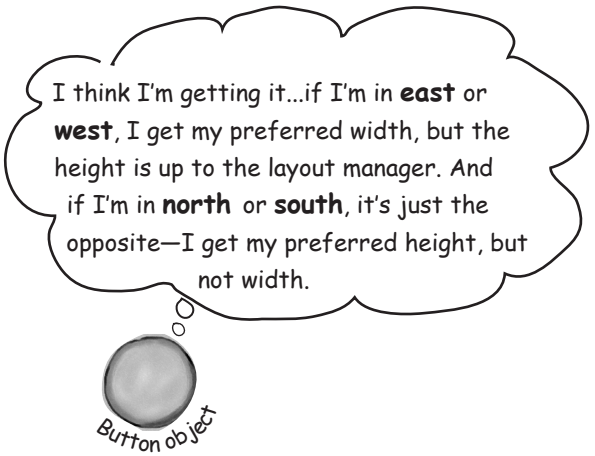
Now let's make the button ask to be taller

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.

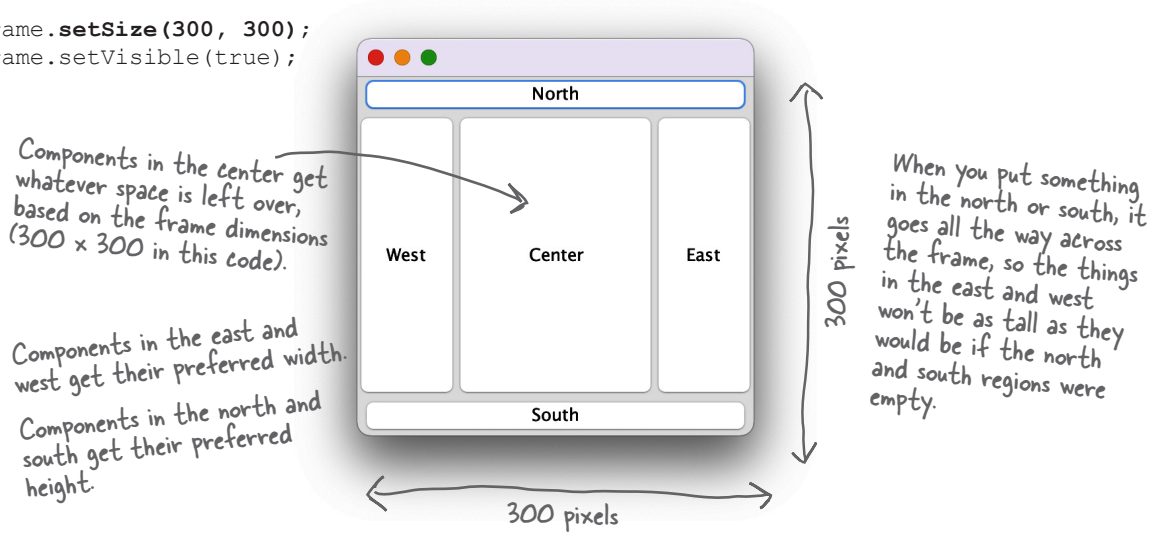


But what happens in the center region?

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);
}
```



FlowLayout cares about the flow of the components:

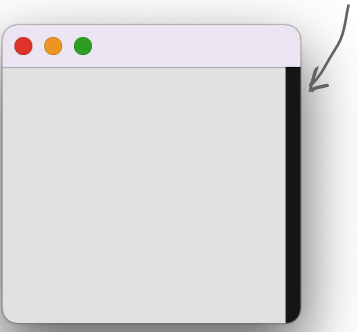
left to right, top to bottom, in the order they were added.

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 are 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.

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

public class Panel1 {

  public static void main(String[] args) {
    Panel1 gui = new Panel1();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(200, 200);
    frame.setVisible(true);
  }
}
```

The panel doesn't have anything in it, so it doesn't ask for much width in the east region.



Make the panel gray so we can see where it is on the frame.

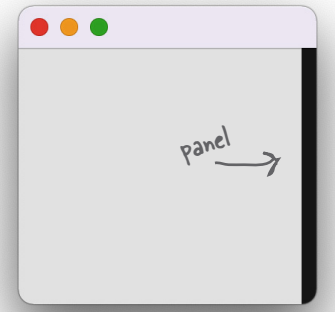
Let's add a button to the panel

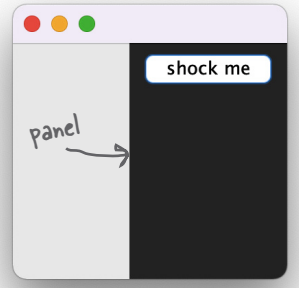
```
public void go() {
  JFrame frame = new JFrame();
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("shock me");
  panel.add(button);

  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(200, 200);
  frame.setVisible(true);
}
```

Add the button to the panel...

...and add the panel to the frame.

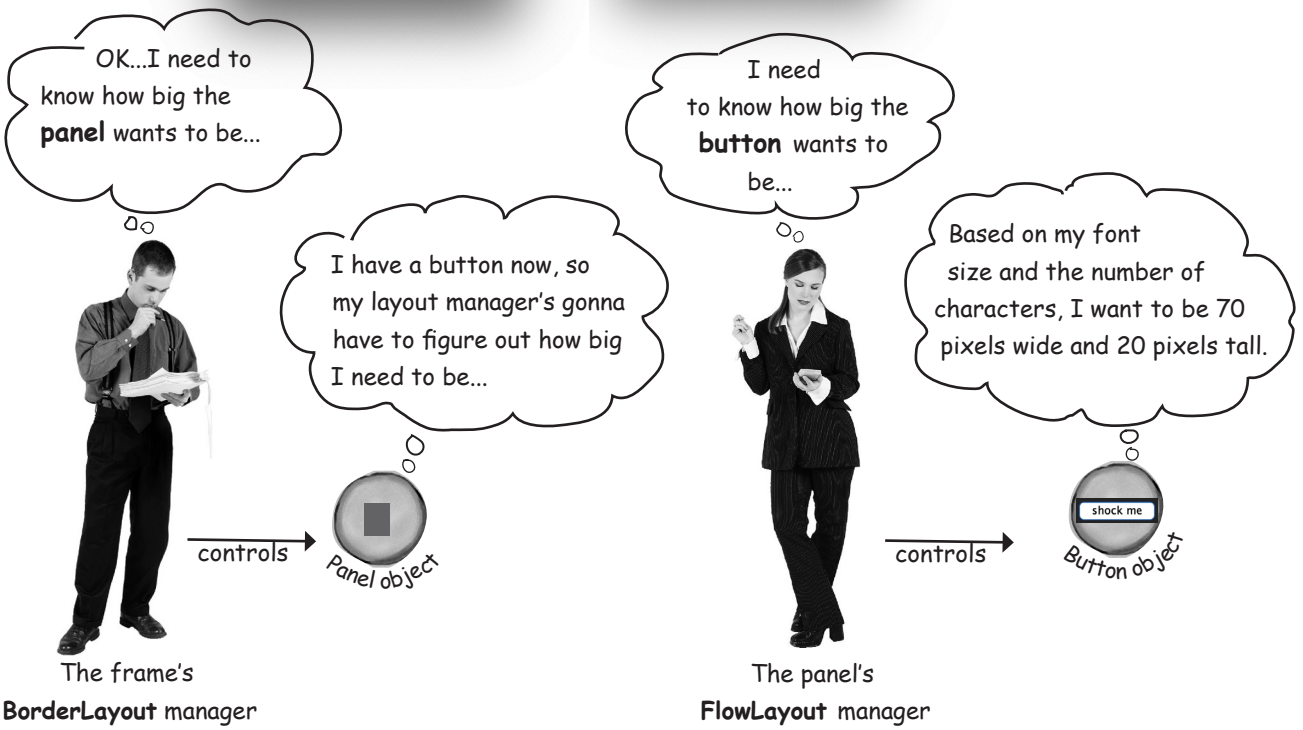
The panel's layout manager (flow) controls the button, and the frame's layout manager (border) controls the panel.





