

It seems to me there should be a way to use polymorphic collection types as method arguments so that a vet program could take Dog lists and Cat lists. Then it would be possible to loop through the lists and call their immunize() method. It would have to be safe so that you couldn't add a Cat in to the Dog list.

We can do this with wildcards

It looks unusual, but there *is* a way to create a method argument that can accept a List of any Animal subtype. The simplest way is to use a **wildcard**.

```
public void takeAnimals(List<? extends Animal> animals) {
    for (Animal a : animals) {
        a.eat();
    }
}
```

Remember, the keyword "extends" here means either extends OR implements.

So now you're wondering, "What's the *difference*? Don't you have the same problem as before?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!

When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.

You can still call methods on the elements in the list, but you cannot add elements to the list.

In other words, you can do things with the list elements, but you can't put new things in the list.



there are no Dumb Questions

Q: Back when we first saw generic methods, there was a similar-looking method that declared the generic type in front of the method name. Does that do the same thing as this takeAnimals method?

A: Well spotted! Back at the start of the chapter, there was a method like this:

```
<T extends Animal> void takeThing(List<T> list)
```

We actually could use this syntax to achieve a similar thing, but it works in a slightly different way. Yes, you can pass List<Animal> and List<Dog> into the method, but you get the added benefit of being able to use the generic type, T, elsewhere too.

Using the method's generic type parameter

What can we do if we define our method like this instead?

```
public <T extends Animal> void takeAnimals(List<T> list) { }
```

Well, not much as the method stands right now, we don't need to use "T" for anything. But if we made a change to our method to return a List, for example of all the animals we had successfully vaccinated, we can declare that the List that's returned has the same generic type as the List that's passed in:

```
public <T extends Animal> List<T> takeAnimals(List<T> list) { }
```

When you call the method, you know you're going to get the same type back as you put in.

```
List<Dog> dogs = List.of(new Dog(), new Dog());  
List<Dog> vaccinatedDogs = takeAnimals(dogs);  
  
List<Animal> animals = List.of(new Dog(), new Cat());  
List<Animal> vaccinatedAnimals = takeAnimals(animals);
```

The List we get back from the
takeAnimals method is always the
same type as the list we pass in.

If the method used the wildcard for both method parameter and return type, there's nothing to guarantee they're the same type. In fact, anything calling the method has almost no idea what's going to be in the collection, other than "some sort of animal."

```
public void go() {  
    List<Dog> dogs = List.of(new Dog(), new Dog());  
    List<? extends Animal> vaccinatedSomethings = takeAnimals(dogs);  
}  
  
public List<? extends Animal> takeAnimals(List<? extends Animal> animals) { }
```

Using the wildcard ("? extends") is fine when you don't care much about the generic type, you just want to allow all subtypes of some type.

Using a type parameter ("T") is more helpful when you want to do more with the type itself, for example in the method's return.



BE the Compiler, advanced

Your job is to play compiler and determine which of these statements would compile. Some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations.

The signatures of the methods used in the exercise are in the boxes.

```
private void takeDogs(List<Dog> dogs) { }
```

```
private void takeAnimals(List<Animal> animals) { }
```

```
private void takeSomeAnimals(List<? extends Animal> animals) { }
```

```
private void takeObjects(ArrayList<Object> objects) { }
```

Compiles?

- `takeAnimals(new ArrayList<Animal>());`
- `takeDogs(new ArrayList<Animal>());`
- `takeAnimals(new ArrayList<Dog>());`
- `takeDogs(new ArrayList<>());`
- `List<Dog> dogs = new ArrayList<>();
takeDogs(dogs);`
- `takeSomeAnimals(new ArrayList<Dog>());`
- `takeSomeAnimals(new ArrayList<>());`
- `takeSomeAnimals(new ArrayList<Animal>());`
- `List<Animal> animals = new ArrayList<>();
takeSomeAnimals(animals);`
- `List<Object> objects = new ArrayList<>();
takeObjects(objects);`
- `takeObjects(new ArrayList<Dog>());`
- `takeObjects(new ArrayList<Object>());`

Exercise Solution

Fill-in-the-blanks (from page 334)

Possible Answers:

Comparator,

Comparable,

compareTo(),

compare(),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in myArrayList implement?
2. What method must the class of the objects stored in myArrayList implement?
3. Can the class of the objects stored in myArrayList implement both Comparator AND Comparable?

Comparable

compareTo()

yes

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in myArrayList implement Comparable?
5. Can the class of the objects stored in myArrayList implement Comparator?
6. Must the class of the objects stored in myArrayList implement Comparable?
7. Must the class of the objects stored in myArrayList implement Comparator?
8. What must the class of the myCompare object implement?
9. What method must the class of the myCompare object implement?

yes

yes

no

no

Comparator

compare()

Solution**“Reverse Engineer” lambdas exercise**

(from page 343)

```

import java.util.*;

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

    public void go() {
        List<Mountain> mountains = new ArrayList<>();
        mountains.add(new Mountain("Longs", 14255));
        mountains.add(new Mountain("Elbert", 14433));
        mountains.add(new Mountain("Maroon", 14156));
        mountains.add(new Mountain("Castle", 14265));
        System.out.println("as entered:\n" + mountains);

        mountains.sort((mount1, mount2) -> mount1.name.compareTo(mount2.name));
        System.out.println("by name:\n" + mountains);

        mountains.sort((mount1, mount2) -> mount2.height - mount1.height);
        System.out.println("by height:\n" + mountains); ←
    }
}

class Mountain {
    String name;
    int height;

    Mountain(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public String toString() {
        return name + " " + height;
    }
}

```

Did you notice that the height list is
in DESCENDING sequence? :)

Output:

```

File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]

```

Solution



Sorting with lambdas (from page 342)

Sort by BPM ascending

```
songList.sort((one, two) -> one.getBpm() - two.getBpm());
```

Sort by title descending

```
songList.sort((one, two) -> two.getTitle().compareTo(one.getTitle()));
```

Output:

```
File Edit Window Help IntNotString
%java SharpenLambdas
[50 ways, havana, $10, somersault, cassidy, Cassidy]
[somersault, havana, cassidy, Cassidy, 50 ways, $10]
```

Solution



TreeSet exercise

(from page 353)

1. What is the result when you compile this code?

It compiles correctly

2. If it compiles, what is the result when you run the TestTree class?

It throws an exception:

```
Exception in thread "main" java.lang.ClassCastException: class Book cannot be cast to class java.lang.Comparable
        at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
        at java.base/java.util.TreeMap.put(TreeMap.java:536)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at TestTree.go(TestTree.java:16)
        at TestTree.main(TestTree.java:7)
```

3. If there is a problem (either compile-time or runtime) with this code, how would you fix it?

Make Book implement Comparable, or pass the TreeSet a Comparator

See page 574



BE the Compiler solution

(from page 363)

Compiles?

- takeAnimals(new ArrayList<Animal>());
- takeDogs(new ArrayList<Animal>());
- takeAnimals(new ArrayList<Dog>());
- takeDogs(new ArrayList<>()); ←
- List<Dog> dogs = new ArrayList<>();
takeDogs(dogs);
- takeSomeAnimals(new ArrayList<Dog>());
- takeSomeAnimals(new ArrayList<>()); ←
- takeSomeAnimals(new ArrayList<Animal>());
- List<Animal> animals = new ArrayList<>();
takeSomeAnimals(animals);
- List<Object> objects = new ArrayList<>();
takeObjects(objects);
- takeObjects(new ArrayList<Dog>());
- takeObjects(new ArrayList<Object>());

If you use the diamond operator here, it works out the type from the method signature. Therefore, the compiler assumes this `ArrayList` is `ArrayList<Dog>`.

Here the diamond operator means this is `ArrayList<Animal>`.

This doesn't compile because `takeObjects` wants an `ArrayList`, not a `List`.

Lambdas and Streams: What, Not How



What if...you didn't need to tell the computer HOW to do something? Programming involves a lot of telling the computer how to do something: **while** this is true **do** this thing; **for** all these items **if** it looks like this **then** do this; and so on.

We've also seen that we don't have to do everything ourselves. The JDK contains library code, like the Collections API we saw in the previous chapter, that we can use instead of writing everything from scratch. This library code isn't just limited to collections to put data into; there are methods that will do common tasks for us, so we just need to tell them **what** we want and not **how** to do it.

In this chapter we'll look at the Streams API. You'll see how helpful lambda expressions can be when you're using streams, and you'll learn how to use the Streams API to query and transform the data in a collection.

Tell the computer **WHAT** you want

Imagine you have a list of colors, and you wanted to print out all the colors. You could use a for loop to do this.

```
List<String> allColors = List.of("Red", "Blue", "Yellow");
for loop {
    for (String color : allColors) {
        System.out.println(color);
    }
}
```

This is a “convenience factory method” for creating a new List from a known group of values. We saw this in Chapter 11.

For each item in the list create a temporary variable, color...

...then print out each color.

But doing something to every item in a list is a really common thing to want to do. So instead of creating a for loop every time we want to do something “for each” item in the list, we can call the **forEach** method from the Iterable interface—remember, List implements Iterable so it has all the methods from the Iterable interface.

```
List<String> allColors = List.of("Red", "Blue", "Yellow");
allColors.forEach(color -> System.out.println(color));
```

For each item in the list...

Create a temporary variable named color

Print out the color

```
File Edit Window Help SingARainbow
% java PrintColors
Red
Blue
Yellow
```

The **forEach** method of a list takes a lambda expression, which we saw for the first time in the previous chapter. This is a way for you to pass behavior (“follow these instructions”) into a method, instead of passing an object containing data (“here is an object for you to use”).

Fireside Chats



for loop

I am the default! The for loop is so important that loads of programming languages have me. It's one of the first things a programmer learns! If someone needs to loop a set number of times to do something, they're going to reach for their trusty for loop.

Sure, fashions change. But sometimes it's just a fad; things fall out of fashion too. A classic like me will be easy to read and write forever, even for non-Java programmers.

So much work?! Ha! A developer isn't scared of a little syntax to clearly specify what to do and how to do it. At least with me, someone reading my code can clearly see what's going on.

Well I'm faster. Everyone knows that.

I said you would disappear soon.

Tonight's Talk: **The for loop and forEach method battle over the question, "Which is better?"**

forEach()

Pff. Please. You are *so old*; that's why you're in all the programming languages. But things change, languages evolve. There's a better way. A more modern way. Me.

But look how much work developers need to do to write you! They have to control when to start, increment, and stop the loop, as well as writing the code that needs to be run inside the loop. All sorts of things could go wrong! If they use me, they just have to think about what needs to happen to each item, they don't have to worry about how to loop to find each item.

Dude, they shouldn't *have* to see what's going on. It says very clearly in my method name exactly what I do—"for each" element I will apply the logic they specify. Job done.

Well actually, under the covers I'm using a for loop myself, but if something is invented later that's even faster, I can use that, and developers don't have to change a single thing to get faster code. In fact we're out of time now so....

When for loops go wrong

Using **forEach** instead of a for loop means a bit less typing, and it's also nice to focus on telling the compiler *what* you want to do and not *how* to do it. There's another advantage to letting the libraries take care of routine code like this—it can mean fewer accidental errors.



A short Java program is listed below. One block of the program is missing. We expect the output of the program should be "1 2 3 4 5" but sometimes it's difficult to get a for loop just right.

Your challenge is to **match the candidate block of code** (on the left) **with the output** that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once.

```
class MixForLoops {  
    public static void main(String [] args) {  
        List<Integer> nums = List.of(1, 2, 3, 4, 5);  
        String output = "";  
  
        System.out.println(output);  
    }  
}
```

Candidate code
goes here

Candidates:

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

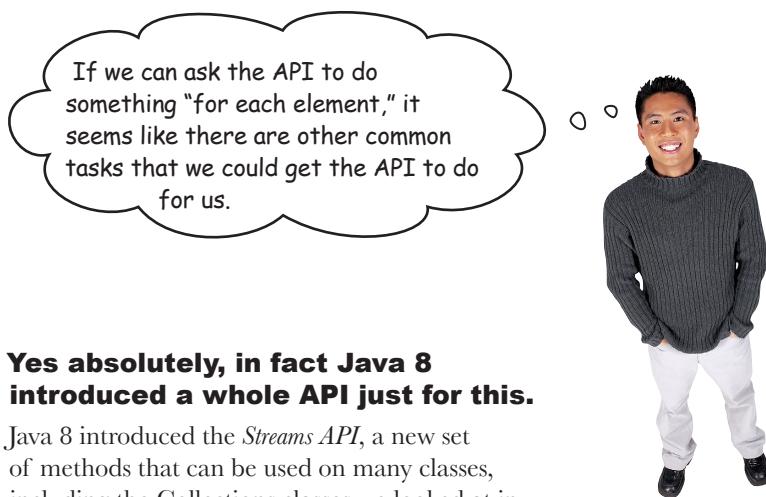
Match each
candidate with
one of the
possible outputs

→ Answers on page 417.

Small errors in common code can be hard to spot

The for loops from the previous exercise all look quite similar, and at first glance they all look like they would print out all the values in the List in order. Compiler errors can be easiest to spot, because your IDE or compiler will tell you the code is wrong, and Exceptions (which we'll see in Chapter 13, *Risky Behavior*) can also point to a problem in the code. But it can be trickier to spot code that produces incorrect output just by looking at the code.

Using a method like **forEach** takes care of the “boilerplate,” the repetitive and common code like the for loop. Using forEach, passing in only the thing we want to do, can reduce accidental errors in our code.



Yes absolutely, in fact Java 8 introduced a whole API just for this.

Java 8 introduced the *Streams API*, a new set of methods that can be used on many classes, including the Collections classes we looked at in the previous chapter.

The Streams API isn’t just a bunch of helpful methods, but also a slightly different way of working. It lets us build up a whole set of requirements, a recipe if you like, of what we want to know about our data.

Brain Barbell

Can you think of more examples of the types of things we might want to do to a collection? Are you going to want to ask similar questions about what’s inside different types of collections? Can you think of different types of information you might want to output from a collection?

Building blocks of common operations

The ways we search our collections, and the types of information we want to output from those collections, can be quite similar even on different types of collections containing different types of Objects.

Imagine what you might want do to with a Collection: “give me just the items that meet some criteria,” “change all the items using these steps,” “remove all duplicates,” and the example we worked through in the previous chapter: “sort the elements in this way.”

It’s not too hard to go one step further and assume each of these collection operations could be given a name that tells us what will happen to our collection.



We know this is all new, but have a go at matching each operation name to the description of what it does. Try not to look at the next page as you complete it, as that will give the game away!

filter	Changes the current element in the stream into something else
skip	Sets the maximum number of elements that can be output from this Stream
limit	While a given criteria is true, will not process elements
distinct	Only allows elements that match the given criteria to remain in the Stream
sorted	Will only process elements while the given criteria is true
map	States the result of the stream should be ordered in some way
dropWhile	This is the number of elements at the start of the Stream that will not be processed
takeWhile	Use this to make sure duplicates are removed

—————> Answers on page 417.

Introducing the Streams API

The Streams API is a set of operations we can perform on a collection, so when we read these operations in our code, we can understand what we're trying to do with the collection data. If you were successful in the “Who Does What?” exercise on the previous page (the complete answers are at the end of this chapter), you should have seen that the names of the operations describe what they do.

java.util.stream.Stream

Stream<T> distinct()	>Returns a stream consisting of the distinct elements
Stream<T> filter(Predicate<? super T> predicate)	>Returns a stream of the elements that match the given predicate.
Stream<T> limit(long maxSize)	>Returns a stream of elements truncated to be no longer than max-size in length.
<R> Stream<R> map(Function<? super T, ? extends R> mapper)	>Returns a stream with the results of applying the given function to the elements of this stream.
Stream<T> skip(long n)	>Returns a stream of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T> sorted()	>Returns a stream of the elements of this stream, sorted according to natural order.

// more

(These are just a few of the methods in Stream... there are many more.)

These generics do look a little intimidating, but don't panic! We'll use the map method later, and you'll see it's not as complicated as it seems.

Streams, and lambda expressions, were introduced in Java 8.



You don't need to worry too much about the generic types on the Stream methods; you'll see that using Streams “just works” the way you'd expect.

In case you are interested:

- <**T**> is usually the Type of the object in the stream.
- <**R**> is usually the type of the Result of the method.

Getting started with Streams

Before we start going into detail about what the Streams API is, what it does, and how to use it, we're going to give you some very basic tools to start experimenting.

To use the Streams methods, we need a Stream object (obviously). If we have a collection like a List, this doesn't implement Stream. However, the Collection interface has a method, **stream**, which returns a Stream object for the Collection.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
Stream<String> stream = strings.stream();
```

Assuming we had a List of Strings like this...

...we can call this method to get a Stream of these Strings.

Now we can call the methods of the Streams API. For example, we could use **limit** to say we want a maximum of four elements.

```
stream<String> limit = stream.limit(4);
```

The limit method returns another Stream of Strings, which we'll assign to another variable

Sets the maximum number of results to return to 4

What happens if we try to print out the result of calling limit()?

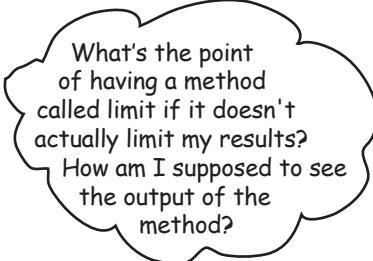
```
System.out.println("limit = " + limit);
```

```
File Edit Window Help SliceAndDice
%java LimitWithStream
limit = java.util.stream.SliceOps$1@7a0ac6e3
```

This doesn't look right at all! What's a SliceOps, and why isn't there a collection of just the first four items from the list?

Like everything in Java, the stream variables in the example are Objects. But a stream does **not** contain the elements in the collection. It's more like the set of instructions for the operations to perform on the Collection data.

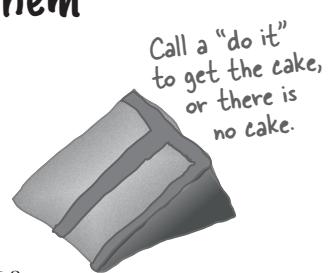
Stream methods that return another Stream are called Intermediate Operations. These are instructions of things to do, but they don't actually perform the operation on their own.



Streams are like recipes: nothing's going to happen until someone actually cooks them

A recipe in a book only tells someone *how* to cook or bake something. Opening the recipe doesn't automatically present you with a freshly baked chocolate cake. You need to gather the ingredients according to the recipe and follow the instructions exactly to come up with the result you want.

Collections are not ingredients, and a list limited to four entries is not a chocolate cake (sadly). But you do need to call one of the Stream's "do it" methods in order to get the result you want. These "do it" methods are called **Terminal Operations**, and these are the methods that will actually return something to you.



(These are some terminal operations on Stream.)

java.util.stream.Stream

boolean anyMatch(Predicate<? super T> predicate)
Returns true if any element matches the provided predicate.

long count()
Returns the number of elements in this stream.

<R,A> R collect(Collector<? super T,A,R> collector)
Performs a mutable reduction operation on the elements of this stream using a Collector.

Optional<T> findFirst()
Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.

// more

Yes, this looks even scarier than the map method! Don't panic, these generic types help the compiler, but you'll see when we actually use this method, we don't have to think about these generic types.

Getting a result from a Stream

Yes, we've thrown a **lot** of new words at you: *streams*; *intermediate operations*; *terminal operations*... And we still haven't told you what streams can do!

To start to get a feel for what we can do with streams, we're going to show code for a simple use of the Streams API. After that, we'll step back and learn more about what we're seeing here.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
```

```
Stream<String> stream = strings.stream();
Stream<String> limit = stream.limit(4);
long result = limit.count();
```

Call the count terminal operator, and store the output in a variable called result

```
File Edit Window Help WellDuh
%java LimitWithStream

result = 4
```

This works, but it's not very useful. One of the most common things to do with Streams is put the results into another type of collection. The API documentation for this method might seem intimidating with all the generic types, but the simplest case is straightforward:

The stream contained Strings, so the output object will also contain Strings.

Terminal operation that will collect the output into some sort of Object.

This method returns a Collector that will output the results of the stream into a List.

```
List<String> result = limit.collect(Collectors.toList());
```

The `toList` Collector will output the results as a List.

A helpful class that contains methods to return common Collector implementations.

```
System.out.println("result = " + result);
```

```
File Edit Window Help FinallyAResult
%java LimitWithStream

result = [I, am, a, list]
```



Relax

We'll see `collect()` and the `Collectors` in more detail later.

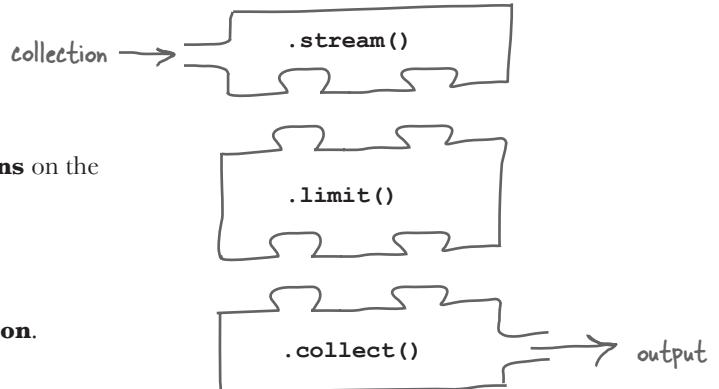
For now, `collect(Collectors.toList())` is a magic incantation to get the output of the stream pipeline in a List.

Finally, we have a result that looks like something we would have expected: we had a List of Strings, and we asked to **limit** that list to the first four items and then **collect** those four items into a new List.

Stream operations are building blocks

We wrote a lot of code just to output the first four elements in the list. We also introduced a lot of new terminology: streams, intermediate operations, and terminal operations. Let's put all this together: you create a **stream pipeline** from three different types of building blocks.

- ① Get the Stream from a **source collection**.



- ② Call zero or more **intermediate operations** on the Stream.

- ③ Output the results with a **terminal operation**.

You need at least the **first** and **last** pieces of the puzzle to use the Streams API. However, you don't need to assign each step to its own variable (which we were doing on the last page). In fact, the operations are designed to be **chained**, so you can call one stage straight after the previous one, without putting each stage in its own variable.

On the last page, all the building blocks for the stream were highlighted (stream, limit, count, collect). We can take these building blocks and rewrite the limit-and-collect operation in this way:

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
```

```

List<String> result = strings.stream()
    .limit(4)
    .collect(Collectors.toList());
  
```

Formatted to align each operation directly underneath the one above, to clearly show each stage.

Get the stream for the collection

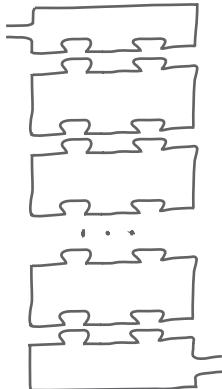
Set a limit to return a maximum of 4 results from the stream

Returns the results of the operation as a List

```
System.out.println("result = " + result);
```

Building blocks can be stacked and combined

Every intermediate operation acts on a Stream and returns a Stream. That means you can stack together as many of these operations as you want, before calling a terminal operation to output the results.



The source, the intermediate operation(s), and the terminal operation all combine to form a Stream Pipeline. This pipeline represents a query on the original collection.

This is where the Streams API becomes really useful. In the earlier example, we needed three building blocks (stream, limit, collect) to create a shorter version of the original List, which may seem like a lot of work for a simple operation.

But to do something more complicated, we can stack together multiple operations in a single **stream pipeline**.

For example, we can sort the elements in the stream before we apply the limit:

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

List<String> result = strings.stream()
    .sorted()           ← Sort what's in the stream (not the
                        original collection), using natural
                        order, before limiting the results.
    .limit(4)          ← Limit the stream to just
                        four elements.
    .collect(Collectors.toList());

System.out.println("result = " + result);
```

```
File Edit Window Help InChains
%java ChainedStream
result = [I, Strings, a, am]
```

← Natural ordering of Strings will place capitalized Strings ahead of lowercase Strings.

Customizing the building blocks

We can stack together operations to create a more advanced query on our collection. We can also customize what the blocks do too. For example, we customized the `limit` method by passing in the maximum number of items to return (four).

If we didn't want to use the natural ordering to sort our Strings, we could define a specific way to sort them. It's possible to set the sort criteria for the `sorted` method (remember, we did something similar in the previous chapter when we sorted Lou's song list).

```
List<String> result = strings.stream()
    .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
    .limit(4)
    .collect(Collectors.toList());
```

File Edit Window Help IgnoreCaps

```
%java ChainedStream
```

```
result = [a, am, I, list]
```

Create complex pipelines block by block

Each new operation you add to the pipeline changes the output from the pipeline. Each operations tell the Streams API *what* it is you want to do.

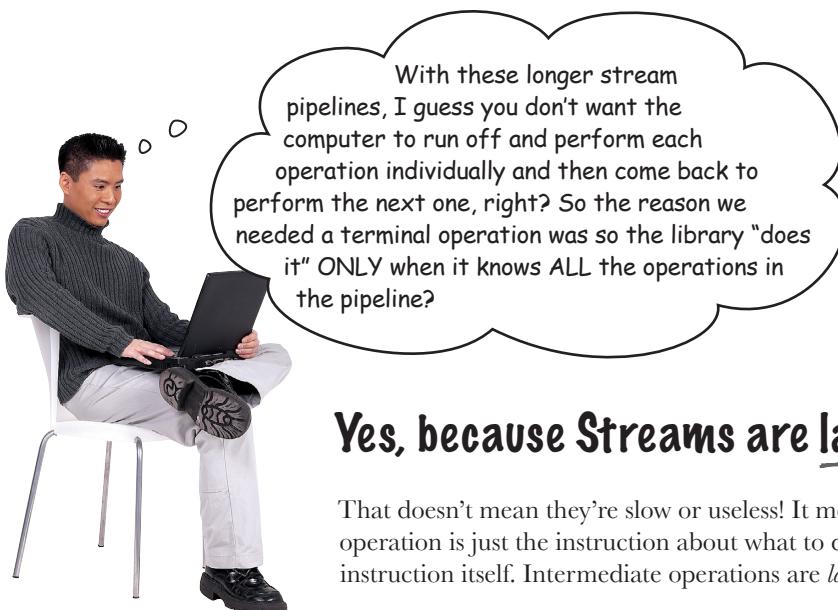
```
List<String> result = strings.stream()
    .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
    .skip(2)
    .limit(4)
    .collect(Collectors.toList());
```

```
File Edit Window Help BoxersDolt
```

```
%java ChainedStream
```

```
result = [I, list, of, Strings]
```

The stream skipped over the first two elements.



Yes, because Streams are lazy

That doesn't mean they're slow or useless! It means that each intermediate operation is just the instruction about what to do; it doesn't perform the instruction itself. Intermediate operations are *lazily evaluated*.

The terminal operation is responsible for looking at the whole list of instructions, all those intermediate operations in the pipeline, and then running the whole set together in one go. Terminal operations are *eager*; they are run as soon as they're called.

This means that in theory it's possible to run the combination of instructions in the most efficient way. Instead of having to iterate over the original collection for each and every intermediate operation, it may be possible to do all the operations while only going through the data once.



Terminal operations do all the work

Since intermediate operations are *lazy*, it's up to the terminal operation to do everything.

- 1** Perform all the intermediate operations as efficiently as possible. Ideally, just going through the original data once.
- 2** Work out the result of the operation, which is defined by the terminal operation itself. For example, this could be a list of values, a single value, or a boolean (true/false).
- 3** Return the result.

Collecting to a List

Now that we know more about what's going on in a terminal operation, let's take a closer look at the "magic incantation" that returns a list of results.

```
List<String> result = strings.stream()
    .sorted()
    .skip(2)
    .limit(4)
    .collect(Collectors.toList());
```

Terminal operation:

1. performs all intermediate operations, in this case: sort; skip; limit.
2. collects the results according to the instructions passed into it
3. returns those results

Collectors is a class that has static methods that provide different implementations of Collector. Look at the Collectors class to find the most common ways to collect up the results.

The collect method takes a Collector, the recipe for how to put together the results. In this case, it's using a helpful predefined Collector that puts the results into a List.

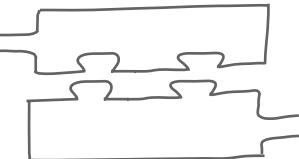
We will look at more Collectors, and other terminal operations, later in the chapter. For now, you know enough to get going with Streams.

Guidelines for working with streams

Like any puzzle or game, there are rules for getting the stream building blocks to work properly.

① You need at least the first and last pieces to create a stream pipeline.

Without the `stream()` piece, you don't get a Stream at all, and without the terminal operation, you're not going to get any results.



② You can't reuse Streams.

It might seem useful to store a Stream representing a query, and reuse it in multiple places, either because the query itself is useful or because you want to build on it and add to it. But once a terminal operation has been called on a stream, you can't reuse any parts of that stream; you have to create a new one. Once a pipeline has executed, that stream is closed and can't be used in another pipeline, even if you stored part of it in a variable for reusing elsewhere. If you try to reuse a stream in any way, you'll get an Exception.

```
Stream<String> limit = strings.stream()
                                .limit(4);
List<String> result = limit.collect(Collectors.toList());
List<String> result2 = limit.collect(Collectors.toList());
```

```
File Edit Window Help ClosingTime
%java LimitWithStream

Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
        at java.base/java.util.stream.AbstractPipeline.
evaluate(AbstractPipeline.java:229)
```

③ You can't change the underlying collection while the stream is operating.

If you do this, you'll see strange results, or exceptions. Think about it—if someone asked you a question about what was in a shopping list and then someone else was scribbling on that shopping list at the same time, you'd give confusing answers too.





So if you shouldn't change the underlying collection while you're querying it, the stream operations don't change the collection either, right?

Correct! Stream operations don't change the original collection.

The Streams API is a way to query a collection, but it **doesn't make changes** to the collection itself. You can use the Streams API to look through that collection and return results based on the contents of the collection, but your original collection will remain the same as it was.

This is actually very helpful. It means you can query collections and output the results from anywhere in your program and know that the data in your original collection is safe; it will not be changed ("mutated") by any of these queries.

You can see this in action by printing out the contents of the original collection after using the Streams API to query it.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

Stream<String> limit = strings.stream()
    .limit(4)
    .collect(Collectors.toList());

System.out.println("strings = " + strings);
System.out.println("result = " + result);
```

```
File Edit Window Help Untouchable
%java LimitWithStream

strings = [I, am, a, list, of, Strings]
result = [I, am, a, list]
```

No changes to original collection after the stream operations are run.

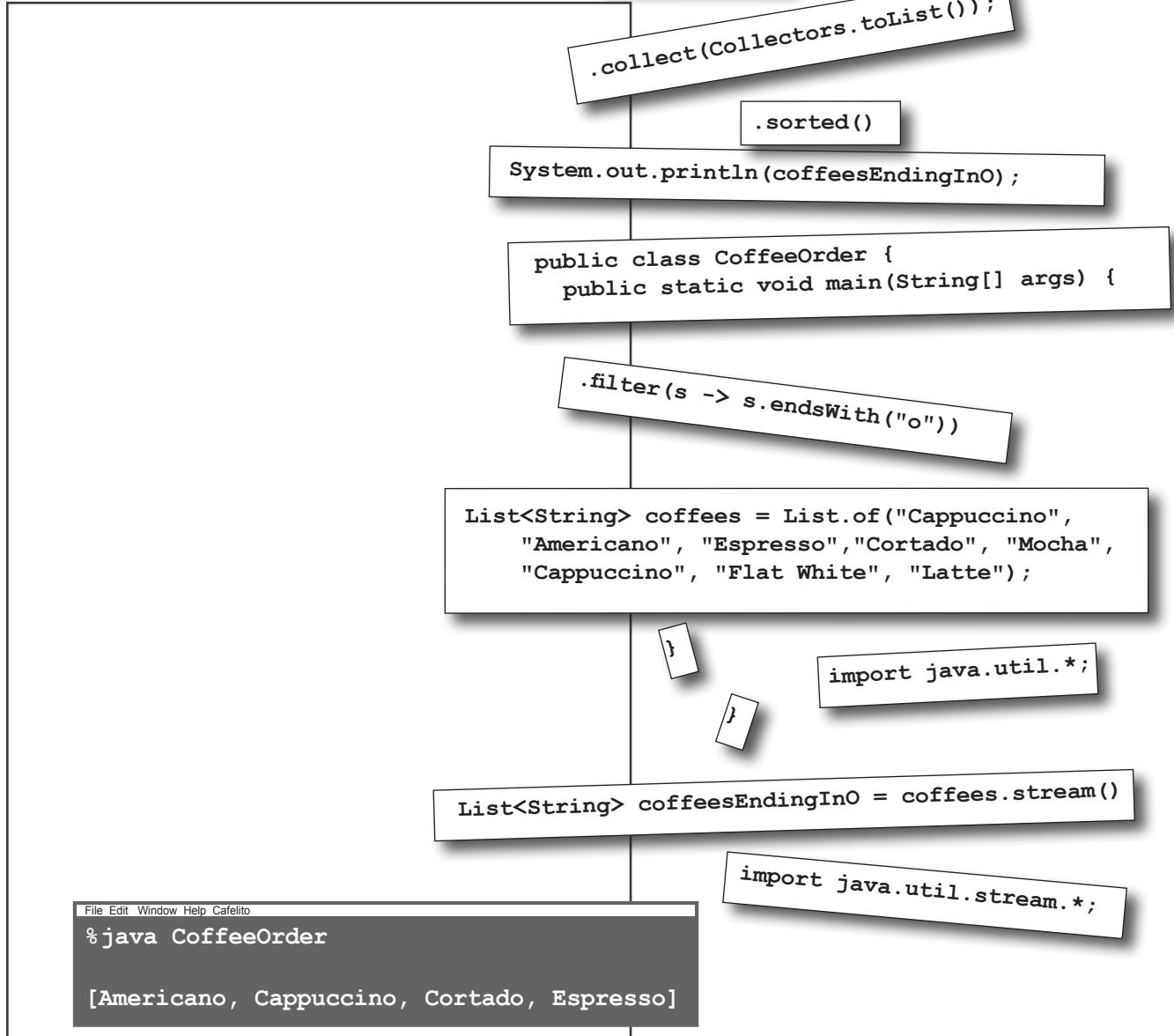
Only the output object has the results of the query. This is a brand new List.



Code Magnets



A Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below?



→ Answers on page 418.

there are no
Dumb Questions

Q: Is there a limit to the number of intermediate operations I can put in a stream pipeline?

A: No, you can keep chaining these operations as much as you like. But do remember that it's not just computers that have to read and understand this code; it's humans too! If the stream pipeline is really long, it might be too complicated to understand. That's when you might want to split it up and assign sections to variables, so you can give these variables useful names.

Q: Is there any point in having a stream pipeline *without* intermediate operations?

A: Yes, you might find that there's a terminal operation that outputs the original collection in some new shape, which is just right for what you need. Be aware, however, that some of the terminal operations are similar to methods that exist on the collection; you don't always need to use streams. For example, if you're just using `count` on a Stream, you could probably use `size` instead, if your original collection is a List. Similarly, anything that is Iterable (like List) already has a `forEach` method; you don't need to use `stream().forEach()`.

Q: You said not to change the source collection while the stream operation is in progress. How is it possible to change the collection from my code, if my code is doing a stream operation?

A: Great question! It's possible to write programs that run different bits of code at the same time. We'll learn about this in Chapters 17 and 18, which cover concurrency. To be safe, it's usually best (not just for Streams, but in general) to create collections that can't be changed if you know they don't need to be changed.

Q: How can I output a List that can't be changed from the `collect` terminal operation?

A: If you're using Java 10 or higher, you can use `Collectors.toUnmodifiableList`, instead of using `Collectors.toList`, when you call `collect`.

Q: Can I get the results of the stream pipeline in a collection that isn't a List?

A: Yes! In the previous chapter we learned that there are a few different kinds of collections for different purposes. The `Collectors` class has convenience methods for collecting `toList`, `toSet`, and `toMap`, as well as (since Java 10) `toUnmodifiableList`, `toUnmodifiableSet`, and `toUnmodifiableMap`.

BULLET POINTS

- You don't have to write detailed code telling the JVM exactly what to do and how to do it. You can use library methods, including the Streams API, to query collections and output the results.
- Use `forEach` on a collection instead of creating a `for` loop. Pass the method a lambda expression of the operation to perform on each element of the collection.
- Create a stream from a collection (a **source**) by calling the **stream** method.
- Configure the query you want to run on the collection by calling one or more **intermediate operations** on the stream.
- You won't get any results until you call a terminal operation. There are a number of different **terminal operations** depending upon what you want your query to output.
- To output the results into a new List, use `collect(Collectors.toList)` as the terminal operation.
- The combination of the source collection, intermediate operations, and terminal operations is a **stream pipeline**.
- Stream operations do not change the original collection; they are a way to query the collection and return a different Object, which is a result of the query.

Hello Lambda, my (not so) old friend

Lambda expressions have cropped up in the streams examples so far, and you can bet your bottom dollar (or euro, or currency of your choice) that you're going to see more of them before this chapter is done.

Having a better understanding of what lambda expressions are will make it easier to work with the Streams API, so let's take a closer look at lambdas.

Passing behavior around

If you wrote a **forEach** method, it might look something like this:

```
void forEach( ?????? ) {  
    for (Element element : list) {  
        }  
    }  
}
```

This is the space where the block of code to run for every list element would go.

What would you put in the place where “?????” is? It would need to somehow **be** the block of code that's going to go into that nice, blank square.

Then you want someone calling the method to be able to say:

```
forEach(do this: System.out.println( item ));
```

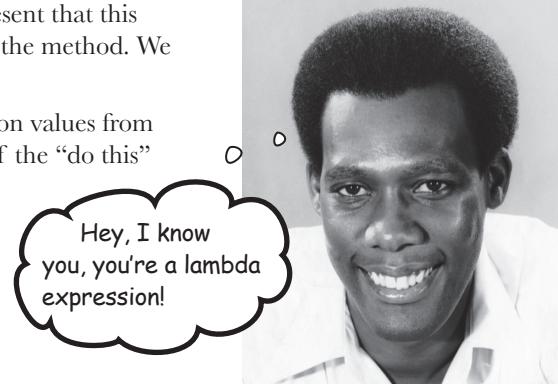
You can't just write this code here, because it will be run straightaway. Instead, we need a way to hand this block of code over to the `forEach` method so that method can call it when it's ready.

This code needs to somehow get hold of the element to print it, but how can it get an element when that code is INSIDE the `forEach` method?

Now, we need to replace the *do this* with some sort of symbol to represent that this code isn't to be run straightaway, but instead needs to be passed into the method. We could use, oh, let's see... “->” as this symbol.

Then we need a way to say “look, this code is going to need to work on values from elsewhere.” We could put the things the code needs on the left side of the “do this” symbol....

```
forEach( item -> System.out.println(item) );
```





OK, so now I get that the lambda I pass in as a method argument is somehow used in the body of that method. But what is the lambda? How can the method USE this chunk of code I just passed it?

Lambda expressions are objects, and you run them by calling their Single Abstract Method

Remember, everything in Java is an Object (well, except for the primitive types), and lambdas are no exception.

A lambda expression implements a Functional Interface.

This means the reference to the lambda expression is going to be a Functional Interface. So, if you want your method to accept a lambda expression, you need to have a parameter whose type is a functional interface. That functional interface needs to be the right “shape” for your lambda.

Back to our imaginary **forEach** example; our parameter needs to implement a Functional Interface. We also need to call that lambda expression somehow, passing in the list element.

Remember, Functional Interfaces have a Single Abstract Method (SAM) . It's this method, whatever its name is, that gets called when we want to run the lambda code.

```
void forEach( SomeFunctionalInterface lambda ) {
    for (Element element : list) {
        lambda.singleAbstractMethodName(element);
    }
}
```

This would be the name of whatever is the Single functional interface.

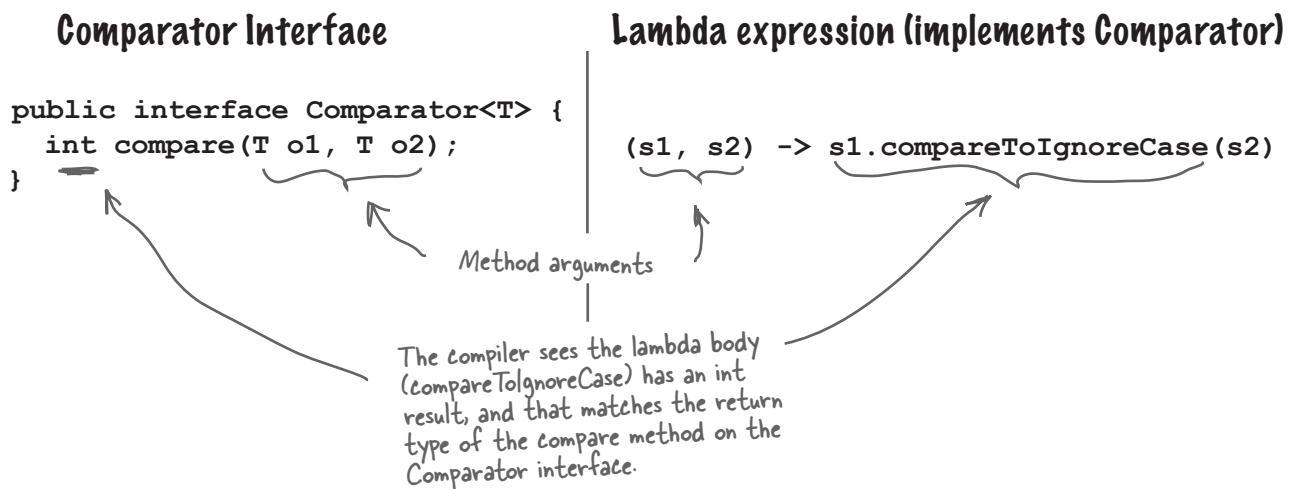
This is a placeholder type to give you an idea of what the method would look like. We'll look at specific Functional Interfaces throughout this chapter.

“element” is the lambda’s parameter, the “item” in the lambda expression on the last page.

Lambdas aren't magic;
they're just classes
like everything else.

The shape of lambda expressions

We've seen two lambda expressions that implement the Comparator interface: the example for sorting Lou's songs in the previous chapter, and the lambda expression we passed into the `sorted()` stream operation on page 381. Look at this last example side by side with the Comparator Functional Interface.



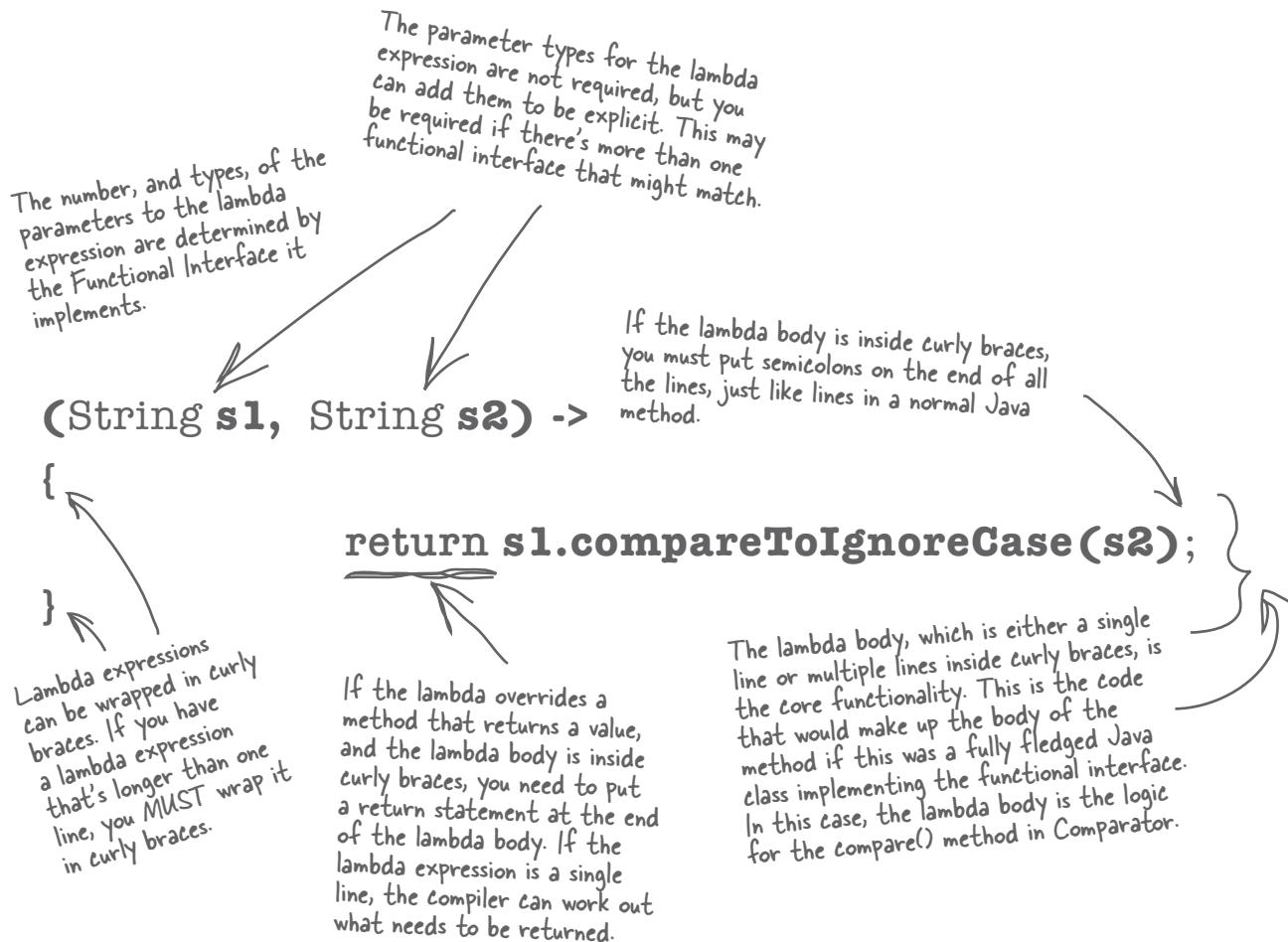
You might be wondering where the **return** keyword is in the lambda expression. The short version is: you don't need it. The longer version is, if the lambda expression is a single line, and if the functional interface's method signature requires a returned value, the compiler just assumes that your one line of code will generate the value that is to be returned.

The lambda expression can also be written like this, if you want to add all the parts a lambda expression can have:

```
(String s1, String s2) -> {
    return s1.compareToIgnoreCase(s2);
}
```

Anatomy of a lambda expression

If you take a closer look at this expanded version of the lambda expression that implements `Comparator<String>`, you'll see it's not so different from a standard Java method.



The shape of the lambda (its parameters, return type, and what it can reasonably be expected to do) is dictated by the Functional Interface it implements.

Variety is the spice of life

Lambda expressions can come in all shapes and sizes, and still conform to the same basic rules that we've seen.

A lambda might have more than one line

A lambda expression is effectively a method, and can have as many lines as any other method. Multiline lambda expressions **must** be inside curly braces. Then, like any other method code, every line **must** end in a semicolon, and if the method is supposed to return something, the lambda body **must** include the word “return” like any normal method.



Life would be boring if we all looked the same.

method.

Here's a lambda expression that implements Comparator<String> and results in the collection being sorted by string length in descending order.