The panel expanded!

And the button got its preferred size in both dimensions, because the panel uses flow layout, and the button is part of the panel (not the frame).



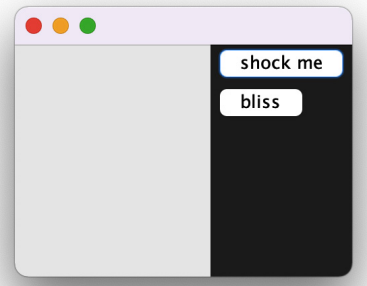
What happens if we add TWO buttons to the panel?

```
public void go() {
  JFrame frame = new JFrame();
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("shock me");
  JButton buttonTwo = new JButton("bliss");

  panel.add(button);
  panel.add(buttonTwo);

  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(250, 200);
  frame.setVisible(true);
}
```

Make TWO buttons

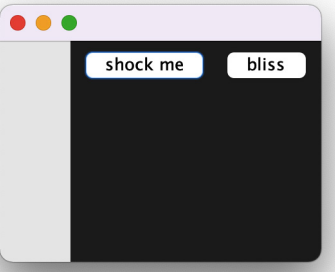
Add BOTH to the panel

what we *wanted*:



We want the buttons stacked on top of each other.

what we *got*:



The panel expanded to fit both buttons side by side.

Notice that the 'bliss' button is smaller than the 'shock me' button...that's how flow layout works. The button gets just what it needs (and no more).

Sharpen your pencil

→ Yours to solve.

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.

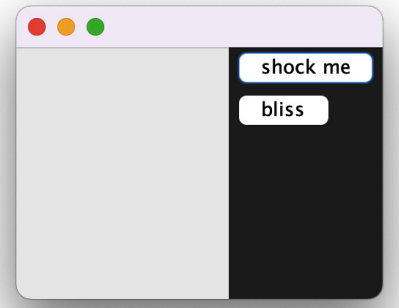
Unlike FlowLayout, BoxLayout can force a "new line" to make the components wrap to the next line, even if there's room for them to fit horizontally.

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."

there are no Dumb Questions

Q: How come you can't add directly to a frame the way you can to a panel?

A: A JFrame is special because it's where the rubber meets the road in making something appear on the screen. While all your Swing components are pure Java, a JFrame has to connect to the underlying OS in order to access the display. Think of the content pane as a 100% pure Java layer that sits on *top* of the JFrame. Or think of it as though JFrame is the window frame and the content pane is the...glass. You know, the window *pane*. And you can even swap the content pane with your own JPanel, to make your JPanel the frame's content pane, using:

```
myFrame.setContentPane(myPanel);
```

Q: Can I change the layout manager of the frame? What if I want the frame to use flow instead of border?

A: The easiest way to do this is to make a panel, build the GUI the way you want in the panel, and then make that panel the frame's content pane using the code in the previous answer (rather than changing 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 container-by-container 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.
- 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's content pane; 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. If you do want to find out more, read *Java Swing* by Dave Wood, Marc Loy, and Robert Eckstein.

JTextField

Constructors

```
JTextField field = new JTextField(20);
```

20 means 20 columns, not 20 pixels. This defines the preferred width of the text field.

```
JTextField field = new JTextField("Your name");
```

How to use it

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);
```

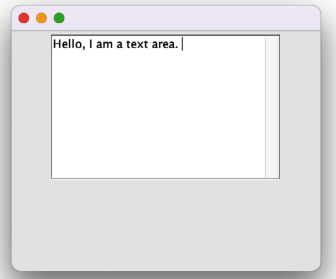
4. Select/Highlight the text in the field

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

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

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

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 JScrollPane. A JScrollPane is an object that really loves to scroll and will take care of the text area's scrolling needs.

Constructor

10 means 10 lines (sets the preferred height).

20 means 20 columns (sets the preferred width).

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

How to use it

① Make it have a vertical scrollbar only

Make a JScrollPane and give it the text area that it's going to scroll for.

Turn on line wrapping

Tell the scroll pane to use only a vertical scrollbar.

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

Important!! You give the text area to the scroll pane (through the scroll pane constructor), and 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

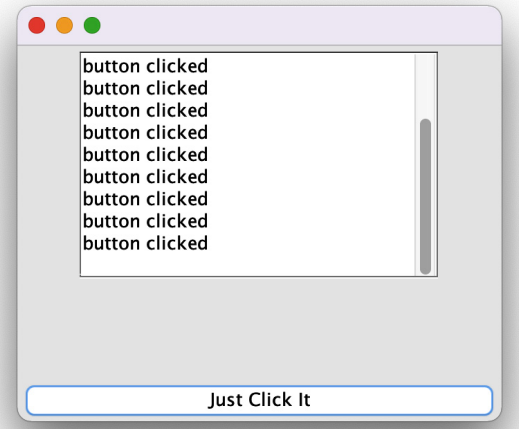
```
text.selectAll();
```

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

```
text.requestFocus();
```

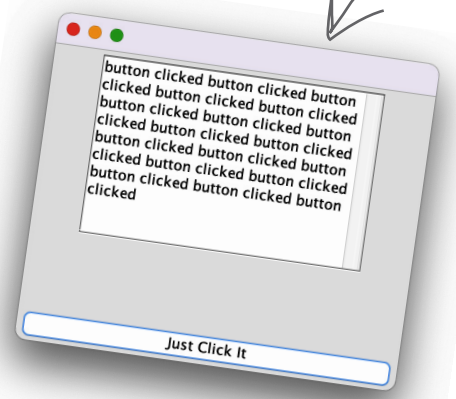
JTextArea example

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

public class TextArea1 {
  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");

    JTextArea text = new JTextArea(10, 20);
    text.setLineWrap(true);
    button.addActionListener(e -> text.append("button clicked \n"));

    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);
  }
}
```



Lambda expression to implement the button's ActionListener.

Insert a new line so the words go on a separate line each time the button is clicked. Otherwise, they'll run together.



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 e) {
  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

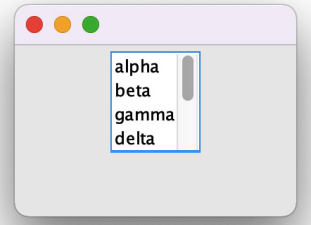
Q: Aren't the layout managers just more trouble than they're worth? If I have to go to all this trouble, I might as well just hard-code the size and coordinates for where everything should go.

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 haven't even covered 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<String> list = new JList<>(listEntries);
```

JList is a generic class, so you can declare what type of objects are in the list.

The diamond operator from Chapter 11.

How to use it

This is just like with JTextArea—you make a JScrollPane (and give it the list), and then add the scroll pane (NOT the list) to the panel.

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);
```