```
(str1, str2) -> {  
    int l1 = str1.length();  
    int l2 = str2.length();  
    return l2 - l1;  
}
```

Semicolons required.

Curly braces for multiline lambda expressions.

Return keyword required.

Single-line lambdas don't need ceremony

If your lambda expression is a single line, it makes it much easier for the compiler to guess what's going on. Therefore, we can leave out a lot of the "boilerplate" syntax. If we shrink the lambda expression from the last example into a single line, it looks like this:

No need for curly braces →
(str1, str2) → str2.length() - str1.length() ←
No need for “return” ← No semicolons

This is the same Functional Interface (Comparator) and performs the same operation. Whether you use multiline lambdas or single-line lambdas is completely up to you. It will probably depend upon how complicated the logic in the lambda expression is, and how easy you think it is to read—sometimes longer code can be more descriptive.

Later, we'll see another approach for handling long lambda expressions.

A lambda might not return anything

The Functional Interface's method might be declared void; i.e., it doesn't return anything. In these cases, the code inside the lambda is simply run, and you don't need to return any values from the lambda body.

This is the case for lambda expressions in a **forEach** method.

Multiline lambda

```

Look! No round brackets! We'll
see this again in a minute.
str -> {
    String output = "str = " + str;
    System.out.println(output);
}
No return value
  
```

@FunctionalInterface
public interface Consumer<T> {
 void accept(T t);
}

Method is void on the Functional Interface

A lambda might have zero, one, or many parameters

The number of parameters the lambda expression needs is dependent upon the number of parameters the Functional Interface's method takes. The parameter types (e.g., the name “String”) are not usually required, but you can add them if you think it makes it easier to understand the code. You may need to add the types if the compiler can't automatically work out which Functional Interface your lambda implements.

If a lambda expression doesn't take any parameters, you need to use empty brackets to show this.

```
(() -> System.out.println("Hello!"))
```

No method parameters

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

No need for round brackets if it's a single parameter without a type (remember param types are optional)

```
str -> System.out.println(str)
```

One method parameter

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
(str1, str2) -> str1.compareToIgnoreCase(str2)
```

Two method parameters

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

How can I tell if a method takes a lambda?

By now you've seen that lambda expressions are implementations of a functional interface—that is, an Interface with a Single Abstract Method. That means the **type** of a lambda expression is this interface.

Go ahead and create a lambda expression. Instead of passing this into some method, as we have been doing so far, assign it to a variable. You'll see it can be treated just like any other Object in Java, because everything in Java is an Object. The variable's type is the Functional Interface.

```
Comparator<String> comparator = (s1, s2) -> s1.compareToIgnoreCase(s2);  
  
Runnable runnable = () -> System.out.println("Hello!");  
  
Consumer<String> consumer = str -> System.out.println(str);
```

How does this help us see if a method takes a lambda expression? Well, the method's parameter type will be a Functional Interface. Take a look at some examples from the Streams API:

Stream<T> filter(**Predicate**<? super T> predicate)

boolean allMatch(**Predicate**<? super T> predicate)

```
@FunctionalInterface  
public interface Predicate<T>
```

<R> Stream<R> map(**Function**<? super T,? extends R> mapper)

```
@FunctionalInterface  
public interface Function<T, R>
```

void forEach(**Consumer**<? super T> action)

```
@FunctionalInterface  
public interface Consumer<T>
```



Exercise

BE the Compiler, advanced

Your job is to play compiler and determine which of these statements would compile. But some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations.

The signatures of the functional interfaces are on the right, for your convenience.

Check the box if the statement would compile.

- Runnable r = () -> System.out.println("Hi!");
- Consumer<String> c = s -> System.out.println(s);
- Supplier<String> s = () -> System.out.println("Some string");
- Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2);
- Runnable r = (String str) -> System.out.println(str);
- Function<String, Integer> f = s -> s.length();
- Supplier<String> s = () -> "Some string";
- Consumer<String> c = s -> "String" + s;
- Function<String, Integer> f = (int i) -> "i = " + i;
- Supplier<String> s = s -> "Some string: " + s;
- Function<String, Integer> f = (String s) -> s.length();

```
public interface Runnable {
    void run();
}
```

```
public interface Consumer<T> {
    void accept(T t);
}
```

```
public interface Supplier<T> {
    T get();
}
```

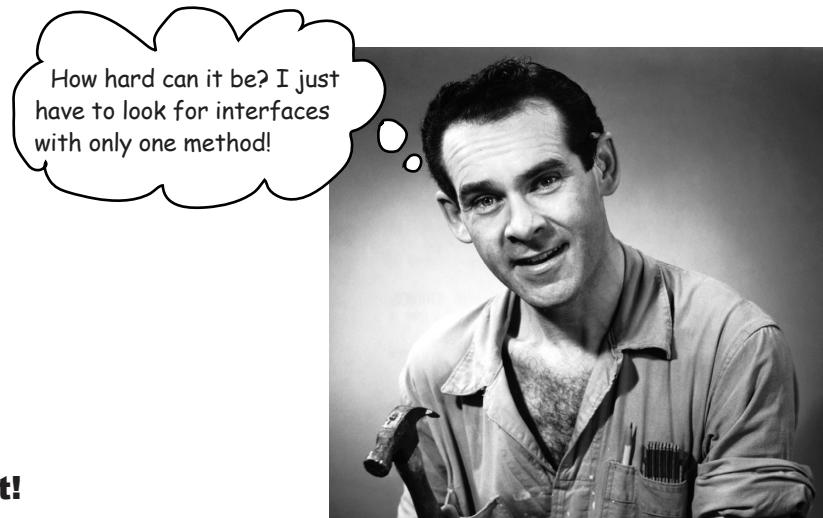
```
public interface Function<T, R> {
    R apply(T t);
}
```

—————> Answers on page 418.

Spotting Functional Interfaces

So far we've seen Functional Interfaces that are marked with a `@FunctionalInterface` annotation (we'll cover annotations in Appendix B), which conveniently tells us this interface has a Single Abstract Method and can be implemented with a lambda expression.

Not all functional interfaces are tagged this way, particularly in older code, so it's useful to understand how to spot a functional interface for yourself.



Not so fast!

Originally, the only kind of methods allowed in interfaces were **abstract** methods, methods that need to be *overridden* by any class that *implements* this interface. But as of Java 8, interfaces can also contain **default** and **static** methods.

You saw static methods in Chapter 10, *Numbers Matter*, and you'll see them later in this chapter too. These are methods that don't need to belong to an instance, and are often used as helper methods.

Default methods are slightly different. Remember abstract classes from Chapter 8, *Serious Polymorphism*? They had abstract methods that need to be overridden, and standard methods with a body. On an interface, a default method works a bit like a standard method in an abstract class—they have a body, and will be inherited by subclasses.

Both default and static methods have a method body, with defined behavior. With interfaces, any method that is not defined as **default** or **static** is an abstract method that *must* be overridden.

Functional interfaces in the wild

Now that we know interfaces can have **non**-abstract methods, we can see there's a bit more of a trick to identifying interfaces with just one abstract method. Take a look at our old friend, Comparator. It has a **lot** of methods! And yet it's still a SAM-type; it has only one Single Abstract Method. It's a Functional Interface we can implement as a lambda expression.

Here it is! This is our Single Abstract Method.

Don't be misled by this method! It's not static or default, but it's not actually abstract either—it's inherited from Object. It does have a method body, defined by the Object class.

Modifier and Type	Method
int	compare(T o1, T o2)
static <T,U extends Comparable<? super U>> Comparator<T>	comparing(Function<? super T,> keyExtractor)
static <T,U> Comparator<T>	comparing(Function<? super T,> keyExtractor, Comparator<U> keyComparator)
static <T> Comparator<T>	comparingDouble(ToDoubleFunction<? super T> keyExtractor)
static <T> Comparator<T>	comparingInt(ToIntFunction<? super T> keyExtractor)
static <T> Comparator<T>	comparingLong(ToLongFunction<? super T> keyExtractor)
boolean	equals(Object obj)
static <T extends Comparable<? super T>> Comparator<T>	naturalOrder()
static <T> Comparator<T>	nullsFirst(Comparator<? super T> comparator)
static <T> Comparator<T>	nullsLast(Comparator<? super T> comparator)
default Comparator<T>	reversed()



Which of these interfaces has a Single Abstract Method and can therefore be implemented as a lambda expression?

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

ActionListener	
Modifier and Type	Method
void	actionPerformed(ActionEvent e)

Iterator

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

Function

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

SocketOption

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

→ Answers on page 419.

Lou's back!

Lou's been running his new jukebox management software from the last chapter for some time now, and he wants to learn so much more about the songs played on the diner's jukebox. Now that he has the data, he wants to slice-and-dice it and put it together in a new shape, just as he does with the ingredients of his famous Special Omelette!

He's thinking there are all kinds of information he could learn about the songs that are played, like:

- What are the top five most-played songs?
- What sort of genres are played?
- Are there any songs with the same name by different artists?

We *could* find these things out writing a for loop to look at our song data, performing checks using if statements, and perhaps putting songs, titles, or artists into different collections to find the answers to these questions.

**But now that we know about the Streams API,
we know there's an easier way....**

The code on the next page is your **mock** code; calling `Songs.getSongs()` will give you a List of Song objects that you can assume looks just like the real data from Lou's jukebox.



Type in the Ready-Bake Code on the next page, including filling out the rest of the Song class. When you've done that, create a main method that prints out all the songs.

What do you expect the output to look like?



Now that I have data about what's been played on my jukebox, I want to know more!



Ready-Bake Code

Here's an updated "mock" method. It will return some test data that we can use on to try out some of the reports Lou wants to create for the jukebox system. There's also an updated Song class.

```

class Songs {
    public List<Song> getSongs() {
        return List.of(
            new Song("$10", "Hitchhiker", "Electronic", 2016, 183),
            new Song("Havana", "Camila Cabello", "R&B", 2017, 324),
            new Song("Cassidy", "Grateful Dead", "Rock", 1972, 123),
            new Song("50 ways", "Paul Simon", "Soft Rock", 1975, 199),
            new Song("Hurt", "Nine Inch Nails", "Industrial Rock", 1995, 257),
            new Song("Silence", "Delerium", "Electronic", 1999, 134),
            new Song("Hurt", "Johnny Cash", "Soft Rock", 2002, 392),
            new Song("Watercolour", "Pendulum", "Electronic", 2010, 155),
            new Song("The Outsider", "A Perfect Circle", "Alternative Rock", 2004, 312),
            new Song("With a Little Help from My Friends", "The Beatles", "Rock", 1967, 168),
            new Song("Come Together", "The Beatles", "Blues rock", 1968, 173),
            new Song("Come Together", "Ike & Tina Turner", "Rock", 1970, 165),
            new Song("With a Little Help from My Friends", "Joe Cocker", "Rock", 1968, 46),
            new Song("Immigrant Song", "Karen O", "Industrial Rock", 2011, 12),
            new Song("Breathe", "The Prodigy", "Electronic", 1996, 337),
            new Song("What's Going On", "Gaye", "R&B", 1971, 420),
            new Song("Hallucinate", "Dua Lipa", "Pop", 2020, 75),
            new Song("Walk Me Home", "P!nk", "Pop", 2019, 459),
            new Song("I am not a woman, I'm a god", "Halsey", "Alternative Rock", 2021, 384),
            new Song("Pasos de cero", "Pablo Alborán", "Latin", 2014, 117),
            new Song("Smooth", "Santana", "Latin", 1999, 244),
            new Song("Immigrant song", "Led Zeppelin", "Rock", 1970, 484)));
    }
}

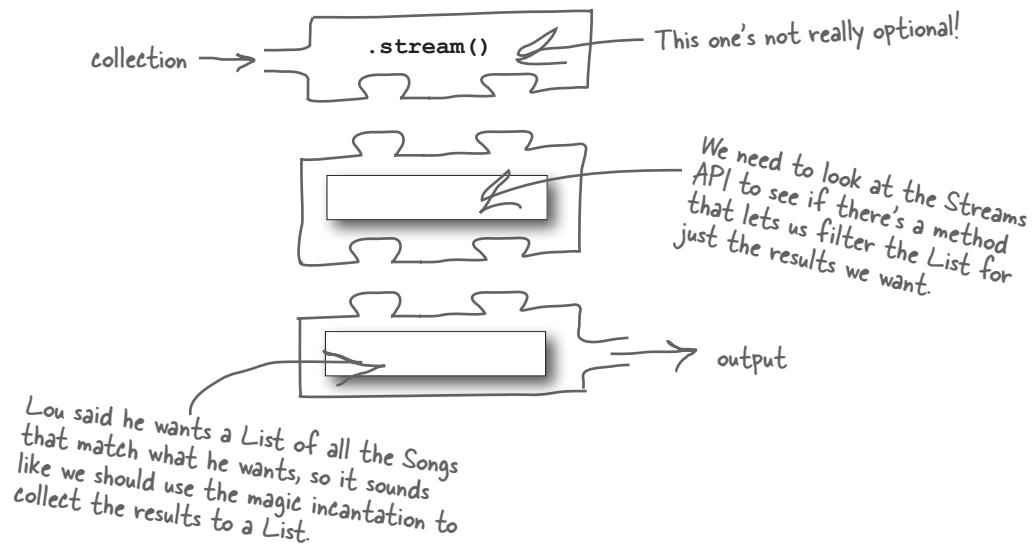
public class Song {
    private final String title;
    private final String artist;
    private final String genre;
    private final int year;
    private final int timesPlayed;
    // Practice for you! Create a constructor, all the getters and a toString()
}

```

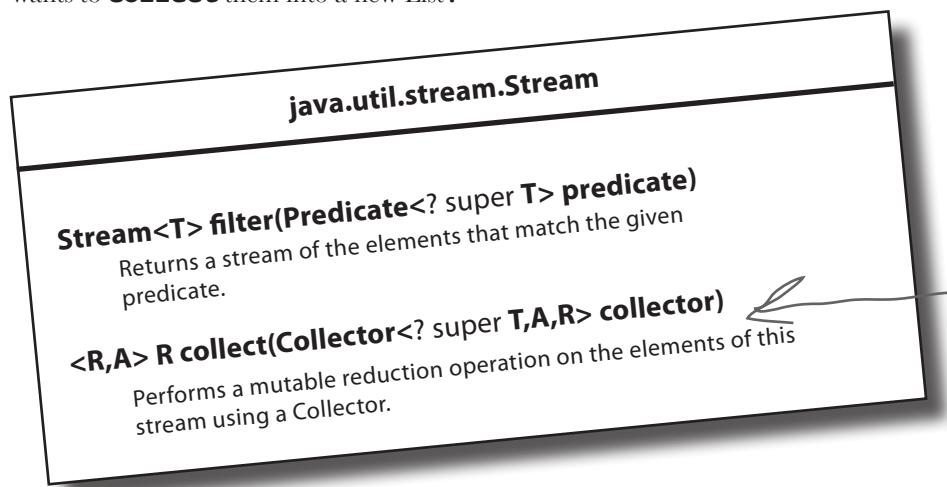
Lou's Challenge #1: Find all the "rock" songs

The data in the updated song list contains the *genre* of the song. Lou's noticed that the diner's clientele seem to prefer variations on rock music, and he wants to see a list of all the songs that fall under some genre of "rock."

This is the Streams chapter, so clearly the solution is going to involve the Streams API. Remember, there are three types of pieces we can put together to form a solution.



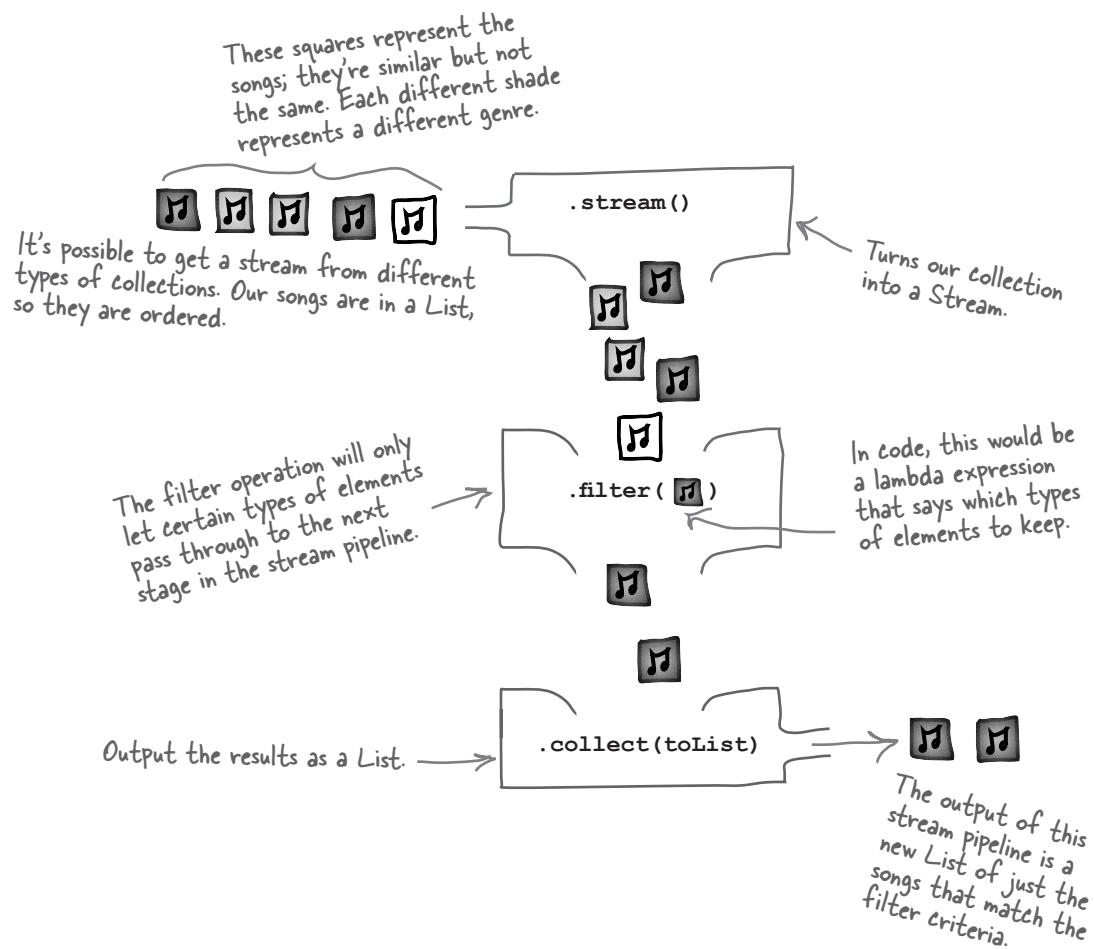
Fortunately, there are hints about how to create a Streams API call based on the requirements Lou gave us: he wants to **filter** for just the Songs with a particular genre, and he wants to **collect** them into a new List.



Remember that for now we're just going to use the "magic incantation" to collect into a List.

Filter a stream to keep certain elements

Let's see how a filter operation might work on the list of songs.

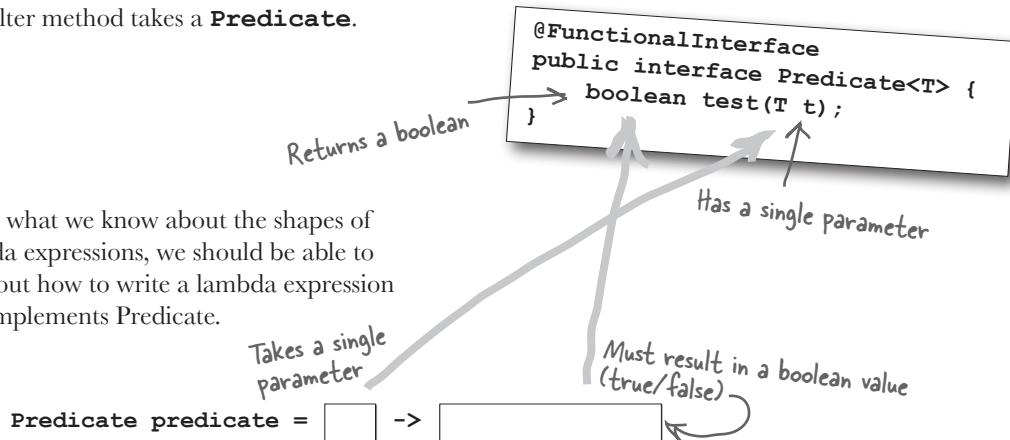


Let's Rock!

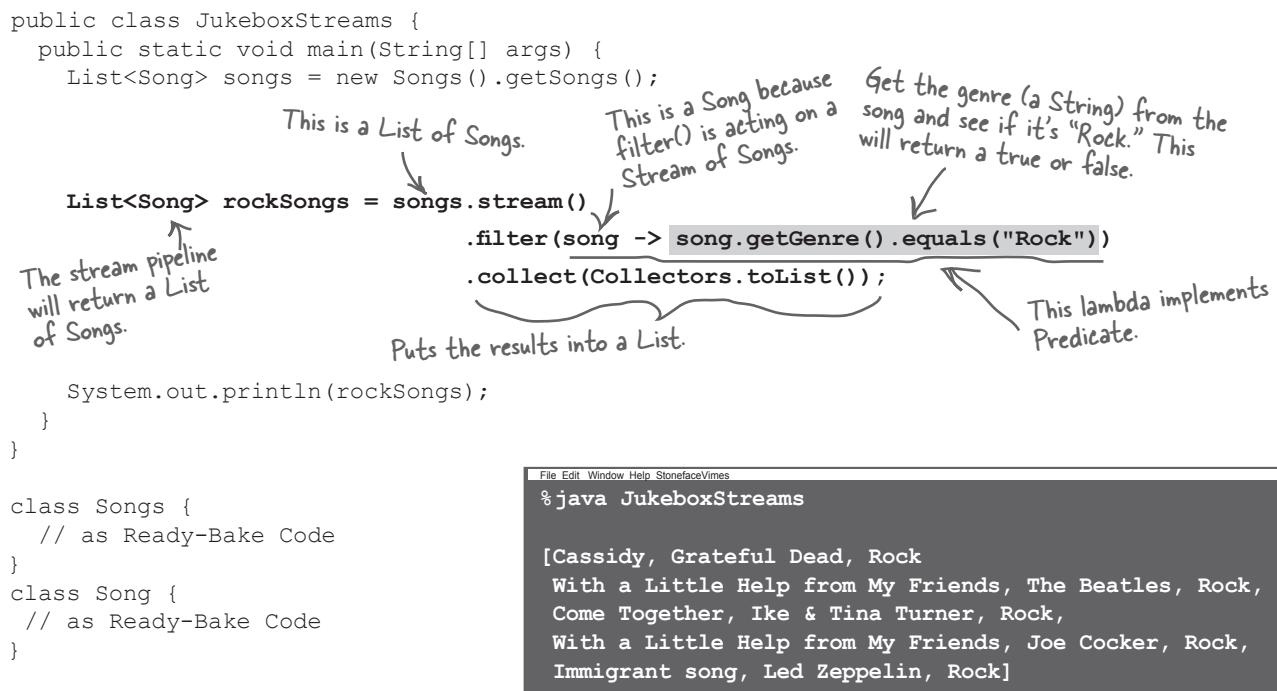
So adding a **filter** operation filters out elements that we don't want, and the stream continues with just the elements that meet our criteria. It should come as no surprise to find that you can use a lambda expression to state which elements we want to keep in the stream.

The filter method takes a **Predicate**.

Given what we know about the shapes of lambda expressions, we should be able to work out how to write a lambda expression that implements Predicate.



We'll know what the type of the single parameter is when we plug it into the Stream operation, since the input type to the lambda will be determined by the types in the stream.



Getting clever with filters

The **filter** method, with its “simple” true or false return value, can contain sophisticated logic to filter elements in, or out, of the stream. Let’s take our filter one step further and actually do what Lou asked:

*He wants to see a list of all the songs that fall under **some genre** of “rock.”*

He doesn’t want to see just the songs that are classed as “Rock,” but any genre that is kinda Rock-like. We should search for any genre that has the word “Rock” in it somewhere.

There’s a method in String that can help us with this, it’s called **contains**.

```
List<Song> rockSongs = songs.stream()
    .filter(song -> song.getGenre().contains("Rock"))
    .collect(Collectors.toList());
```

Returns true if the genre
has the word “Rock” in it
anywhere

```
File Edit Window Help YouRock
%java JukeboxStreams

[Cassidy, Grateful Dead, Rock
 50 ways, Paul Simon, Soft Rock
 Hurt, Nine Inch Nails, Industrial Rock
 Hurt, Johnny Cash, Soft Rock
 ...

```

Now the stream returns different
types of rock songs.

*Output chopped down to save space
in the book—save the trees!*



Brain Barbell

Can you write a filter operation that can select songs:

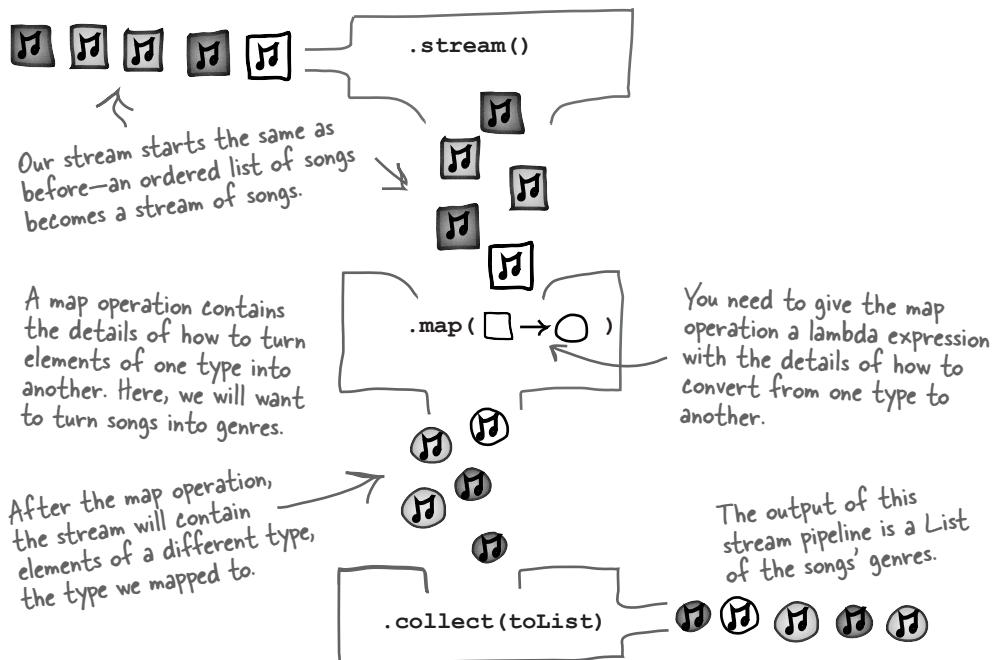
- By The Beatles
- That start with “H”
- More recent than 1995

Lou's Challenge #2: List all the genres

Lou now senses that the genres of music that the diners are listening to are more complicated than he thought. He wants a list of all the genres of the songs that have been played.

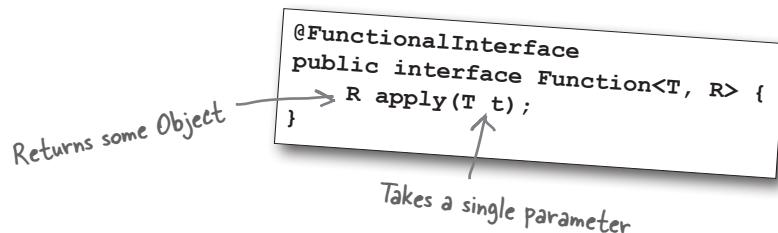
So far, all of our streams have returned the same types that they started with. The earlier examples were Streams of Strings, and returned Lists of Strings. Lou's previous challenge started with a List of Songs and ended up with a (smaller) List of Songs.

Lou now wants a list of genres, which means we need to somehow turn the song elements in the stream into genre (String) elements. This is what **map** is for. The map operation states how to map *from* one type *to* another type.



Mapping from one type to another

The map method takes a **Function**. The generics by definition are a bit vague, which makes it a little tricky to understand, but Functions do one thing: they take something of one type and return something of a different type. Exactly what's needed for mapping varies from one type to another.



Let's see what it looks like when we use `map` in a stream pipeline.

```

List<String> genres = songs.stream()
    .map(song -> song.getGenre())
    .collect(toList());
  
```

Annotations on the code:

- `The result will be a List of Strings, because genre is a String.`
- `This is a List of Song objects.`
- `A single parameter, a Song because this map() is on a Stream of Songs.`
- `Puts the results into a List`
- `The lambda body can return an object of any type. By calling getGenre on the song, the stream after this point will be a stream of (genre) Strings.`

The map's lambda expression is similar to the one for filter; it takes a song and turns it into something else. Instead of returning a boolean, it returns some other object, in this case a String containing the song's genre.

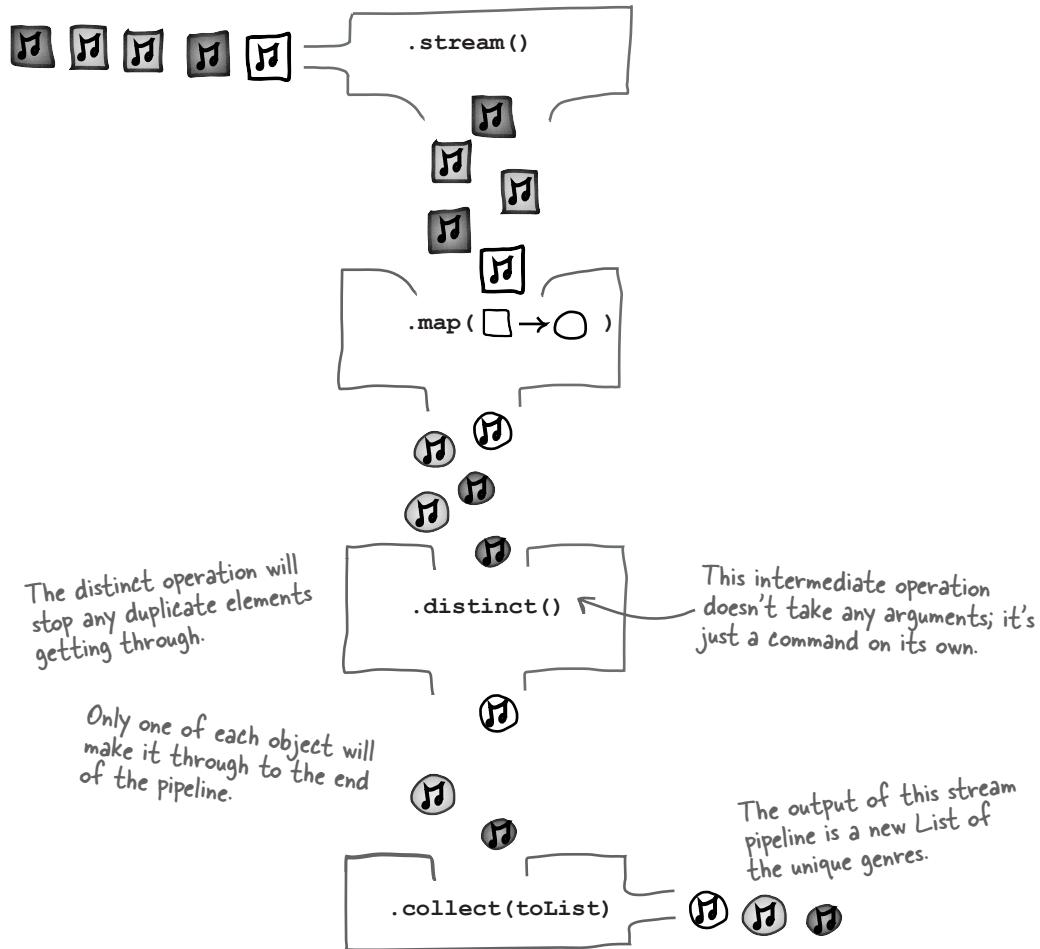
```

File Edit Window Help RoadToNowhere
%java JukeboxStreams

[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Electronic, Soft Rock, Electronic, Alternative Rock, Rock, Blues rock, Rock, Rock, Industrial Rock, Electronic, R&B, Pop, Pop, Alternative Rock, Latin, Latin, Rock]
  
```

Removing duplicates

We've got a list of all the genres in our test data, but Lou probably doesn't want to wade through all these duplicate genres. The `map` operation on its own will result in an output List that's the same size as the input List. Since stream operations are designed to be stacked together, perhaps there's another operation we can use to get just one of every element in the stream?



Only one of every genre

All we need to do is to add a distinct operation to the stream pipeline, and we'll get just one of each genre.

```
List<String> genres = songs.stream()
    .map(song -> song.getGenre())
    ↗ .distinct()
    .collect(Collectors.toList());
```

Having this in the stream pipeline means there will be no duplicates after this point

Outputs a much more readable list of all the genres

```
File Edit Window Help UniqueIsGood
% java JukeboxStreams
[Electronic, R&B, Rock, Soft Rock,
Industrial Rock, Alternative Rock,
Blues rock, Pop, Latin]
```

Just keep building!

A stream pipeline can have any number of intermediate operations. The power of the Streams API is that we can build up complex queries with understandable building blocks. The library will take care of running this in a way that is as efficient as possible. For example, we could create a query that returns a list of all the artists that have covered a specific song, excluding the original artists, by using a map operation and multiple filters.

```
String songTitle = "With a Little Help from My Friends";
List<String> result = allSongs.stream()
    .filter(song -> song.getTitle().equals(songTitle))
    .map(song -> song.getArtist())
    .filter(artist -> !artist.equals("The Beatles"))
    .collect(Collectors.toList());
```



Sharpen your pencil

Try annotating this code yourself.
What do each of the filters do?
What does the map do?

→ Yours to solve.

Sometimes you don't even need a lambda expression

Some lambda expressions do something simple and predictable, given the type of the parameter or the shape of the functional interface. Look again at the lambda expression for the `map` operation.

```
Function<Song, String> getGenre = song -> song.getGenre();
```

Instead of spelling this whole thing out, you can point the compiler to a method that does the operation we want, using a **method reference**.

```
Function<Song, String> getGenre = Song::getGenre;
```

The input parameter to this Function is a Song.

The output of this Function needs to be a String.

The output of getGenre() is a String, just like the Function needs.

This is the method call in the lambda body.

A method reference—instead of using a “.” that would cause the compiler to call the method, use a “::” to point the compiler in the direction of the method.

Method references can replace lambda expressions in a number of different cases. Generally, we might use a method reference if it makes the code easier to read.

Take our old friend the `Comparator`, for example. There are a lot of helper methods on the `Comparator` interface that, when combined with a method reference, let you see which value is being used for sorting and in which direction. Instead of doing this, to order the songs from oldest to newest:

```
List<Song> result = allSongs.stream()
    .sorted(o1, o2) -> o1.getYear() - o2.getYear())
    .collect(toList());
```

Use a method reference combined with a `static` helper method from `Comparator` to state what the comparison should be:

```
List<Song> result = allSongs.stream()
    .sorted(Comparator.comparingInt(Song::getYear))
    .collect(toList());
```



You don't need to use method references if you don't feel comfortable with them. What's important is to be able to recognize the “::” syntax, especially in a stream pipeline.

Method references can replace lambda expressions, but you don't have to use them.

Sometimes method references make the code easier to understand.

Collecting results in different ways

While `Collectors.toList` is the most commonly used Collector, there are other useful Collectors. For example, instead of using `distinct` to solve the last challenge, we could collect the results into a Set, which does not allow duplicates. The advantage of using this approach is that anything else that uses the results knows that because it's a Set, *by definition* there will be no duplicates.

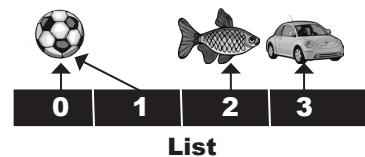
```
Set<String> genres = songs.stream()
    .map(song -> song.getGenre())
    .collect(Collectors.toSet());
```

Store the results in a Set of Strings, not a List. Sets cannot contain duplicates.

Put the results into a Set, which will automatically only have unique entries.

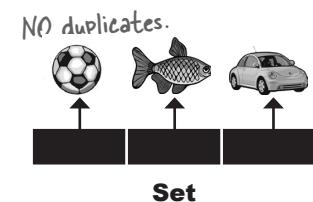
Collectors.toList and Collectors.toUnmodifiableList

You've already seen `toList`. Alternatively, you can get a List that can't be changed (no elements can be added, replaced or removed) by using `Collectors.toUnmodifiableList` instead. This is only available from Java 10 onward.



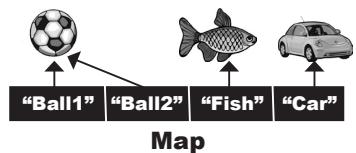
Collectors.toSet and Collectors.toUnmodifiableSet

Use these to put the results into a Set, rather than a List. Remember that a Set cannot contain duplicates, and is not usually ordered. If you're using Java 10 or higher, you can use `Collectors.toUnmodifiableSet` if you want to make sure your results aren't changed by anything.



Collectors.toMap and Collectors.toUnmodifiableMap

You can collect your stream into a Map of key/value pairs. You will need to provide some functions to tell the collector what will be the key and what will be the value. You can use `Collectors.toUnmodifiableMap` to create a map that can't be changed, from Java 10 onward.



Collectors.joining

You can create a String result from the stream. It will join together all the stream elements into a single String. You can optionally define the *delimiter*, the character to use to separate each element. This can be very useful if you want to turn your stream into a String of Comma Separated Values (CSV).

But wait, there's more!

Collecting the results is not the only game in town; `collect` is just one of many terminal operations.

Checking if something exists

You can use terminal operations that return a boolean value to look for certain things in the stream. For example, we can see if any R&B songs have been played in the diner.

```
boolean result =
    songs.stream()
        .anyMatch(s -> s.getGenre().equals("R&B"));
```

```
boolean anyMatch(Predicate p);
boolean allMatch(Predicate p);
boolean noneMatch(Predicate p);
```

Find a specific thing

Terminal operations that return an `Optional` value look for certain things in the stream. For example, we can find the first song played that was released in 1995.

```
Optional<Song> result =
    songs.stream()
        .filter(s -> s.getYear() == 1995)
        .findFirst();
```

```
Optional<T> findAny();
Optional<T> findFirst();
Optional<T> max(Comparator c);
Optional<T> min(Comparator c);
Optional<T> reduce(BinaryOperator a);
```

Count the items

There's a count operation that you can use to find out the number of elements in your stream. We could find the number of unique artists, for example.

```
long result =
    songs.stream()
        .map(Song::getArtist)
        .distinct()
        .count();
```

```
long count();
```

There are even more terminal operations, and some of them depend upon the type of Stream you're working with.

Remember, the API documentation can help you figure out if there's a built-in operation that does what you want.



Wait a minute. How can a result be "Optional"? What does that even mean?

Well, some operations may return something, or may not return anything at all

It might seem weird that a method *may* or *may not* return a value, but it happens all the time in real life.

Imagine you're at an ice-cream stand, and you ask for strawberry ice cream.

Strawberry ice cream, please!

Here you go!

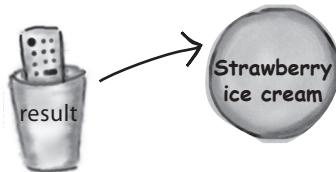
```
IceCream iceCream =  
    getIceCream("Strawberry");
```

Easy, right? But what if they don't have any strawberry? The ice-cream person is likely to tell you "we don't have that flavor."

We don't have any, sorry.

It's then up to you what you do next—perhaps order chocolate instead, find another ice-cream place, or maybe just go home and sulk about your lack of ice cream.

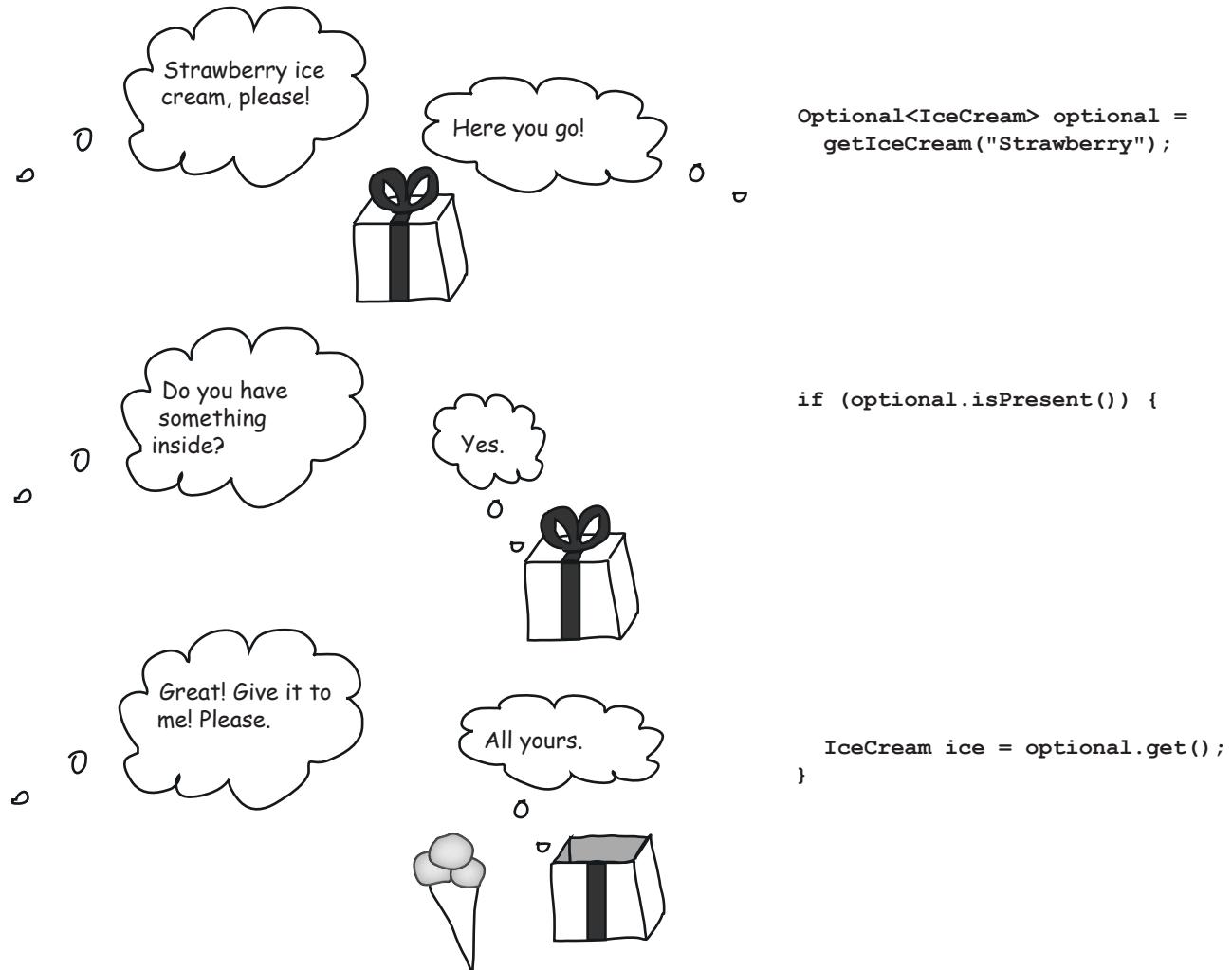
Imagine trying to do this in the Java world. In the first example, you get an ice-cream instance. In the second, you get...a String message? But a message doesn't fit into an ice-cream-shaped variable. A null? But what does null really mean?



optional

Optional is a wrapper

Since Java 8, the normal way for a method to declare that *sometimes it might not return a result* is to return an **Optional**. This is an object that *wraps* the result, so you can ask “Did I get a result? Or is it empty?” Then you can make a decision about what to do next.