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);
```

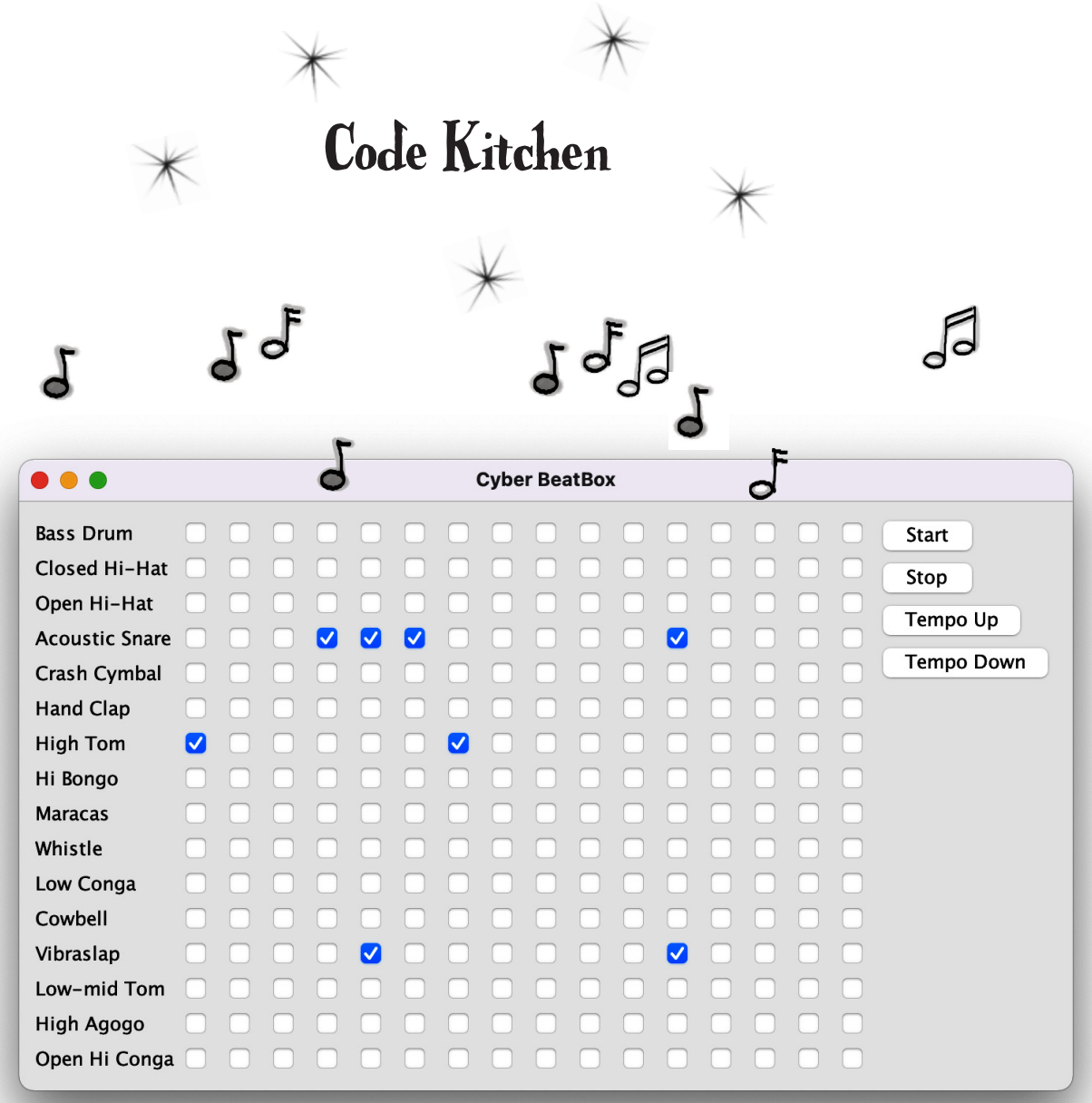
5. Handle events (find out which thing in the list was selected)

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

```
public void valueChanged(ListSelectionEvent e) {
  if (!e.getValueIsAdjusting()) {
    String selection = list.getSelectedValue();
    System.out.println(selection);
  }
}
```

getSelectedValue() actually returns an Object. A list isn't limited to only String objects.

Code Kitchen



This part's optional. We're making the full BeatBox, GUI and all. In Chapter 16, Saving Objects, we'll learn how to save and restore drum patterns. Finally, in Chapter 17, 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, setLoopCount(), that 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, and 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."

```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;

import static javax.sound.midi.ShortMessage.*;

public class BeatBox {
  private ArrayList<JCheckBox> checkboxList;
  private Sequencer sequencer;
  private Sequence sequence;
  private 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};

  public static void main(String[] args) {
    new BeatBox().buildGUI();
  }

  public void buildGUI() {
    JFrame frame = new JFrame("Cyber BeatBox");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    BorderLayout layout = new BorderLayout();
    JPanel background = new JPanel(layout);
    background.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

    Box buttonBox = new Box(BoxLayout.Y_AXIS);

    JButton start = new JButton("Start");
    start.addActionListener(e -> buildTrackAndStart());
    buttonBox.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(e -> sequencer.stop());
    buttonBox.add(stop);

    JButton upTempo = new JButton("Tempo Up");
    upTempo.addActionListener(e -> changeTempo(1.03f));
    buttonBox.add(upTempo);

    JButton downTempo = new JButton("Tempo Down");
    downTempo.addActionListener(e -> changeTempo(0.97f));
    buttonBox.add(downTempo);
```

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.

Lambda expressions are perfect for these event handlers, since when these buttons are pressed, all we want to do is call a specific method.

The default tempo is 1.0, so we're adjusting +/- 3% per click.

```
    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (String instrumentName : instrumentNames) {
      JLabel instrumentLabel = new JLabel(instrumentName);
      instrumentLabel.setBorder(BorderFactory.createEmptyBorder(4, 1, 4, 1));
      nameBox.add(instrumentLabel);
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    frame.getContentPane().add(background);

    GridLayout grid = new GridLayout(16, 16);
    grid.setVgap(1);
    grid.setHgap(2);

    JPanel mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    checkboxList = new ArrayList<>();
    for (int i = 0; i < 256; i++) {
      JCheckBox c = new JCheckBox();
      c.setSelected(false);
      checkboxList.add(c);
      mainPanel.add(c);
    }

    setUpMidi();

    frame.setBounds(50, 50, 300, 300);
    frame.pack();
    frame.setVisible(true);
  }

  private 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();
    }
  }
```

This border on each instrument name helps them line up with the checkboxes.

Still more GUI setup code. Nothing remarkable.

Another layout manager, this one lets you put the components in a grid with rows and columns.

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 setup stuff for getting the Sequencer, the Sequence, and the Track. Again, nothing special.

This is where it all happens! Where we turn checkbox state into MIDI events and add them to the Track.

```
private void buildTrackAndStart() {
  int[] trackList;

  sequence.deleteTrack(track);
  track = sequence.createTrack();

  for (int i = 0; i < 16; i++) {
    trackList = new int[16];

    int key = instruments[i];


    for (int j = 0; j < 16; j++) {
      JCheckBox jc = checkboxList.get(j + 16 * i);
      if (jc.isSelected()) {
        trackList[j] = key;
      } else {
        trackList[j] = 0;
      }
    }

    makeTracks(trackList);
    track.add(makeEvent(CONTROL_CHANGE, 1, 127, 0, 16));
  }

  track.add(makeEvent(PROGRAM_CHANGE, 9, 1, 0, 15));


  try {
    sequencer.setSequence(sequence);
    sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
    sequencer.setTempoInBPM(120);
    sequencer.start();
  } catch (Exception e) {
    e.printStackTrace();
  }
}

private void changeTempo(float tempoMultiplier) {
  float tempoFactor = sequencer.getTempoFactor();
  sequencer.setTempoFactor(tempoFactor * tempoMultiplier);
}
```

We'll make a 16-element array to hold the values for one instrument, across all 16 beats. If the instrument is supposed to play on that beat, the value at that element will be the key. If that instrument is NOT supposed to play on that beat, put in a zero.

Get rid of the old track, make a fresh one.

do this for each of the 16 ROWS (i.e., Bass, Congo, etc.)

Set the "key" that represents which instrument this is (Bass, Hi-Hat, etc.). The instruments array holds the actual MIDI numbers for each instrument.

Do this for each of the BEATS for this row.

Is the checkbox at this beat selected? If yes, put the key value in this slot in the array (the slot that represents this beat). Otherwise, the instrument is NOT supposed to play at this beat, so set it to zero.

For this instrument, and for all 16 beats, make events and add them to the track.

We always want to make sure that there IS an event at beat 16 (it goes 0 to 15). Otherwise, the BeatBox might not go the full 16 beats before it starts over.

Lets you specify the number of loop iterations, or in this case, continuous looping.

NOW PLAY THE THING!!

The Tempo Factor scales the sequencer's tempo by the factor provided, slowing the beat down or speeding it up.

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.

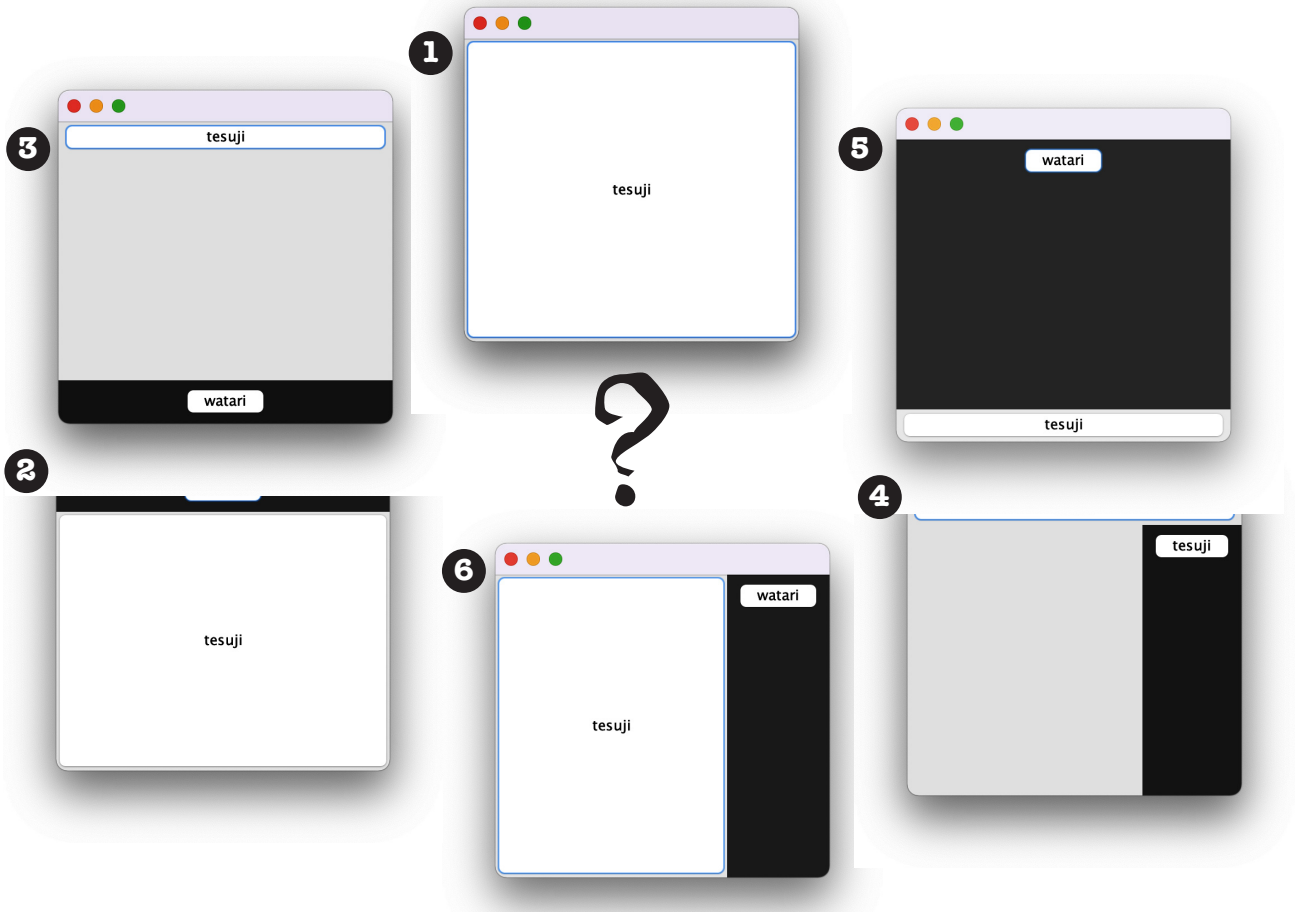
```
  private void makeTracks(int[] list) {
    for (int i = 0; i < 16; i++) {
      int key = list[i];

      if (key != 0) {
        track.add(makeEvent(NOTE_ON, 9, key, 100, i));
        track.add(makeEvent(NOTE_OFF, 9, key, 100, i + 1));
      }
    }
  }

  public static MidiEvent makeEvent(int cmd, int chnl, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage msg = new ShortMessage();
      msg.setMessage(cmd, chnl, one, two);
      event = new MidiEvent(msg, tick);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return event;
  }

}
```

Make the NOTE ON and NOTE OFF events, and add them to the Track.

This is the utility method from the previous chapter's Code Kitchen. Nothing new.

Which code goes with which layout?

Five of the six screens below were made from one of the code fragments on the opposite page. Match each of the five code fragments with the layout that fragment would produce.



Answers on page 537.

Code Fragments

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);
```

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);
```

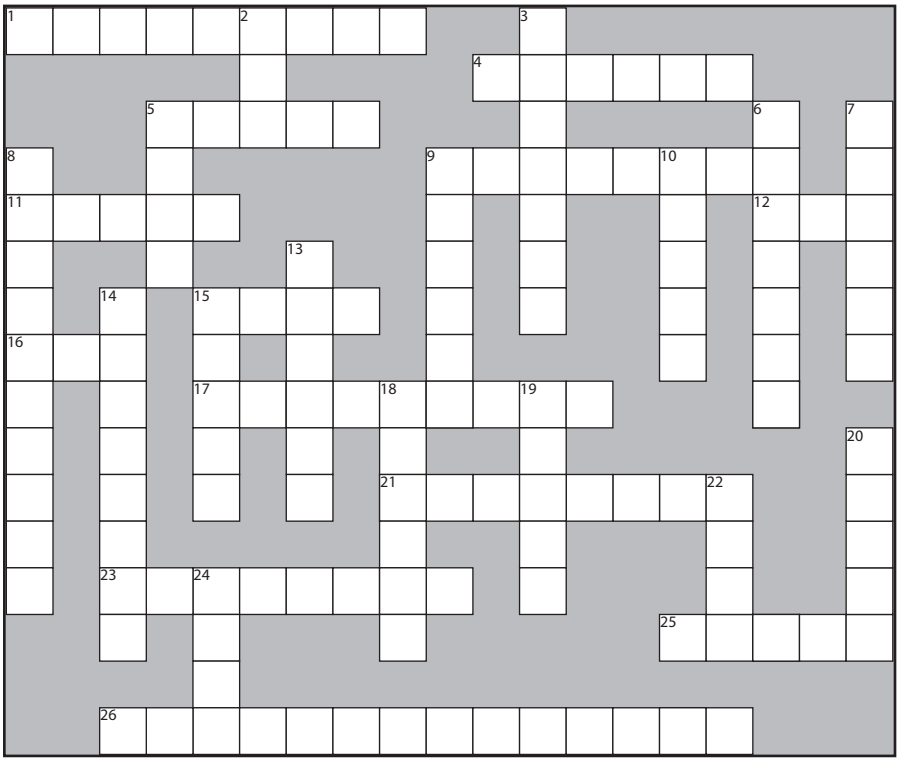
D

```
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);
```

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);
```

GUI-Cross



You can do it.

Across

1. Artist's sandbox
4. Border's catchall
5. Java look
9. Generic waiter
11. A happening
12. Apply a widget
15. JPanel's default
16. Polymorphic test
17. Shake it, baby
21. Lots to say
23. Choose many
25. Button's pal
26. Home of actionPerformed

Down

2. Swing's dad
3. Frame's purview
5. Help's home
6. More fun than text
7. Component slang
8. Romulin command
9. Arrange
10. Border's top
13. Manager's rules
14. Source's behavior
15. Border by default
18. User's behavior
19. Inner's squeeze
20. Backstage widget
22. Classic Mac look
24. Border's right

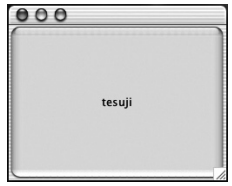
→ Answers on page 538.