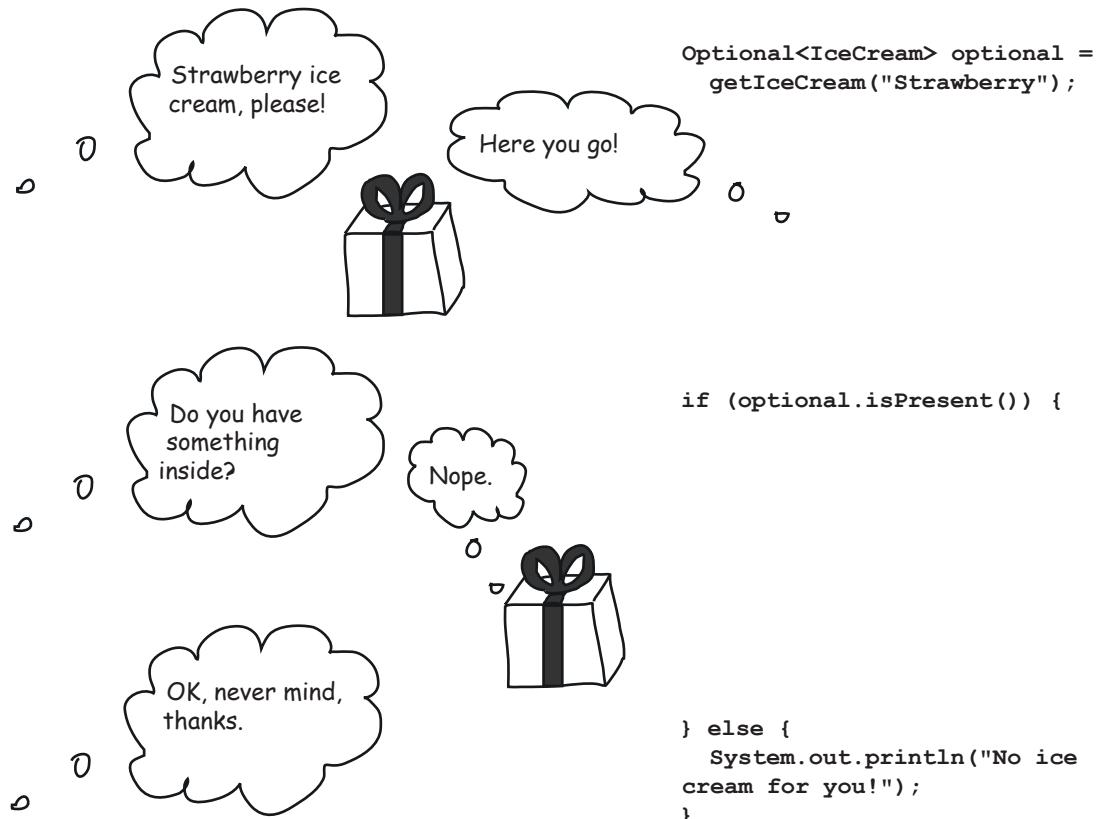




You've just introduced two new steps for me to get my ice cream!

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.

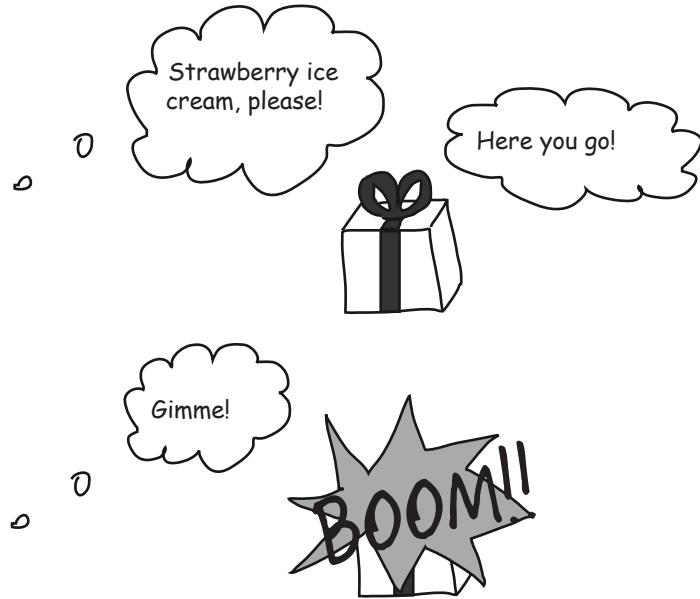


In the past, methods might have thrown Exceptions for this case, or return “null”, or a special type of “Not Found” ice-cream instance. Returning an *Optional* from a method makes it really clear that anything calling the method **needs** to check if there’s a result first, and **then** make their own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

optional

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.

puzzle: Pool Puzzle



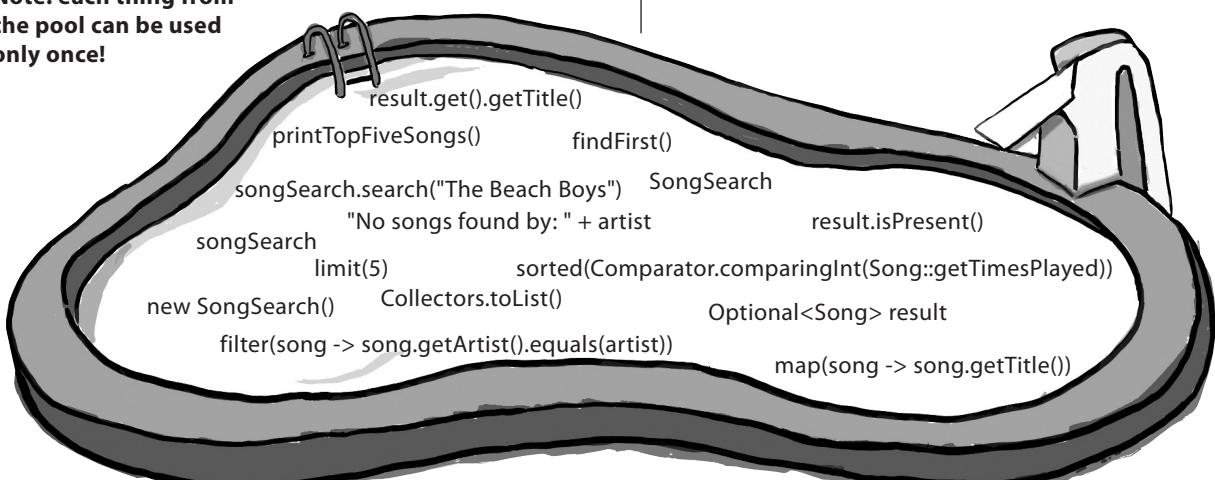
Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **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.

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!



```
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(_____ );
                }
            }
        }
    }
}
```



Mixed Messages (from page 372)

Candidates:

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

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

```
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

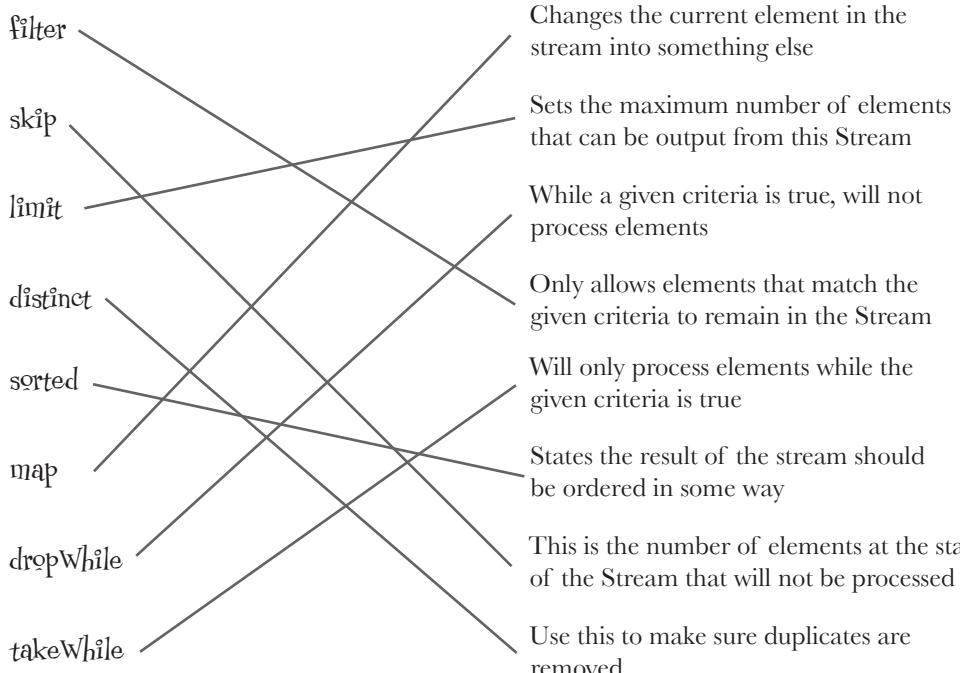
2 3 4 5

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]



WHO DOES WHAT? (from page 374)





Code Magnets (from page 386)

```

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);
    }
}

```

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

File Edit Window Help Cafelito
% java CoffeeOrder
[Americano, Cappuccino,
Cortado, Espresso]

BE the Compiler (from page 395)

- Runnable r = () -> System.out.println("Hi!");
- Consumer<String> c = s -> System.out.println(s); *Should return a String but doesn't*
- Supplier<String> s = () -> System.out.println("Some string"); *Should take only one parameter but has two*
- Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2); *Should not have parameters*
- Runnable r = (String str) -> System.out.println(str); *Should not have parameters*
- Function<String, Integer> f = s -> s.length(); *This single-line lambda effectively returns a String*
- Supplier<String> s = () -> "Some 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.*
- Consumer<String> c = s -> "String" + s; *Should have a String parameter and return an int, but instead it has an int param and returns a String*
- Function<String, Integer> f = (int i) -> "i = " + i; *Should not have any parameters*
- Supplier<String> s = s -> "Some string: " + s; *Should take a String parameter. Should return an int, but actually returns nothing.*
- Function<String, Integer> f = () -> System.out.println("Some string"); *Should return an int, but actually returns nothing.*



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.



Poōl 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



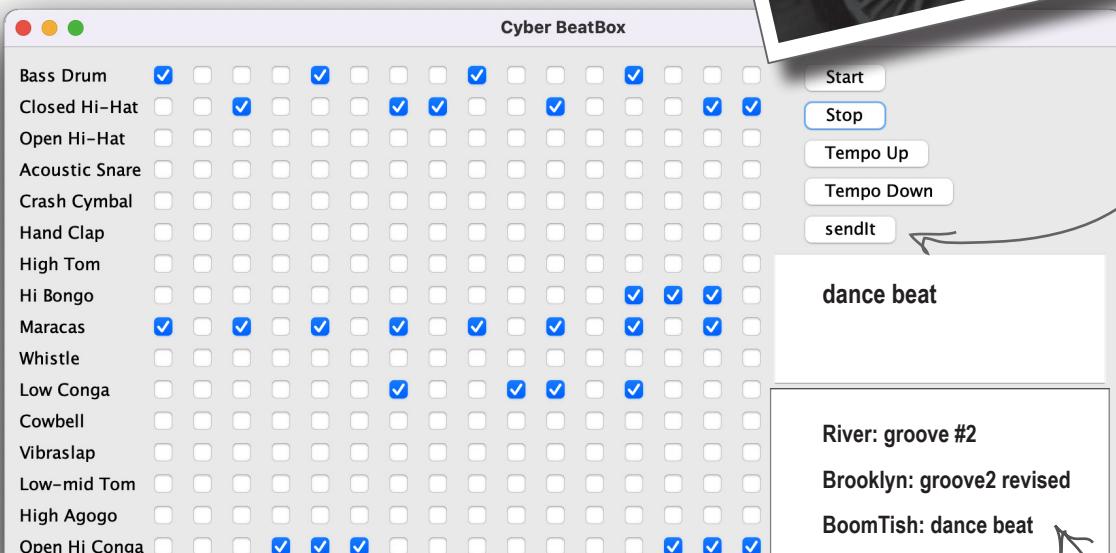
Stuff happens. The file isn't there. The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very* wrong. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? And 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.

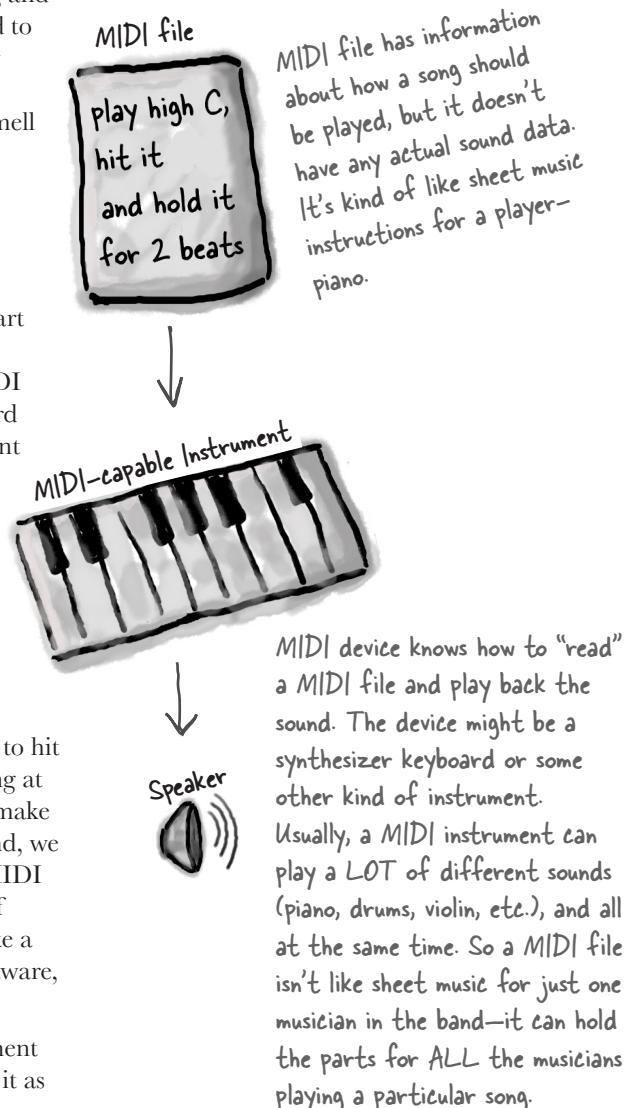
Oh yeah, and the JavaSound API. *That's* where we'll start in this chapter. For now, you can forget the GUI, forget the networking and the I/O, and focus only on getting some MIDI-generated sound to come out of your computer. And don't worry if you don't know a thing about MIDI or a thing about reading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



but it looked so simple

First we need a Sequencer

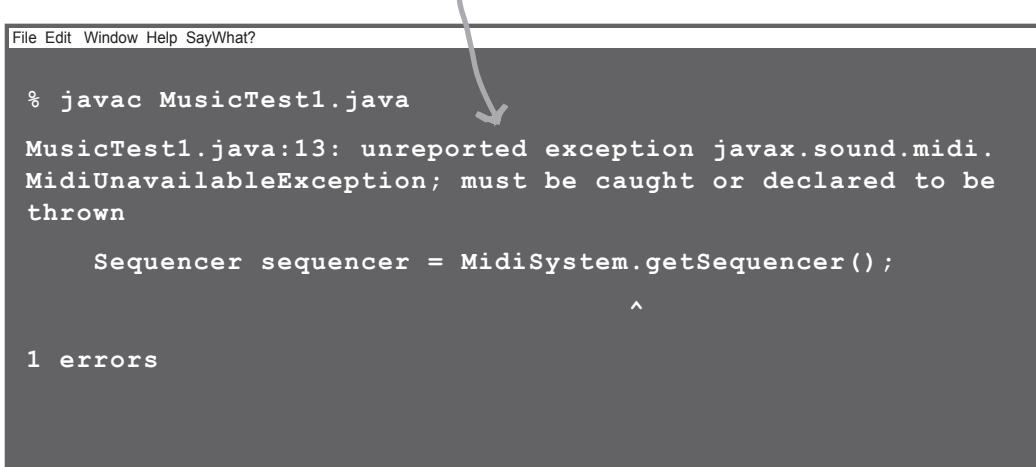
Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback 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.*;  
    ← import the javax.sound.midi package  
  
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();  
    }  
}
```

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!

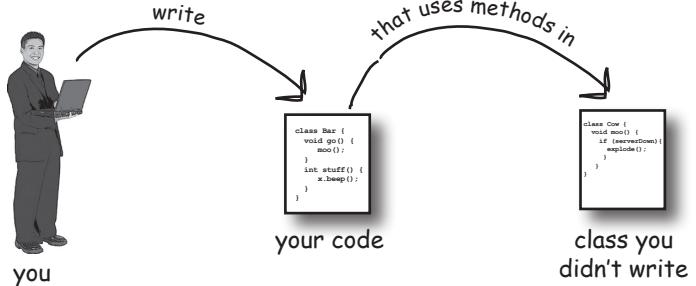
This code won't compile! The compiler says there's an "unreported exception" that must be caught or declared.



```
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.**

A diagram illustrating the second step. A code block labeled "class you didn't write" contains a method definition:

```

class Cow {
    void moo() {
        if (serverDown) {
            explode();
        }
    }
}

```

An arrow points from this code to a copy of the same code within a code block labeled "void moo() { ... }" under the heading "class you didn't write".

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

A diagram illustrating the third step. A person labeled "you" stands holding a laptop. Two thought bubbles appear above them:

- "I wonder if that method could blow up..."
- "My moo() method will explode if the server is down."

Below the person is a code block labeled "class you didn't write" containing the same risky code as before.

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

A diagram illustrating the fourth step. A person labeled "you" stands holding a laptop. An arrow labeled "write safely" points from the person to a code block labeled "your code".

The "your code" block contains a try-catch block that handles the potential exception from the external method:

```

class Bar {
    void goo() {
        try {
            moo();
        } catch (Exception e) {
            cry();
        }
    }
}

```

when things might go wrong

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()`

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).

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

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.

```
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");
        }
    }

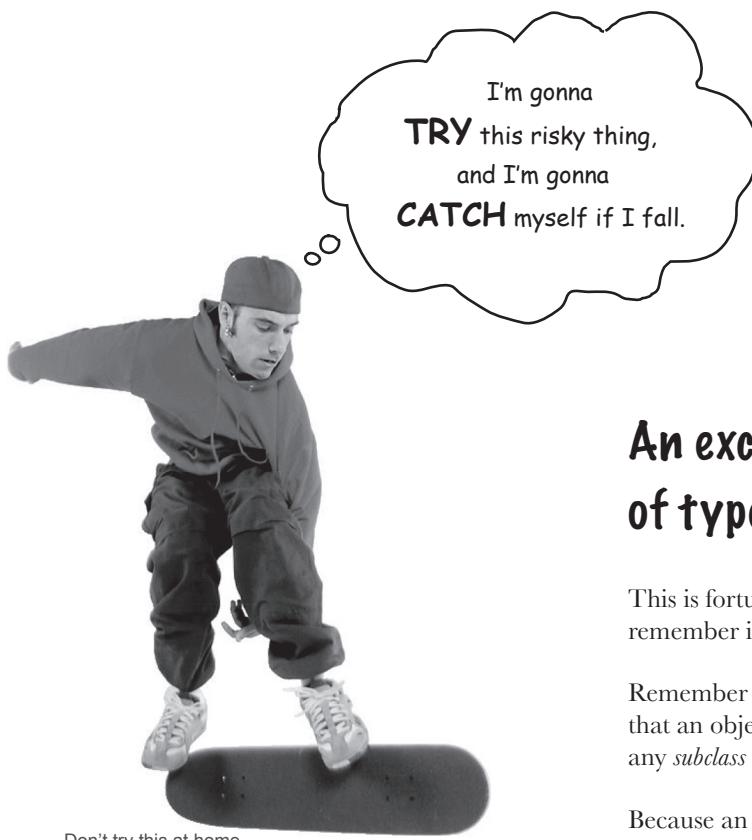
    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    }
}
```

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

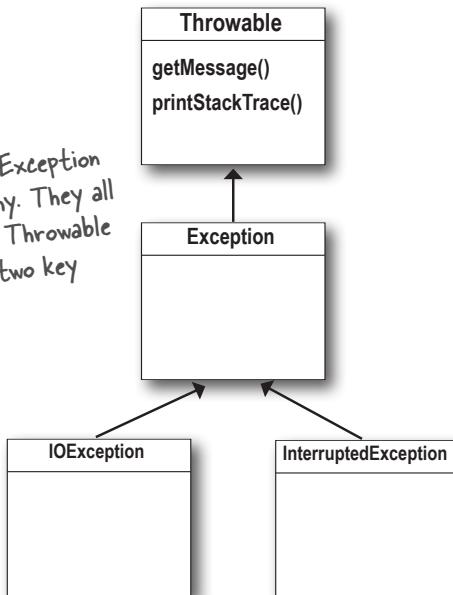
Dear Geeky,
*Life is short (especially on the
 heap). Take the risk. Try it. But
 just in case things don't work out, be
 sure to catch any problems before all
 hell breaks loose.*

} 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().



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



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 `e`.

```
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.

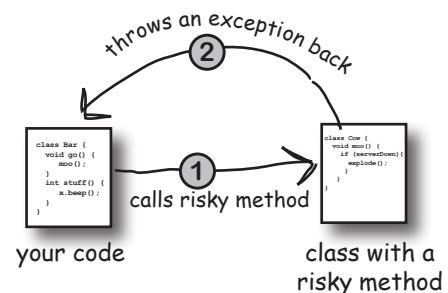
What you write in a catch block depends on the exception that was thrown. For example, if a server is down, you might use the catch block to try another server. If the file isn't there, you might ask the user for help finding it.

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:

```

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

```

Create a new Exception object and throw it.

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

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:

```

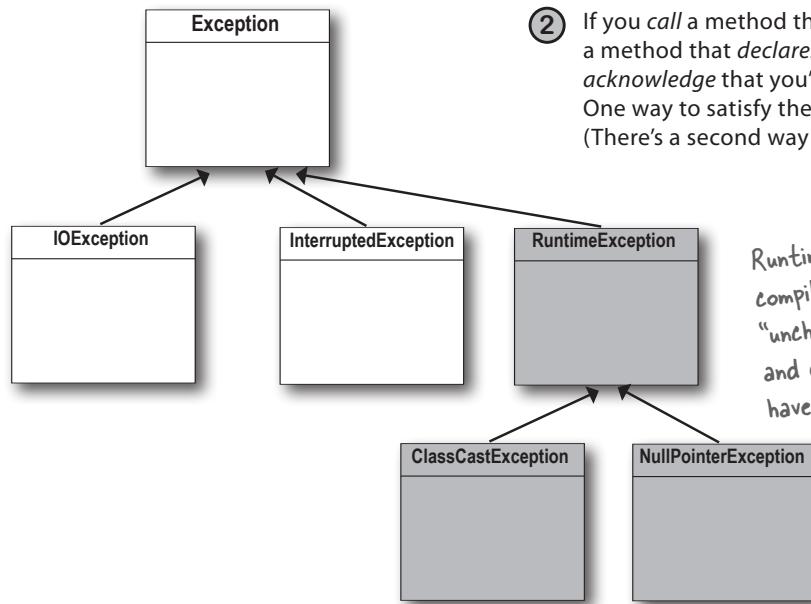
public void crossFingers() {
    try {
        anObject.takeRisk();
    } catch (BadException e) {
        System.out.println("Aaargh!");
        e.printStackTrace();
    }
}

```

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:

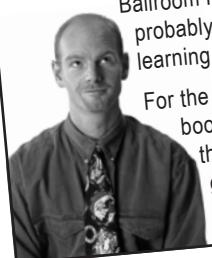
```
throw new NoCaffeineException();
```

- 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

- ✓ Connect to a remote server
- 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

What might go wrong

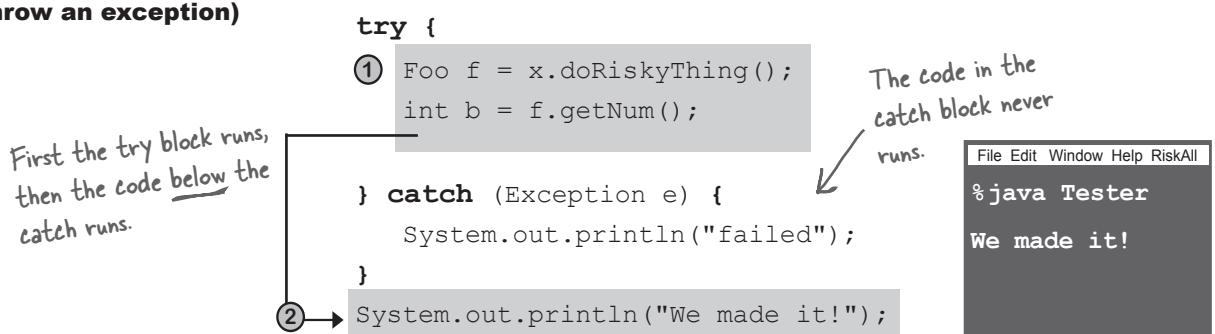
The server is down

Flow control in try/catch blocks

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

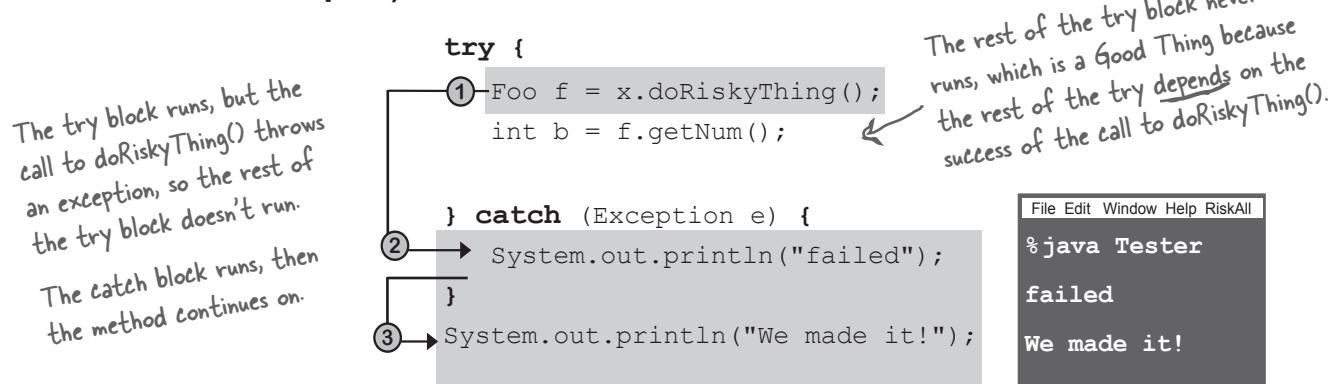
If the try succeeds

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



If the try fails

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



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

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

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

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

You have to turn off the oven no matter what!

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

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



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 = "yes"`: start try - start risky - scary exception - finally - end of main

When `test = "no"`: start try - start risky - end risky - 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.

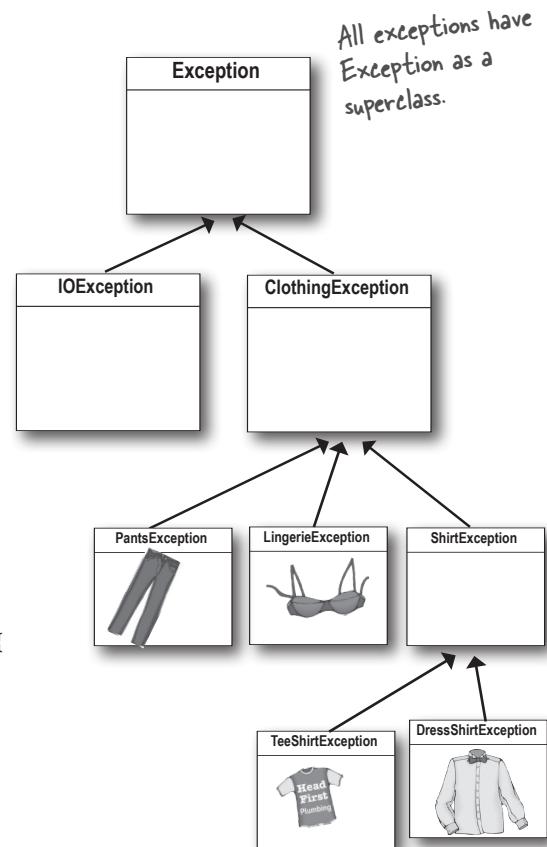
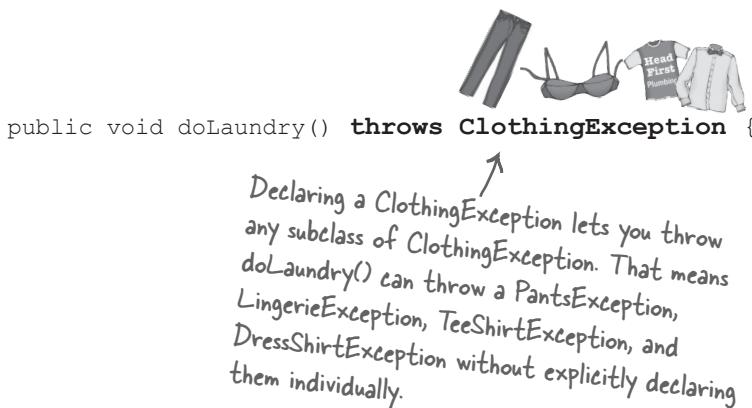


If doLaundry() throws a LingerieException, it lands in the LingerieException catch block.

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.



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



**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



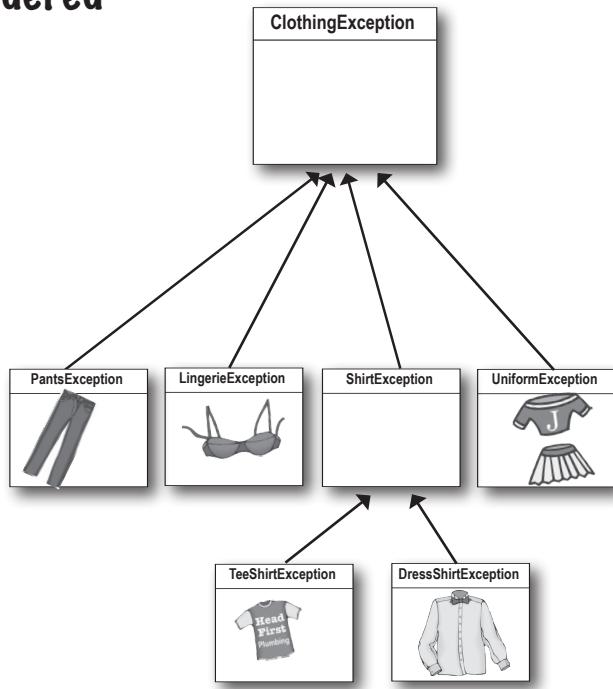
```
catch(TeeShirtException tex)
```



```
catch(ShirtException sex)
```



```
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.

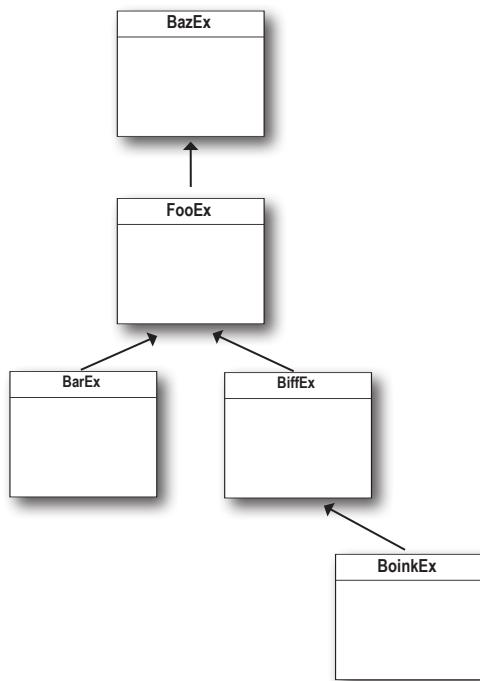
You could put `ShirtException` above `LingerieException`, and nobody would mind. Because even though `ShirtException` is a bigger (broader) type because it can catch other classes (its own subclasses), `ShirtException` can't catch a `LingerieException`, so there's no problem.

polymorphic puzzle



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

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



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

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

just duck it

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

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

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

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

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

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



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

Ducking (by declaring) only delays the inevitable

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

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

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

1 doLaundry() throws a ClothingException



main() calls foo()
foo() calls doLaundry()
doLaundry() is running and throws a ClothingException

2 foo() ducks the exception



doLaundry() pops off the stack immediately, and the exception is thrown back to foo().
But foo() doesn't have a try/catch, so...

3 main() ducks the exception



foo() pops off the stack, 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."



We're using the T-shirt to represent a Clothing Exception. We know, we know...you would have preferred the blue jeans.

4 The JVM shuts down

Handle or Declare. It's the law.

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

① HANDLE

Wrap the risky call in a try/catch

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

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

② DECLARE (duck it)

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

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

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

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

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

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

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

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

TROUBLE!!

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

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), the code would not compile, because polymorphically a MidiUnavailableException won't fit into a FileNotFoundException.

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

Exception Rules

- ① You cannot have a catch or finally without a try.**

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

NOT LEGAL!
Where's the try?

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

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

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

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

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

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

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

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

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

Code Kitchen



Everything
you see I made
myself from scratch.

Was that more
satisfying than
using Ready-Bake
Code?

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.

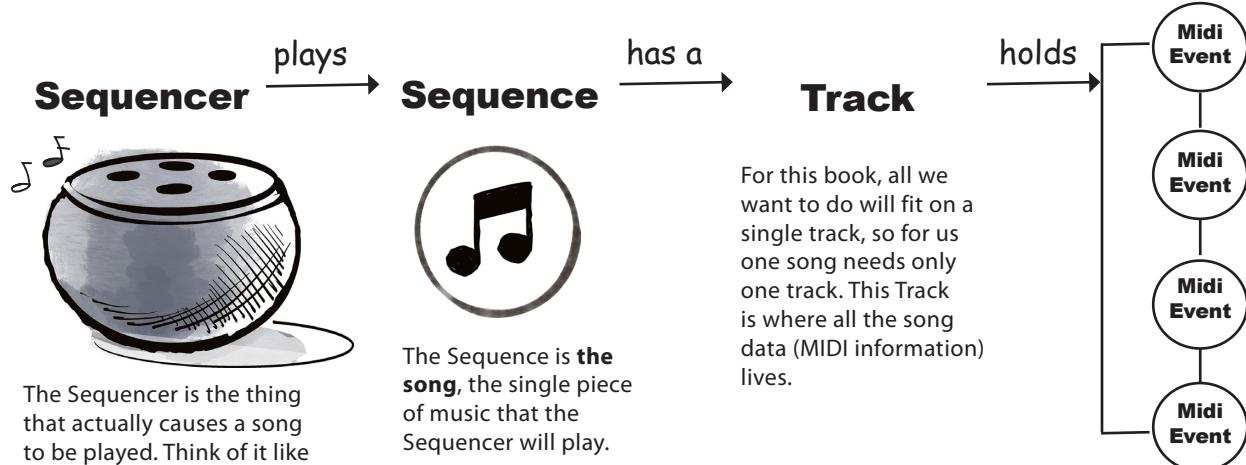
But if you want to learn
more about JavaSound,
turn the page.

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.



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

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.

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.”

And you need **FIVE** steps:

- ① Get a **Sequencer** and open it

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

- ② Make a new **Sequence**

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

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

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

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

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



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

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 *someone*, but *something*.)

```

import javax.sound.midi.*; ← Don't forget to import the midi package.
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 {
            ① Sequencer player = MidiSystem.getSequencer(); ← We're using a static import here so we can
            player.open(); ← use the constants in the ShortMessage class.

            ② Sequence seq = new Sequence(Sequence.PPQ, 4); ← Get a Sequencer and open it
                (so we can use it...a Sequencer ← (so we can use it...a Sequencer
                doesn't come already open).
                (think of 'em as Ready-bake arguments).

            ③ Track track = seq.createTrack(); ← Don't worry about the arguments to the
                Track lives in the Sequence, and the MIDI data
                lives in the Track.

            ④ 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); ← Give the Sequence to the Sequencer (like
                selecting the song to play).
            player.start(); ← start() the Sequencer (play the song).

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

```



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)
to instrument 102



change the instrument and note

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

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

```
import javax.sound.midi.*;
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();
        }
    }
}
```

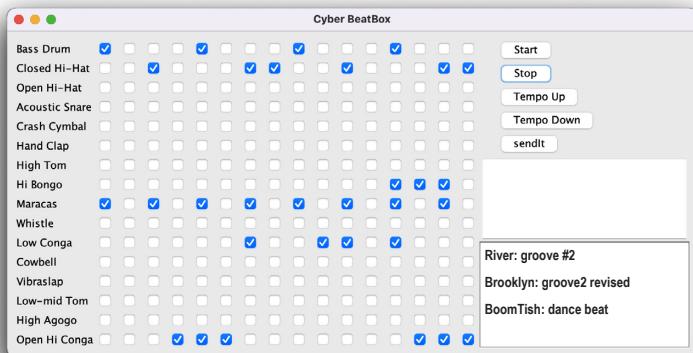
Run it with two int args from 0 to 127. Try these for starters:

```
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

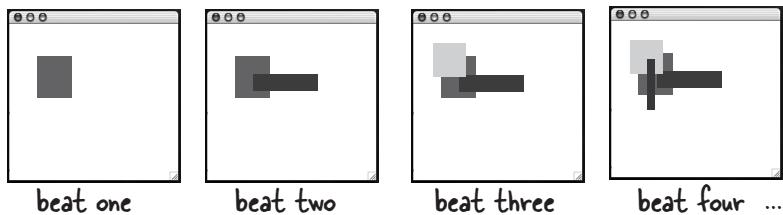
Chapter 17: the goal

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.



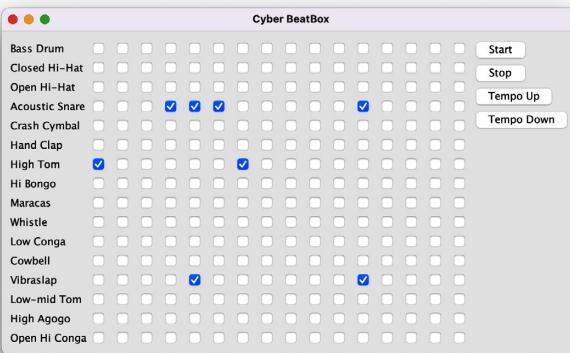
Chapter 14: MIDI events

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).



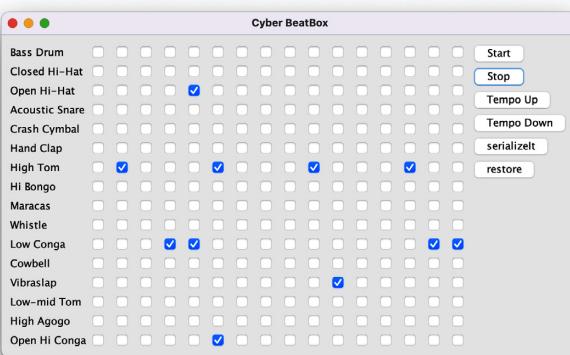
Chapter 15: Standalone BeatBox

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.



exercise: True or False



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

TRUE OR FALSE

1. A try block must be followed by a catch and a finally block.
2. If you write a method that might cause a compiler-checked exception, you must wrap that risky code in a try/catch block.
3. Catch blocks can be polymorphic.
4. Only “compiler checked” exceptions can be caught.
5. If you define a try/catch block, a matching finally block is optional.
6. If you define a try block, you can pair it with a matching catch or finally block, or both.
7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try/catch block.
8. The main() method in your program must handle all unhandled exceptions thrown to it.
9. A single try block can have many different catch blocks.
10. A method can 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!

```

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];
}

```

File Edit Window Help ThrowUp

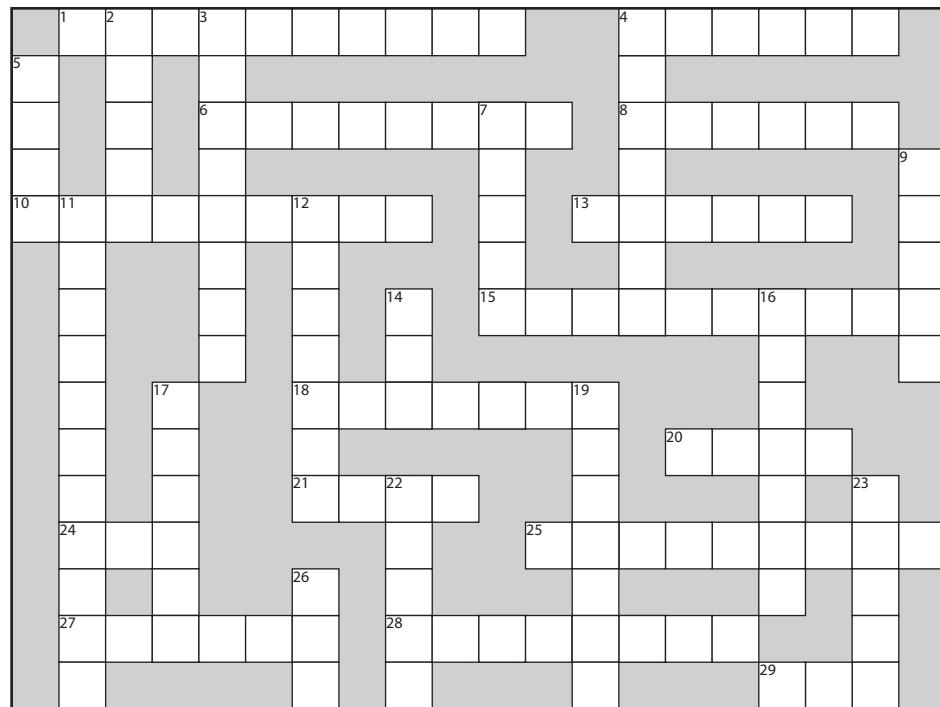
```
% java ExTestDrive yes
thaws
```

```
% java ExTestDrive no
throws
```

→ Answers on page 458.