Exercise Solutions

Which code goes with which layout?

(from pages 534–535)

1



C

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

2

D

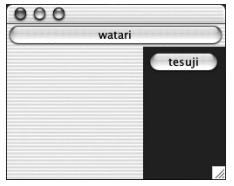
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,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

3

E

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

4



A

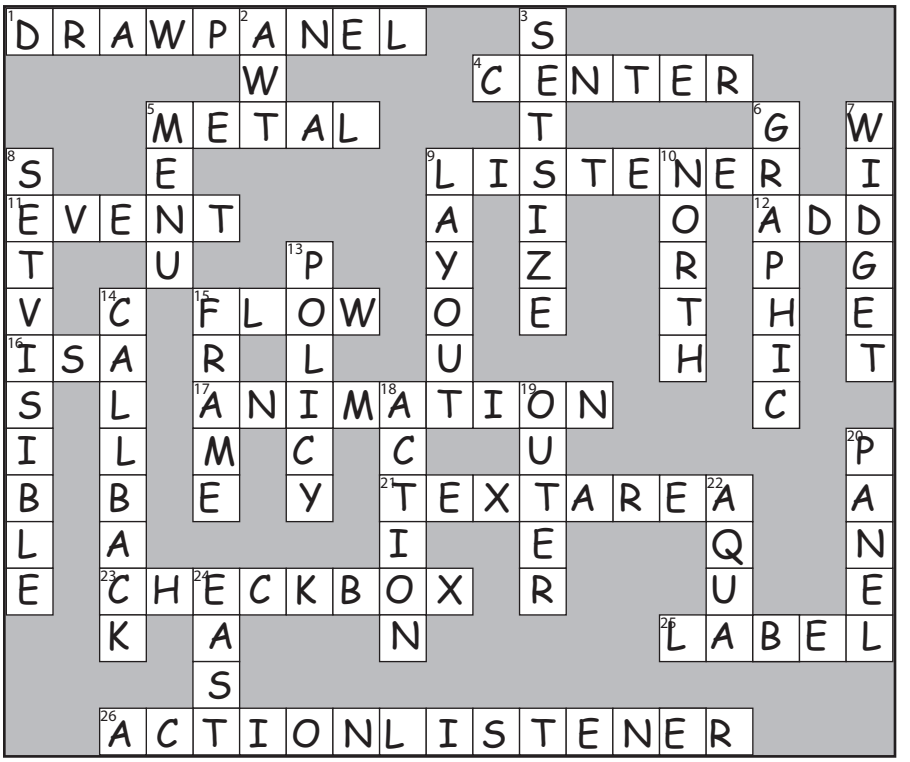
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH, buttonTwo);
frame.getContentPane().add(BorderLayout.EAST,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

6

B

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

GUI-Cross (from page 536)



Saving Objects (and Text)

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. And since all I/O operations are risky, we'll take a look at how to do even better exceptions handling.

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 during a random computer crash.