→ 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

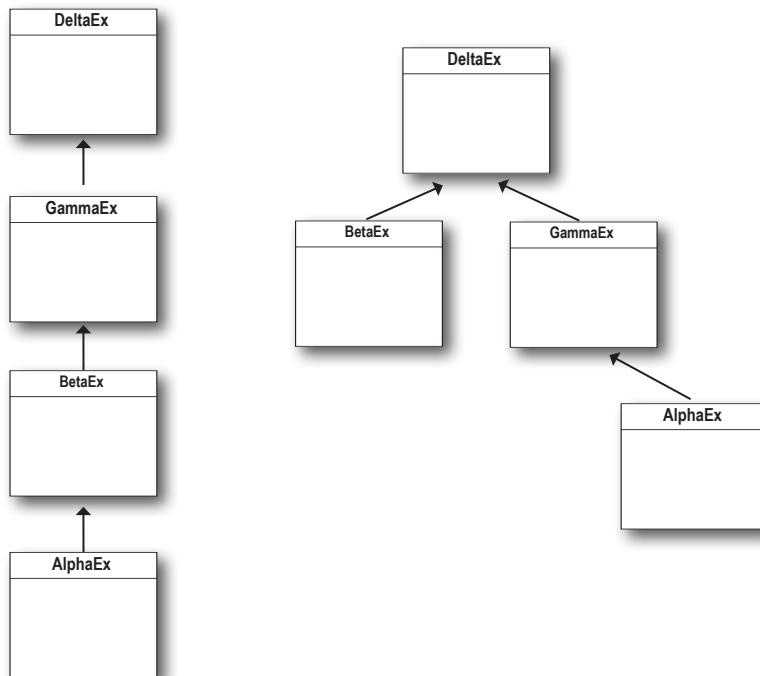
More Hints:

- | | | | | | |
|--------|---------------------------|-------------------------------|----------------------------|----------------------|------------------|
| Across | 6. A Java child | 20. Also a type of collection | 21. Duck | 27. Starts a problem | 28. Not Abstract |
| | 8. Start a method | 2. Or a mouthwash | 3. For _____ (not example) | 17. Not a getter | 5. Numbers ... |
| | 9. Only public or default | 16. _____ the family fortune | | | |

Solution



(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");
}
```

A screenshot of a terminal window titled "Chill". The window shows the following command-line interaction:
% java ExTestDrive yes
throws

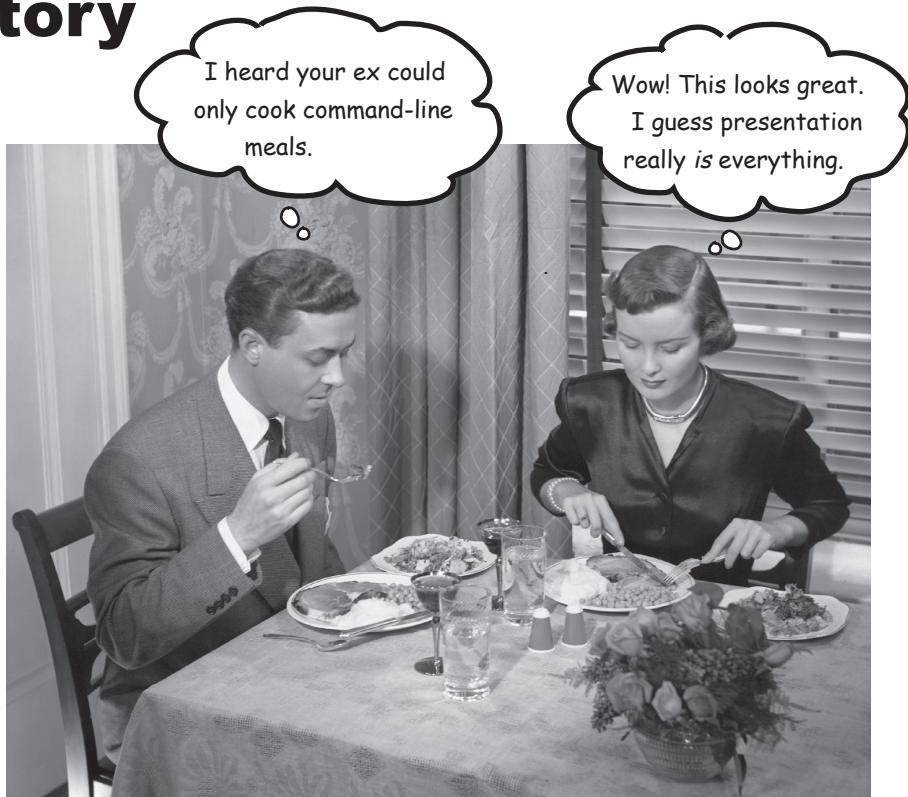
% java ExTestDrive no
throws



JavaCross (from page 456)

	1	2	ASSIGNMENT		4	POPPED	
5	M	C	N		R		
A	O	S	SUBCLASS	7	I	INVOKE	9
T	P	T		8	V		C
H	I	E	11 HIERARCHY	12	A	13 HANDLE	L
N	N	N		14	T	T	A
S	C	E		15	EXCEPTIONS		S
T	E	C	R			N	
A	S	K	KEYWORD	18		H	
N	E	E		19			
T	T	D	DUCK	21	E	20 TREE	23
I	N	A		22	C	R	T
A	E	T		25	ALGORITHM	M	
27	THROWS	I		26	A	T	R
E		A		28	CONCRETE		O
		H		29	E	NEW	

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)

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.

Two issues!

1. Swing? This looks like Swing code.
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."



Making a GUI is easy:

- ① Make a frame (a JFrame)

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

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

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

- ③ Add the widget to the frame

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

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

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

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

Your first GUI: a button on a frame

```

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

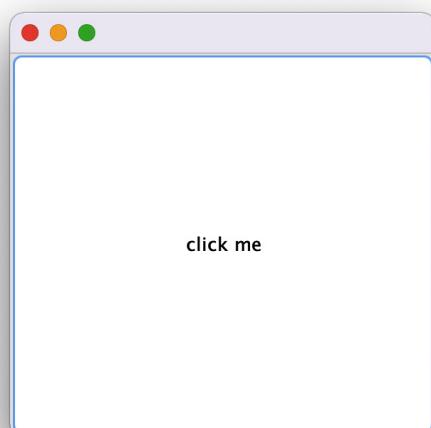
public class SimpleGuil {
    public static void main(String[] args) {
        JFrame frame = new JFrame(); ← make a frame and a button
        JButton button = new JButton("click me"); ← (you can pass the button constructor
                                                    the text you want on the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

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ← this line makes the program quit as soon as you
                                                    close the window (if you leave this out it will
                                                    just sit there on the screen forever)
        frame.getContentPane().add(button); ← add the button to the frame's
                                             content pane
        frame.setSize(300, 300); ← give the frame a size, in pixels
        frame.setVisible(true); ← finally, make it visible!! (if you forget
                               this step, you won't see anything when
                               you run this code)
    }
}

```

Let's see what happens when we run it:

%java SimpleGuil



Whoa! That's a
Really Big Button.

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

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!



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):

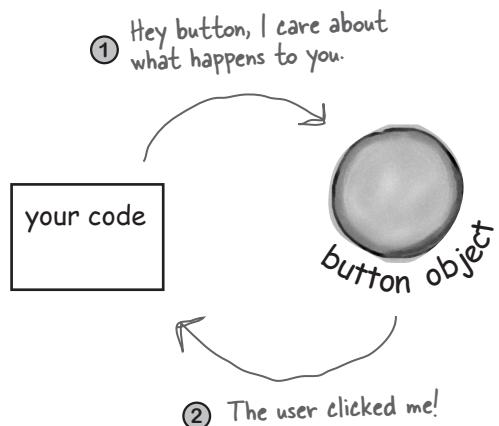
```
public void changeIt() {
    button.setText("I've been clicked!");
}
```

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).

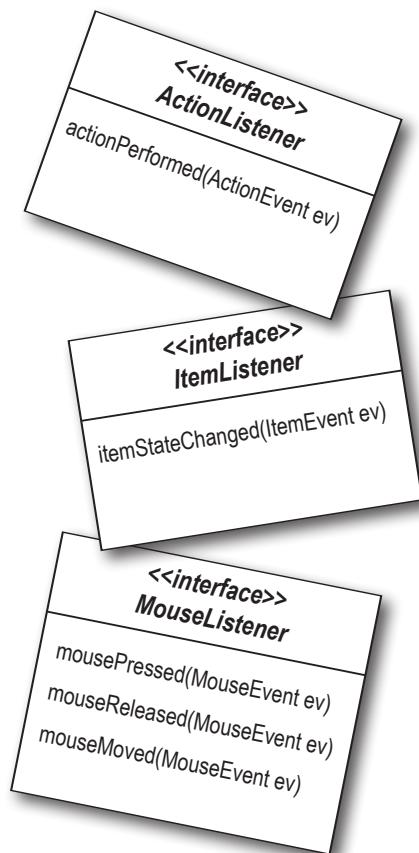
The Swing GUI components are event sources. In Java terms, an event source is an object that can turn user actions (click a mouse, type a key, close a window) into events. And like virtually everything else in Java, an event is represented as an object. An object of some event class. If you scan through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

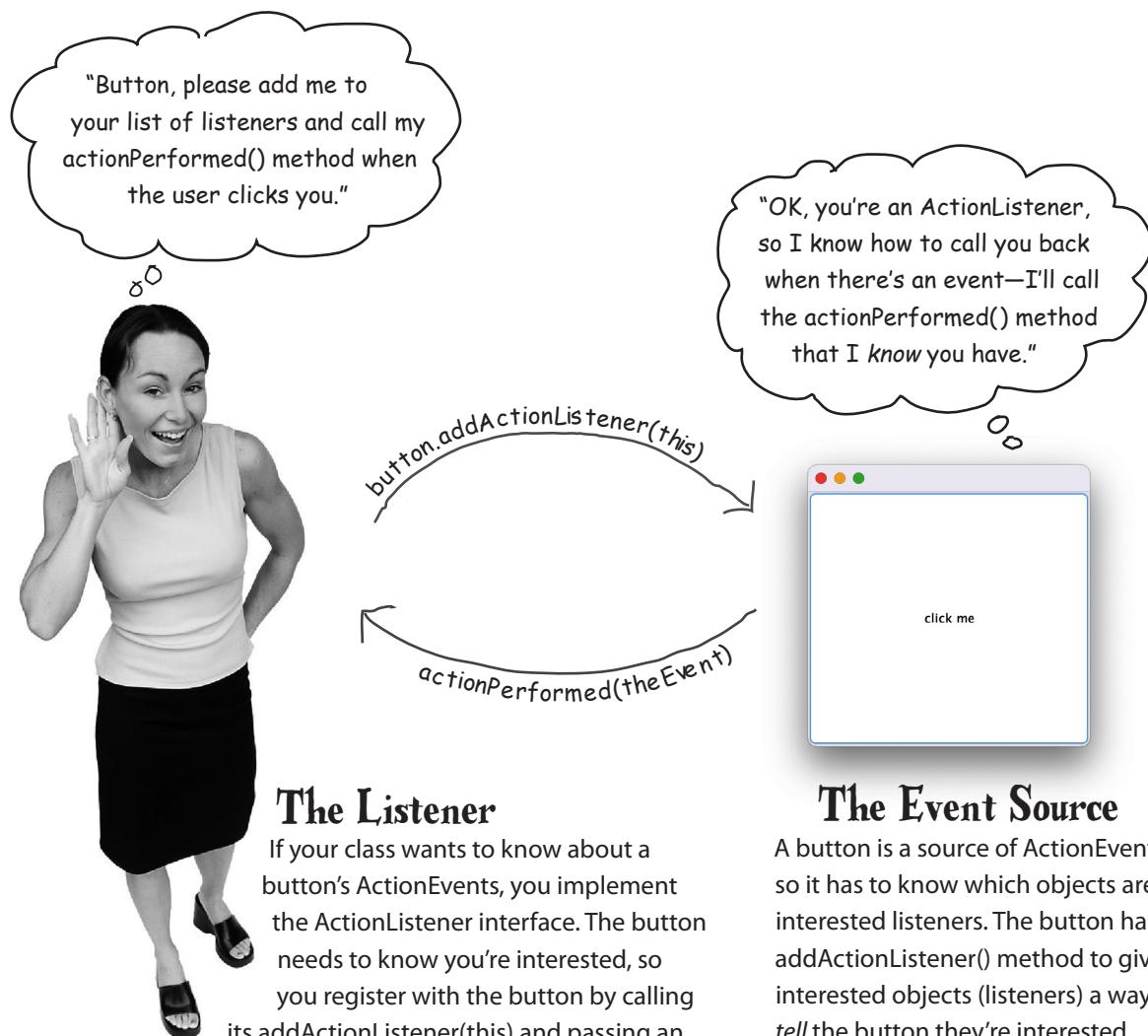
Every event type has a matching listener interface. If you want `MouseEvents`, implement the `MouseListener` interface. Want `WindowEvents`? Implement `WindowListener`. You get the idea. And remember your interface rules—to implement an interface you *declare* that you implement it (class `Dog` implements `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.*; ← A new import statement for the package
                           that ActionListener and ActionEvent are in.

① public class SimpleGui2 implements ActionListener { ← Implement the interface. This says,
                           "an instance of SimpleGui2 IS-A
                           ActionListener."
                           (The button will give events only to
                           ActionListener implementers.)
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); ← 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!!
}

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

```

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.

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

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

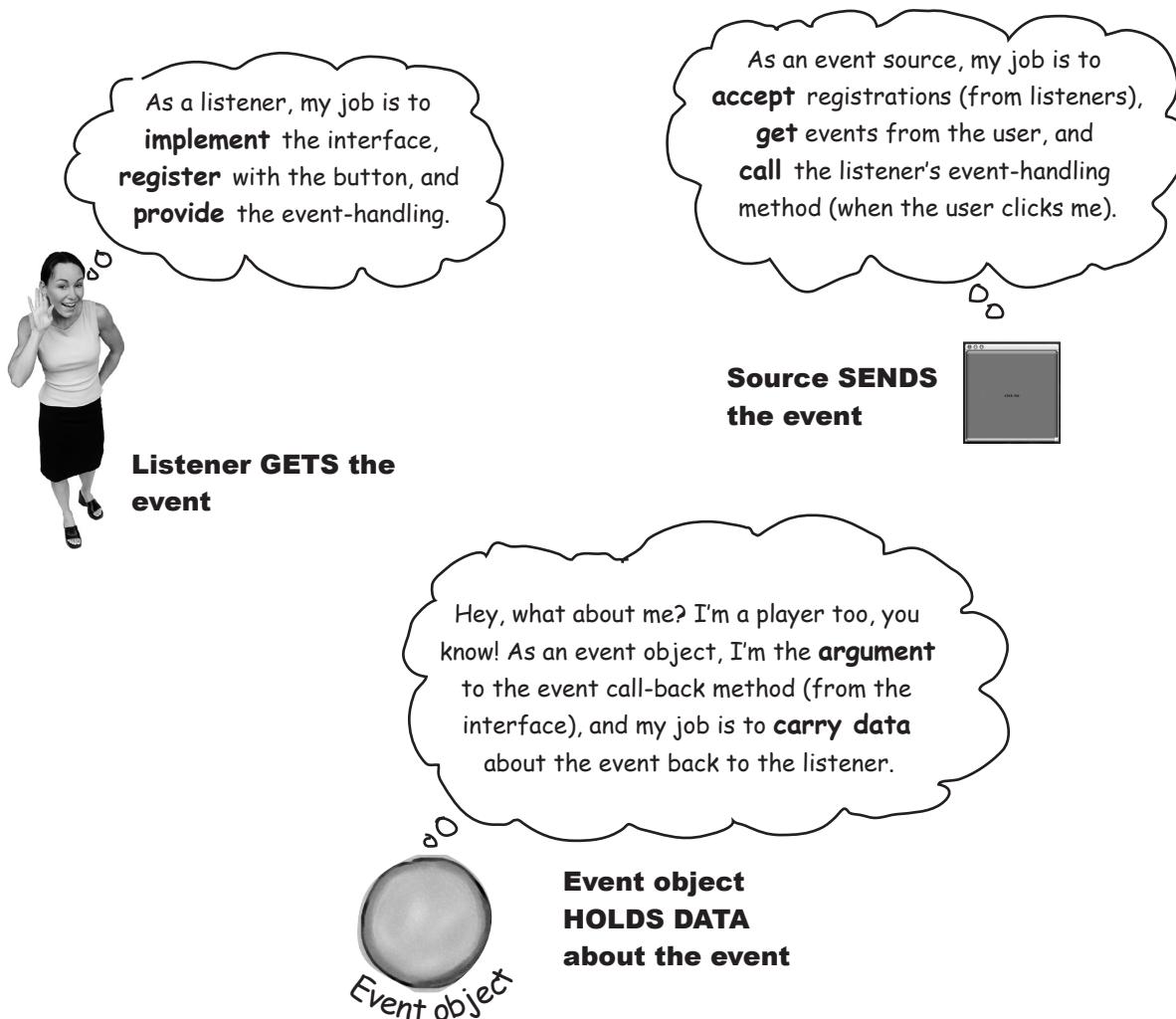
Listeners, Sources, and Events

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

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

Event methods

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

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

Look in the API.

OK. Look for what?

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

`addKeyListener(KeyListener k)`

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

Getting back to graphics...

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

Three ways to put things on your GUI:

① Put widgets on a frame

Add buttons, menus, radio buttons, etc.

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

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

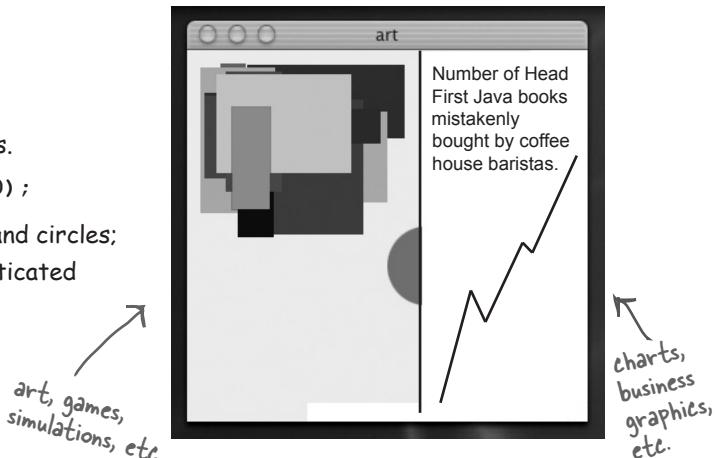


② Draw 2D graphics on a widget

Use a graphics object to paint shapes.

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

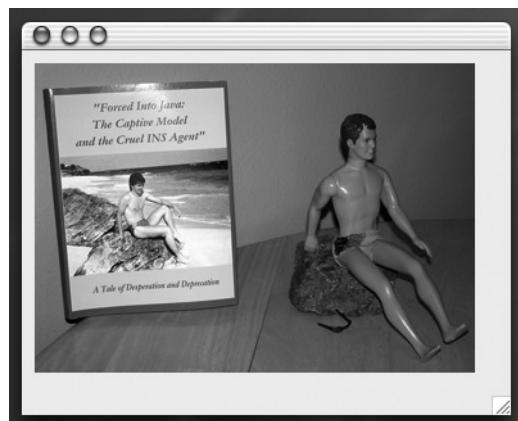
You can paint a lot more than boxes and circles; the Java2D API is full of fun, sophisticated graphics methods.



③ Put a JPEG on a widget

You can put your own images on a widget.

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



Make your own drawing widget

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

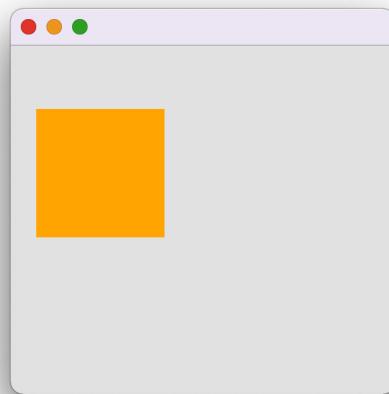
It's a piece of cake.

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

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

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

A Swing frame with a custom drawing panel



```

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);
}
```

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);
}
```

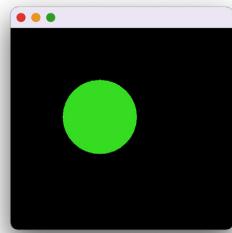
Fill the entire panel with black (the default color).

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.

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

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



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:

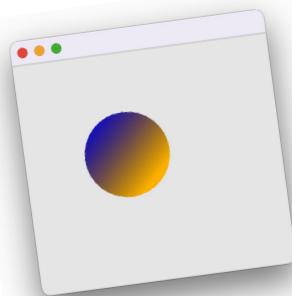
```
Graphics2D g2d = (Graphics2D) g;
```

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.

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;`

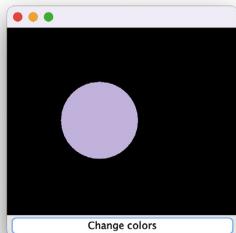
We can get an event.

We can paint graphics.

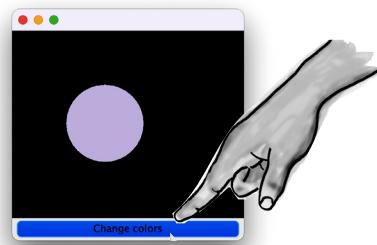
But can we paint graphics when we get an event?

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

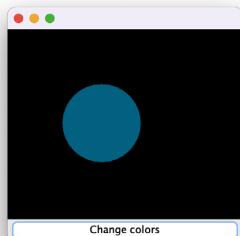


- 1 The frame is built with the two widgets (your drawing panel and a button). A listener is created and registered with the button. Then the frame is displayed, and it just waits for the user to click.

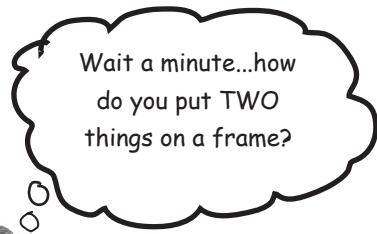


- 2 The user clicks the button, and the button creates an event object and calls the listener's event handler.

- 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:

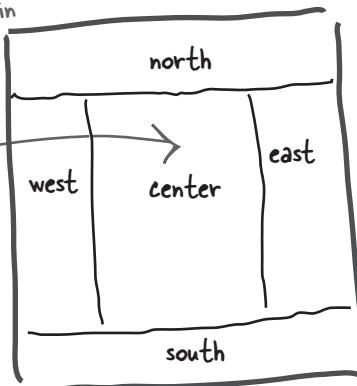
```
frame.getContentPane().add(button);
```

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

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

default region

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



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

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



Sharpen your pencil —

Given the pictures on page 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.

```

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();
}
}

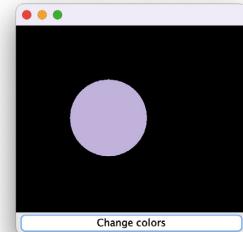
```

```

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

Button is in the SOUTH region of the frame.

Add the listener (this) to the button.

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

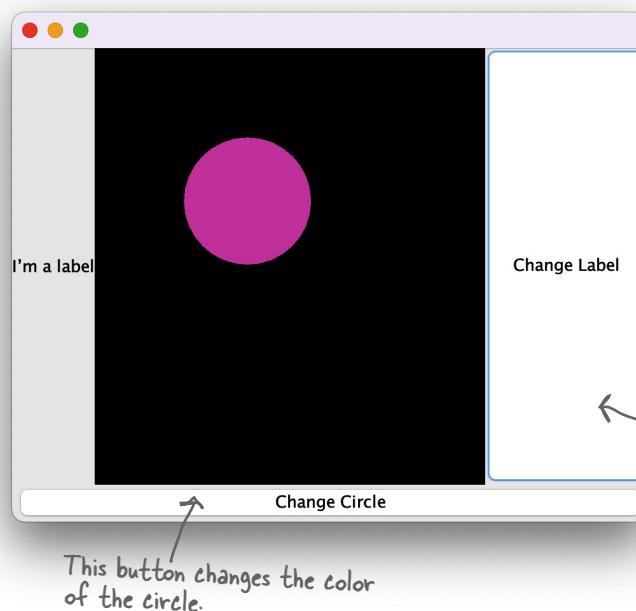
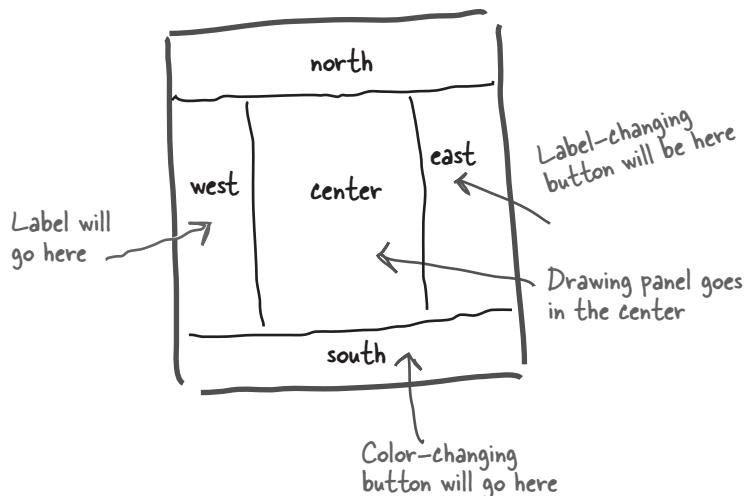
When the user clicks, tell the frame to repaint() itself. That means paintComponent() is called on every widget in the frame!

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

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint(); ← Query the event object
        } else {           to find out which button
            label.setText("That hurt!");   actually fired it, and use
        }                   that to decide what to do.
    }
}
```

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

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

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

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

Inner class using an outer class variable

```
class MyOuterClass {
    private int x;

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

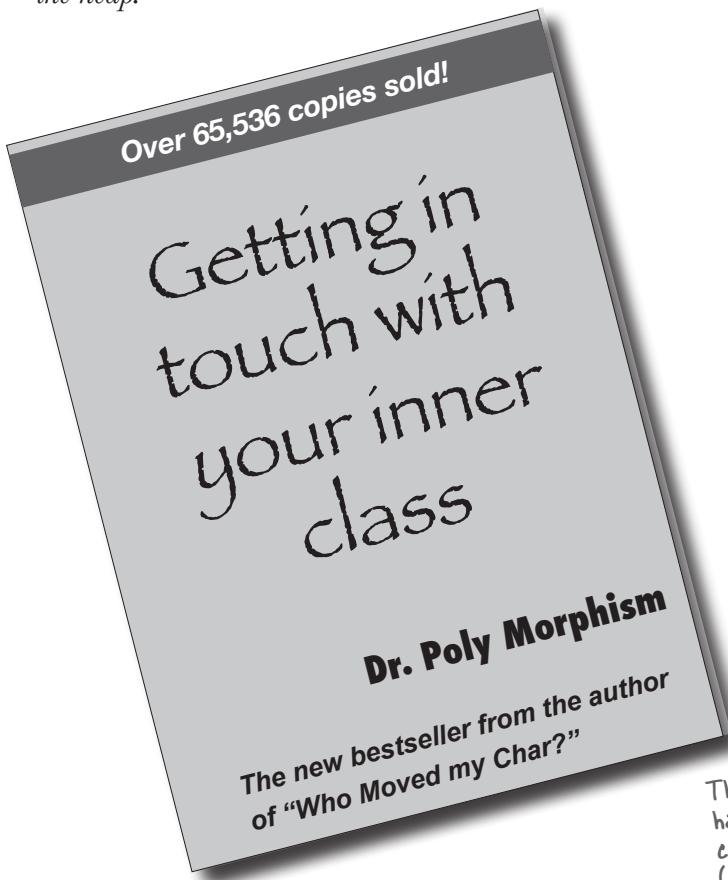
of the inner class!

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

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

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

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



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

- ① Make an instance of the outer class

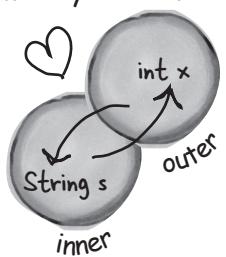


- ② Make an instance of the inner class, by using the instance of the outer class.



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

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



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

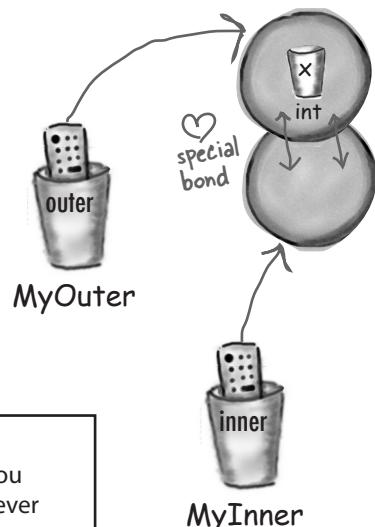
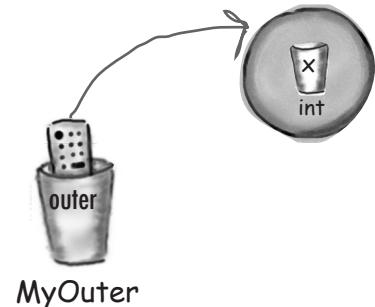
How to make an instance of an inner class

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

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

```
class MyOuter {
    private int x;           ← The outer class has a private
    MyInner inner = new MyInner(); ← instance variable "x."
                                Make an instance of the
    public void doStuff() {
        inner.go();          ← Call a method on the
    }                         inner class.
}
```

```
class MyInner {
    void go() {
        x = 42;             ← The method in the inner class uses the
    } // close inner class   outer class instance variable "x," as if "x"
} // close outer class    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...

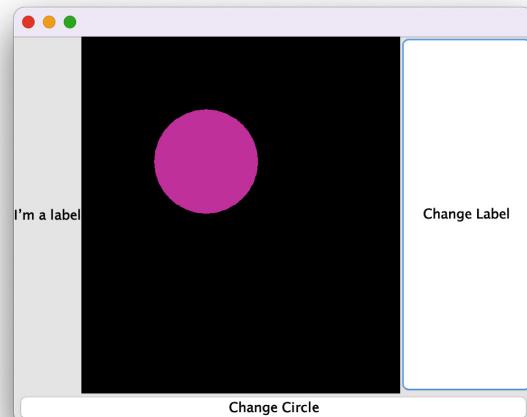
```
class Foo {
    public static void main (String[] args) {
        MyOuter outerObj = new MyOuter();
        MyOuter.MyInner innerObj = outerObj.new MyInner();
    }
}
```

Now we can get the two-button code working

```
public class TwoButtons { ← Much better: the main GUI
    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()); ← Instead of passing (this) to the
                                                    button's listener registration
                                                    method, pass a new instance of
                                                    the appropriate listener class.

    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);
}
```

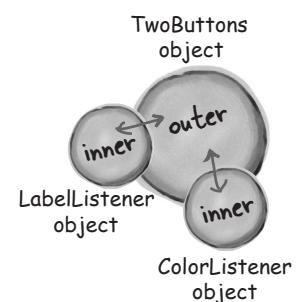
```
class LabelListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("Ouch!");
    }
}
```

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

Inner class knows
about "label."

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

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.

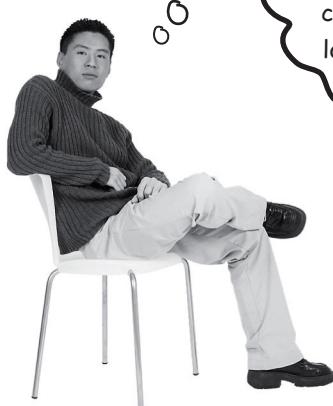
Inner objects: You got it!

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...

He thinks he's got it made, having two inner class objects. But we have access to all his private data, so just imagine the damage we could do...





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? Hmm...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);
}

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

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

```

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

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

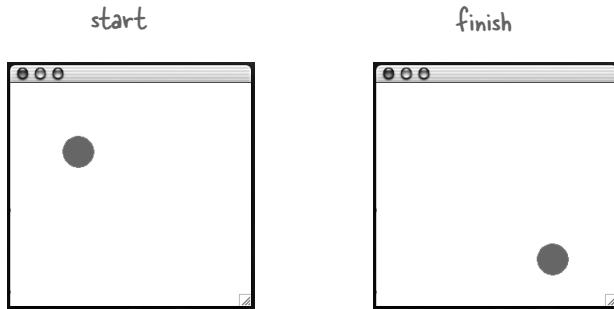
Lambdas, clearer and more concise

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);
    }
}
```

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

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



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

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

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

        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(drawPanel);
        frame.setSize(300, 300);
        frame.setVisible(true); } ← Nothing new here. Make the widgets
                           and put them in the frame.

This is where the action is! ← Repeat this 130 times.

for (int i = 0; i < 130; i++) {
    xPos++;
    yPos++; ← increment the x and y
               coordinates
    drawPanel.repaint(); ← Tell the panel to repaint itself (so we
                         can see the circle in the new location).

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

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

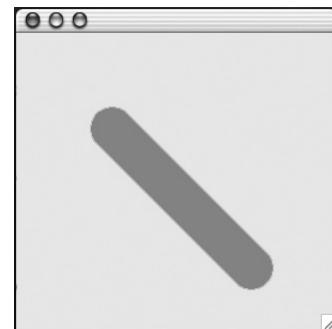
What did we do wrong?

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

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

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);
}
```



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

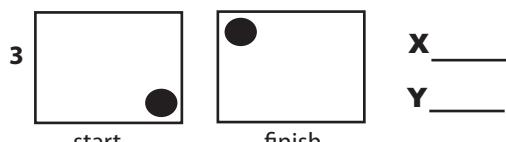
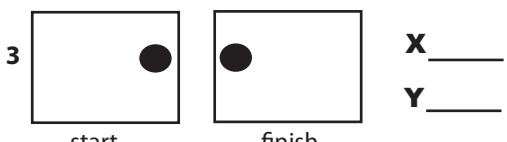
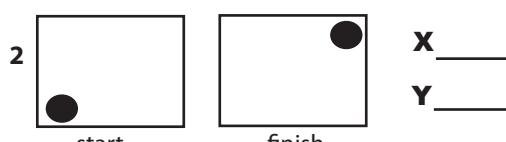
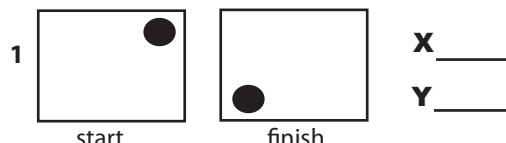


Sharpen your pencil (optional, just for fun)

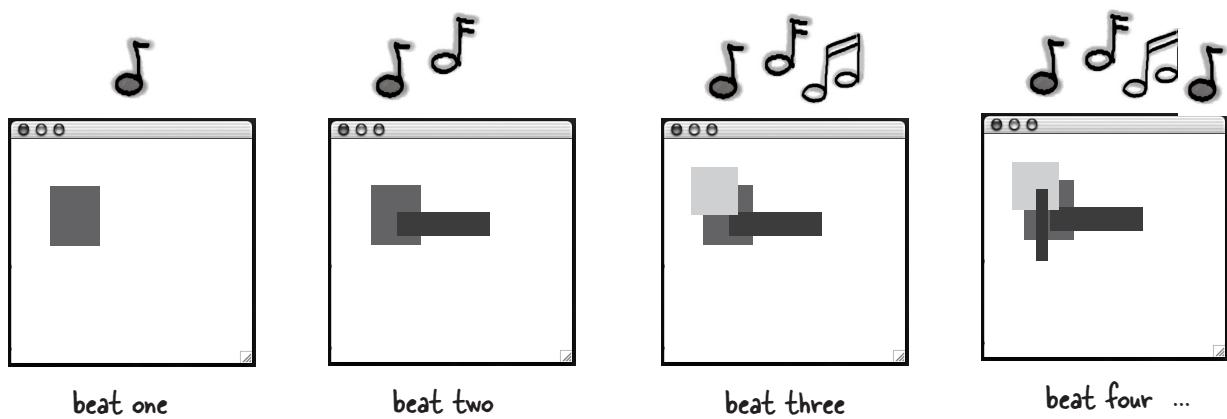
→ Yours to solve.

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

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; // Return the event (a MidiEvent all loaded up with the message).
}
```

The annotations include:

- A bracket above the parameters "command", "channel", "one", "two", and "tick" with the label "The four arguments for the message".
- A bracket below the "return event;" line with the label "Return the event (a MidiEvent all loaded up with the message)."
- A bracket on the right side of the code with the label "The event 'tick' for WHEN this message should happen".
- A bracket on the right side of the code with the label "Whoo! A method with five parameters."
- A bracket on the right side of the code with the label "Make the message and the event, using the method parameters."

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.*;   ← Don't forget the imports.
import static javax.sound.midi.ShortMessage.*;

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

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

            for (int i = 5; i < 61; i += 4) {   ← Make a bunch of events to make the notes keep
                track.add(makeEvent(NOTE_ON, 1, i, 100, i));   going up (from piano note 5 to piano note b1).
                track.add(makeEvent(NOTE_OFF, 1, i, 100, i + 2));   ← Call our new makeEvent()
            }                                         method to make the message
            sequencer.setSequence(seq);   ← and event; then add the result
            sequencer.setTempoInBPM(220);   } Start it running
            sequencer.start();   ← (the MidiEvent returned from
            } catch (Exception ex) {   makeEvent()) to the track.
                ex.printStackTrace();   These are NOTE ON and NOTE
            } OFF pairs.

        }

        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;
        }
    }
}

```

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:

```

class MyDrawPanel extends JPanel implements ControllerEventListener {
    private boolean msg = false; ← We set a flag to false, and we'll set it
                                    to true only when we get an event.

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

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

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

```



→ Yours to solve.

```

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));
            }

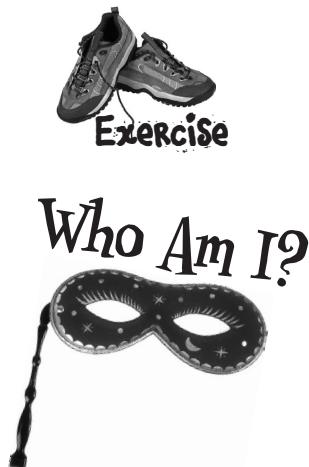
            sequencer.setSequence(seq);
            sequencer.start();
            sequencer.setTempoInBPM(120);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

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.

```
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;  
        }  
    }  
}
```

exercise: 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.



```

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");
        }
    }
}
  
```

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?



→ Answers on page 507.

puzzle: Pool Puzzle



Pool Puzzle



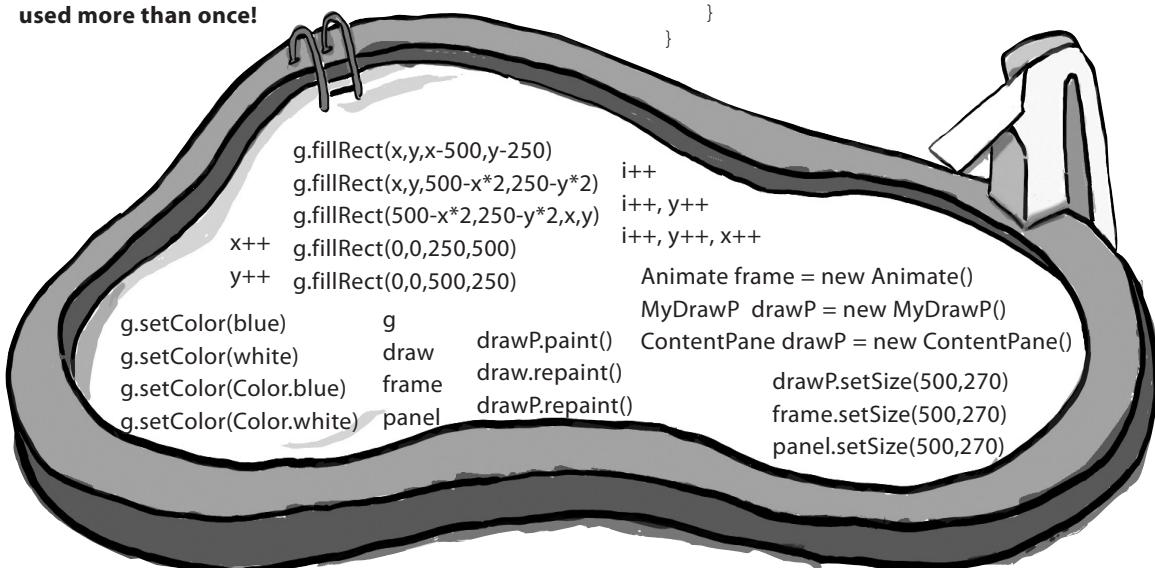
Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

Output

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



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





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)

```
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);
```

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

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

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



Poo! Puzzle (from page 506)

The Amazing, Shrinking, Blue Rectangle.



```
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);
        }
    }
}
```

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`.

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.

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

Four steps to making a GUI (review)

- ① Make a window (a JFrame)

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

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

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

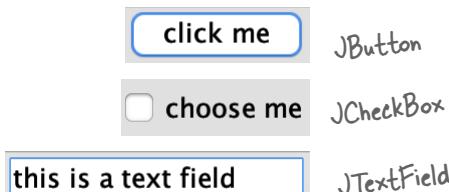
- ③ Add the component to the frame

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

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

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

Put interactive components:



Into background components:



Layout Managers

A layout manager is a Java object associated with a particular component, almost always a *background* component. The layout manager controls the components contained *within* the component the layout manager is associated with. In other words, if a frame holds a panel, and the panel holds a button, the panel's layout manager controls the size and placement of the button, while the frame's layout manager controls the size and placement of the panel. The button, on the other hand, doesn't need a layout manager because the button isn't holding other components.

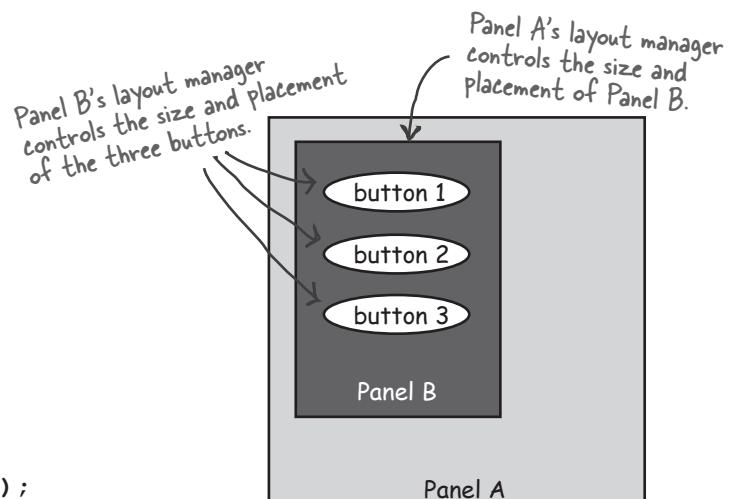
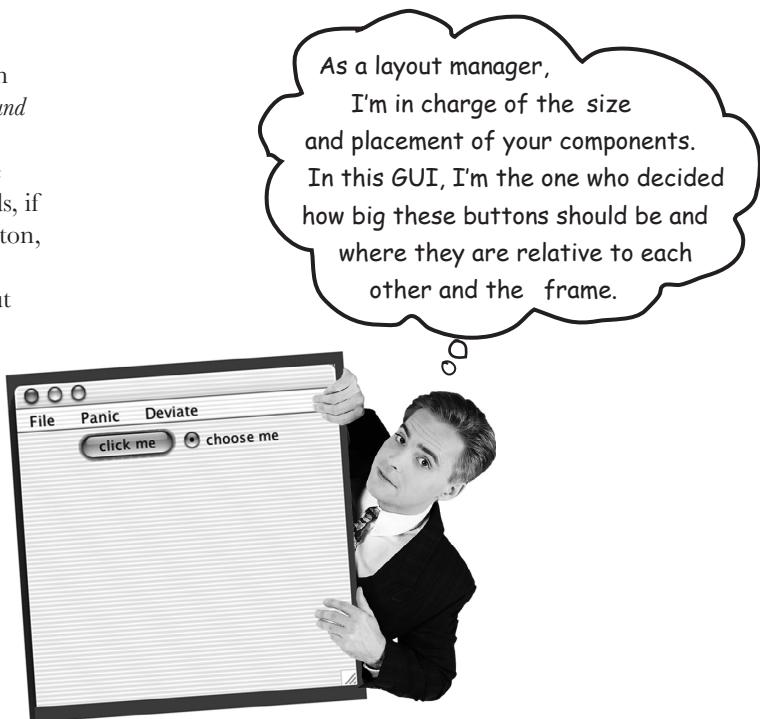
If a panel holds five things, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things (e.g., if any of those five things are panels or other containers that hold other things), then those *other* things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

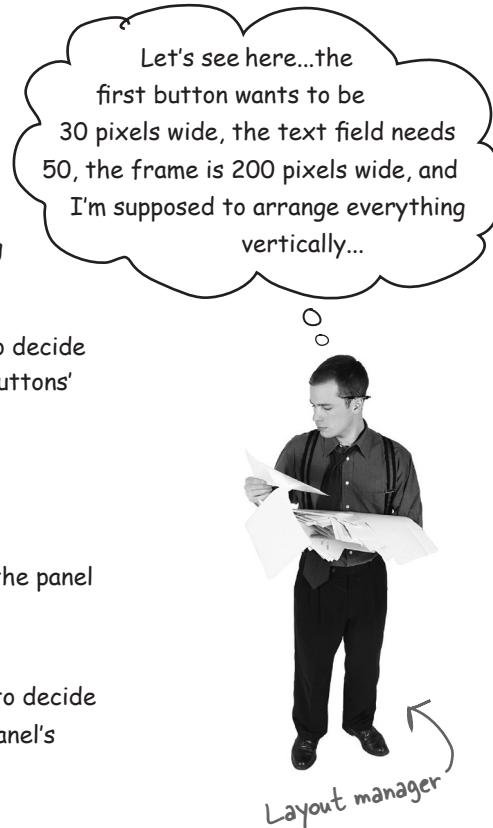


How does the layout manager decide?

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.



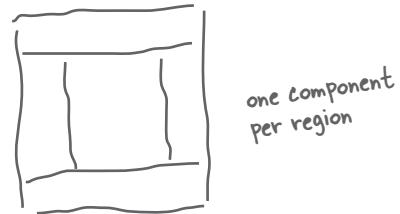
Different layout managers have different policies

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

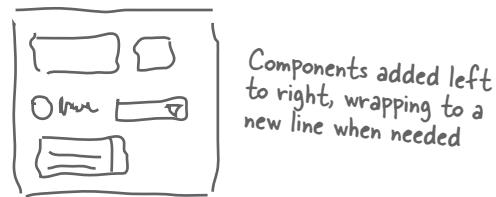
BorderLayout

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!**



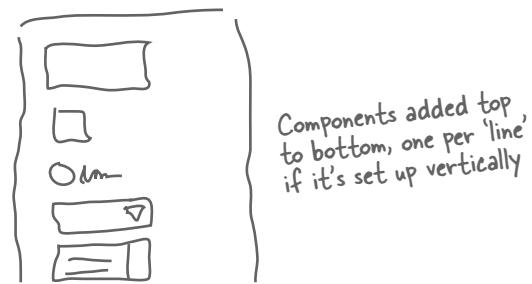
FlowLayout

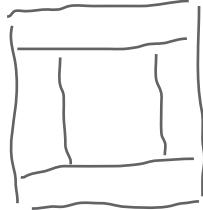
A FlowLayout manager acts kind of like a word processor, except with components instead of words. Each component is the size it wants to be, and they're laid out left to right in the order that they're added, with "word wrap" turned on. So when a component won't fit horizontally, it drops to the next "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.*;    ← BorderLayout is in the java.awt package.  
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);  
    }  
}
```

Specify the region.