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

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

① Use serialization

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

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

② Write a plain-text file

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

These aren't the only options, but if we had to pick only two approaches to doing I/O in Java, we'd probably pick these. 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. 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:

```
¨ÌsrGameCharacter
¨%gê8MÛIpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp2tlfur[Ljava.lang.String;≠“VÁ
È{Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

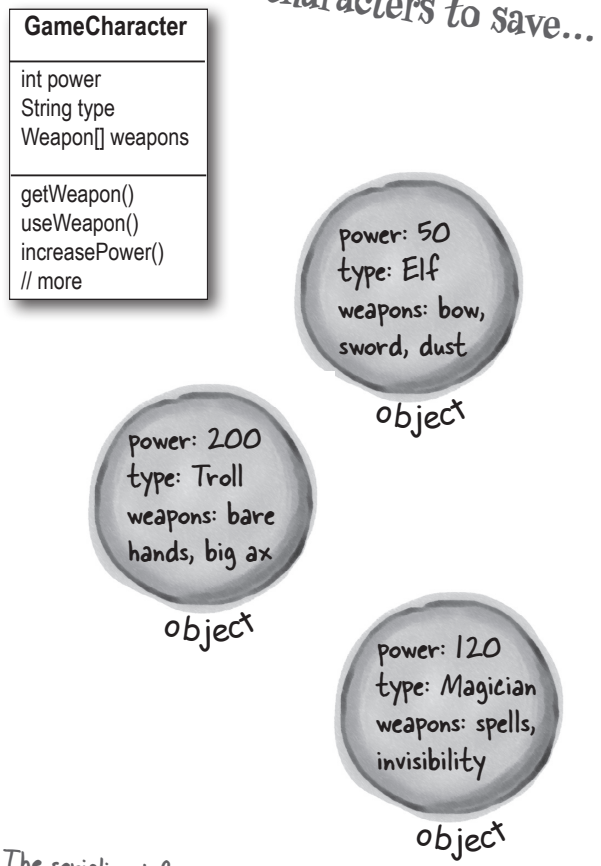
② Option two

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...



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

Writing a serialized object to a file

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

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

1 Make a FileOutputStream

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

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

2 Make an ObjectOutputStream

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

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

3 Write the object

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

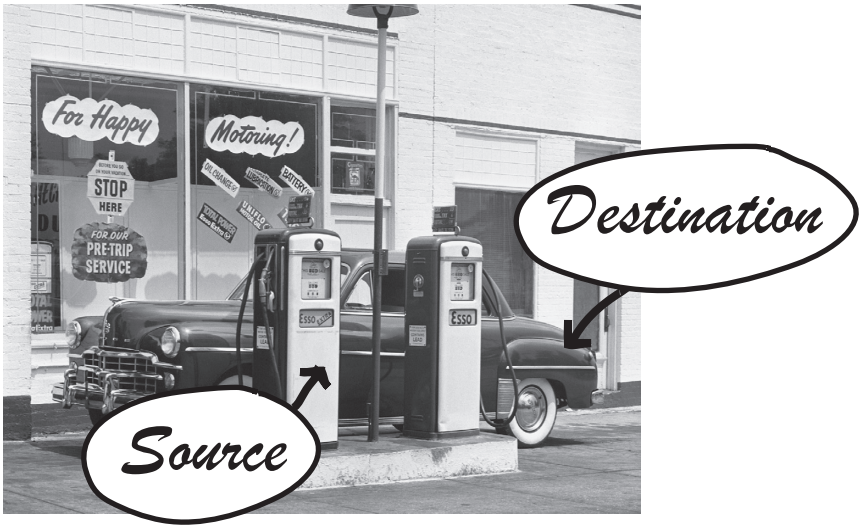
Serializes the objects referenced by characterOne, characterTwo, and characterThree, and writes them in this order to the file "MyGame.ser."

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



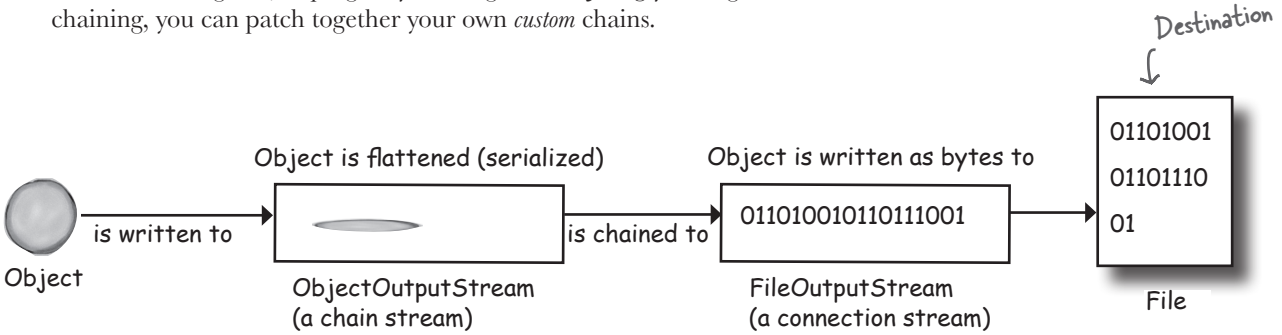
Connection streams represent a connection to a source or destination (file, network socket, etc.), while chain streams can't connect on their own and must be chained to a connection stream.

The Java I/O API has ***connection*** streams, which 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 (a connection stream) that lets us write to a file, and we hook an ObjectOutputStream (a chain stream) on the end of it. When we call writeObject() on the ObjectOutputStream, the object gets pumped into the stream and then moves to the FileOutputStream where it ultimately gets written as bytes to a file.

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



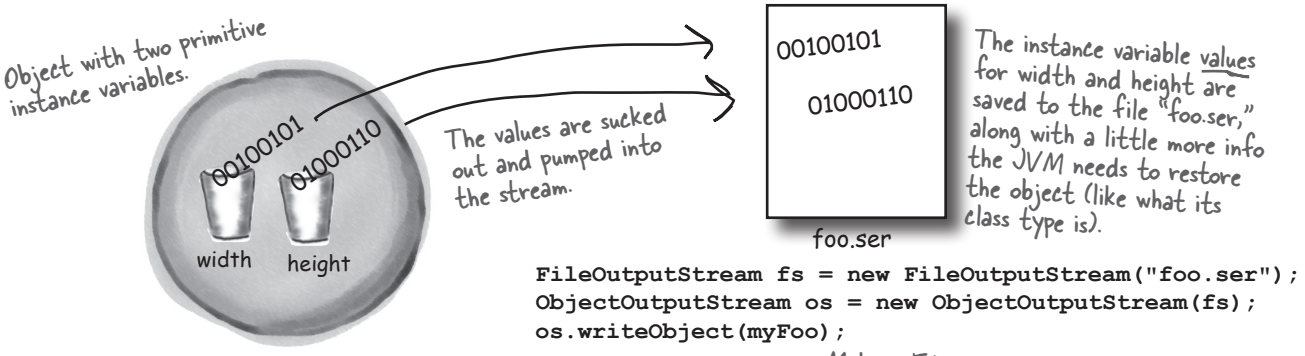
What really happens to an object when it's serialized?

1 Object on the heap

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

2 Object serialized

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




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

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

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

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

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

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

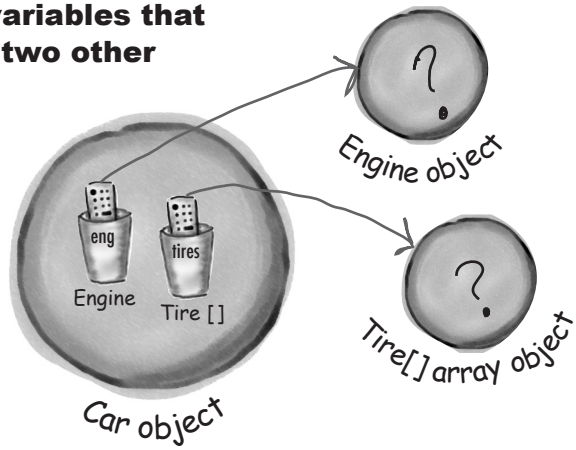
Brain Barbell

What has to happen for the Car object to be saved in such a way that it can be restored 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.



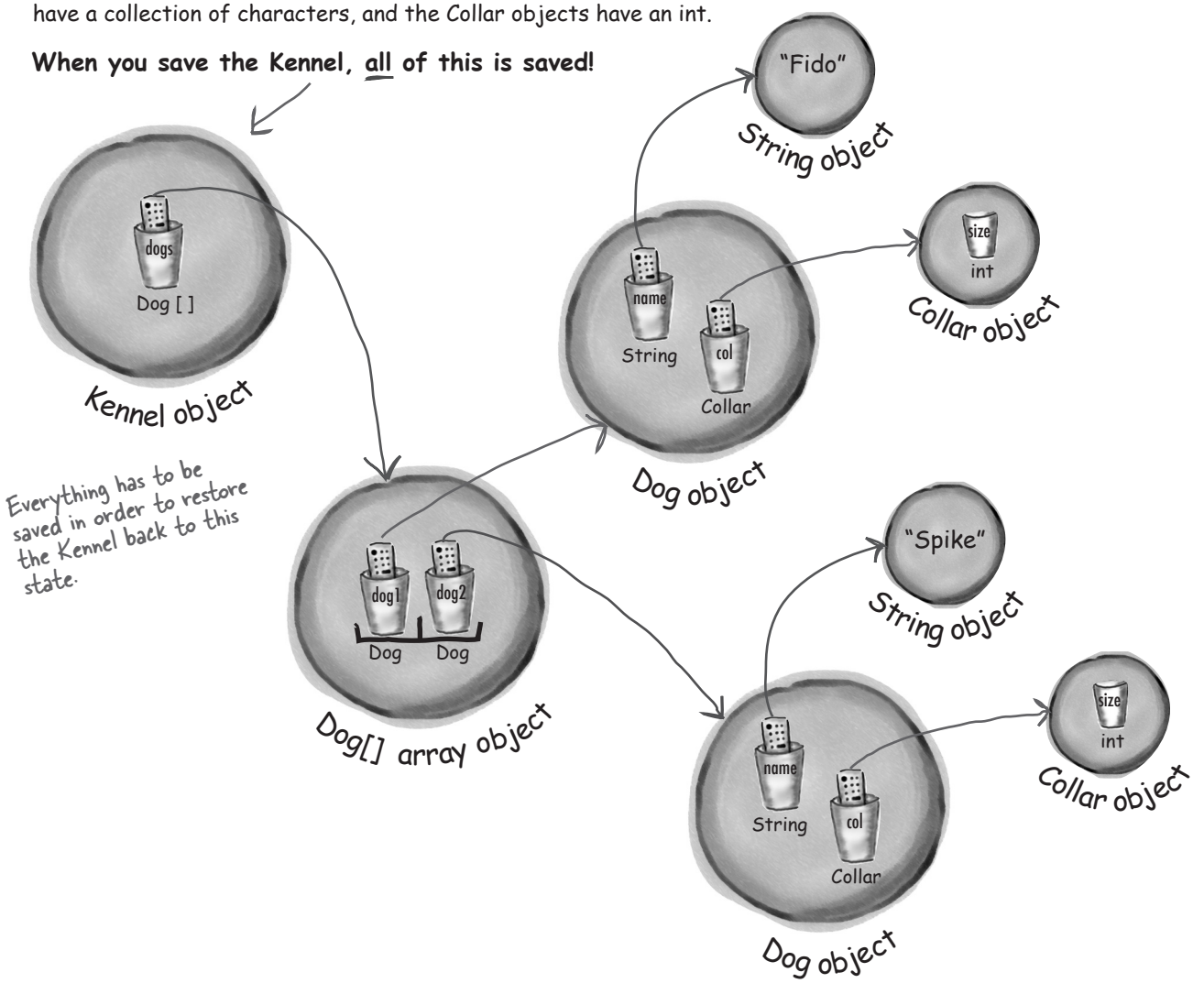
What does it take to save a Car object?

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(mySquare);
```

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 Square 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 Square(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public static void main(String[] args) {
    Square mySquare = new Square(50, 20);

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

I/O operations can throw exceptions.

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

Make an ObjectOutputStream chained to the connection stream.

Tell it to write the object.

Serialization is all or nothing.

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

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

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

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

```
import java.io.*;

public class Pond implements Serializable {

  private Duck duck = new Duck();

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

      os.writeObject(myPond);
      os.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Pond objects can be serialized.

Class Pond has one instance variable, a Duck.

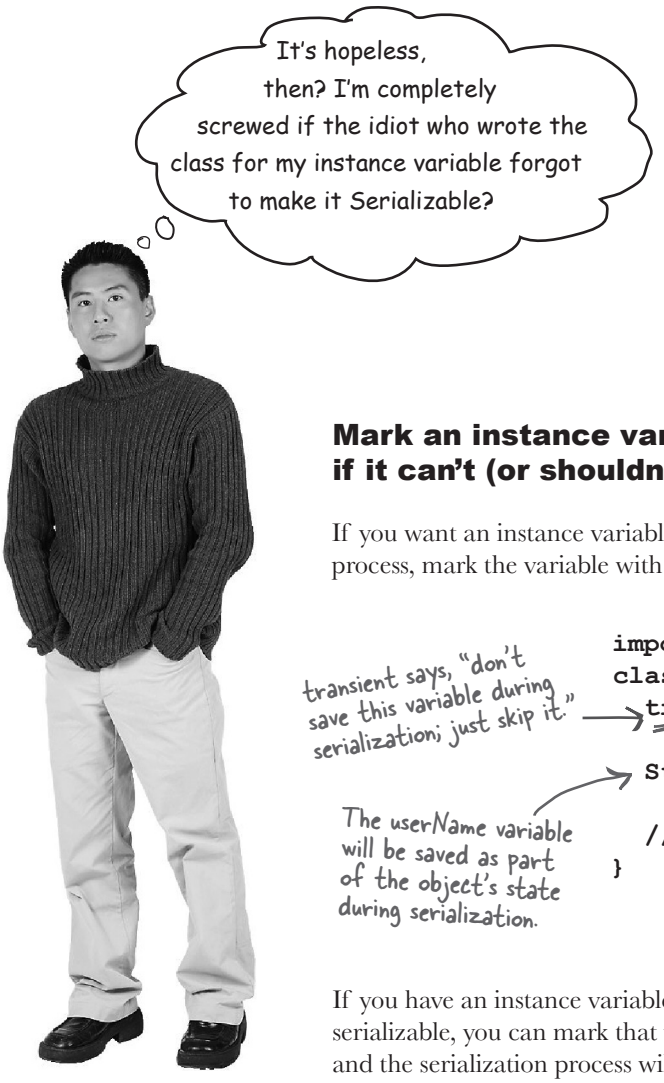
When you serialize myPond (a Pond object), its Duck instance variable automatically gets serialized.

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

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

When you try to run the main in class Pond:

```
File Edit Window Help Regret

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



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

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

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

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

  String userName;

  // more code
}
```

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

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

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

there are no Dumb Questions

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

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

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

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

Q: If a class I'm using isn't serializable but there's no good reason, 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.) That 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.

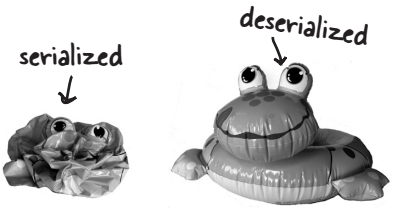
2. If the value of the transient variable *does* matter (say, if the color and design of the transient Collar are unique for each Dog), then you need to save the key values of the Collar and use them when the Dog is brought back to essentially re-create a brand new Collar that's identical to the original.

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 to its original state at some later date, in a different "run" of the JVM (which might not even be the same JVM that was running at the time the object was serialized). Deserialization is a lot like serialization in reverse.



1 Make a FileInputStream

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

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

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

2 Make an ObjectInputStream

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

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

3 Read the objects

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

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

4 Cast the objects

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

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

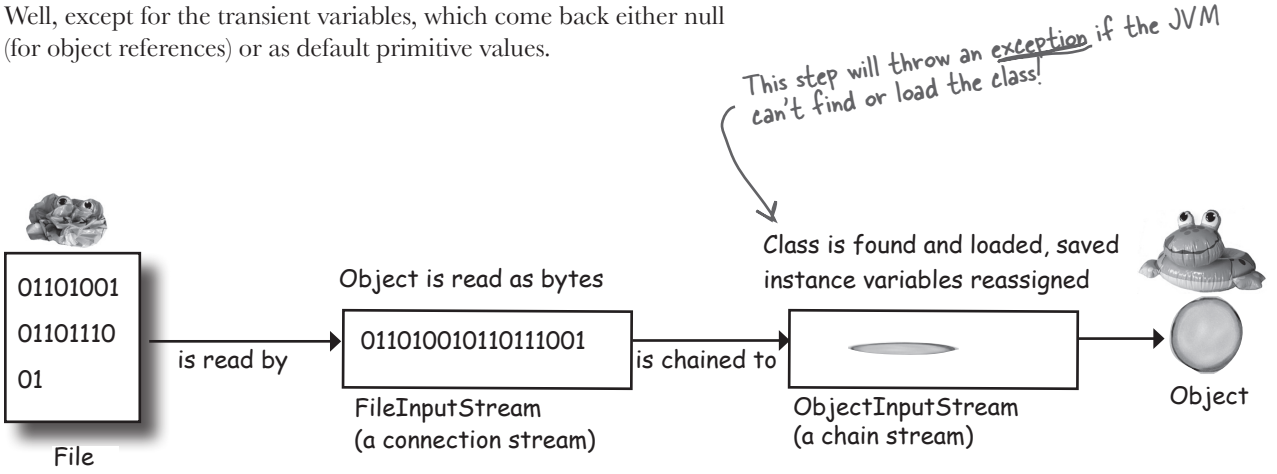
5 Close the ObjectInputStream

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

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

What happens during deserialization?

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



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

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

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

4. A new object is given space on the heap, but the **serialized object's constructor does NOT run**! Obviously, if the constructor ran, it would restore the state of the object 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 i**nstance variables are given the values from the serialized state**. Transient variables are given a value of null for object references and defaults (0, false, etc.) for primitives.

there are no Dumb Questions

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

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

For objects serialized to ship over a network, though, there actually *is* a mechanism where the serialized object can be "stamped" with a URL for where its class can be found. This is used in Java's Remote Method Invocation (RMI) so that you can send a serialized object as part of, say, a method argument, and if the JVM receiving the call doesn't have the class, it can use the URL to fetch the class from the network and load it, all automatically. You may see RMI used in the wild, although you may also see objects serialized to XML or JSON (or other human-readable formats) to send over a network.

Q: What about static variables? Are they serialized?

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

Saving and restoring the game characters

```
import java.io.*;

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

    // imagine code that does things with the characters that changes 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();
    }

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

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

Make some characters...

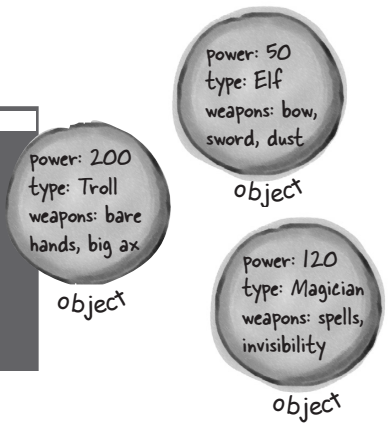
Serialize the characters.

Now read them back in from the file...

Restore the characters.