Brain Barbell

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

What are the factors the layout manager has to consider?

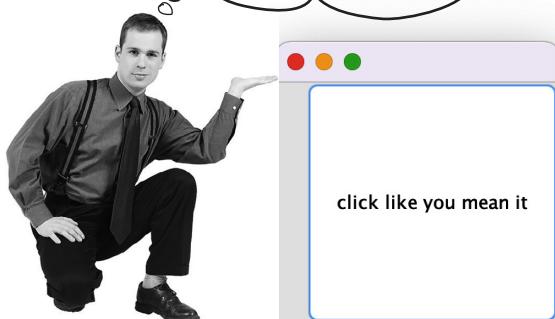
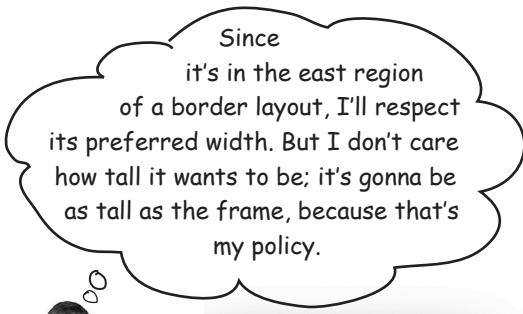
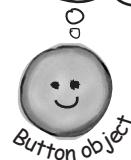
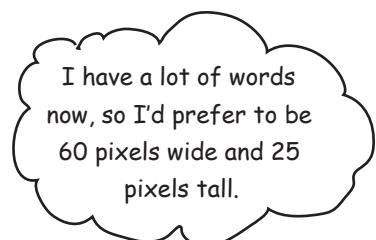
Why isn't it wider or taller?



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

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

We changed only the text on the button.

The button gets its preferred width, but not height.



border layout

Let's try a button in the NORTH region

```
public void go() {  
    JFrame frame = new JFrame();  
    JButton button = new JButton("There is no spoon...");  
    frame.getContentPane().add(BorderLayout.NORTH, button);  
    frame.setSize(200, 200);  
    frame.setVisible(true);  
}
```



The button is as tall as it wants to be, but as wide as the frame.

Now let's make the button ask to be taller

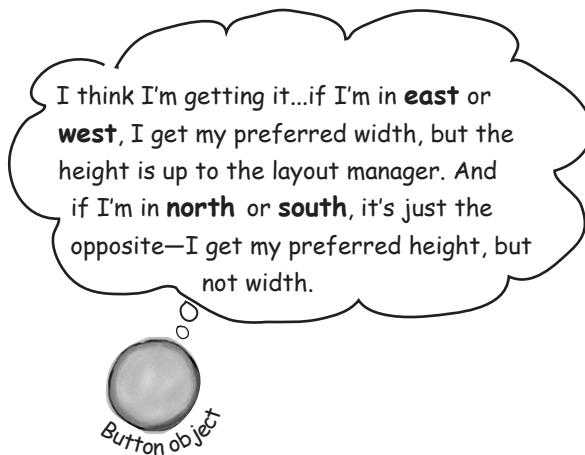
How do we do that? The button is already as wide as it can ever be—as wide as the frame. But we can try to make it taller by giving it a bigger font.

```
public void go() {  
    JFrame frame = new JFrame();  
    JButton button = new JButton("Click This!");  
    Font bigFont = new Font("serif", Font.BOLD, 28);  
    button.setFont(bigFont);  
    frame.getContentPane().add(BorderLayout.NORTH, button);  
    frame.setSize(200, 200);  
    frame.setVisible(true);  
}
```



A bigger font will force the frame to allocate more space for the button's height.

The width stays the same, but now the button is taller. The north region stretched to accommodate the button's new preferred height.



But what happens in the center region?

The center region gets whatever's left!

(except in one special case we'll look at later)

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

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

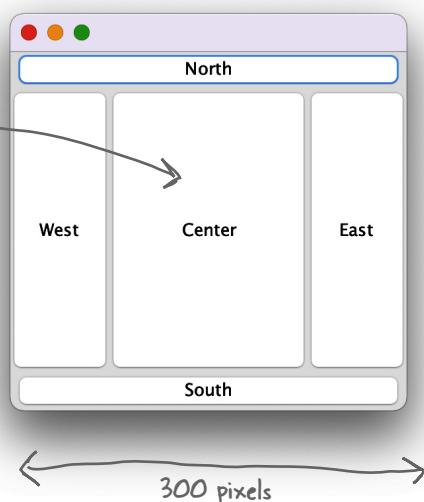
    frame.getContentPane().add(BorderLayout.EAST, east);
    frame.getContentPane().add(BorderLayout.WEST, west);
    frame.getContentPane().add(BorderLayout.NORTH, north);
    frame.getContentPane().add(BorderLayout.SOUTH, south);
    frame.getContentPane().add(BorderLayout.CENTER, center);

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

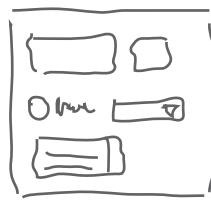
Components in the center get whatever space is left over, based on the frame dimensions (300 x 300 in this code).

Components in the east and west get their preferred width.

Components in the north and south get their preferred height.



When you put something in the north or south, it goes all the way across the frame, so the things in the east and west won't be as tall as they would be if the north and south regions were empty.



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.

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

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

public class Panel1 {

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

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



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

Let's add a button to the panel

```

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

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

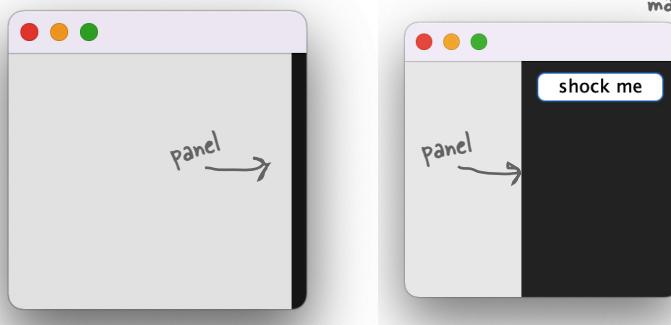
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(200, 200);
    frame.setVisible(true);
}

```

Add the button to the panel...

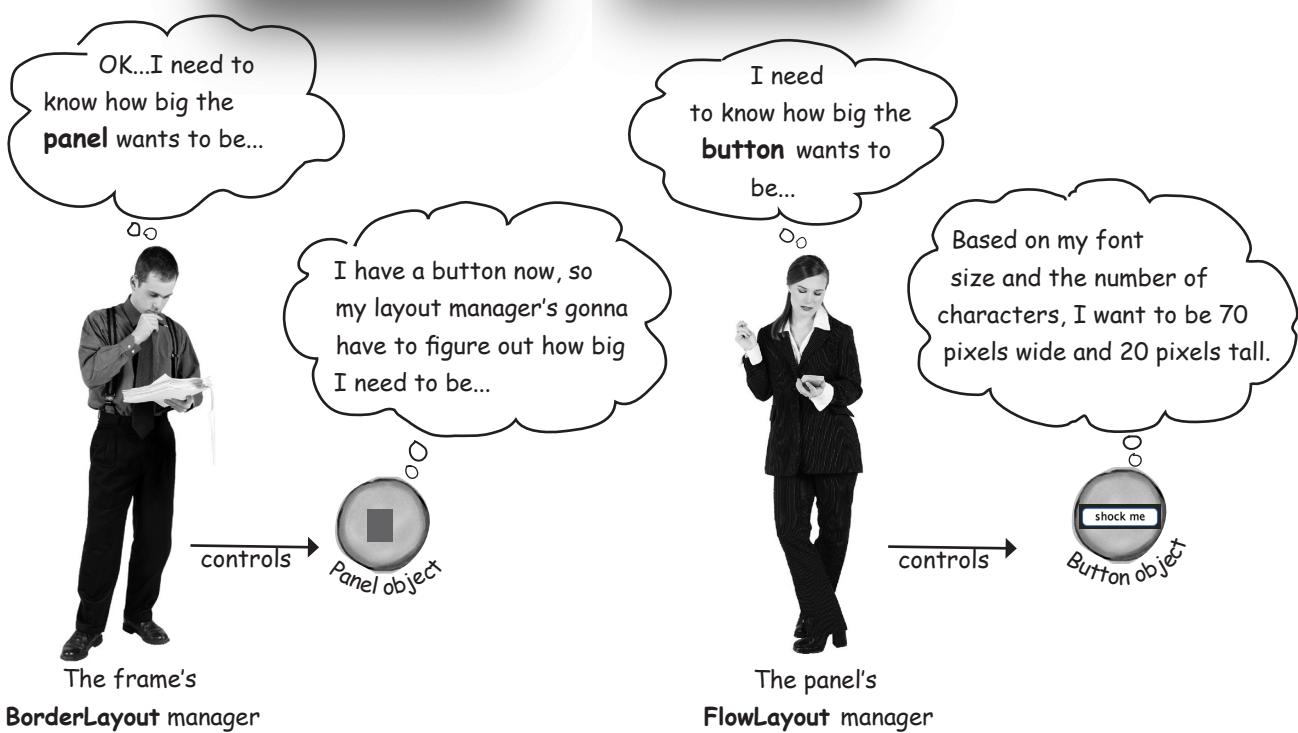
...and add the panel to the frame.

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).



The frame's
BorderLayout manager

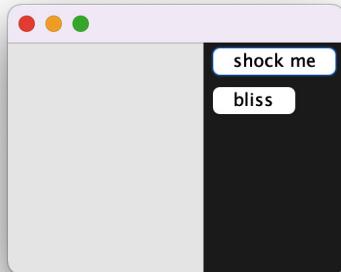
The panel's
FlowLayout manager

flow layout

What happens if we add TWO buttons to the panel?

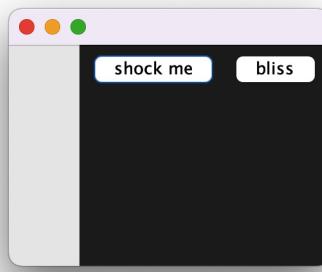
```
public void go() {  
    JFrame frame = new JFrame();  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
  
    JButton button = new JButton("shock me"); ← Make TWO buttons  
    JButton buttonTwo = new JButton("bliss"); ←  
  
    panel.add(button); ← Add BOTH to the panel  
    panel.add(buttonTwo);  
  
    frame.getContentPane().add(BorderLayout.EAST, panel);  
    frame.setSize(250, 200);  
    frame.setVisible(true);  
}
```

what we wanted:



We want the buttons stacked on top of each other.

what we got:



The panel expanded to fit both buttons side by side.

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

Sharpen your pencil

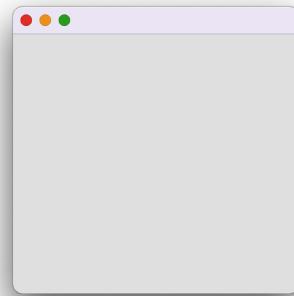
→ Yours to solve.

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.

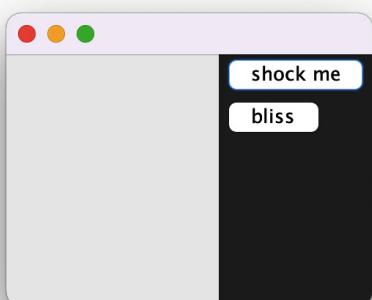
Unlike FlowLayout, BoxLayout can force a “new line” to make the components wrap to the next line, even if there’s room for them to fit horizontally.

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.getContentPane(myPanel);
```

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

A: The easiest way to do this is to make a panel, build the GUI the way you want in the panel, and then make that panel the frame's content pane using the code in the previous answer (rather than 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.



How to use it

① Get text out of it

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

② Put text in it

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

This clears the field

③ Get an ActionEvent when the user presses return or enter

You can also register for key events if you
really want to hear about it every time the
user presses a key.

```
field.addActionListener(myActionListener);
```

④ Select/Highlight the text in the field

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

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

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

text area

JTextArea

Unlike JTextField, JTextArea can have more than one line of text. It takes a little configuration to make one, because it doesn't come out of the box with scroll bars or line wrapping. To make a JTextArea scroll, you have to stick it in a JScrollPane. A JScrollPane is an object that really loves to scroll and will take care of the text area's scrolling needs.



Constructor

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

10 means 10 lines (sets
the preferred height).

20 means 20 columns (sets
the preferred width).

How to use it

- ① Make it have a vertical scrollbar only

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

Turn on line wrapping

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

```
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 TextAreal {
    public static void main(String[] args) {
        TextAreal gui = new TextAreal();
        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"));
        // Lambda expression to implement the
        // button's ActionListener.

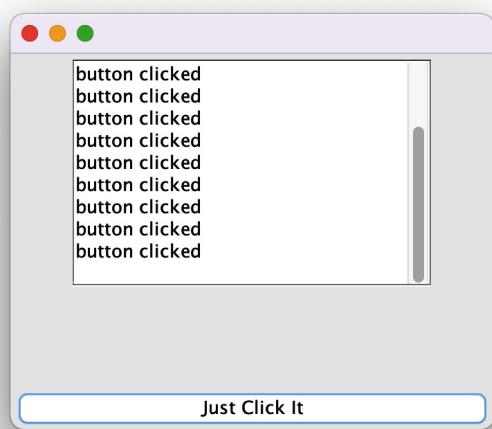
        JScrollPane scroller = new JScrollPane(text);
        scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        panel.add(scroller);

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

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

```



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

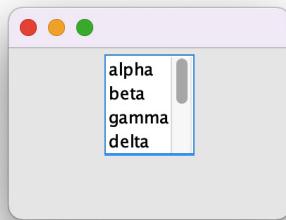
Q: Aren't the layout managers just more trouble than they're worth? If I have to go to all this trouble, I might as well just hard-code the size and coordinates for where everything should go.

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

① Make it have a vertical scrollbar

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

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

② Set the number of lines to show before scrolling

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

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

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

④ Register for list selection events

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

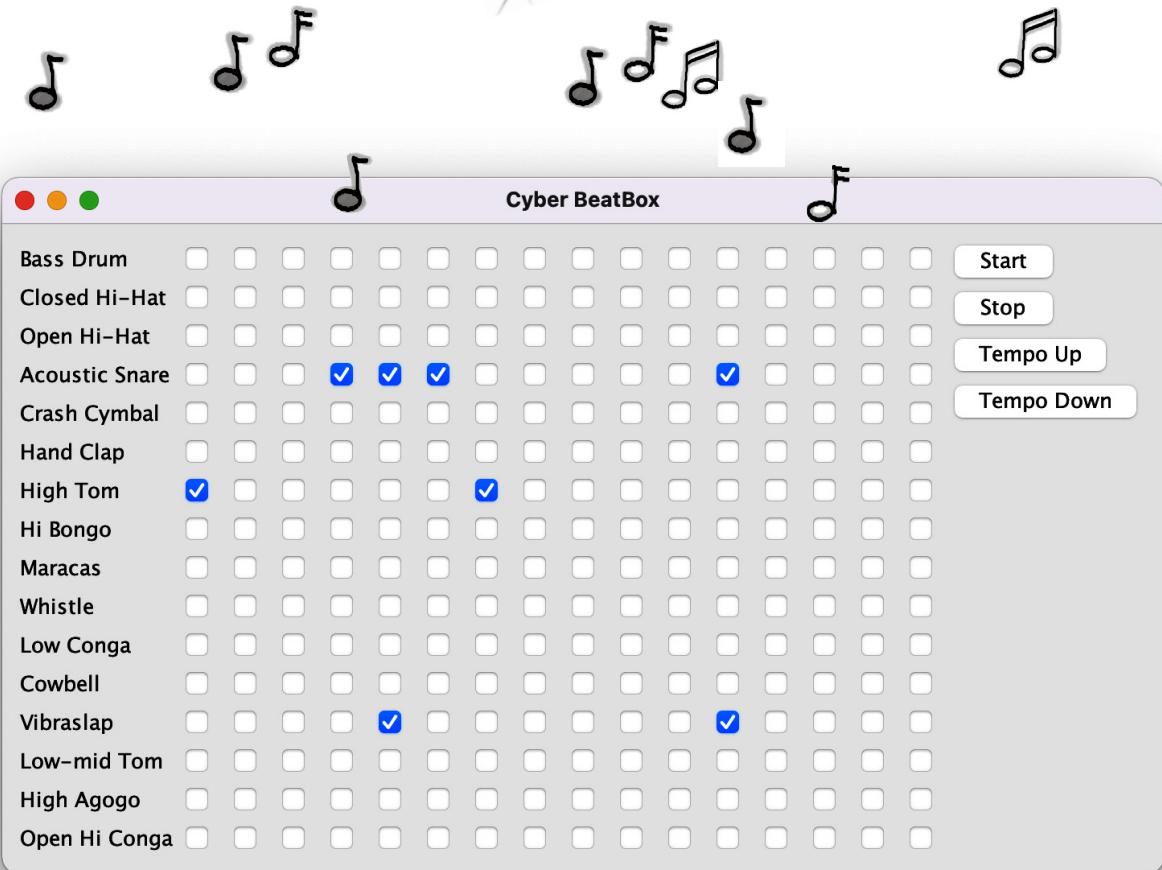
You'll get the event TWICE if you don't
put in this if test.

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

```
public void valueChanged(ListSelectionEvent 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."

BeatBox code

```
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 checkbox helps them line up with the name.

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.

BeatBox code

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();
```

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.

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

} 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.

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

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.

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

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.

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

NOW PLAY THE THING!!

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

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

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

```

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;
}
}

```

This makes events for one instrument at a time, for all 16 beats. So it might get an int[] for the Bass drum, and each index in the array will hold either the key of that instrument or a zero. If it's a zero, the instrument isn't supposed to play at that beat. Otherwise, make an event and add it to the track.

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

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

exercise: Which Layout?



Which code goes with which layout?

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

The image shows six mobile device screens, each with a large question mark in the center, indicating they are待匹配 (waiting to be matched) with their corresponding code fragments. The screens are arranged in two columns of three. Each screen has a large number (1 through 6) in a black circle at its top-left corner.

- Screen 1:** A white screen with a single text field at the top containing "tesuji".
- Screen 2:** A white screen with a single text field in the bottom right corner containing "tesuji".
- Screen 3:** A white screen with a text field at the top containing "tesuji" and a button at the bottom containing "watari".
- Screen 4:** A white screen with a text field in the bottom right corner containing "tesuji" and a button at the bottom containing "watari".
- Screen 5:** A black screen with a small white text field at the top containing "watari" and a white button at the bottom containing "tesuji".
- Screen 6:** A white screen with a text field in the bottom right corner containing "tesuji" and a black vertical bar on the right side containing a white text field at the top with "watari" and a white button at the bottom.

→ 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

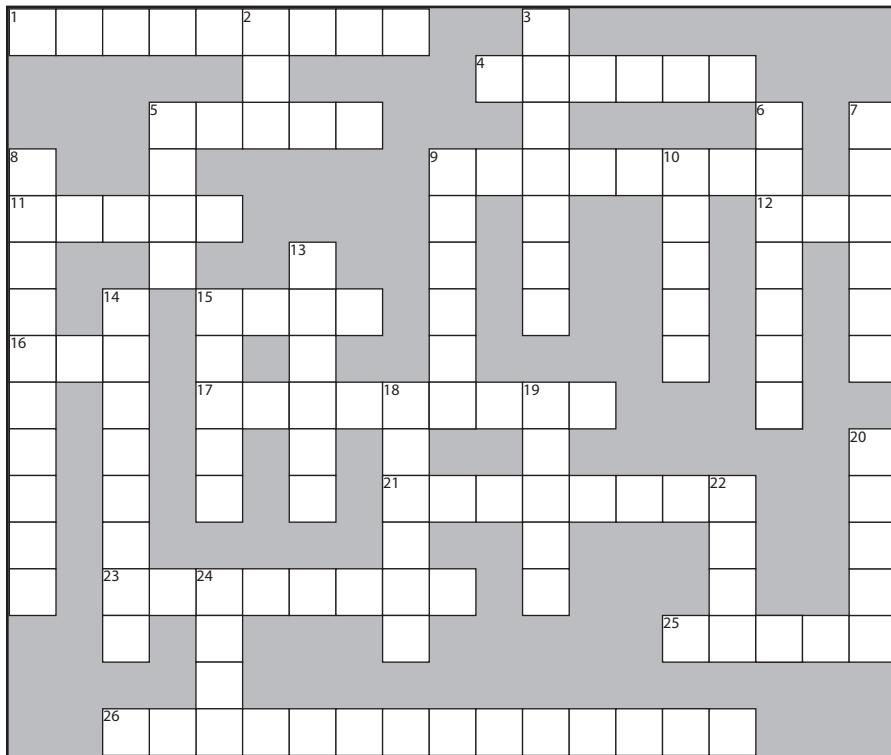
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,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

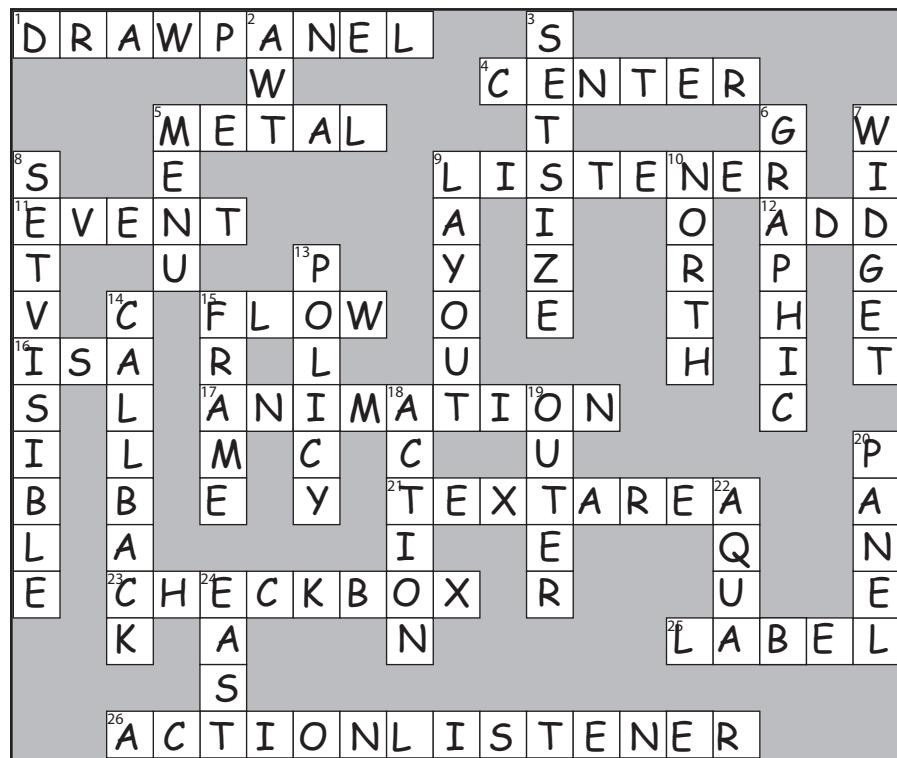


B

```
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)



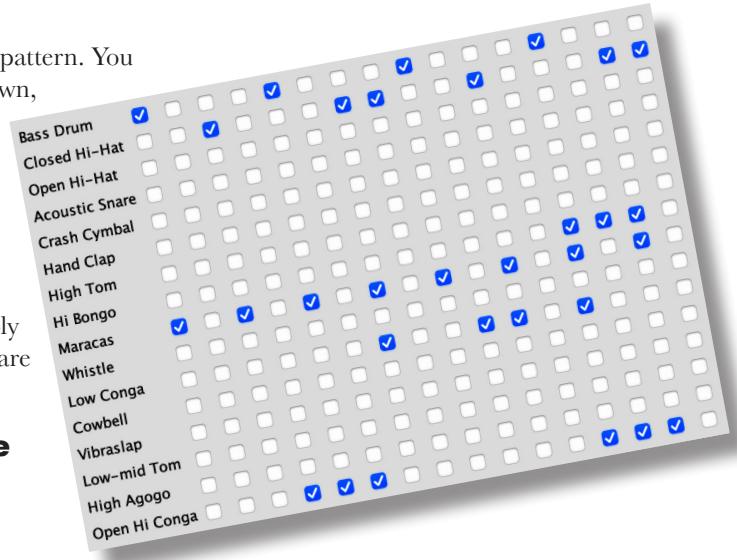
Objects can be flattened and inflated. Objects have state and behavior. *Behavior* lives in the *class*, but *state* lives within each individual *object*. So what happens when it's time to *save* the state of an object? If you're writing a game, you're gonna need a Save/Restore Game feature. If you're 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:

```
“IsrGameCharacter  
“%g 8MUIpowerIjava/lang/  
String;[weaponst[Ijava/lang/  
String;xp2tlfur[Ijava.lang.String;*“V   
È(Gxptbowtsworthdustsq~»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**

```
GameCharacter
int power
String type
Weapon[] weapons

getWeapon()
useWeapon()
increasePower()
// more
```

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);
```

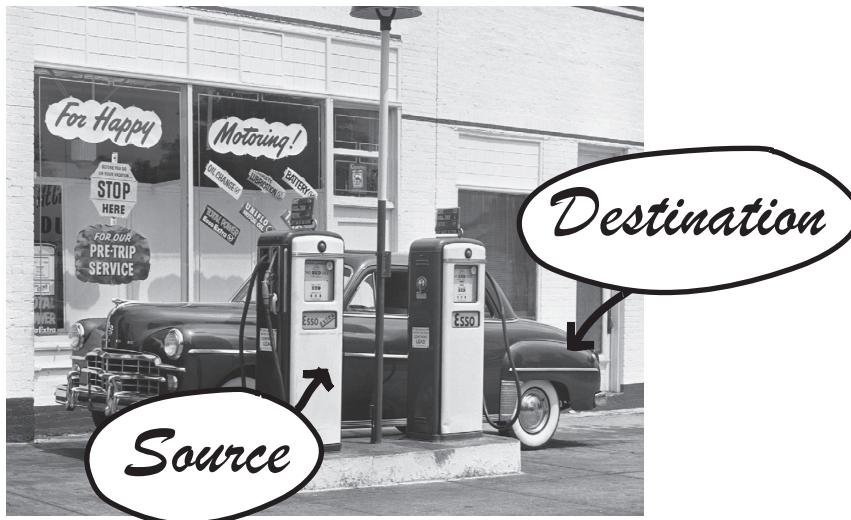
Serializes the objects referenced by characterOne, characterTwo, and characterThree, and writes them in this order to the file "MyGame.ser."

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



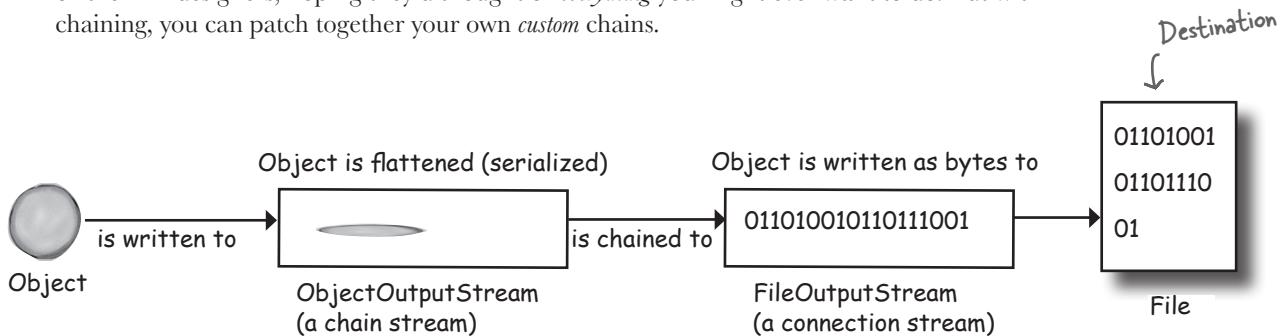
Connection streams represent a connection to a source or destination (file, network socket, etc.), while chain streams can't connect on their own and must be chained to a connection stream.

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.

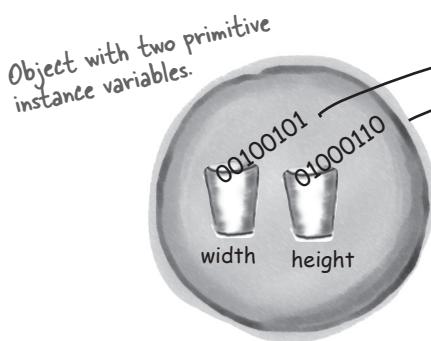


What really happens to an object when it's serialized?

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.

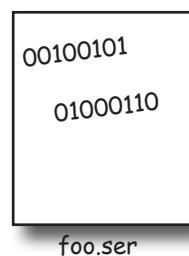


```
Foo myFoo = new Foo();  
myFoo.setWidth(37);  
myFoo.setHeight(70);
```

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.



```
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.



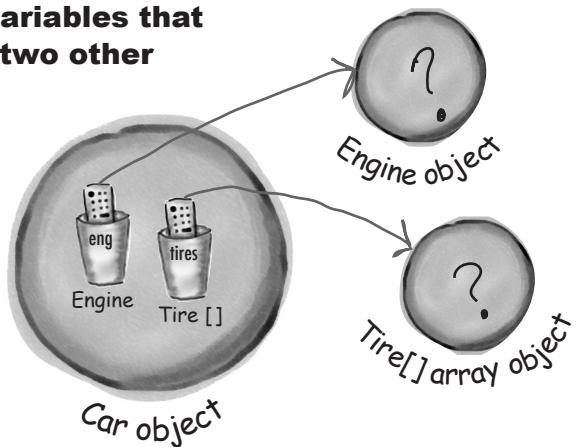
Brain Barbell

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.



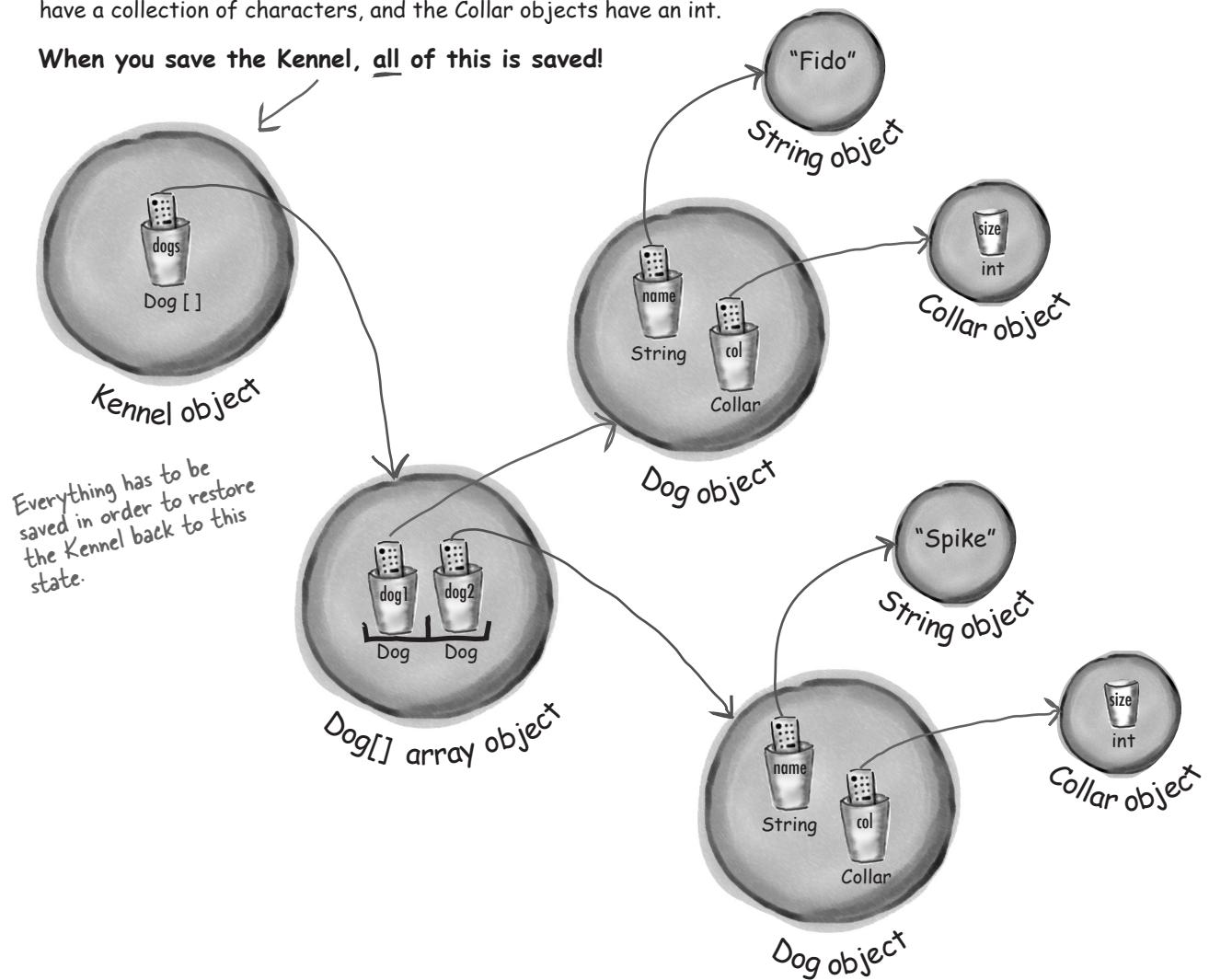
What does it take to save a Car object?

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.

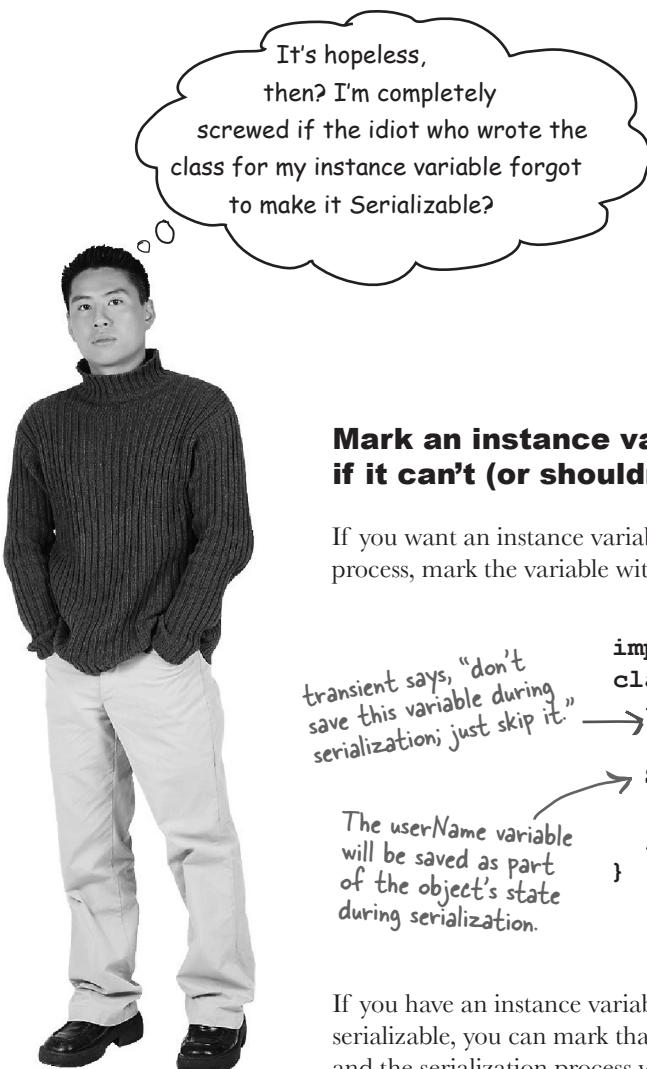
You can't serialize a Pond object if its Duck instance variable refuses to be serialized (by not implementing Serializable).

```
import java.io.*;
public class Pond implements Serializable {
    private Duck duck = new Duck(); ← Class Pond has one instance
    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); ← When you serialize myPond (a Pond
            os.close(); object), its Duck instance variable
        } catch (Exception ex) { automatically gets serialized.
            ex.printStackTrace();
        }
    }
}

public class Duck { ← Argh!! Duck is not serializable!
    // duck code here
}
```

When you try to run the main in class Pond:

```
File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
at Pond.main(Pond.java:13)
```



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

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

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

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

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 serialized 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:

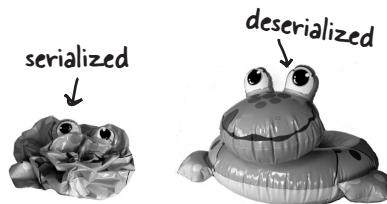
1. When the object is brought back, reinitialize that null instance variable back to some default state. This works if your serialized object isn't dependent on a particular value for that transient variable. In other words, it might be important that the Dog have a Collar, but perhaps all Collar objects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



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

1 Make a FileInputStream

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

Make a FileInputStream object. 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

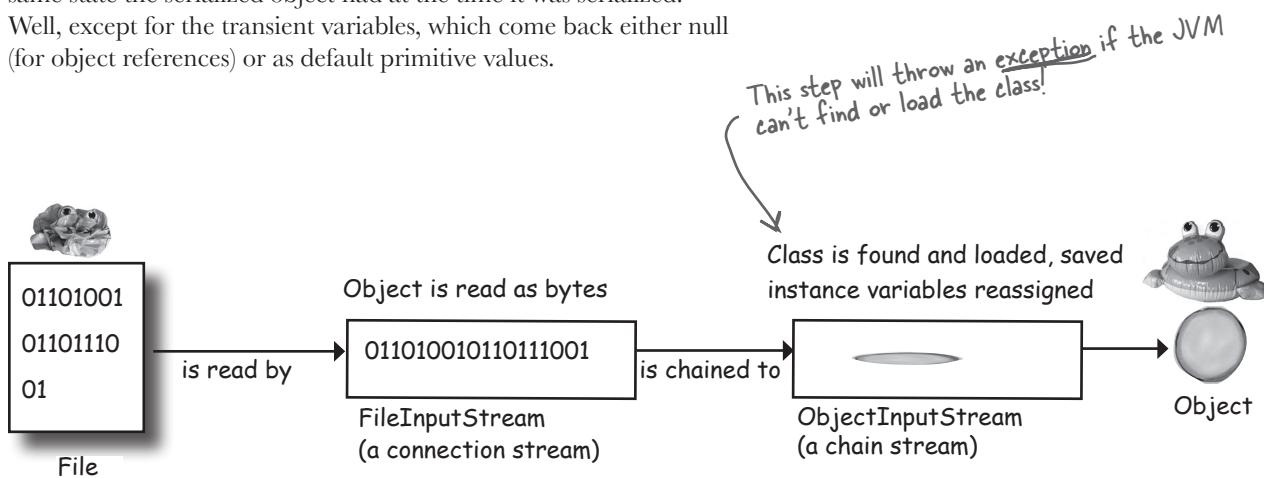
```
os.close();
```

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.



- ➊ The object is **read** from the stream.
- ➋ The JVM determines (through info stored with the serialized object) the object's **class type**.
- ➌ 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.
- ➍ 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.

- ➅ 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.

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

there are no Dumb Questions

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

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

For objects serialized to ship over a network, though, there actually *is* a mechanism where the serialized object can be "stamped" with a URL for where its class can be found. This is used in Java's Remote Method Invocation (RMI) so that

you can send a serialized object as part of, say, a method argument, and if the JVM receiving the call doesn't have the class, it can use the URL to fetch the class from the network and load it, all automatically. 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) {           Make some characters...
        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);           Serialize the characters.
            os.writeObject(two);
            os.writeObject(three);
            os.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        try {                                Now read them back in from the file...
            ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
            GameCharacter oneRestore = (GameCharacter) is.readObject();           Check to see if it worked.
            GameCharacter twoRestore = (GameCharacter) is.readObject();           Restore the characters.
            GameCharacter threeRestore = (GameCharacter) is.readObject();

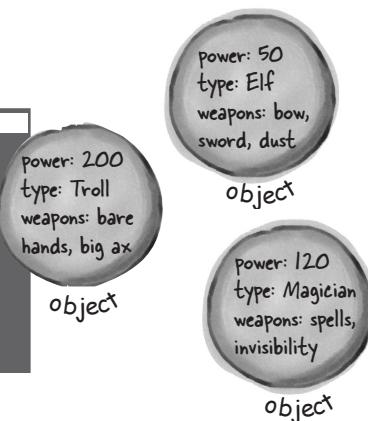
            System.out.println("One's type: " + oneRestore.getType());
            System.out.println("Two's type: " + twoRestore.getType());
            System.out.println("Three's type: " + threeRestore.getType());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

```

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)

- ① You write a Dog class.

```

101101
101101
10100000010
1010 10 0
01010 1
1010101
10101010
1001010101

```

Dog.class

class version ID
#343

- ② You serialize a Dog object using that class.

Dog object

Object is stamped with version #343

- ③ You change the Dog class.

```

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

```

Dog.class

class version ID
#728

- ④ You deserialize a Dog object using the changed class.

Dog object

Object is stamped with version #343

Dog.class

class version is #728

- ⑤ Serialization fails!!

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

Using the serialVersionUID

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

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

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

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

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

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

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

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

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

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

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

- ② Paste the output into your class.

```
public class Dog {

    static final long serialVersionUID =
        -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 `OutputStream` (and you don't chain it to an `ObjectOutputStream`).

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

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

To write a serialized object:

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

To write a String:

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

```
import java.io.*; // We need the java.io package for FileWriter.

class WriteAFile {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("Foo.txt");
            writer.write("hello foo!"); // The write() method takes
                                         // a String.
            writer.close(); // Close it when you're done!
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

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

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

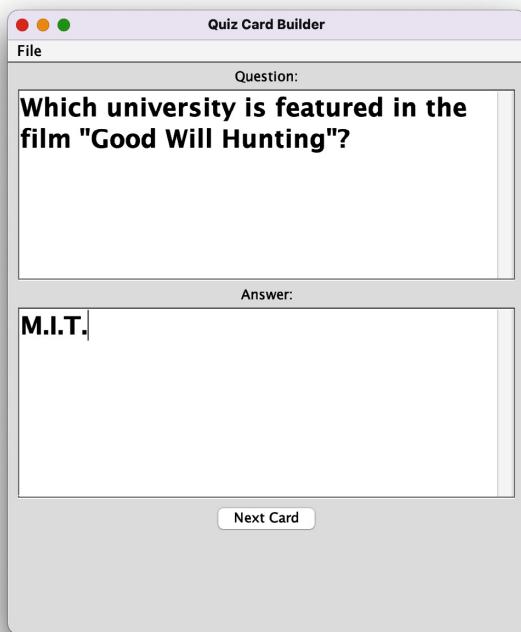
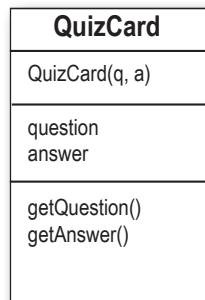
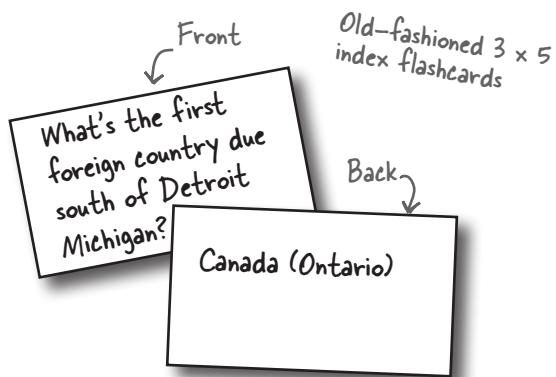
writing a text file

Text file example: e-Flashcards

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

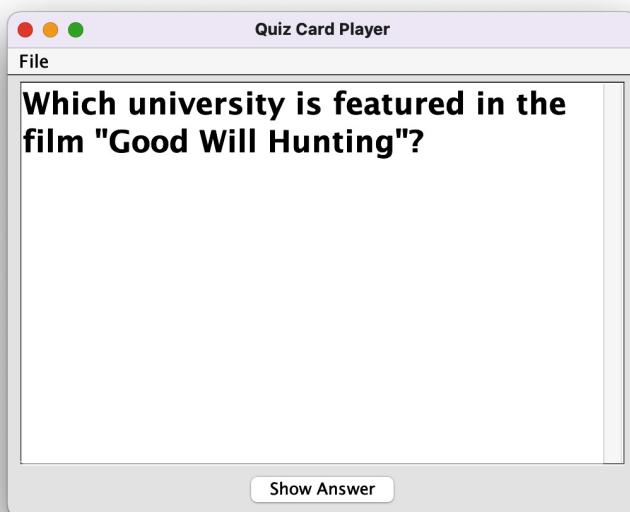
We're going to make an electronic version that has three classes:

1. **QuizCardBuilder**, a simple authoring tool for creating and saving a set of e-Flashcards.
2. **QuizCardPlayer**, a playback engine that can load a flashcard set and play it for the user.
3. **QuizCard**, a simple class representing card data. We'll walk through the code for the builder and the player, and have you make the QuizCard class yourself, using this: →



QuizCardBuilder

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



QuizCardPlayer

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

Quiz Card Builder (code outline)

```

public class QuizCardBuilder {
    public void go() {
        // 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 SaveMenuItemListener; 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.