Check to see if it worked.

```
File Edit Window Help Resuscitate
% java GameSaverTest
One's type: Elf
Two's type: Troll
Three's type: Magician
```



The GameCharacter class

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

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

  public GameCharacter(int power, String type, String[] weapons) {
    this.power = power;
    this.type = type;
    this.weapons = weapons;
  }

  public int getPower() {
    return power;
  }

  public String getType() {
    return type;
  }

  public String getWeapons() {
    return Arrays.toString(weapons);
  }
}
```

This is a basic class just for testing the Serialization code on the last page. We don't have an actual game, but we'll leave that to you to experiment.

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!

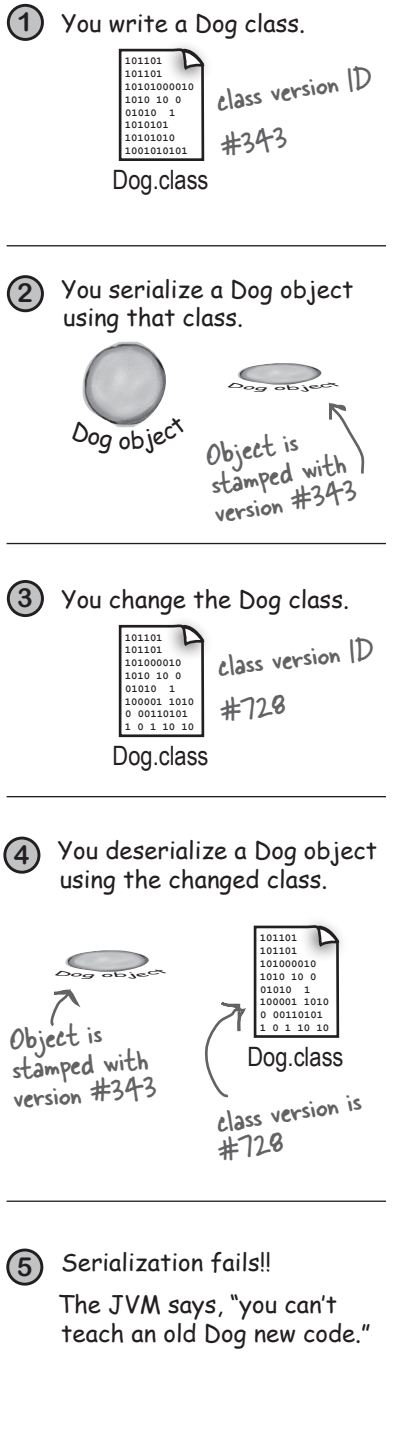
If you serialize an object, you must have the class in order to deserialize and use the object. OK, that's obvious. But it might be less obvious what happens if you ***change the class*** in the meantime. Yikes. Imagine trying to bring back a Dog object when one of its instance variables (non-transient) 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 (public, private, etc.) 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)



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 =
                  -5849794470654667210L;

    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.

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.
- If a class that implements Serializable might change over time, put a *static final long serialVersionUID* on that class. This version ID should be changed when the serialized variables in that class change.

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 something like a FileWriter instead of a FileOutputStream (and you don't chain it to an ObjectOutputStream).

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

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

To write a serialized object:

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

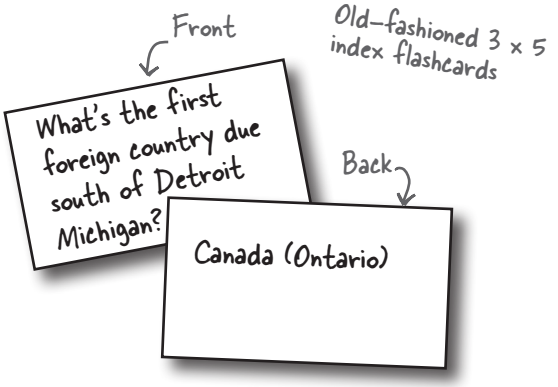
To write a String:

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

```
import java.io.*;                                   // We need the java.io package for FileWriter.

class WriteAFile {
  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter("Foo.txt");   // If the file "Foo.txt" does not exist, FileWriter will create it.

      writer.write("hello foo!");                   // The write() method takes a String.

      writer.close();                               // Close it when you're done!
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

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

Text file example: e-Flashcards



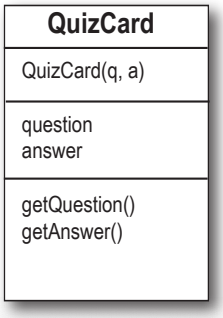
Remember those flashcards you used in school? Where you had a question on one side and the answer on the back? They aren't much help when you're trying to understand something, but nothing beats 'em for raw drill-and-practice and rote memorization. *When you have to burn in a fact.* And they're also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

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

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

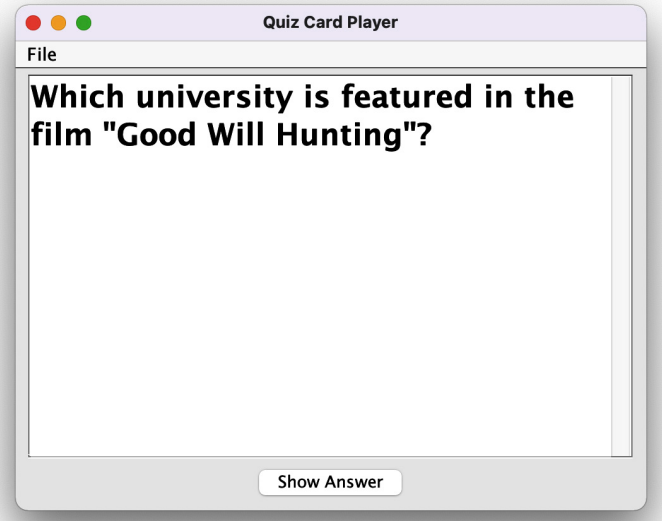
2. ***QuizCardPlayer***, a playback engine that can load a flashcard set and play it for the user.

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



QuizCardBuilder

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



QuizCardPlayer

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

Quiz Card Builder (code outline)

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

  private void nextCard() {
    // add the current card to the list
    // and clear the text areas
  }

  private void saveCard() {
    // bring up a file dialog box
    // let the user name and save the set
  }

  private void clearCard() {
    // clear out the text areas
  }

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

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

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

Call 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.).

Will need to clear the screen when the user chooses 'New' from the File menu or moves to the next card.

Called by the SaveMenuListener; does the actual file writing.

Java I/O to NIO to NIO.2

The Java API has included I/O features since day one, you know, back in the last millennium. In 2002, Java 1.4 was released, and it included a new approach to I/O called "NIO," short for non-blocking I/O. In 2011, Java 7 was released, and it included big enhancements to NIO. This yet again newer approach to I/O was dubbed "NIO.2." Why should you care? When you're writing new I/O, you should use the latest and greatest features. But you're almost certainly going to encounter older code that uses the NIO approach. We want you to be covered for both situations, so in this chapter:

- We'll use original I/O for a while.
- Then we'll show some NIO.2.

You'll see more I/O, NIO, and NIO.2 features in Chapter 17, *Make a Connection*, when we look at network connections.

Reminder: For the next eight pages or so we'll be using older-style I/O code!

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

public class QuizCardBuilder {
  private ArrayList<QuizCard> cardList = new ArrayList<>();
  private JTextArea question;
  private JTextArea answer;
  private JFrame frame;

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

  public void go() {
    frame = new JFrame("Quiz Card Builder");
    JPanel mainPanel = new JPanel();
    Font bigFont = new Font("sanserif", Font.BOLD, 24);

    question = createTextArea(bigFont);
    JScrollPane qScroller = createScroller(question);
    answer = createTextArea(bigFont);
    JScrollPane aScroller = createScroller(answer);

    mainPanel.add(new JLabel("Question:"));
    mainPanel.add(qScroller);
    mainPanel.add(new JLabel("Answer:"));
    mainPanel.add(aScroller);

    JButton nextButton = new JButton("Next Card");
    nextButton.addActionListener(e -> nextCard());
    mainPanel.add(nextButton);

    JMenuBar menuBar = new JMenuBar();
    JMenu fileMenu = new JMenu("File");

    JMenuItem newMenuItem = new JMenuItem("New");
    newMenuItem.addActionListener(e -> clearAll());

    JMenuItem saveMenuItem = new JMenuItem("Save");
    saveMenuItem.addActionListener(e -> saveCard());

    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);
  }
```

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

Next Card button calls the nextCard method when it's pressed.

When the user clicks "New" on the menu, the clearAll method is called.

When the user clicks "Save" on the menu, the saveCard method is called.

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, and then tell the frame to use this menu bar. Menu items can fire an ActionEvent.