Quiz Card Builder 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 newItem = new JMenuItem("New");
        newItem.addActionListener(e -> clearAll());

        JMenuItem saveMenuItem = new JMenuItem("Save");
        saveMenuItem.addActionListener(e -> saveCard());

        fileMenu.add(newItem);
        fileMenu.add(saveMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(500, 600);
        frame.setVisible(true);
    }
}
```

Reminder: For the next eight pages or so we'll be using older-style I/O code!

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 `MenuItem`.

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.

```

private JScrollPane createScroller(JTextArea textArea) {
    JScrollPane scroller = new JScrollPane(textArea);
    scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    return scroller;
}

private JTextArea createTextArea(Font font) {
    JTextArea textArea = new JTextArea(6, 20);
    textArea.setLineWrap(true);
    textArea.setWrapStyleWord(true);
    textArea.setFont(font);
    return textArea;
}

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

private void saveCard() {
    QuizCard card = new QuizCard(question.getText(), answer.getText());
    cardList.add(card);

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

```

Creating a scroll pane or a text area needs a lot of similar-looking code. We've put the code into a couple of helper methods that we can call when we need a text area or scroll pane.

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

private void clearAll() {

```

    cardList.clear();
    clearCard(); ←
}

```

When we want a new set of cards, we need to clear out the card list AND the text areas.

private void clearCard() {

```

    question.setText("");
    answer.setText("");
    question.requestFocus();
}

```

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

private void saveFile(File file) {

```

try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(file));
    for (QuizCard card : cardList) {
        writer.write(card.getQuestion() + "/");
        writer.write(card.getAnswer() + "\n"); ←
    }
    writer.close();
} catch (IOException e) {
    System.out.println("Couldn't write the cardList out: " + e.getMessage());
}
}

```

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

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

The `java.io.File` class

The `java.io.File` class is another example of an older class in the Java API. It's been "replaced" by two classes in the newer `java.nio.file` package, but you'll undoubtedly encounter code that uses the `File` class. **For new code, we recommend using the `java.nio.file` package instead of the `java.io.File` class.** In a few pages, we'll take a look at a few of the most important capabilities in the `java.nio.file` package. With that said...

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

Some things you can do with a File object:

- ① Make a File object representing an existing file

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

- ② Make a new directory

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

- ③ List the contents of a directory

```
if (dir.isDirectory()) {
    String[] dirContents = dir.list();
    for (String dirContent : dirContents) {
        System.out.println(dirContent);
    }
}
```

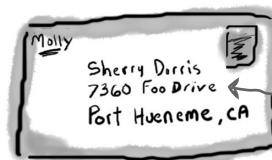
- ④ Delete a file or directory (returns true if successful)

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

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

`/Users/Kathy/Data/Game.txt`

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



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

A File object represents the filename "GameFile.txt"

GameFile.txt

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

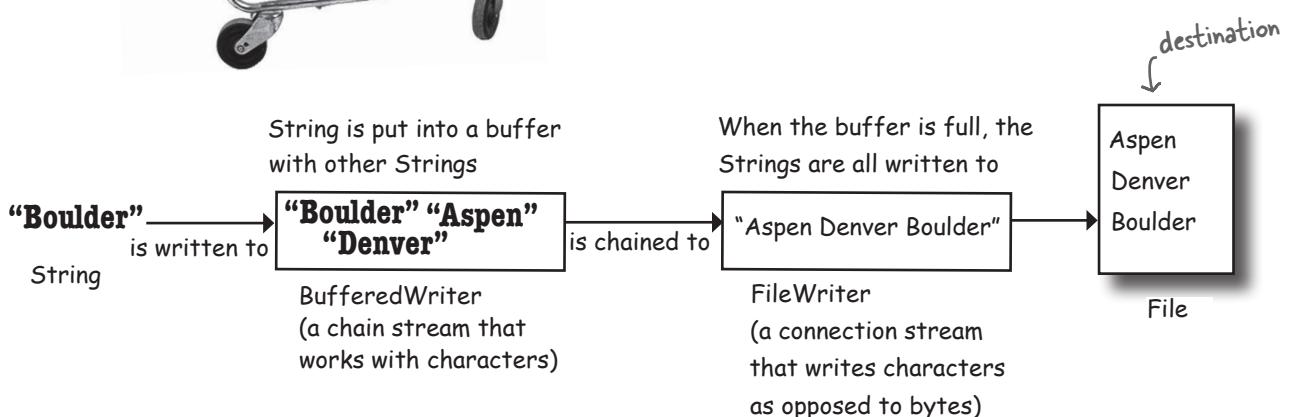
A File object does NOT represent (or give you direct access to) the data inside the file!

The beauty of buffers

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



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



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

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

If you do want to send data *before* the buffer is full, you do have control.

Just Flush It. Calls to `writer.flush()` say, “send whatever’s in the buffer, **now!**”

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

Reading from a text file

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

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

```

import java.io.*;           Don't forget the import.

class ReadAFile {
    public static void main(String[] args) {
        try {
            File myFile = new File("MyText.txt");
            FileReader fileReader = new FileReader(myFile);

            BufferedReader reader = new BufferedReader(fileReader);
            Make a String variable to hold
            each line as the line is read
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
  
```

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

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

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

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

A file with two lines of text.

What's $2 + 27/4$

What's $20+22/42$

MyText.txt

Java 8 Streams and I/O

If you're using Java 8 and you feel comfortable using the Streams API, you can replace all the code inside the try block with the following:

```

Files.lines(Path.of("MyText.txt"))
    .forEach(line -> System.out.println(line));
  
```

We'll see the Files and Path classes later in this chapter.

Quiz Card Player (code outline)

```
public class QuizCardPlayer {  
  
    public void go() {  
        // build and display gui  
    }  
  
    private void nextCard() {  
        // if this is a question, show the answer, otherwise show  
        // next question set a flag for whether we're viewing a  
        // question or answer  
    }  
  
    private void open() {  
        // bring up a file dialog box  
        // let the user navigate to and choose a card set to open  
    }  
  
    private void loadFile(File file) {  
        // must build an ArrayList of cards, by reading them from  
        // a text file called from the OpenMenuItem event handler,  
        // reads the file one line at a time and tells the makeCard()  
        // method to make a new card out of the line (one line in the  
        // file holds both the question and answer, separated by a "/")  
    }  
  
    private void makeCard(String lineToParse) {  
        // called by the loadFile method, takes a line from the text file  
        // and parses into two pieces—question and answer—and creates a  
        // new QuizCard and adds it to the ArrayList called CardList  
    }  
}
```

Quiz Card Player code

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

public class QuizCardPlayer {
    private ArrayList<QuizCard> cardList;
    private int currentCardIndex;
    private QuizCard currentCard;
    private JTextArea display;
    private JFrame frame;
    private JButton nextButton;
    private boolean isShowAnswer;

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

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

        display = new JTextArea(10, 20);
        display.setFont(bigFont);
        display.setLineWrap(true);
        display.setEditable(false);

        JScrollPane scroller = new JScrollPane(display);
        scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        mainPanel.add(scroller);

        nextButton = new JButton("Show Question");
        nextButton.addActionListener(e -> nextCard());
        mainPanel.add(nextButton);

        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem loadMenuItem = new JMenuItem("Load card set");
        loadMenuItem.addActionListener(e -> open());
        fileMenu.add(loadMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(500, 400);
        frame.setVisible(true);
    }
}
```

*Just GUI code on this page;
nothing special.*

```

private void nextCard() {
    if (isShowAnswer) {
        // show the answer because they've seen the question
        display.setText(currentCard.getAnswer());
        nextButton.setText("Next Card");
        isShowAnswer = false;
    } else { // show the next question
        if (currentCardIndex < cardList.size()) {
            showNextCard();
        } else {
            // there are no more cards!
            display.setText("That was last card");
            nextButton.setEnabled(false);
        }
    }
}

```

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

```

private void open() {
    JFileChooser fileOpen = new JFileChooser();
    fileOpen.showOpenDialog(frame);
    loadFile(fileOpen.getSelectedFile());
}

```

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

```

private void loadFile(File file) {
    cardList = new ArrayList<>();
    currentCardIndex = 0;
    try {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line;
        while ((line = reader.readLine()) != null) {
            makeCard(line);
        }
        reader.close();
    } catch (IOException e) {
        System.out.println("Couldn't write the cardList out: " + e.getMessage());
    }
    showNextCard(); ← Now time to start,
}

```

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

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

```

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

```

```

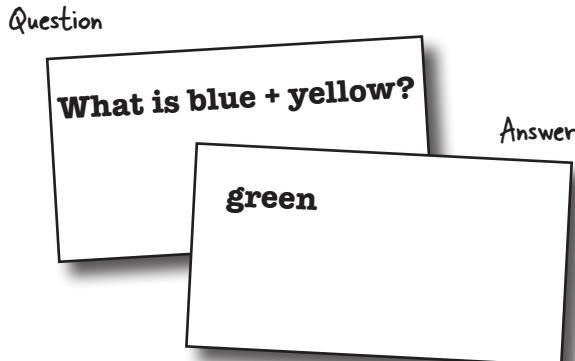
private void showNextCard() {
    currentCard = cardList.get(currentCardIndex);
    currentCardIndex++;
    display.setText(currentCard.getQuestion());
    nextButton.setText("Show Answer");
    isShowAnswer = true;
}

```

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

Parsing with String split()

Imagine you have a flashcard like this:



Saved in a question file like this:

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

How do you separate the question and answer?

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

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

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

What is blue + yellow?

token 1



token 2

green

separator

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

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

The split() method takes the "/" and uses it to break apart the String into (in this case) two pieces, token 1 and token 2. (Note: split() is FAR more powerful than what we're using it for here. It can do extremely complex parsing with filters, wildcards, etc.)

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

there are no Dumb Questions

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

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

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

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

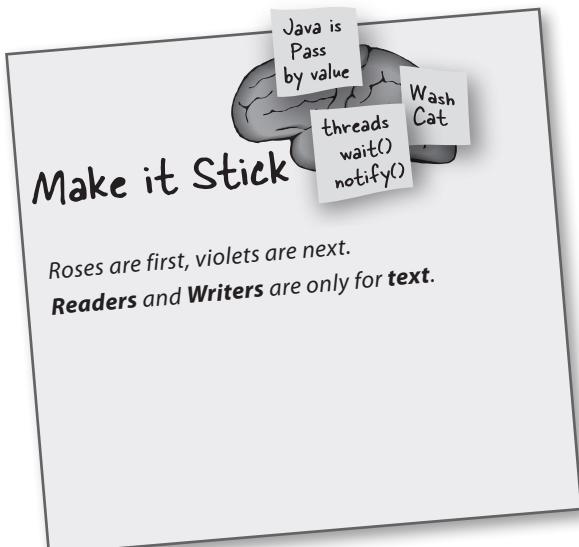
In other words, 90% of what you might typically do with Java I/O can use what we’ve already covered.

Q: You just said we’ve already learned 90% of what we’ll probably use, but we haven’t seen the fabled NIO.2 stuff yet. What gives?

A: NIO.2 is coming up on the very next page! But for reading and writing text files, `BufferedReader`s and `BufferedWriter`s are still usually the way to go. So we’ll be looking at how NIO.2 makes using them easier.

Q: My brain is a little tired, and I’ve heard NIO.2 is pretty complicated.

A: We’re going to focus on a few key concepts in the `java.nio.file` package.



BULLET POINTS

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

NIO.2 and the `java.nio.file` package

Java NIO.2 is usually taken to mean two packages added in Java 7:

`java.nio.file`

`java.nio.file.attribute`

The `java.nio.file.attribute` package lets you manipulate the *metadata* associated with a computer's files and directories. For example, you would use the classes in this package if you wanted to read or change a file's permissions settings. We WON'T be discussing this package further. (phew)

The `java.nio.file` package is all you need to do common text file reading and writing, and it also provides you with the ability to manipulate a computer's directories and directory structure. Most of the time you'll use three types in `java.nio.file`:

- The Path interface: You'll always need a Path object to locate the directories or files you want to work with.
- The Paths class: You'll use the `Paths.get()` method to make the Path object you'll need when you use methods in the Files class.
- The Files class: This is the class whose (static) methods do all the work you'll want to do: making new Readers and Writers, and creating, modifying, and searching through directories and files on file systems.

A mini-tutorial, creating a `BufferedWriter` with NIO.2

- ① Import Path, Paths, and Files:

```
import java.nio.file.*;
```

- ② Make a Path object using the Paths class:

```
Path myPath = Paths.get("MyFile.txt");
```

Or, if the file is in a subdirectory like:

/myApp/files/MyFile.txt :

```
Path myPath = Paths.get("/myApp", "files", "MyFile.txt");
```

A Path object is used to locate a file on a computer (i.e., in the file system). A path can be used to locate files in the current directory or in other directories.

- ③ Make a new BufferedWriter using a Path and the Files class:

```
BufferedWriter writer = Files.newBufferedWriter(myPath);
```

The "/" in "/myApp" is called the name-separator. Depending on which OS you're using, your name-separator might be different; for example, it might be "\".

Somewhere—under the covers—some method is saying:

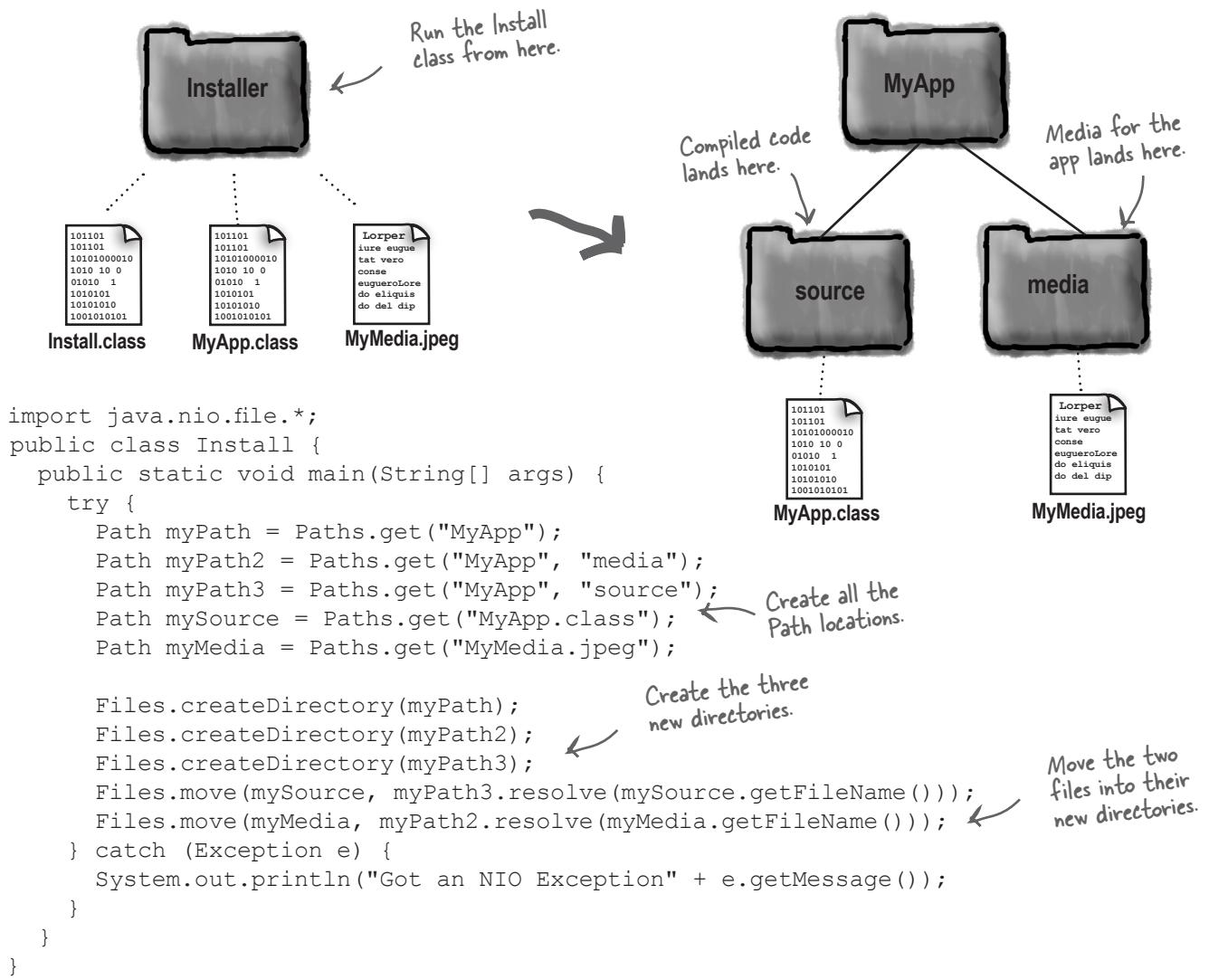
```
BufferedWriter writer =
new BufferedWriter(..)
```

Path, Paths, and Files (messing with directories)

In Appendix B, we'll be discussing how to split your Java app into packages. This includes creating the proper directory structure for all of your app's files. In most cases you'll make and move directories and files by hand, using the command line or utilities like the Finder or Windows Explorer. But you can also do it from within your Java code.

Warning! Goofing around with directories in a Java program is a real “can of worms” topic. To do it correctly you need to learn about paths, absolute paths, relative paths, OS permissions, file attributes, and on and on. Below is a greatly simplified example of messing around with directories, just to give you a feel for what's possible.

Suppose you wanted to make an installer program to install your killer app. You start with the directory and files on the left, and want to end up with the directory structure and files on the right.



Finally, a closer look at finally

Several chapters ago we looked at how try-catch-finally worked. Kind of. All we said about finally was that it was a good place to put your “cleanup code.” That’s true, but let’s get more specific. Most of the time, when we talk about “cleanup code,” we mean closing resources we borrowed from the operating system. When we open a file or a socket, the OS is giving us some of its resources. When we’re done with them, we need to give them back. Below is a snippet of code from the QuizCardBuilder class. We highlighted a call to a constructor and three separate method calls...

That's FOUR places an exception can be thrown!

```
private void saveFile(File file) {  
    try {  
        BufferedWriter writer = new BufferedWriter(new FileWriter(file));  
        for (QuizCard card : cardList) {  
            writer.write(card.getQuestion() + "/");  
            writer.write(card.getAnswer() + "\n");  
        }  
        writer.close();  
    } catch (IOException e) {  
        System.out.println("Couldn't write the cardList out: " + e.getMessage());  
    }  
}
```

The code is annotated with four arrows pointing to the following lines:

- A curved arrow points to the line `new FileWriter(file))`, indicating it's a constructor call.
- A horizontal arrow points to the line `writer.write(card.getAnswer() + "\n");`, indicating it's a method call.
- A horizontal arrow points to the line `writer.close();`, indicating it's another method call.
- A curved arrow points to the line `catch (IOException e)`, indicating it's a catch block.

A handwritten note to the right of the annotations reads: **All the places an exception could be thrown!**

If the call to make a new `FileWriter` fails, if ANY of the many `write()` invocations fail, or the `close()` itself fails, an exception will be thrown, the JVM will jump to the catch block, and the writer will never be closed. Yikes!

Remember, finally **ALWAYS** runs!!

Since we REALLY want to make sure we close the writer file, let’s put the `close()` invocation in a finally block.



Sharpen your pencil

→ Yours to solve.

Coding the new finally block

What changes will we have to make to the code above to move the `close()` to a finally block?

There might be more than you first imagine.

Finally, a closer look at finally, cont.

The amount of code required to put the close() in the finally block might surprise you; let's take a look.

```
private void saveFile(File file) {
    BufferedWriter writer = null; ←
    try {
        writer = new BufferedWriter(new FileWriter(file));
        for (QuizCard card : cardList) {
            writer.write(card.getQuestion() + "/");
            writer.write(card.getAnswer() + "\n");
        }
        writer.close();
    } catch (IOException e) {
        System.out.println("Couldn't write the cardList out: " + e.getMessage());
    } finally {
        try {
            writer.close(); ←
        } catch (Exception e) {
            System.out.println("Couldn't close writer: " + e.getMessage());
        }
    }
}
```

We had to declare the writer reference outside of the try block so that it's visible in the finally block.

Yup, we had to put the close() in yet another try-catch block!

Are you kidding me right now?
I have to write all of this code
every time I want to do a little
I/O? Verbose much?



There IS a better way!

In the early days of Java, this is how you had to make sure you were really closing a file. You are very likely to encounter finally blocks that look like this when you're looking at existing code. But for new code, there is a better way:

Try-With-Resources

We'll look at that next.

The try-with-resources (TWR) statement

If you're using Java 7 or later (and we sure hope you are!), you can use the try-with-resources version of try statements to make doing I/O easier. Let's compare the try code we've been looking at with try-with-resources code that does the same thing:

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

        for (QuizCard card : cardList) {
            writer.write(card.getQuestion() + "/");
            writer.write(card.getAnswer() + "\n");
        }
    } catch (IOException e) {
        System.out.println("Couldn't write the cardList out: " + e.getMessage());
    } finally {
        try {
            writer.close();
        } catch (Exception e) {
            System.out.println("Couldn't close writer: " + e.getMessage());
        }
    }
}
```

*Old style,
try-catch-finally
code*

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

        for (QuizCard card : cardList) {
            writer.write(card.getQuestion() + "/");
            writer.write(card.getAnswer() + "\n");
        }
    } catch (IOException e) {
        System.out.println("Couldn't write the cardList out: " + e.getMessage());
    }
}
```

*Modern,
try-with-resources
code*

*there are no
Dumb Questions*

Q: Wait, what? You told us that a try statement needs a catch and/or a finally?

A: Nice catch! It turns out that when you use try-with-resources, the compiler makes a finally block for you. You can't see it, but it's there.

Autocloseable, the very small catch

On the last page we saw a different kind of try statement, the try-with-resources statement (TWR). Let's take a look at how to write and use TWR statements by first, deconstructing the following:

```
try (BufferedWriter writer =
      new BufferedWriter(new FileWriter(file))) {
```

Writing a try-with-resources statement

- ① Add a set of parentheses between "try" and "{":

```
try ( ... ) {
```

- ② Inside the parentheses, declare an object whose type implements Autocloseable:

```
try (BufferedWriter writer =
      new BufferedWriter(new FileWriter(file))) {
```

Like all of the I/O classes we've been using this chapter, BufferedWriter implements Autocloseable.

- ③ Use the object you declared inside the try block (just like you always did):

```
writer.write(card.getQuestion() + "/");
writer.write(card.getAnswer() + "\n");
```

Autocloseable, it's everywhere you do I/O

Autocloseable is an interface that was added to `java.lang` in Java 7. Almost all of the I/O you're ever going to do uses classes that implement Autocloseable. You mostly won't have to think about it.

There are a few more things worth knowing about TWR statements:

- You can declare and use more than one I/O resource in a single TWR block:

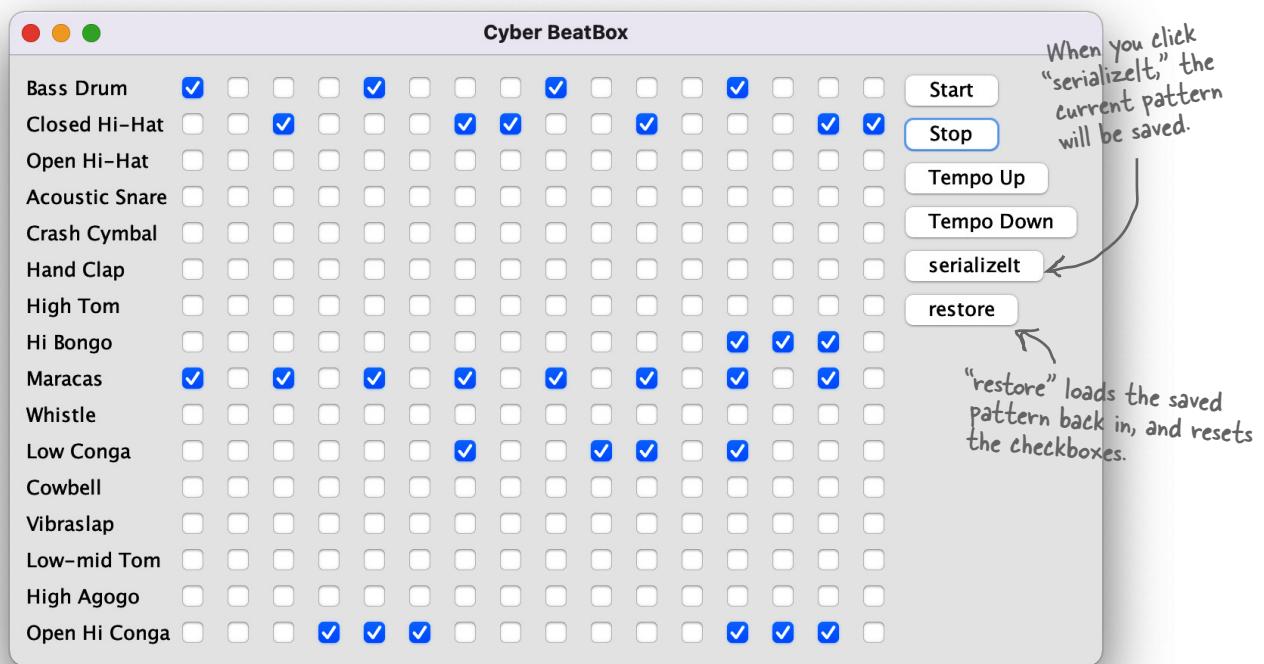
```
try (BufferedWriter writer =
      new BufferedWriter(new FileWriter(file));
      BufferedReader reader =
      new BufferedReader(new FileReader(file))) {
```

Separate the resources using semicolons, ;.

- If you declare more than one resource, they will be closed in the order OPPOSITE to which they were declared; i.e., first declared is last closed.
- If you add catch or finally blocks, the system will handle multiple close() invocations gracefully.

ONLY classes that implement Autocloseable can be used in TWR statements!

Code Kitchen



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

Saving a BeatBox pattern

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

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

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

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

Serializing a pattern

This is a method in the BeatBox code. We can call this from a lambda expression when we add an ActionListener to the serialize button, or create an ActionListener inner class that calls this.

```
private void writeFile() {  
  
    boolean[] checkboxState = new boolean[256];  
  
    for (int i = 0; i < 256; i++) {  
        JCheckBox check = checkboxList.get(i);  
        if (check.isSelected()) {  
            checkboxState[i] = true;  
        }  
    }  
  
    try (ObjectOutputStream os =  
         new ObjectOutputStream(new FileOutputStream("Checkbox.ser"))){  
        os.writeObject(checkboxState);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



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

Try-with-resources

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

Restoring a BeatBox pattern

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

Restoring a pattern

This is another method in the BeatBox class.

```
private void readFile() {
    boolean[] checkboxState = null;
    try (ObjectInputStream is =
        new ObjectInputStream(new FileInputStream("Checkbox.ser"))) {
        checkboxState = (boolean[]) is.readObject(); ← Try-with-resources
    } catch (Exception e) {
        e.printStackTrace();
    }

    for (int i = 0; i < 256; i++) {
        JCheckBox check = checkboxList.get(i);
        check.setSelected(checkboxState[i]); } Now restore the state of each of the checkboxes in the ArrayList of actual JCheckBox objects (checkboxList).
    }

    sequencer.stop();
    buildTrackAndStart(); } Now stop whatever is currently playing,
    and rebuild the sequence using the new state of the checkboxes in the ArrayList.
}
```

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



→ Yours to solve.

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

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



→ Yours to solve.

Can they be saved?

Which of these do you think are, or should be, serializable? If not, why not? Not meaningful?

Security risk? Only works for the current execution of the JVM? Make your best guess, without looking it up in the API.

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

What's Legal?

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



→ Yours to solve.

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

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

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

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

exercise: True or False



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

TRUE OR FALSE

1. Serialization is appropriate when saving data for non-Java programs to use.
2. Object state can be saved only by using serialization.
3. ObjectOutputStream is a class used to save serializable objects.
4. Chain streams can be used on their own or with connection streams.
5. A single call to writeObject() can cause many objects to be saved.
6. All classes are serializable by default.
7. The java.nio.file.Path class can be used to locate files.
8. If a superclass is not serializable, then the subclass can't be serializable.
9. Only classes that implement AutoCloseable can be used in try-with-resources statements.
10. When an object is deserialized, its constructor does not run.
11. Both serialization and saving to a text file can throw exceptions.
12. BufferedWriter can be chained to FileWriter.
13. File objects represent files, but not directories.
14. You can't force a buffer to send its data before it's full.
15. Both file readers and file writers can optionally be buffered.
16. The methods on the Files class let you operate on files and directories.
17. Try-with-resources statements cannot include explicit finally blocks.

—————> Answers on page 584.



Code Magnets

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

```

class DungeonGame implements Serializable {
    try {
        FileOutputStream fos = new
            FileOutputStream("dg.ser");
        e.printStackTrace();
        short getZ() {
            return z;
        }
        oos.close();
        int getX() {
            return x;
        }
        System.out.println(d.getX() + d.getY() + d.getZ());
    }
    FileInputStream fis = new
        FileInputStream("dg.ser");
    public int x = 3;
    transient long y = 4;
    private short z = 5;
    long getY() {
        return y;
    }
    class DungeonTest {
        import java.io.*;
        ois.close();
        fos.writeObject(d);
        } catch (Exception e) {
        d = (DungeonGame) ois.readObject();
    }
}

```

```

File Edit Window Help Torture
% java DungeonTest
12
8

```

```

ObjectOutputStream oos = new
    ObjectOutputStream(fos);
    oos.writeObject(d);
    public static void main(String[] args) {
        DungeonGame d = new DungeonGame();
}

```

→ Answers on page 585.



Exercise Solutions

TRUE OR FALSE

(from page 582)

- | | |
|--|--------------|
| 1. Serialization is appropriate when saving data for non-Java programs to use. | False |
| 2. Object state can be saved only by using serialization. | False |
| 3. ObjectOutputStream is a class used to save serializable objects. | True |
| 4. Chain streams can be used on their own or with connection streams. | False |
| 5. A single call to writeObject() can cause many objects to be saved. | True |
| 6. All classes are serializable by default. | False |
| 7. The java.nio.file.Path class can be used to locate files. | False |
| 8. If a superclass is not serializable, then the subclass can't be serializable. | False |
| 9. Only classes that implement AutoCloseable can be used in try-with-resources statements. | True |
| 10. When an object is deserialized, its constructor does not run. | True |
| 11. Both serialization and saving to a text file can throw exceptions. | True |
| 12. BufferedWriter can be chained to FileWriters. | True |
| 13. File objects represent files, but not directories. | False |
| 14. You can't force a buffer to send its data before it's full. | False |
| 15. Both file readers and file writers can optionally be buffered. | True |
| 16. The methods on the Files class let you operate on files and directories. | True |
| 17. Try-with-resources statements cannot include explicit finally blocks. | False |



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

Code Magnets

(from page 583)

```

import java.io.*;

class DungeonGame implements Serializable {
    public int x = 3;
    transient long y = 4;
    private short z = 5;

    int getX() {
        return x;
    }
    long getY() {
        return y;
    }
    short getZ() {
        return z;
    }
}

class DungeonTest {
    public static void main(String[] args) {
        DungeonGame d = new DungeonGame();
        System.out.println(d.getX() + d.getY() + d.getZ());
        try {
            FileOutputStream fos = new FileOutputStream("dg.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(d);
            oos.close();

            FileInputStream fis = new FileInputStream("dg.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (DungeonGame) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(d.getX() + d.getY() + d.getZ());
    }
}

```

```

File Edit Window Help Escape
% java DungeonTest
12
8
}

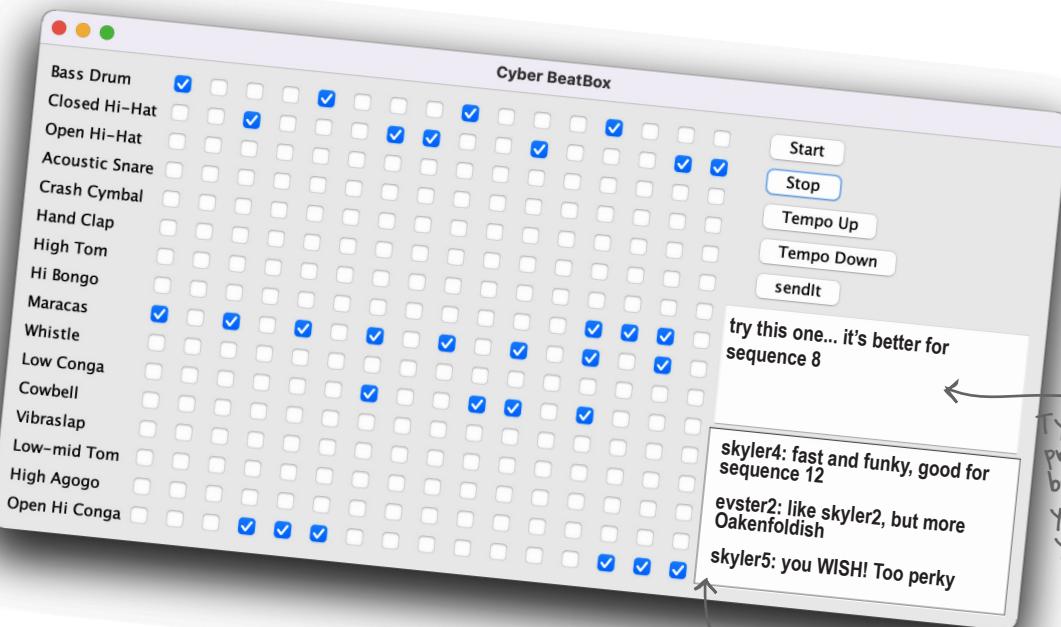
```


Make a Connection



Connect with the outside world. Your Java program can talk to a program on another machine. It's easy. All the low-level networking details are taken care of by the built-in Java libraries. One of Java's big benefits is that sending and receiving data over a network can be just I/O with a slightly different connection at the end of the I/O chain. In this chapter we'll connect to the outside world with *channels*. We'll make *client* channels. We'll make *server* channels. We'll make *clients* and *servers*, and we'll make them talk to each other. And we'll also have to learn how to do more than one thing at once. Before the chapter's done, you'll have a fully functional, multithreaded chat client. Did we just say *multithreaded*? Yes, now you *will* learn the secret of how to talk to Bob while simultaneously listening to Suzy.

Real-time BeatBox chat

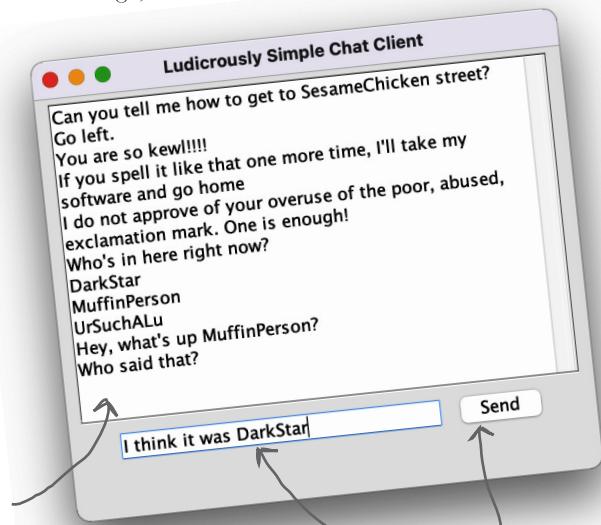


Type a message and press the sendit button to send your message AND your current beat pattern.

Clicking on a received message loads the pattern that went with it.

You're working on a computer game. You and your team are doing the sound design for each part of the game. Using a “chat” version of the BeatBox, your team can collaborate—you can send a beat pattern along with your chat message, and everybody in the BeatBox Chat gets it. So you don’t just get to *read* the other participants’ messages; you get to load and *play* a beat pattern simply by clicking the message in the incoming messages area.

In this chapter we’re going to learn what it takes to make a chat client like this. We’re even going to learn a little about making a chat *server*. We’ll save the full BeatBox Chat for the Code Kitchen, but in this chapter you *will* write a Ludicrously Simple Chat Client and Very Simple Chat Server that send and receive text messages.



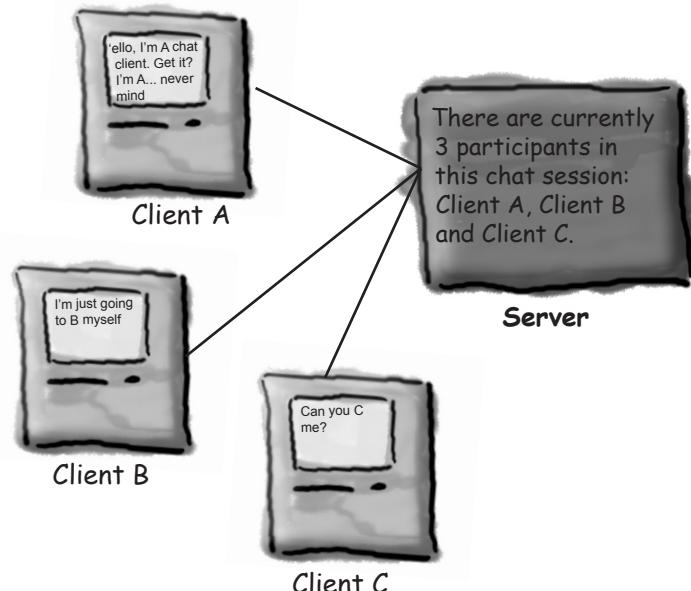
You can have completely authentic, intellectually stimulating chat conversations. Every message is sent to all participants.

Send your message to the server.

Chat program overview

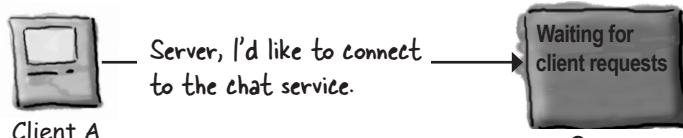
Each Client has to know about the Server.

The Server has to know about ALL the Clients.

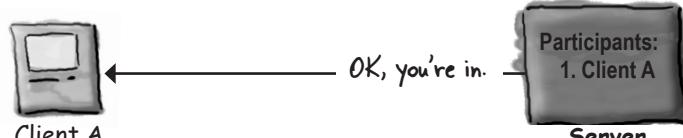


How it works:

- 1 Client connects to the server



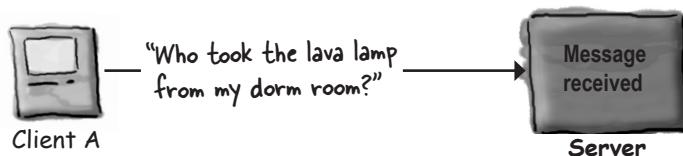
- 2 The server makes a connection and adds the client to the list of participants



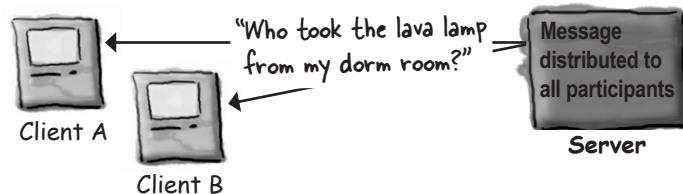
- 3 Another client connects



- 4 Client A sends a message to the chat service



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



Connecting, sending, and receiving

The three things we have to learn to get the client working are:

1. How to establish the initial **connection** between the client and server
2. How to **receive** messages *from* the server
3. How to **send** messages *to* the server

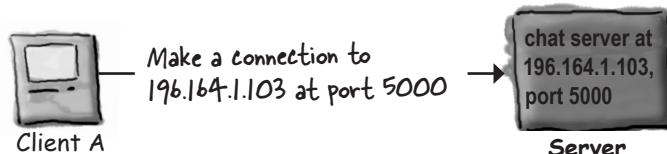
There's a lot of low-level stuff that has to happen for these things to work. But we're lucky, because the Java APIs make it a piece of cake for programmers. You'll see a lot more GUI code than networking and I/O code in this chapter.

And that's not all.

Lurking within the simple chat client is a problem we haven't faced so far in this book: doing two things at the same time. Establishing a connection is a one-time operation (that either works or fails). But after that, a chat participant wants to *send outgoing messages* and **simultaneously** receive incoming messages from the other participants (via the server). Hmm...that one's going to take a little thought, but we'll get there in just a few pages.

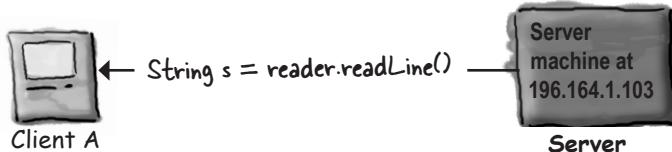
➊ Connect

Client **connects** to the server



➋ Receive

Client **reads** a message from the server



➌ Send

Client **writes** a message to the server



1. Connect

To talk to another machine, we need an object that represents a network connection between two machines. We can open a java.nio.channels.SocketChannel to give us this connection object.

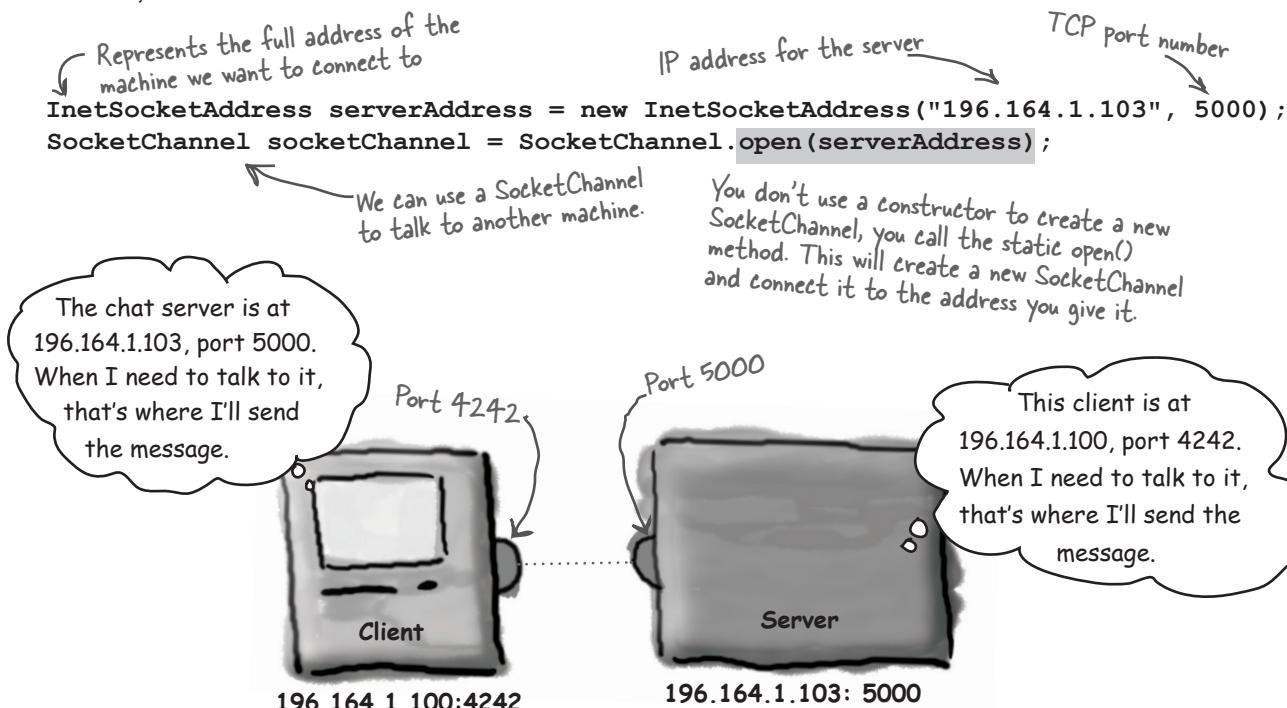
What's a connection? A *relationship* between two machines, where **two pieces of software know about each other**. Most importantly, those two pieces of software know how to *communicate* with each other. In other words, how to send *bits* to each other.

We don't care about the low-level details, thankfully, because they're handled at a much lower place in the “networking stack.” If you don't know what the “networking stack” is, don't worry about it. It's just a way of looking at the layers that information (bits) must travel through to get from a Java program running in a JVM on some OS, to physical hardware (Ethernet cables, for example), and back again on some other machine.

The part that you have to worry about is high-level. You just have to create an object for the server's address and then open a channel to that server. Ready?

To make a connection,
you need to know two
things about the server:
where it is and which
port it's running on.

In other words,
**IP address and TCP
port number**.



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

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

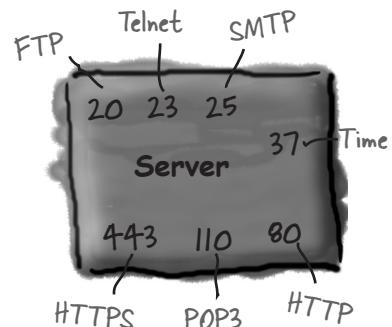
Your internet web (HTTP) server runs on port 80. That's a standard. If you've got a Telnet server, it's running on port 23. FTP? 20. POP3 mail server? 110. SMTP? 25. The Time server sits at 37. Think of port numbers as unique identifiers. They represent a logical connection to a particular piece of software running on the server. That's it. You can't spin your hardware box around and find a TCP port. For one thing, you have 65,536 of them on a server (0–65535). So they obviously don't represent a place to plug in physical devices. They're just a number representing an application.

Without port numbers, the server would have no way of knowing which application a client wanted to connect to. And since each application might have its own unique protocol, think of the trouble you'd have without these identifiers. What if your web browser, for example, landed at the POP3 mail server instead of the HTTP server? The mail server won't know how to parse an HTTP request! And even if it did, the POP3 server doesn't know anything about servicing the HTTP request.

When you write a server program, you'll include code that tells the program which port number you want it to run on (you'll see how to do this in Java a little later in this chapter). In the Chat program we're writing in this chapter, we picked 5000. Just because we wanted to. And because it met the criteria that it be a number between 1024 and 65535. Why 1024? Because 0 through 1023 are reserved for the well-known services like the ones we just talked about.

And if you're writing services (server programs) to run on a company network, you should check with the sysadmins to find out which ports are already taken. Your sysadmins might tell you, for example, that you can't use any port number below, say, 3000. In any case, if you value your limbs, you won't assign port numbers with abandon. Unless it's your *home* network. In which case you just have to check with your *kids*.

Well-known TCP port numbers
for common server applications:



A server can have up to 65,536 different server apps running, one per port.

The TCP port numbers from 0 to 1023 are reserved for well-known services. Don't use them for your own server programs!*

The chat server we're writing uses port 5000. We just picked a number between 1024 and 65535.

*Well, you *might* be able to use one of these, but the sysadmin where you work will write you a strongly worded message and CC your boss.

there are no Dumb Questions

Q: How do you know the port number of the server program you want to talk to?

A: That depends on whether the program is one of the well-known services. If you're trying to connect to a well-known service, like the ones on the opposite page (HTTP, SMTP, FTP, etc.), you can look these up on the internet (Google "Well-Known TCP Port"). Or ask your friendly neighborhood sysadmin.

But if the program isn't one of the well-known services, you need to find out from whoever is deploying the service. Ask them. Typically, if someone writes a network service and wants others to write clients for it, they'll publish the IP address, port number, and protocol for the service. For example, if you want to write a client for a GO game server, you can visit one of the GO server sites and find information about how to write a client for that particular server.

Q: Can there ever be more than one program running on a single port? In other words, can two applications on the same server have the same port number?

A: No! If you try to bind a program to a port that is already in use, you'll get a `BindException`. To *bind* a program to a port just means starting up a server application and telling it to run on a particular port. Again, you'll learn more about this when we get to the server part of this chapter.

IP address is the mall



Port number is the specific store in the mall



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

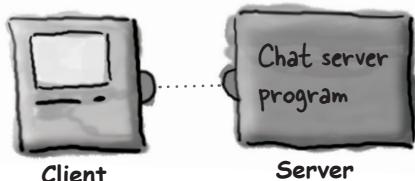
Port number is like naming a specific store, say, "Bob's CD Shop"



Brain Barbell

OK, you got a connection. The client and the server know the IP address and TCP port number for each other. Now what? How do you communicate over that connection? In other words, how do you move bits from one to the other? Imagine the kinds of messages your chat client needs to send and receive.

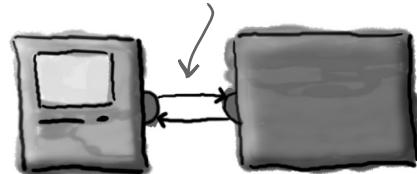
How do these two actually talk to each other?



2. Receive

To communicate over a remote connection, you can use regular old I/O streams, just like we used in the previous chapter. One of the coolest features in Java is that most of your I/O work won't care what your high-level chain stream is actually connected to. In other words, you can use a **BufferedReader** just like you did when you were reading from a file; the difference is that the underlying connection stream is connected to a *Channel* rather than a *File*!

Channels between the client and server



Reading from the network with BufferedReader

1 Make a connection to the server

```
SocketAddress serverAddr = new InetSocketAddress("127.0.0.1", 5000);
```

```
SocketChannel socketChannel = SocketChannel.open(serverAddr);
```

You need to open a SocketChannel
that connects to this address.

127.0.0.1 is the IP address for "localhost," in other words, the one this code is running on. You can use this when you're testing your client and server on a single, stand-alone machine. You could also use "localhost" here instead.

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

2 Create or get a Reader from the connection

```
Reader reader = Channels.newReader(socketChannel, StandardCharsets.UTF_8);
```

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

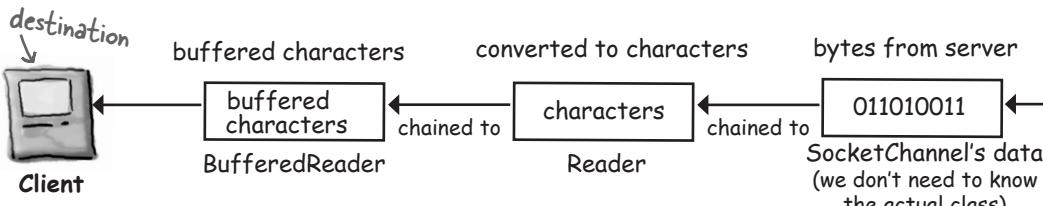
You can use the static helper methods on the Channels class to create a Reader from your SocketChannel.

You need to say which Charset to use for reading the values from the network. UTF 8 is a common one to use.

Chain the BufferedReader to the Reader (which is from our SocketChannel).

3 Make a BufferedReader and read!

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



3. Send

In the previous chapter, we used BufferedWriter. We have a choice here, but when you're writing one String at a time, **PrintWriter** is a standard choice. And you'll recognize the two key methods in PrintWriter, print() and println(). Just like good ol' System.out.

Writing to the network with PrintWriter

1 Make a connection to the server

```
SocketAddress serverAddr = new InetSocketAddress("127.0.0.1", 5000);
SocketChannel socketChannel = SocketChannel.open(serverAddr);
```

This part's the same as it was on the last page—to write to the server, we still have to connect to it.

2 Create or get a Writer from the connection

```
Writer writer = Channels.newWriter(socketChannel, StandardCharsets.UTF_8);
```

Writer acts as a bridge between character data and the bytes to be written to the Channel.

The Channels class contains utility methods to create a Writer.

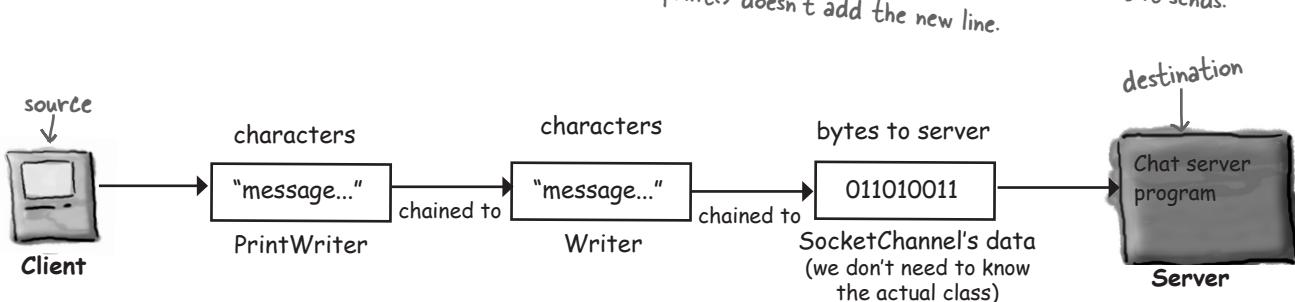
You need to say which Charset to use to write Strings. You should use the same one for reading as for writing!

3 Make a PrintWriter and write (print) something

```
PrintWriter printWriter = new PrintWriter(writer);
```

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

By chaining a PrintWriter to the Channel's Writer, we can write Strings to the Channel, which will be sent over the connection.



There's more than one way to make a connection

If you look at real life code that talks to a remote machine, you'll probably see a number of different ways to make connections and to read from and write to a remote computer.

Which approach you use depends on a number of things, including (but not limited to) the version of Java you're using and the needs of the application (for example, how many clients connect at once, the size of messages sent, frequency or message, etc). One of the simplest approaches is to use a **java.net.Socket** instead of a Channel.

Using a Socket

You can get an *InputStream* or *OutputStream* from a Socket, and read and write from it in a very similar way to what we've already seen.



```
Instead of using an InetSocketAddress and
opening a SocketChannel, you can create a
Socket with the host and port number.
```

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

To read from the Socket, we need to get an *InputStream* from the Socket.

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

Reader code is exactly the same as we've already seen.

```
BufferedReader reader = new BufferedReader(in);}
```

```
String message = reader.readLine();
```

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

To write to the socket, we need to get an *OutputStream* from the Socket, which we can chain to the *PrintWriter*.

```
writer.println("message to send");}
writer.print("another message");}
```

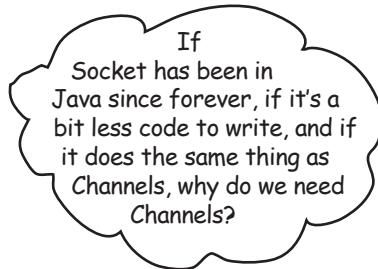
Writer code is exactly the same as we've already seen.

The **java.net.Socket** class is available in all versions of Java.

It supports simple network I/O via the I/O streams we've already used for file I/O.



o o



**As we've become an increasingly connected world,
Java has evolved to offer more ways to work with
remote machines.**

Remember that Channels are in the `java.nio.channels` package? The `java.nio` package (NIO) was introduced in Java 1.4, and there were more changes and additions made (sometimes called NIO.2) in Java 7.

There are ways to use Channels and NIO to get better performance when you're working with lots of network connections, or lots of data coming over those connections.

In this chapter, we're using Channels to provide the same very basic connection functionality we could get from

Sockets. However, if our application needed to work well with a very busy network connection (or lots of them!), we could configure our Channels differently and use them to their full potential, and our program would cope better with a high network I/O load.

We've chosen to teach you the **simplest way to get started** with network I/O using *Channels* so that if you need to "level up" to working with more advanced features, it shouldn't be such a big step.

If you do want to learn more about NIO, read *Java NIO* by Ron Hitchens and *Java I/O, NIO and NIO.2* by Jeff Friesen.



Channels support advanced networking features that you don't need for these exercises.

Channels can support nonblocking I/O, reading and writing via ByteBuffers, and asynchronous I/O. We're not going to show you any of this!

But at least now you have some keywords to put into your search engine when you want to know more.

The DailyAdviceClient

Before we start building the Chat app, let's start with something a little smaller. The Advice Guy is a server program that offers up practical, inspirational tips to get you through those long days of coding.

We're building a client for The Advice Guy program, which pulls a message from the server each time it connects.

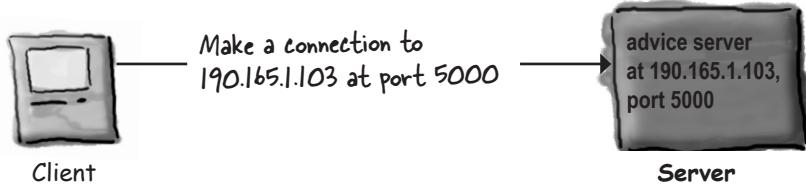
What are you waiting for? Who *knows* what opportunities you've missed without this app.



The Advice Guy

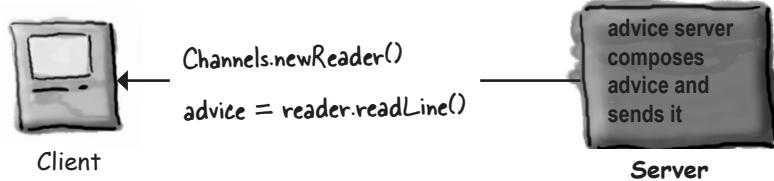
➊ Connect

Client connects to the server



➋ Read

Client gets a Reader for the Channel, and reads a message from the server



DailyAdviceClient code

This program makes a SocketChannel, makes a BufferedReader (with the help of the channel's Reader), and reads a single line from the server application (whatever is running at port 5000).

```

import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.Channels;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;

public class DailyAdviceClient {
    public void go() {
        InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000);
        try (SocketChannel socketChannel = SocketChannel.open(serverAddress)) {
            Reader channelReader = Channels.newReader(socketChannel, StandardCharsets.UTF_8);
            BufferedReader reader = new BufferedReader(channelReader);
            String advice = reader.readLine();
            System.out.println("Today you should: " + advice);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

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

```

This uses try-with-resources to automatically close the SocketChannel when the code is complete.

Define the server address as being port 5000, on the same host this code is running on (the "localhost").

Create a SocketChannel by opening one for the server's address.

Create a Reader that reads from the SocketChannel.

Chain a BufferedReader to the Reader from the SocketChannel.

This readLine() is EXACTLY the same as if you were using a BufferedReader chained to a FILE.. In other words, by the time you call a BufferedReader method, the reader doesn't know or care where the characters came from.

exercise: sharpen your pencil



Sharpen your pencil

Test your memory of the classes for reading and writing from a SocketChannel. Try not to look at the opposite page!

→ Yours to solve.

To **read** text from a SocketChannel:



Client

Write/draw in the chain of classes the client uses to read from the server.



Server

To **send** text to a SocketChannel:



Client

Write/draw in the chain of classes the client uses to send something to the server.



Server



Sharpen your pencil

Fill in the blanks:

→ Yours to solve.

What two pieces of information does the client need in order to make a connection with a server?

Which TCP port numbers are reserved for “well-known services” like HTTP and FTP? _____

TRUE or FALSE: The range of valid TCP port numbers can be represented by a short primitive.

Writing a simple server application

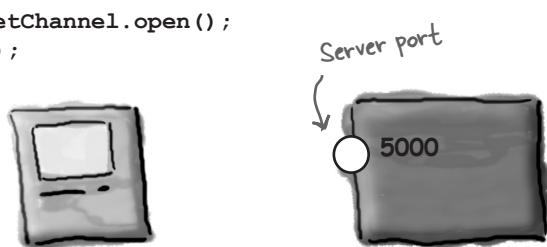
So what's it take to write a server application? Just a couple of Channels. Yes, a couple as in two. A ServerSocketChannel, which waits for client requests (when a client connects) and a SocketChannel to use for communication with the client. If there's more than one client, we'll need more than one channel, but we'll get to that later.

How it works:

- 1 Server application makes a ServerSocketChannel and binds it to a specific port

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.bind(new InetSocketAddress(5000));
```

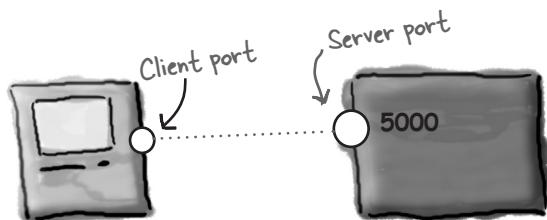
This starts the server application listening for client requests coming in for port 5000.



- 2 Client makes a SocketChannel connected to the server application

```
SocketChannel svr = SocketChannel.open(new InetSocketAddress("190.165.1.103", 5000));
```

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

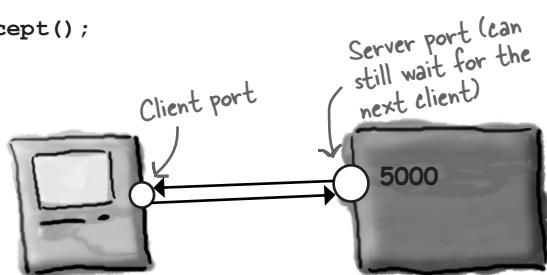


- 3 Server makes a new SocketChannel to communicate with this client

```
SocketChannel clientChannel = serverChannel.accept();
```

The accept() method blocks (just sits there) while it's waiting for a client connection. When a client finally connects, the method returns a SocketChannel that knows how to communicate with this client.

The ServerSocketChannel can go back to waiting for other clients. The server has just one ServerSocketChannel, and a SocketChannel per client.



DailyAdviceServer code

This program makes a ServerSocketChannel and waits for client requests. When it gets a client request (i.e., client created a new SocketChannel to this server), the server makes a new SocketChannel to that client. The server makes a PrintWriter (using a Writer created from the SocketChannel) and sends a message to the client.

```

import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.Random;

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

    private final Random random = new Random();

    public void go() {
        try (ServerSocketChannel serverChannel = ServerSocketChannel.open()) {
            serverChannel.bind(new InetSocketAddress(5000));      Daily advice comes from this array.

            while (serverChannel.isOpen()) {                      ServerSocketChannel makes this server
                SocketChannel clientChannel = serverChannel.accept();  application "listen" for client requests on the
                PrintWriter writer = new PrintWriter(Channels.newOutputStream(clientChannel));   port it's bound to.

                String advice = getAdvice();
                writer.println(advice);                                You have to bind the ServerSocketChannel to
                writer.close();                                         the port you want to run the application on.

                System.out.println(advice);                            The accept method blocks (just sits there) until a
                }                                                       request comes in, and then the method returns a
            }                                                       SocketChannel for communicating with the client.

            catch (IOException ex) {                                Create an output stream for the client's
                ex.printStackTrace();                               channel, and wrap it in a PrintWriter. You can
            }                                                       use newOutputStream or newWriter here.

            Print in the server console, so
            we can see what's happening.                         Send the client a String advice message.

        }
    }

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

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

```

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

The accept method blocks (just sits there) until a request comes in, and then the method returns a SocketChannel for communicating with the client.

Create an output stream for the client's channel, and wrap it in a PrintWriter. You can use newOutputStream or newWriter here.

Close the writer, which will also close the client SocketChannel.

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

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



Brain Barbell



Yes, that's right, **the server can't accept a request from a client until it has finished with the current client**. At which point, it starts the next iteration of the infinite loop, sitting, waiting, at the `accept()` call until a new request comes in, at which point it makes a `SocketChannel` to send data to the new client and starts the process over again.

To get this to work with multiple clients *at the same time*, we need to use separate threads.

We'd give each new client's `SocketChannel` to a new thread, and each thread can work independently.

We're just about to learn how to do that!

BULLET POINTS

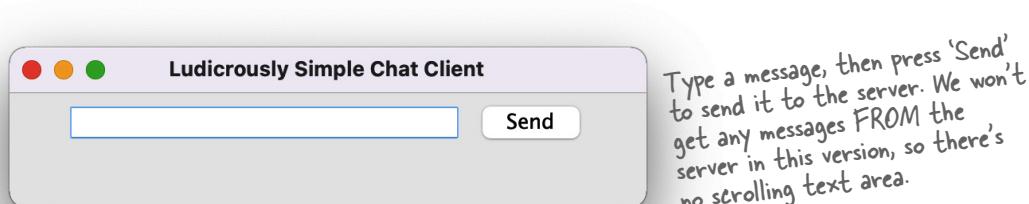
- Client and server applications communicate using Channels.
- A Channel represents a connection between two applications that may (or may not) be running on two different physical machines.
- A client must know the IP address (or host name) and TCP port number of the server application.
- A TCP port is a 16-bit unsigned number assigned to a specific server application. TCP port numbers allow different server applications to run on the same machine; clients connect to a specific application using its port number.
- The port numbers from 0 through 1023 are reserved for "well-known services" including HTTP, FTP, SMTP, etc.
- A client connects to a server by opening a `SocketChannel`:
`SocketChannel.open(
 new InetSocketAddress("127.0.0.1", 4200))`
- Once connected, a client can create readers (to read data from the server) and writers (to send data to the server) for the channel:
`Reader reader = Channels.newReader(sockCh,
 StandardCharsets.UTF_8);`
`Writer writer = Channels.newWriter(sockCh,
 StandardCharsets.UTF_8);`
- To read text data from the server, create a `BufferedReader`, chained to the Reader. The Reader is a "bridge" that takes in bytes and converts them to text (character) data. It's used primarily to act as the middle chain between the high-level `BufferedReader` and the low-level connection.
- To write text data to the server, create a `PrintWriter` chained to the Writer. Call the `print()` or `println()` methods to send Strings to the server.
- Servers use a `ServerSocketChannel` that waits for client requests on a particular port number.
- When a `ServerSocketChannel` gets a request, it "accepts" the request by making a `SocketChannel` for the client.

Writing a Chat Client

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

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

Version One: send-only



Code outline

Here's an outline of the main functionality the chat client needs to provide. The full code is on the next page.

```
public class SimpleChatClientA {
    private JTextField outgoing;
    private PrintWriter writer;

    public void go() {
        // call the setUpNetworking() method
        // make gui and register a listener with the send button
    }

    private void setUpNetworking() {
        // open a SocketChannel to the server
        // make a PrintWriter and assign to writer instance variable
    }

    private void sendMessage() {
        // get the text from the text field and
        // send it to the server using the writer (a PrintWriter)
    }
}
```

```

import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import static java.nio.charset.StandardCharsets.UTF_8; This is a static import; we looked at static imports in Chapter 10.

public class SimpleChatClientA {
    private JTextField outgoing;
    private PrintWriter writer;

    public void go() { Call the method that will
        setUpNetworking(); connect to the server.

        outgoing = new JTextField(20);

        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(e -> sendMessage());
    }

    JPanel mainPanel = new JPanel();
    mainPanel.add(outgoing);
    mainPanel.add(sendButton);
    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(400, 100);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}

private void setUpNetworking() { We're using localhost so you
    try { can test the client and
        InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000); server on one machine.

        SocketChannel socketChannel = SocketChannel.open(serverAddress); Open a SocketChannel
        writer = new PrintWriter(Channels.newWriter(socketChannel, UTF_8)); that connects to the
        System.out.println("Networking established."); server.

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

private void sendMessage() { Now we actually do the writing. Remember,
    writer.println(outgoing.getText()); } the writer is chained to the writer from the
    writer.flush(); SocketChannel, so whenever we do a println(),
    outgoing.setText(""); it goes over the network to the server!
    outgoing.requestFocus();

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

```

Imports for writing (java.io), network connections (java.nio.channels) and the GUI stuff (awt and swing)

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

This is where we make the PrintWriter from a writer that writes to the SocketChannel.

If you want to try this now, type in the Ready-bake chat server code listed on the next page.

First, start the server in one terminal. Next, use another terminal to start this client.



Ready-Bake Code

The really, really simple Chat Server

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

After you finish this chapter, you should be able to annotate this code yourself. You'll understand it much better if *you* work out what's happening than if we explained it to you. Then again, this is Ready-Bake Code, so you really don't have to understand it at all. It's here just to support the two versions of the Chat Client.

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

```

import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.*;
import java.util.concurrent.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class SimpleChatServer {
    private final List<PrintWriter> clientWriters = new ArrayList<>();

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

    public void go() {
        ExecutorService threadPool = Executors.newCachedThreadPool();
        try {
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.bind(new InetSocketAddress(5000));

            while (serverSocketChannel.isOpen()) {
                SocketChannel clientSocket = serverSocketChannel.accept();
                PrintWriter writer = new PrintWriter(Channels.newWriter(clientSocket, UTF_8));
                clientWriters.add(writer);
                threadPool.submit(new ClientHandler(clientSocket));
                System.out.println("got a connection");
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

```

private void tellEveryone(String message) {
    for (PrintWriter writer : clientWriters) {
        writer.println(message);
        writer.flush();
    }
}

public class ClientHandler implements Runnable {
    BufferedReader reader;
    SocketChannel socket;

    public ClientHandler(SocketChannel clientSocket) {
        socket = clientSocket;
        reader = new BufferedReader(Channels.newReader(socket, UTF_8));
    }

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

```

```

File Edit Window Help TakesTwoToTango
%java SimpleChatServer
got a connection
read Nice to meet you

```

Runs in the background

```

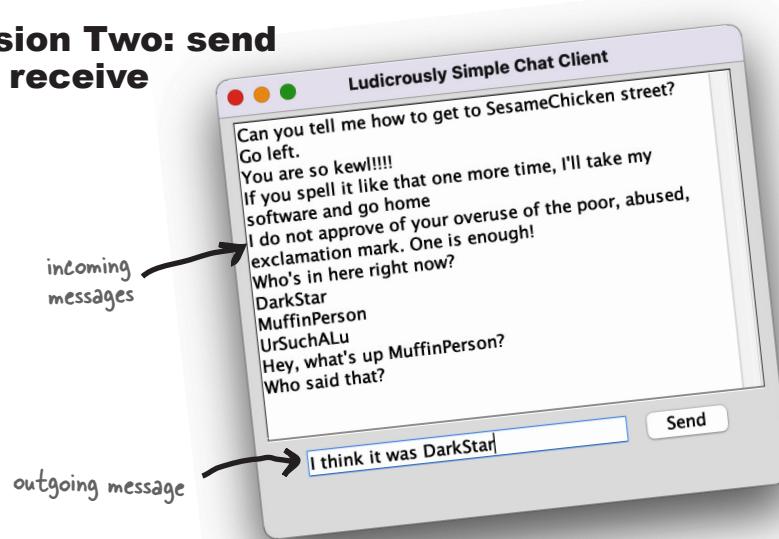
File Edit Window Help MayIHaveThisDance?
%java SimpleChatClientA
Networking established. Client
running at: /127.0.0.1:57531

```

Connects to the server and launches GUI



Version Two: send and receive



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

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

Should be easy; when you set up the networking, make a Reader as well. Then read messages using `readLine`.

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

Think about that. What are the options?

① Option One: Read something in from the server each time the user sends a message.

Pros: Doable, very easy.

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

② Option Two: Poll the server every 20 seconds.

Pros: It's doable, and it fixes the lurker problem.

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

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

Pros: Most efficient, best usability.

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



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

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

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

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

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

Well, not quite. Each new Java thread is not actually a separate process running on the OS. But it almost *feels* as though it is.

We're going to take a break from the chat application for a bit while we explore how this works. Then we'll come back and add it to our chat client at the end of the chapter.

Multithreading in Java

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

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

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

Except for one problem.

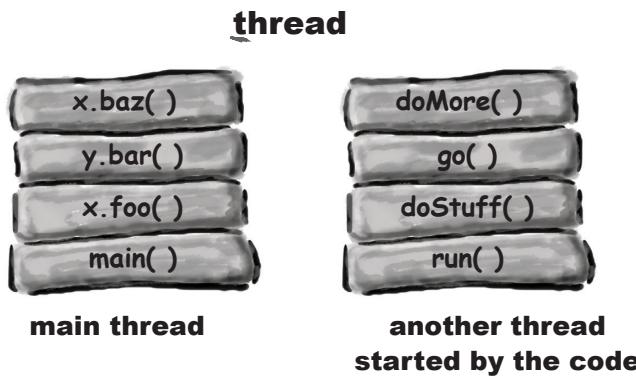
That thread doesn't actually *do* anything, so the thread "dies" virtually the instant it's born. When a thread dies, its new stack disappears again. End of story.

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

Multiple threading in Java means we have to look at both the *thread* and the *job* that's *run* by the thread. In fact, **there's more than one way to run multiple jobs in Java**, not just with the Thread *class* in the java.lang package. (Remember, java.lang is the package you get imported for free, implicitly, and it's where the classes most fundamental to the language live, including String and System.)

Java has multiple threads but only one Thread class

We can talk about *thread* with a lowercase “t” and **Thread** with a capital “T.” When you see *thread*, we’re talking about a separate thread of execution. In other words, a separate call stack. When you see **Thread**, think of the Java naming convention. What, in Java, starts with a capital letter? Classes and interfaces. In this case, **Thread** is a class in the `java.lang` package. A **Thread** object represents a *thread of execution*. In older versions of Java, you always had to create an instance of class **Thread** each time you wanted to start up a new *thread* of execution. Java has evolved over time, and now using the **Thread** class directly is not the only way. We’ll see this in more detail as we go through the rest of the chapter.



A *thread* (lowercase “t”) is a separate thread of execution. That means a separate call stack. Every Java application starts up a main thread—the thread that puts the `main()` method on the bottom of the stack. The JVM is responsible for starting the main thread (and other threads, as it chooses, including the garbage collection thread). As a programmer, you can write code to start other threads of your own.

A **thread** is a separate “thread of execution,” a separate call stack.

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

Using the **Thread** class is not the only way to do multithreading in Java.

Thread

Thread
<code>void join()</code>
<code>void start()</code>

`static void sleep()`

`java.lang.Thread` class

Thread (capital “T”) is a class that represents a thread of execution. It has methods for starting a thread, joining one thread with another, putting a thread to sleep, and more.

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

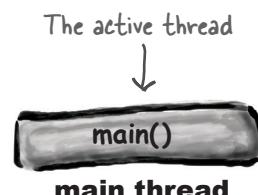
With more than one call stack, you can have multiple things happen at the same time. If you're running on a multiprocessor system (like most modern computers and phones), you can actually do more than one thing at a time. With Java threads, even if you're not running on a multiprocessor system or if you're running more processes than available cores, it can *appear* that you're doing all these things simultaneously. In other words, execution can move back and forth between stacks so rapidly that you feel as though all stacks are executing at the same time. Remember, Java is just a process running on your underlying OS. So first, Java *itself* has to be "the currently executing process" on the OS. But once Java gets its turn to execute, exactly *what* does the JVM *run*? Which bytecodes execute? Whatever is on the top of the currently running stack! And in 100 milliseconds, the currently executing code might switch to a *different* method on a *different* stack.

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

It might look something like this:

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

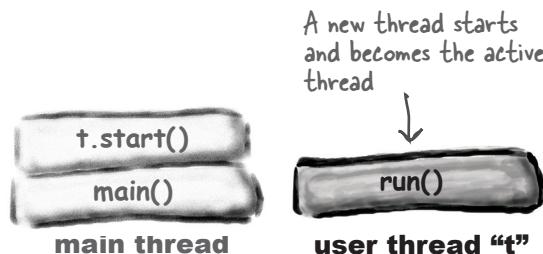
```
public static void main(String[] args) {  
    ...  
}
```



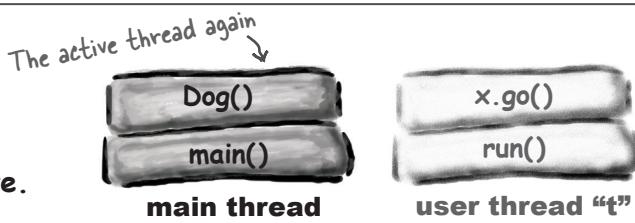
- 2** `main()` starts a new thread. The main thread may be temporarily frozen while the new thread starts running.

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

You'll learn what this means in just a moment...



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



To create a new call stack you need a job to run



To start a new call stack the thread needs a job—a job the thread will run when it's started. That job is actually the first method that goes on the new thread's stack, and it must always be a method that looks like this:

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

How does the thread know which method to put at the bottom of the stack? Because Runnable defines a contract. Because Runnable is an interface. A thread's job can be defined in any class that implements the Runnable interface, or a lambda expression that is the right shape for the run method.

Once you have a Runnable class or lambda expression, you can tell the JVM to run this code in a separate thread; you're giving the thread its job.

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

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

The Runnable interface defines only one method, public void run(). Since it has only a single method, it's a SAM type, a Functional Interface, and you can use a lambda instead of creating a whole class that implements Runnable if you want.

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

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

```
public class MyRunnable implements Runnable {
```

```
    public void run() {
        go();
    }
```

```
    public void go() {
        doMore();
    }
```

```
    public void doMore() {
        System.out.println(Thread.currentThread().getName() +
            ": top o' the stack");
        Thread.dumpStack();
    }
```

*dumpStack will output the current call stack,
just like an Exceptions stack trace. Using it here
only use this for debugging (it might slow real
code down).*

We'll see the
stack for this
thread on the
next page. Yes,
we've gone
straight to "2,"
you should see
why over the
page...

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

This Runnable doesn't really need three
tiny methods, which all call each other
like this; we're using it to demonstrate
what the call stack running this code
looks like.

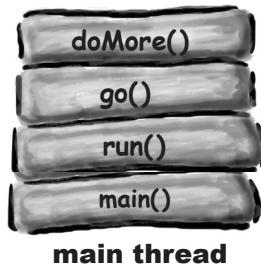
How NOT to run the Runnable

It may be tempting to create a new instance of the Runnable and call the run method,
but that's **not enough to create a new call stack**.

```
class RunTester {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        runnable.run();
        System.out.println(Thread.currentThread().getName() +
            ": back in main");
        Thread.dumpStack();
    }
}
```

This will NOT do what we want!

The `run()` method was called directly from
inside the `main()` method, so it's part of
the call stack of the main thread.



How we used to launch a new thread

The simplest way to launch a new thread is with the Thread class that we mentioned earlier. This method has been around in Java since the very beginning, but **it is no longer the recommended approach to use**. We're showing it here because a) it's simple, and b) you'll see it in the Real World. We will talk later about why it might not be the best approach.

The diagram illustrates the creation of a new thread and its call stack.

Code Snippet:

```

class ThreadTester {
    public static void main(String[] args) {
        Runnable threadJob = new MyRunnable();
        Thread myThread = new Thread(threadJob);
        myThread.start();
        System.out.println(
            Thread.currentThread().getName() +
            ": back in main");
        Thread.dumpStack();
    }
}

```

Annotations:

- ①** A brace groups the line `myThread.start();` and the block containing `System.out.println` and `Thread.dumpStack()`. A callout notes: "This thread's stack is below."
- ②** An annotation points to the line `new Thread(threadJob);` with the text: "Pass the new Runnable instance into the new Thread constructor. This tells the thread what job to run. In other words, the Runnable's run() method will be the first method that the new thread will run."
- A callout for the `myThread.start();` line states: "You won't get a new thread of execution until you call start() on the Thread instance. A thread that's just a Thread instance, like any other object, but it won't have any real 'threadness.'"
- main thread:** A call stack diagram showing the sequence: `main()`.
- new thread:** A call stack diagram showing the sequence: `doMore()`, `go()`, `run()`.
- Call stacks:**
 - Call stack for the "main" thread of the application (started by the "public static void main" method).
 - Call stack for the new thread we started with the `MyRunnable` job.
- Terminal Output:**

```

File Edit Window Help Duo
% java ThreadTester

main: back in main
Thread-0: top o' the stack
java.lang.Exception: Stack trace
    at java.base/java.lang.Thread.dumpStack(Thread.java:1383)
    at ThreadTester.main(MyRunnable.java:38)

java.lang.Exception: Stack trace
    at java.base/java.lang.Thread.dumpStack(Thread.java:1383)
    at MyRunnable.doMore(MyRunnable.java:15)
    at MyRunnable.go(MyRunnable.java:10)
    at MyRunnable.run(MyRunnable.java:6)
    at java.base/java.lang.Thread.run(Thread.java:829)

```
- Annotations on terminal output:**
 - An arrow from the text "dumpStack() called from doMore() in MyRunnable" points to the line `at MyRunnable.doMore(MyRunnable.java:15)`.
 - An arrow from the text "dumpStack() called from main() method" points to the line `at ThreadTester.main(MyRunnable.java:38)`.
 - An arrow from the text "Note the main method is NOT the bottom of the call stack of the Runnable." points to the line `at java.base/java.lang.Thread.run(Thread.java:829)`.

A better alternative: don't manage the Threads at all

Creating and starting a new Thread gives you a lot of control over that Thread, but the downside is you *have* to control it. You have to keep track of all the Threads and make sure they're shut down at the end. Wouldn't it be better to have something else that starts, stops, and even reuses the Threads so you don't have to?

Allow us to introduce an interface in `java.util.concurrent.ExecutorService`. Implementations of this interface will *execute* jobs (Runnables). Behind the scenes the ExecutorService will create, reuse, and kill threads in order to run these jobs.

The `java.util.concurrent.Executors` class has *factory methods* to create the ExecutorService instances we'll need.

Executors have been around since Java 5 and so should be available to you even if you're working with quite an old version of Java. There's no real need to use Thread directly at all these days.

Running one job

For the simple cases we're going to get started with, we'll want to run only one job in addition to our main class. There's a *single thread executor* that we can use to do this.

```
class ExecutorTester {
    public static void main(String[] args) {
        Runnable job = new MyRunnable();
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(job);
        System.out.println(Thread.currentThread().getName() +
            ": back in main");
        Thread.dumpStack();
        executor.shutdown();
    }
}
```

Tell the ExecutorService to run the job. It will take care of starting a new thread for the job if it needs to.

Remember to shut down the ExecutorService when you've finished with it. If you don't shut it down, the program will hang around waiting for more jobs.

Instead of creating a Thread instance, use a method on the Executors class to create an ExecutorService.

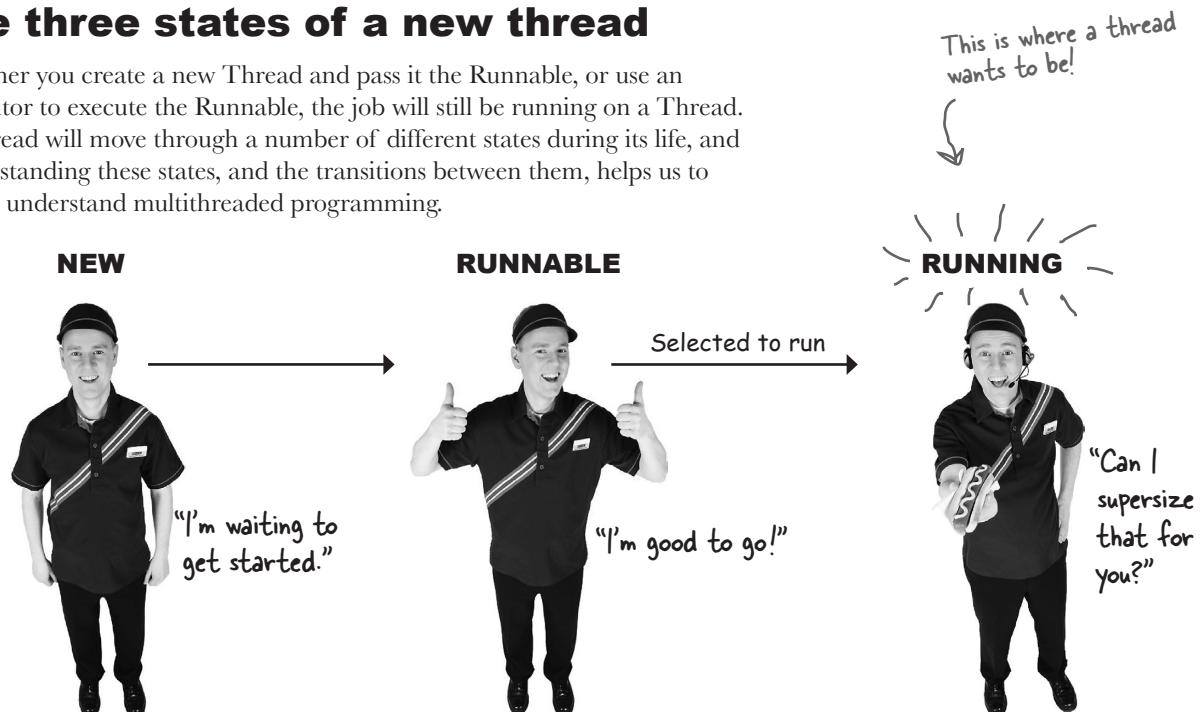
In our case, we only want to start a single job, so it's logical to create a single thread executor.

We'll come back to the Executors factory methods later, and we'll see why it might be better to use ExecutorServices rather than managing the Thread itself.

Factory methods return exactly the implementation of an interface that we need. We don't need to know the concrete classes or how to create them.

The three states of a new thread

Whether you create a new Thread and pass it the Runnable, or use an Executor to execute the Runnable, the job will still be running on a Thread. A Thread will move through a number of different states during its life, and understanding these states, and the transitions between them, helps us to better understand multithreaded programming.



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

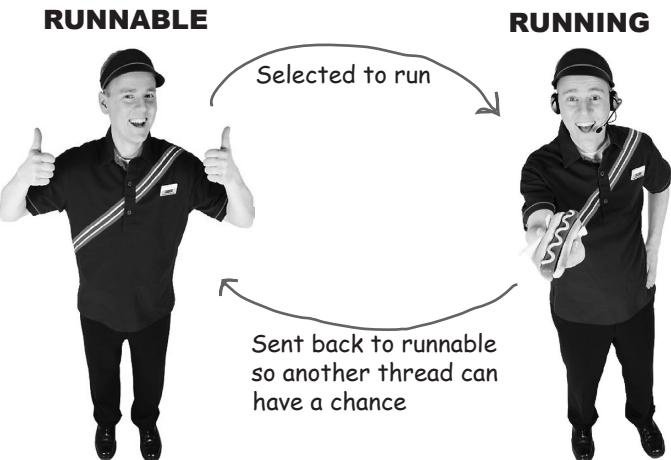
When you start the thread, it moves into the runnable state. This means the thread is ready to run and just waiting for its Big Chance to be selected for execution. At this point, there is a new call stack for this thread.

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

But there's more. Once the thread becomes runnable, it can move back and forth between runnable, running, and an additional state: temporarily not runnable.

Typical runnable/running loop

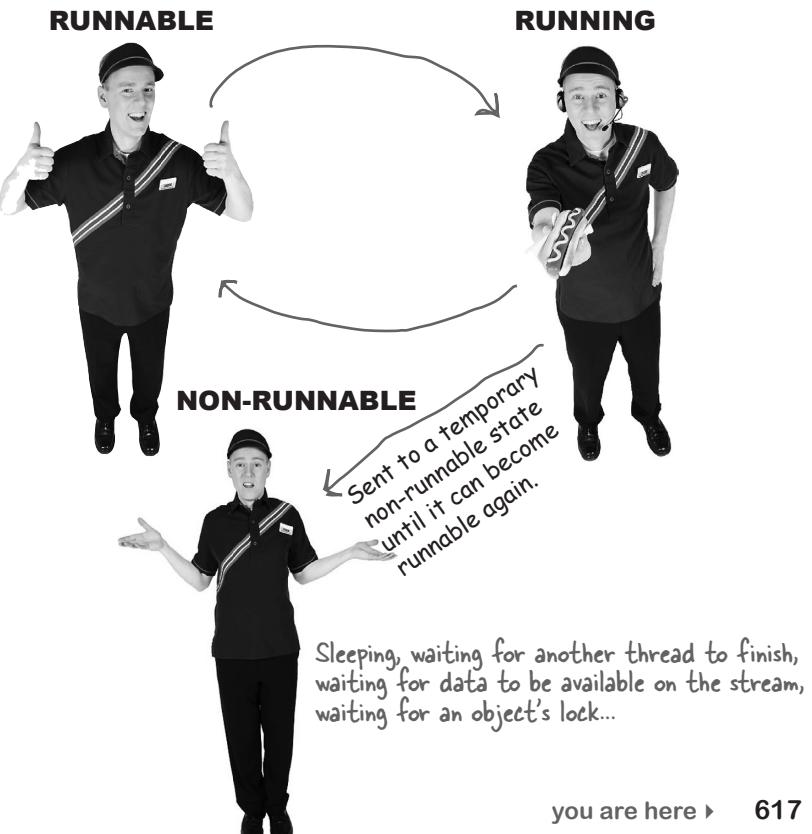
Typically, a thread moves back and forth between runnable and running, as the JVM thread scheduler selects a thread to run and then kicks it back out so another thread gets a chance.



A thread can be made temporarily not-runnable

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

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



The thread scheduler

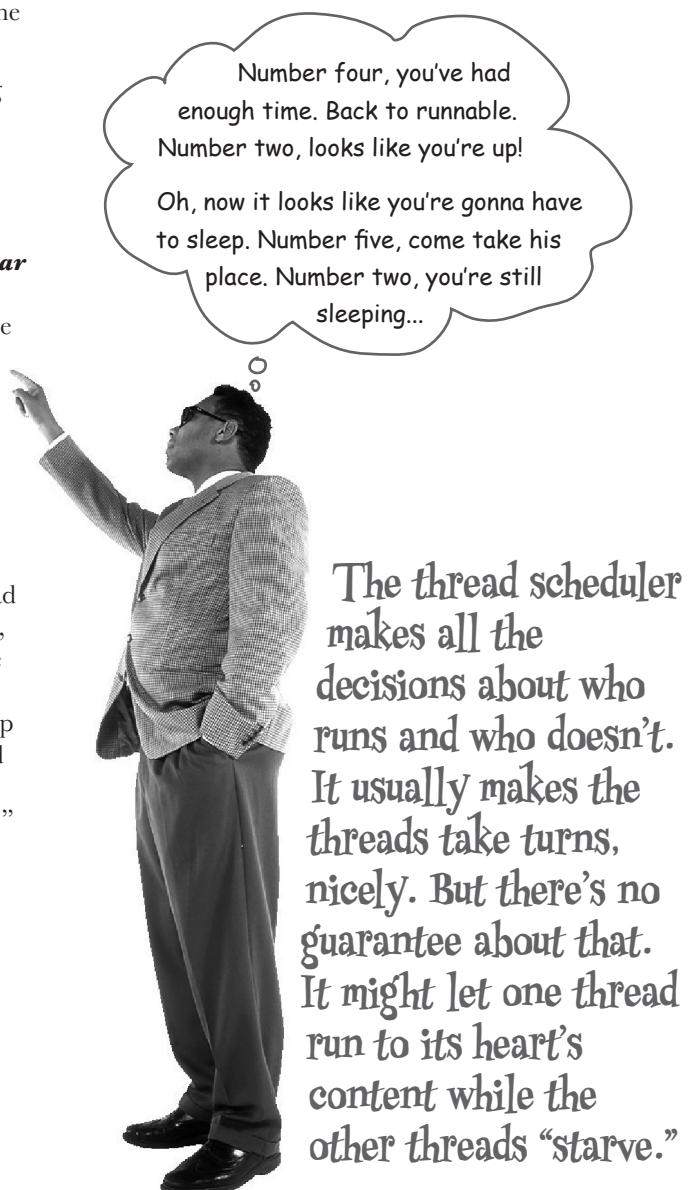
The thread scheduler makes all the decisions about who moves from runnable to running, and about when (and under what circumstances) a thread leaves the running state. The scheduler decides who runs, for how long, and where the threads go when it decides to kick them out of the currently running state.

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

The bottom line is this: ***do not base your program's correctness on the scheduler working in a particular way!*** The scheduler implementations are different for different JVMs, and even running the same program on the same machine can give you different results. One of the worst mistakes new Java programmers make is to test their multithreaded program on a single machine, and assume the thread scheduler will always work that way, regardless of where the program runs.

So what does this mean for write-once-run-anywhere? It means that to write platform-independent Java code, your multithreaded program must work no matter *how* the thread scheduler behaves. That means you can't be dependent on, for example, the scheduler making sure all the threads take nice, perfectly fair, and equal turns at the running state.

Although highly unlikely today, your program might end up running on a JVM with a scheduler that says, “OK, thread five, you're up, and as far as I'm concerned, you can stay here until you're done, when your `run()` method completes.”



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

Running this code on one machine:

```
class ExecutorTestDrive {
    public static void main (String[] args) {
        ExecutorService executor =
            Executors.newSingleThreadExecutor();

        executor.execute(() ->
            System.out.println("top o' the stack"));

        System.out.println("back in main");
        executor.shutdown();
    }
}
```

Runnable is a Functional Interface and can be represented as a lambda expression. Our job is just a single line of code, so a lambda expression makes sense here.

It doesn't matter if you run this using an ExecutorService, like the code above, or with Threads directly, like the code below; both show the same symptoms.

```
class ThreadTestDrive {
    public static void main (String[] args) {
        Thread myThread = new Thread(() ->
            System.out.println("top o' the stack"));
        myThread.start();
        System.out.println("back in main");
    }
}
```

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

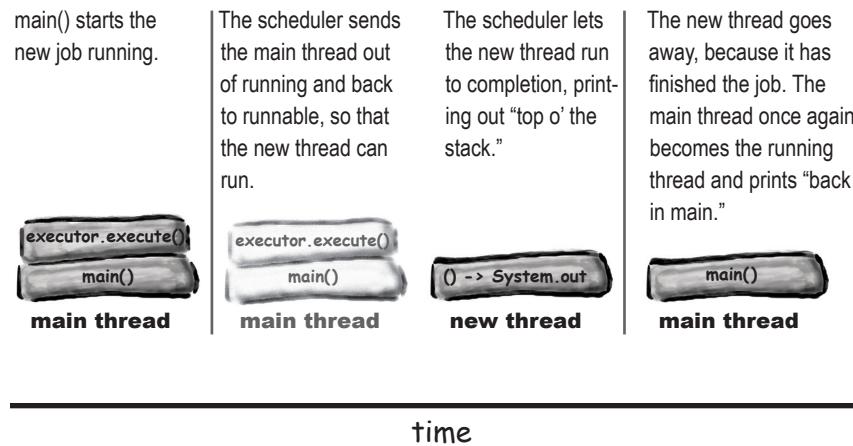
Produced this output:

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

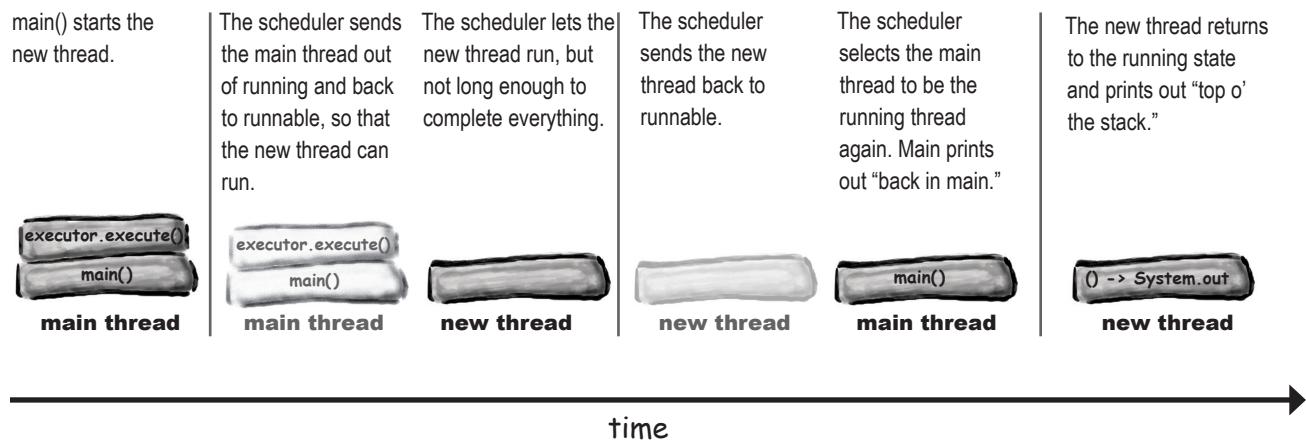
How did we end up with different results?

Multithreaded programs are *not deterministic*; they don't run the same way every time. The thread scheduler can schedule each thread differently each time the program runs.

Sometimes it runs like this:



And sometimes it runs like this:



Even if the new thread is tiny, if it has only one line of code to run like our lambda expression, it can still be interrupted by the thread scheduler.

there are no Dumb Questions

Q: Should I use a lambda expression for my Runnable or create a new class that implements Runnable?

A: It depends upon how complicated your job is, and also on whether you think it's easier to understand as a lambda expression or a class. Lambda expressions are great for when the job is really tiny, like our single-line "print" example. Lambda expressions (or method references) may also work if you have a few lines of code in another method that you want to turn into a job:

```
executor.execute(() -> printMsg());
```

You'll most likely want to use a full Runnable class if your job needs to store things in fields and/or if your job is made up of a number of methods. This is more likely when your jobs are more complex.

Q: What's the advantage of using an ExecutorService? So far, it works the same as creating a Thread and starting it.

A: It's true that for these simple examples, where we're starting just one thread, letting it run, and then stopping our application, the two approaches seem similar. ExecutorServices become really helpful when we're starting lots of independent jobs. We don't necessarily want to create a new Thread for each of these jobs, and we don't want to keep track of all these Threads. There are different ExecutorService implementations depending upon how many threads we'll want to start (or especially if we don't know how many Threads we'll need), including ExecutorServices that create Thread pools. Thread pools let us reuse Thread instances, so we don't have to pay the cost of starting up new Threads for every job. We'll explore this in more detail later.

BULLET POINTS

- A thread with a lowercase "t" is a separate thread of execution in Java.
- Every thread in Java has its own call stack.
- A Thread with a capital "T" is the java.lang.Thread class. A Thread object represents a thread of execution.
- A thread needs a job to do. The job can be an instance of something that implements the Runnable interface.
- The Runnable interface has just a single method, run(). This is the method that goes on the bottom of the new call stack. In other words, it is the first method to run in the new thread.
- Because the Runnable interface has just a single method, you can use lambda expressions where a Runnable is expected.
- Using the Thread class to run separate jobs is no longer the preferred way to create multithreaded applications in Java. Instead, use an Executor or an ExecutorService.
- The Executors class has helper methods that can create standard ExecutorServices to use to start new jobs.
- A thread is in the NEW state when it has not yet started.
- When a thread has been started, a new stack is created, with the Runnable's run() method on the bottom of the stack. The thread is now in the RUNNABLE state, waiting to be chosen to run.
- A thread is said to be RUNNING when the JVM's thread scheduler has selected it to be the currently running thread. On a single-processor machine, there can be only one currently running thread.
- Sometimes a thread can be moved from the RUNNING state to a temporarily NON-RUNNABLE state. A thread might be blocked because it's waiting for data from a stream, because it has gone to sleep, or because it is waiting for an object's lock. We'll see locks in the next chapter.
- Thread scheduling is not guaranteed to work in any particular way, so you cannot be certain that threads will take turns nicely.



Putting a thread to sleep

One way to help your threads take turns is to put them to sleep periodically. All you need to do is call the static `sleep()` method, passing it the amount of time you want the thread to sleep for, in milliseconds.

For example:

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

will knock a thread out of the running state and keep it out of the runnable state for two seconds. The thread *can't* become the running thread again until after at least two seconds have passed.

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

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

Now you know that your thread won't wake up *before* the specified duration, but is it possible that it will wake up some time *after* the “timer” has expired? Effectively, yes. The thread won't automatically wake up at the designated time and become the currently running thread. When a thread wakes up, the thread is once again at the mercy of the thread scheduler; therefore, there are no guarantees about how long the thread will be out of action.

Putting a thread to sleep gives the other threads a chance to run.

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

It can be hard to understand how much time a number of milliseconds represents. There's a convenience method on `java.util.concurrent.TimeUnit` that we can use to make a more readable sleep time:

```
TimeUnit.MINUTES.sleep(2);
```

which may be easier to understand than:

```
Thread.sleep(120000);
```

(You still need to wrap both in a try-catch, though.)

Using sleep to make our program more predictable

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

```
class PredictableSleep {
    public static void main (String[] args) {
        ExecutorService executor =
            Executors.newSingleThreadExecutor();
        executor.execute(() -> sleepThenPrint());
        System.out.println("back in main");
        executor.shutdown();
    }

    private static void sleepThenPrint() {
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("top o' the stack");
    }
}
```

Thread.sleep() throws a checked Exception that we need to catch or declare. Because catching the Exception makes the job’s code a bit longer, we’ve put it into its own method.

There will be a pause (for about two seconds) before we get to this line, which prints out “top o’ the stack.”

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

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

Instead of putting a lambda with an ugly try-catch inside, we’ve put the job code inside a method. We’re calling the method from this lambda

Calling sleep here will force the new thread to leave the currently running state. The main thread will get a chance to print out “back in main.”



Brain Barbell

Can you think of any problems with forcing your threads to sleep for a set amount of time? How long will it take to run this code 10 times?

There are downsides to forcing the thread to sleep

① The program has to wait for at least that amount of time.

If we put the thread to sleep for two seconds, the thread will be non-runnable for that time.

When it wakes up, it won't automatically become the currently running thread. When a thread wakes up, the thread is once again at the mercy of the thread scheduler. Our application is going to be hanging around for at least those two seconds, probably more. This might not sound like a big deal, but imagine a bigger program full of these pauses *intentionally* slowing down the application.

② How do you know the other job will finish in that time?

We put the new thread to sleep for two seconds, assuming that the main thread would be the running thread, and complete its work in that time. But what if the main thread took longer to finish than that? What if another thread, running a longer job, was scheduled instead? One of the ways people deal with this is to set sleep times that are much longer than they'd expect a job to take, but then our first problem becomes even more of a problem.



Is there a way for one thread to tell another that it has finished what it's working on? That way, the main thread could just wait for that signal and then carry on when it knows it's safe to go.

A better alternative: wait for the perfect time.

What we really wanted in our example was to wait until a specific thing had happened in our main thread before carrying on with our new thread. Java supports a number of different mechanisms to do this, like Future, CyclicBarrier, Semaphore, and CountDownLatch.

To coordinate events happening on multiple threads, one thread may need to wait for a specific signal from another thread before it can continue.

Counting down until ready

You can make threads *count down* when significant events have happened. A thread (or threads) can wait for all these events to complete before continuing. You might be counting down until a minimum number of clients have connected, or a number of services have been started.

This is what **`java.util.concurrent.CountDownLatch`** is for. You set a number to count down from. Then any thread can tell the latch to count down when a relevant event has happened.

In our example, we have only one thing we want to count—our new thread should wait until the main thread has printed “back in main” before it can continue.

```
import java.util.concurrent.*;
class PredictableLatch {
    public static void main (String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        CountDownLatch latch = new CountDownLatch(1); ←

        executor.execute(() -> waitForLatchThenPrint(latch));
        System.out.println("back in main");
        latch.countDown(); ← Tell the latch to count down when the
        executor.shutdown(); main method has printed its message.
    }
}
```

```
private static void waitForLatchThenPrint(CountDownLatch latch) {  
    try {  
        latch.await(); // Wait for the main thread to print out its message. This  
    } catch (InterruptedException e) { // thread will be in a non-runnable state while it's waiting.  
        e.printStackTrace();  
    }  
    System.out.println("top o' the stack");  
}
```

await() can throw an InterruptedException, which needs to be caught or declared.

The code is really similar to the code that performs a sleep; the main difference is the latch.countDown in the main method. The performance difference is significant, though. Instead of having to wait *at least* two seconds to make sure main has printed its message, the new thread waits only until the main method has printed its “back in main” message.

To get an idea of the performance difference this might make on a real system, when this latch code was run on a MacBook 100 times, it took around 50 milliseconds to finish *all* one hundred runs, and the output was in the correct order *every time*. If running the sleep() version just one time takes over 2 seconds (2000 milliseconds), imagine how long it took to run 100 times*....

* 200331 milliseconds. That's over 4000x slower.

you are here ►

Making and starting two threads (or more!)

What happens if we want to start more than one job in addition to our main thread? Clearly, we can't use Executors.newSingleThreadExecutor() if we want to run more than one thread. What else is available?

(Just a few of the factory methods)

java.util.concurrent.Executors

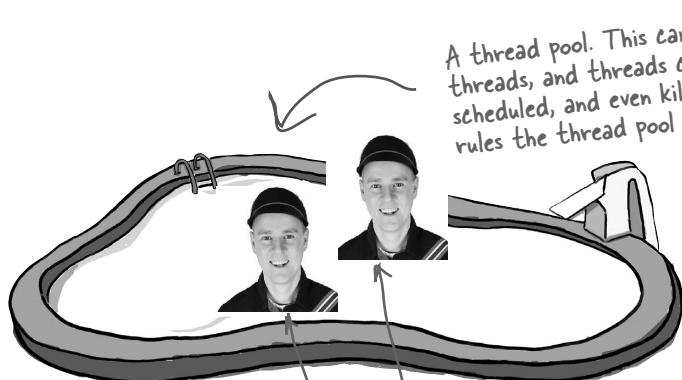
- ExecutorService newCachedThreadPool()**
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
- ExecutorService newFixedThreadPool(int nThreads)**
Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
- ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**
Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
- ExecutorService newSingleThreadExecutor()**
Creates an Executor that uses a single worker thread operating off an unbounded queue.
- ScheduledExecutorService newSingleThreadScheduledExecutor()**
Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.
- ExecutorService newWorkStealingPool()**
Creates a work-stealing thread pool using the number of available processors as its target parallelism level.

These ExecutorServices use some form of Thread Pool. This is a collection of Thread instances that can be used (and reused) to perform jobs.

How many threads are in the pool, and what to do if there are more jobs to run than threads available, depends on the ExecutorService implementation.

Pooling Threads

Using a pool of resources, especially ones that are expensive to create like Threads or database connections, is a common pattern in application code.



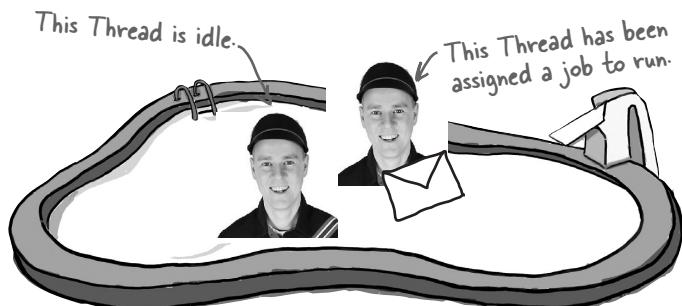
A thread pool. This can contain one or more threads, and threads can be added, used, reused, scheduled, and even killed according to whatever rules the thread pool was set up with.

The threads available for running jobs. How many threads are allowed and how they are used is determined by the pool.

When you create a new ExecutorService, its pool may be started with some threads to begin with, or the pool may be empty.

You can create an ExecutorService with a thread pool using one of the helper methods from the Executors class.

```
ExecutorService threadPool =  
    Executors.newCachedThreadPool();
```



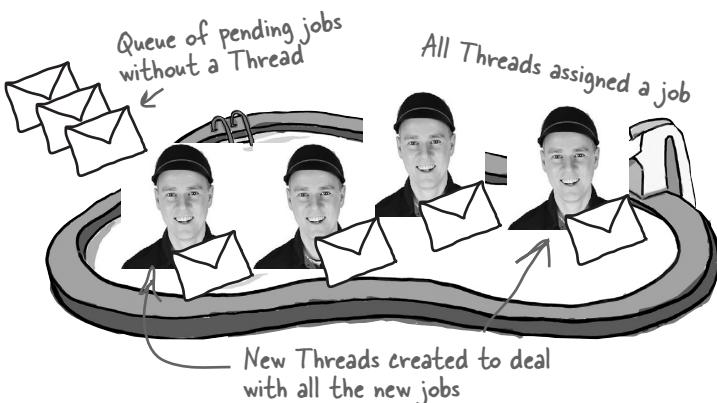
This Thread is idle.

This Thread has been assigned a job to run.

You can use the pool's threads to run your job by giving the job to the ExecutorService. The ExecutorService can then figure out if there's a free Thread to run the job.

```
threadPool.execute(() -> run("Job 1"));
```

This means an ExecutorService can **reuse** threads; it doesn't just create and destroy them.



Queue of pending jobs without a Thread

All Threads assigned a job

New Threads created to deal with all the new jobs

As you give the ExecutorService more jobs to run, it *may* create and start new Threads to handle the jobs. It *may* store the jobs in a queue if there are more jobs than Threads.

How an ExecutorService deals with additional jobs depends on how it is set up.

```
threadPool.execute(() -> run("Job 324"));
```

The ExecutorService may also **terminate** Threads that have been idle for some period of time. This can help to minimize the amount of hardware resources (CPU, memory) your application needs.

Running multiple threads

The following example runs two jobs, and uses a fixed-sized thread pool to create two threads to run the jobs. Each thread has the same job: run in a loop, printing the currently running thread's name with each iteration.

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class RunThreads {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(2);
        threadPool.execute(() -> runJob("Job 1"));
        threadPool.execute(() -> runJob("Job 2"));
        threadPool.shutdown();
    }

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

```

Create an ExecutorService with a fixed-sized thread pool (we know we're going to run only two jobs).

A lambda expression that represents our Runnable job. If you don't want to use lambdas, here you'd pass in a new instance of your Runnable class, like we did when we created MyRunnable earlier in the chapter.

The job is to run through this loop, printing the thread's name each time.

What will happen?

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

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

Running this on a modern multicore system, the two jobs will likely run in parallel, but there's no guarantee that this means they will complete in the same amount of time or output values at the same rate.

```

File Edit Window Help Globetrotter
% java RunThreads
Job 1 is running on pool-1-thread-1

```

Closing time at the thread pool

You may have noticed that our examples have a `threadPool.shutdown()` at the end of the main methods. Although the thread pools will take care of our individual Threads, we do need to be responsible adults and close the pool when we're finished with it. That way, the pool can empty its job queue and shut down all of its threads to free up system resources.

ExecutorService has two shutdown methods. You can use either, but to be safe we'd use both:

① ExecutorService.shutdown()

Calling `shutdown()` asks the ExecutorService nicely if it wouldn't mind awfully wrapping things up so everyone can go home.

All of the Threads that are currently running jobs are allowed to finish those jobs, and any jobs waiting in the queue will also be finished off. The ExecutorService will reject any new jobs too.

If you need your code to wait until all of those things are finished, you can use `awaitTermination` to sit and wait until it's finished. You give `awaitTermination` a maximum amount of time to wait for everything to end, so `awaitTermination` will hang around until either the ExecutorService has finished everything or the timeout has been reached, whichever is earlier.

② ExecutorService.shutdownNow()

Everybody out! When this is called, the ExecutorService will try to stop any Threads that are running, will not run any waiting jobs, and definitely won't let anyone else into the pool.

Use this if you need to put a stop to everything. This is sometimes used after first calling `shutdown()` to give the jobs a chance to finish before pulling the plug entirely.

```

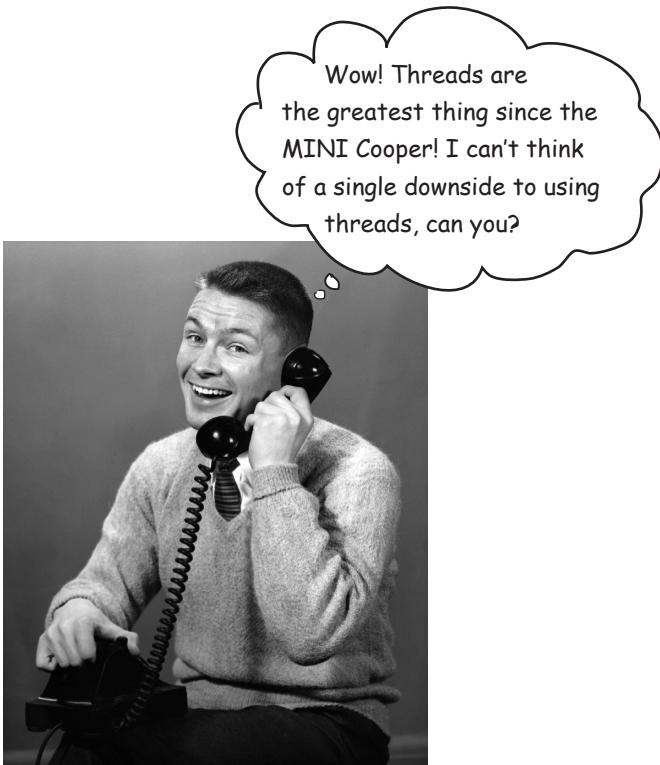
public class ClosingTime {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        threadPool.execute(new LongJob("Long Job"));
        threadPool.execute(new ShortJob("Short Job"));

        threadPool.shutdown();
        try {
            boolean finished = threadPool.awaitTermination(5, TimeUnit.SECONDS);
            System.out.println("Finished? " + finished);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        threadPool.shutdownNow();
    }
}

Ask the ExecutorService to shut down. If you call execute() with a job after this, you will get a RejectedExecutionException. The ExecutorService will continue to run all the jobs that are running, and run any waiting jobs too.
Create a thread pool with just two threads.
Start two jobs, a short one that just prints the name and a "long" one that uses a sleep so it can pretend to be a long-running job (LongJob and ShortJob are classes that implement Runnable).
Wait up to 5 seconds for the ExecutorService to finish everything. If this method hits the timeout before everything has finished, it returns "false".
At this point, we tell the ExecutorService to stop everything right now. If everything was already shut down, that's fine; this won't do anything.

```



Um, yes. There IS a dark side. Multithreading can lead to concurrency "issues."

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

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

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

BULLET POINTS

- The static `Thread.sleep()` method forces a thread to leave the running state for at least the duration passed to the sleep method. `Thread.sleep(200)` puts a thread to sleep for 200 milliseconds.
- You can also use the sleep method on `java.util.concurrent.TimeUnit`, for example `TimeUnit.SECONDS.sleep(2)`.
- The `sleep()` method throws a checked exception (`InterruptedException`), so all calls to `sleep()` must be wrapped in a try/catch, or declared.
- There are different mechanisms to give threads a chance to wait for each other. You can use `sleep()`, but you can also use `CountDownLatch` to wait for the right number of events to have happened before continuing.
- Managing threads directly can be a lot of work. Use the factory methods in `Executors` to create an `ExecutorService`, and use this service to run `Runnable` jobs.
- Thread pools can manage creation, reuse, and destruction of threads so you don't have to.
- `ExecutorServices` should be shut down correctly so the jobs are finished and threads terminated. Use `shutdown()` for graceful shutdown, and `shutdownNow()` to kill everything.



It's a cliff-hanger!

A dark side? Race conditions?? Data corruption?! But what can we DO about those things? Don't leave us hanging!

Stay tuned for the next chapter, where we address these issues and more...



Exercise

A bunch of Java and network terms, in full costume, are playing a party game, “Who am I?” They’ll give you a clue—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 attendee, 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:

InetSocketAddress, SocketChannel, IP address, host name, port, Socket, ServerSocketChannel, Thread, thread pool, Executors, ExecutorService, CountDownLatch, Runnable, InterruptedException, Thread.sleep()



I need to be shut down or I might live forever _____

I let you talk to a remote machine _____

I might be thrown by sleep() and await() _____

If you want to reuse Threads, you should use me _____

You need to know me if you want to connect to another machine _____

I’m like a separate process running on the machine _____

I can give you the ExecutorService you need _____

You need one of me if you want clients to connect to me _____

I can help you make your multithreaded code more predictable _____

I represent a job to run _____

I store the IP address and port of the server _____

—————> Answers on page 636.

New and improved SimpleChatClient

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

This is the New Improved chat client that can both send and receive messages, thanks to the power of multithreading! Remember, you need to run the chat server first to run this code.

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.concurrent.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class SimpleChatClient {
    private JTextArea incoming;
    private JTextField outgoing;
    private BufferedReader reader;
    private PrintWriter writer;

    public void go() {
        setUpNetworking();

        JScrollPane scroller = createScrollableTextArea();
        outgoing = new JTextField(20);

        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(e -> sendMessage());

        JPanel mainPanel = new JPanel();
        mainPanel.add(scroller);
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);

        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new IncomingReader());
    }

    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(400, 350);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}

```

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

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

We've got a new job, an inner class, which is a Runnable. The job is to read from the server's socket stream, displaying any incoming messages in the scrolling text area. We start this job using a single thread executor since we know we want to run only this one job.

```

private JScrollPane createScrollableTextArea() {
    incoming = new JTextArea(15, 30);
    incoming.setLineWrap(true);
    incoming.setWrapStyleWord(true);
    incoming.setEditable(false);
    JScrollPane scroller = new JScrollPane(incoming);
    scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    return scroller;
}

private void setUpNetworking() {
    try {
        InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000);
        SocketChannel socketChannel = SocketChannel.open(serverAddress);

        reader = new BufferedReader(Channels.newReader(socketChannel, UTF_8));
        writer = new PrintWriter(Channels.newWriter(socketChannel, UTF_8));

        System.out.println("Networking established.");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

private void sendMessage() {
    writer.println(outgoing.getText());
    writer.flush();
    outgoing.setText("");
    outgoing.requestFocus();
}
}

public class IncomingReader implements Runnable {
    public void run() {
        String message;
        try {
            while ((message = reader.readLine()) != null) {
                System.out.println("read " + message);
                incoming.append(message + "\n");
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

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

```

A helper method, like we saw back in Chapter 1b, to create a scrolling text area.

We're using `Channels` to create a new reader and writer for the `SocketChannel` that's connected to the server. The writer sends messages to the server, and now we're using a reader so that the reader job can get messages from the server.

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

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

Remember, the Chat Server code was the Ready-Bake Code from page 606.

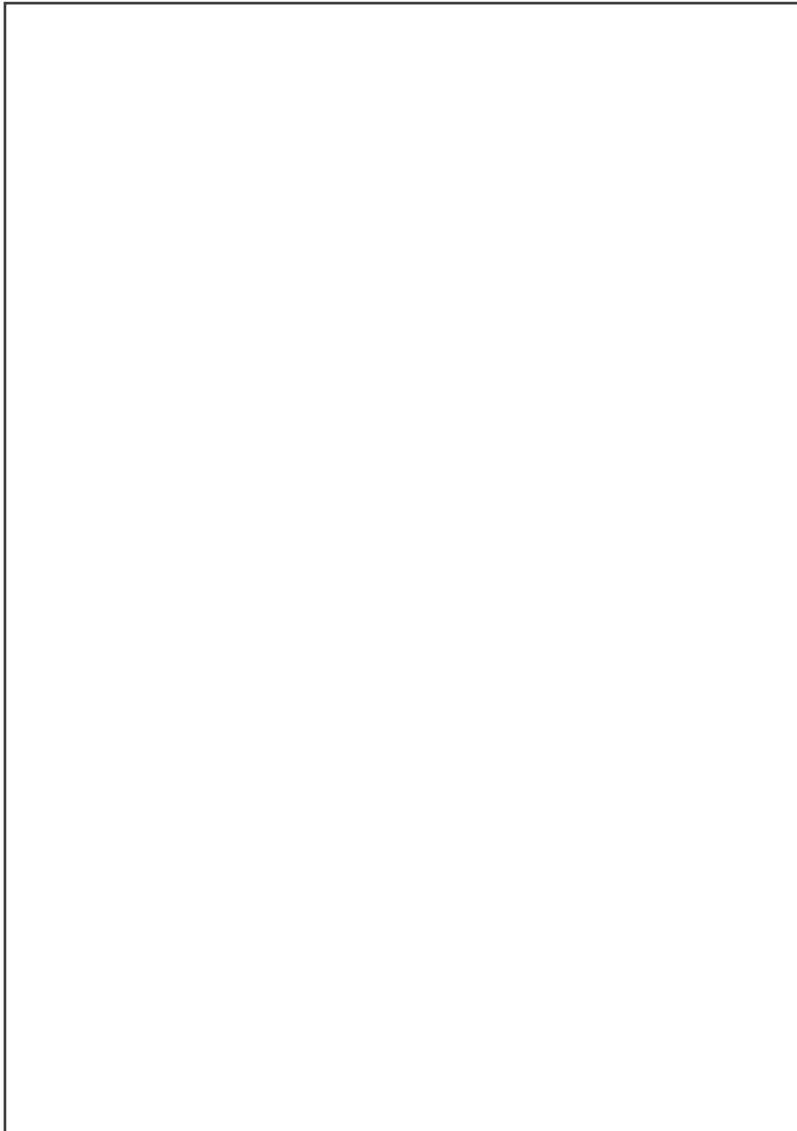
exercise: code magnets



Code Magnets

A working Java program is scrambled up on the fridge (see the next page). Can you reconstruct the code snippets on the next page to make a working Java program that produces the output listed below?

To get it to work, you will need to be running the **SimpleChatServer** from page 606.



→ Answers on page 636.

```
File Edit Window Help StillThere
% java PingingClient
Networking established
09:27:06 Sent ping 0
09:27:07 Sent ping 1
09:27:08 Sent ping 2
09:27:09 Sent ping 3
09:27:10 Sent ping 4
09:27:11 Sent ping 5
09:27:12 Sent ping 6
09:27:13 Sent ping 7
09:27:14 Sent ping 8
09:27:15 Sent ping 9
```

Code Magnets, continued

```

String message = "ping " + i;

try (SocketChannel channel = SocketChannel.open(server)) {
    import static java.nio.charset.StandardCharsets.UTF_8;
    import static java.time.LocalDateTime.now;
    import static java.time.format.DateTimeFormatter.ofLocalizedTime;
    e.printStackTrace();
} catch (IOException | InterruptedException e) {
    public class PingingClient {
        }
        writer.println(message);
    }

PrintWriter writer = new PrintWriter(Channels.newWriter(channel, UTF_8));

import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.time.format.FormatStyle;
import java.util.concurrent.TimeUnit;
for (int i = 0; i < 10; i++) {
    System.out.println(currentTime + " Sent " + message);
    writer.flush();
}

InetSocketAddress server = new InetSocketAddress("127.0.0.1", 5000);
}

System.out.println("Networking established");
    TimeUnit.SECONDS.sleep(1);

public static void main(String [] args) {
    String currentTime = now().format(ofLocalizedTime(FormatStyle.MEDIUM));
}

```

exercise solutions

Who Am I? (from page 631)

I need to be shut down or I might live forever
I let you talk to a remote machine
I might be thrown by sleep() and await()
If you want to reuse Threads, you should use me
You need to know me if you want to connect to another machine
I'm like a separate process running on the machine
I can give you the ExecutorService you need
You need one of me if you want clients to connect to me
I can help you make your multithreaded code more predictable
I represent a job to run
I store the IP address and port of the server

ExecutorService
SocketChannel, Socket
InterruptedException
Thread pool, ExecutorService
IP Address, Host name, port
Thread
Executors
ServerSocketChannel
Thread.sleep(), CountDownLatch
Runnable
InetSocketAddress

Exercise Solutions



Code Magnets

(from pages 634–635)

```
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.time.format.FormatStyle;
import java.util.concurrent.TimeUnit;

import static java.nio.charset.StandardCharsets.UTF_8;
import static java.time.LocalDateTime.now;
import static java.time.format.DateTimeFormatter.ofLocalizedTime;

public class PingingClient {

    public static void main(String[] args) {
        InetSocketAddress server = new InetSocketAddress("127.0.0.1", 5000);
        try (SocketChannel channel = SocketChannel.open(server)) {
            PrintWriter writer = new PrintWriter(Channels.newWriter(channel, UTF_8));
            System.out.println("Networking established");

            for (int i = 0; i < 10; i++) {
                String message = "ping " + i;
                writer.println(message);
                writer.flush();
                String currentTime = now().format(ofLocalizedTime(FormatStyle.MEDIUM));
                System.out.println(currentTime + " Sent " + message);
                TimeUnit.SECONDS.sleep(1);
            }
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

You should get the same output even if you move the sleep() to somewhere else inside this for loop.

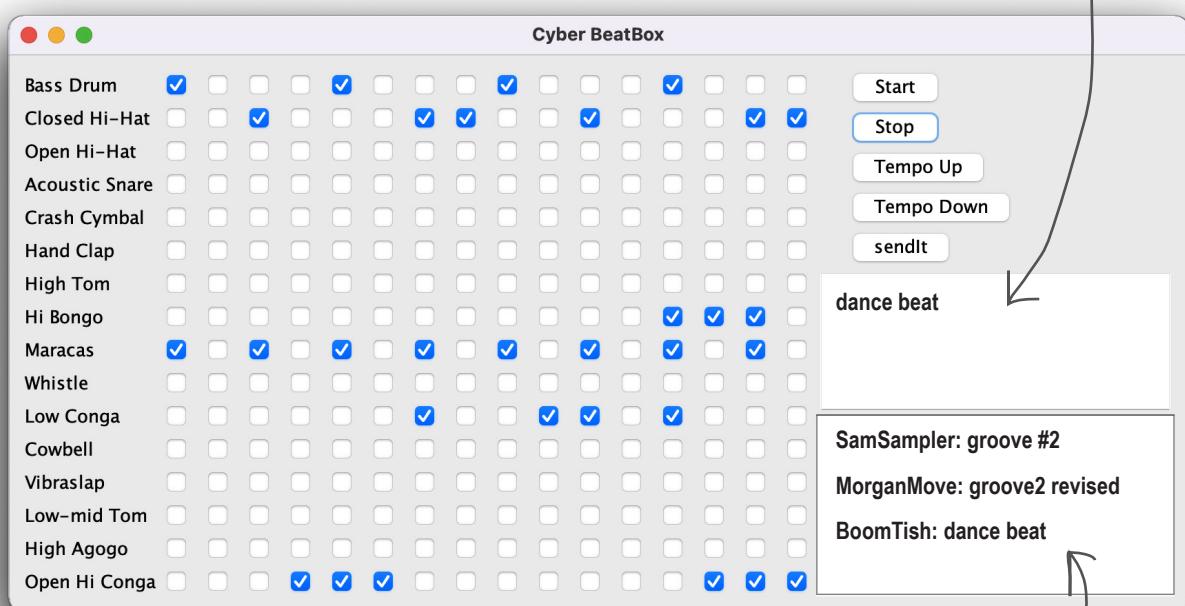
Catching all Exceptions at the end because we do the same thing with them all.

This is one way of getting the current time and turning it into a String in the format of Hours:Minutes:Seconds.

You can catch the InterruptedException thrown by sleep() inside the for loop, or you can catch all the Exceptions at the end of the method.

Code Kitchen

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



Now you've seen how to build a chat client, we have the last version of the BeatBox!

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

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

Dealing with Concurrency Issues



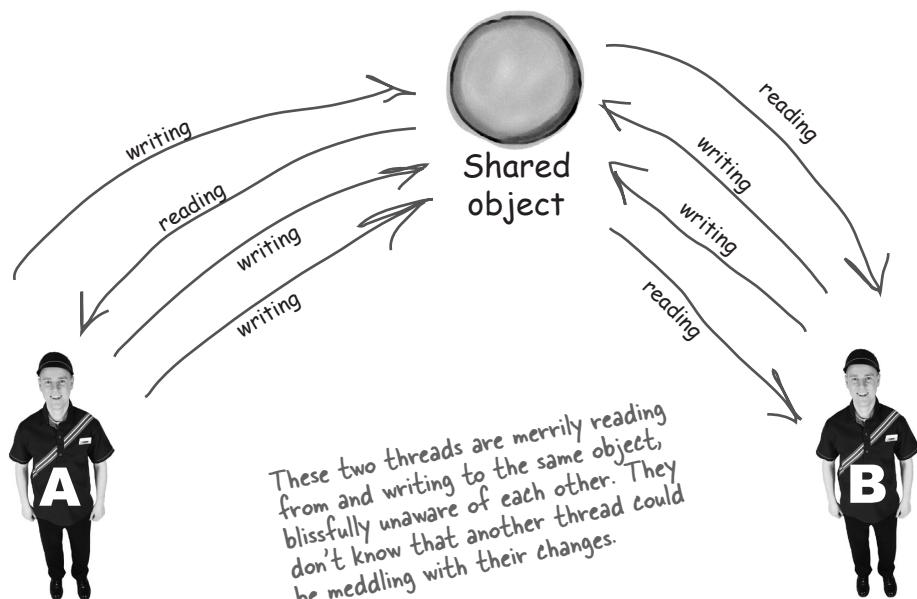
Dave hasn't noticed that Helen is taking bites out of his sandwich while he's eating it! Those two should figure out they're sharing something, or it's going to get messy...

Doing two or more things at once is hard. Writing multithreaded code is easy. Writing multithreaded code that works the way you expect can be much harder. In this final chapter, we're going to show you some of the things that can go wrong when two or more threads are working at the same time. You'll learn about some of the tools in `java.util.concurrent` that can help you to write multithreaded code that works correctly. You'll learn how to create immutable objects (objects that don't change) that are safe for multiple threads to use. By the end of the chapter, you'll have a lot of different tools in your toolkit for working with concurrency.

What could possibly go wrong?

At the end of the last chapter we hinted that things may not all be rainbows and sunshine when you're working with multithreaded code. Well, actually, we did more than hint! We outright said:

"It all comes down to one potentially deadly scenario: two or more threads have access to a single object's data."



Brain Barbell

Why is it a problem if two threads are both reading and writing?

If a thread reads the object's data before changing it, why is it a problem that another thread might also be writing at the same time?

Marriage in Trouble.

Can this couple be saved?

Next, on a very special Dr. Steve Show

[Transcript from episode #42]

Welcome to the Dr. Steve show.



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

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

Here's how Monica described it to me:

"Ryan and I agreed that neither of us will overdraw the checking account. So the procedure is, whoever wants to spend money *must* check the balance in the account *before* withdrawing cash or spending on a card. It all seemed so simple. But suddenly we're getting hit with overdraft fees!"

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

Ryan had a full online shopping cart totalling \$50. He checked the balance in the account and saw that it was \$100. No problem. So he started the checkout procedure.

And that's where *I* come in; while Ryan was filling in the shipping details, I wanted to spend \$100. I checked the balance, and it's \$100 (because Ryan hasn't clicked the "Pay" button yet), so I think, no problem. So I spend the money, and again no problem. But then Ryan finally pays, and we're suddenly overdrawn! He didn't know that I was spending money at the same time, so he just went ahead and completed his transaction without checking the balance again. You've got to help us, Dr. Steve!"

Is there a solution? Are they doomed? We can't help them with their online shopping addiction, but can we make sure one of them can't start spending while the other one is shopping?

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



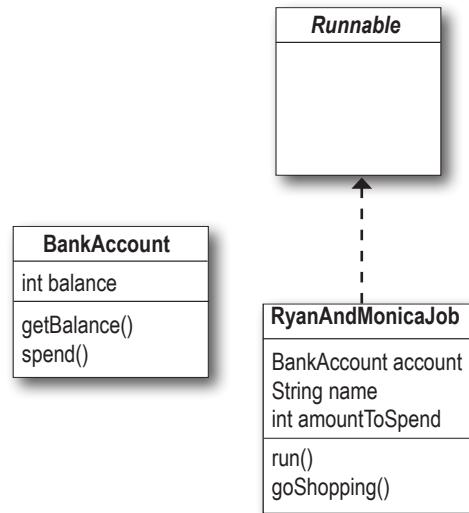
Ryan and Monica: victims of the "two people, one account" problem.

The Ryan and Monica problem, in code

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

The code has two classes, BankAccount and RyanAndMonicaJob. There's also a RyanAndMonicaTest with a main method to run everything. The RyanAndMonicaJob class implements Runnable, and represents the behavior that Ryan and Monica both have—checking the balance and spending money.

The RyanAndMonicaJob class has instance variables for the shared BankAccount, the person's name, and the amount they want to spend. The code works like this:



① Make an instance of the shared bank account

Creating a new one will set up all the defaults correctly.

```
BankAccount account = new BankAccount();
```

② Make one instance of RyanAndMonicaJob for each person

We need one job for each person. We also need to give them access to the BankAccount and tell them how much to spend.

```
RyanAndMonicaJob ryan = new RyanAndMonicaJob("Ryan", account, 50);
RyanAndMonicaJob monica = new RyanAndMonicaJob("Monica", account, 100);
```

③ Create an ExecutorService and give it the two jobs

Since we know we have two jobs, one for Ryan and one for Monica, we can create a fixed-sized thread pool with two threads.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(ryan);
executor.execute(monica);
```

④ Watch both jobs run

One thread represents Ryan, the other represents Monica. Both threads check the balance before spending money. Remember that when more than one thread is running at a time, you can't assume that your thread is the only one making changes to a shared object (e.g., the BankAccount). Even if there's only two lines of code related to the shared object, and they're right next to each other.

```
if (account.getBalance() >= amount) {
    account.spend(amount);
} else {
    System.out.println("Sorry, not enough money");
}
```

In the `goShopping()` method, do exactly what Ryan and Monica would do—check the balance and, if there's enough money, spend.

This should protect against overdrawing the account.

Except...if Ryan and Monica are spending money at the same time, the money in the bank account might be gone before the other one can spend it!

The Ryan and Monica example

```

import java.util.concurrent.*;

public class RyanAndMonicaTest {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        RyanAndMonicaJob ryan = new RyanAndMonicaJob("Ryan", account, 50);
        RyanAndMonicaJob monica = new RyanAndMonicaJob("Monica", account, 100);
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(ryan); // Start both jobs
        executor.execute(monica); // running.
        executor.shutdown();
    } // Don't forget to shut the pool down.
}

class RyanAndMonicaJob implements Runnable {
    private final String name;
    private final BankAccount account;
    private final int amountToSpend;
    RyanAndMonicaJob(String name, BankAccount account, int amountToSpend) {
        this.name = name;
        this.account = account;
        this.amountToSpend = amountToSpend;
    }
    public void run() {
        goShopping(amountToSpend); // The run() method just calls goShopping()
    }
    private void goShopping(int amount) {
        if (account.getBalance() >= amount) {
            System.out.println(name + " is about to spend");
            account.spend(amount);
            System.out.println(name + " finishes spending");
        } else {
            System.out.println("Sorry, not enough for " + name);
        }
    }
}

class BankAccount {
    private int balance = 100; // The account starts with a
    public int getBalance() { // balance of $100.
        return balance;
    }
    public void spend(int amount) {
        balance = balance - amount;
        if (balance < 0) {
            System.out.println("Overdrawn!");
        }
    }
}

```

There will be only ONE instance of the BankAccount. That means both threads will access this one account.

Make two jobs that will do the withdrawal from the shared bank account, one for Monica and one for Ryan, passing in the amount they're going to spend.

Create a new thread pool with two threads for our two jobs.

The run() method just calls goShopping() with the amount they need to spend.

Check the account balance, and if there's enough money, we go ahead and spend the money, just like Ryan and Monica did.

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

Ryan and Monica output

How did this happen? →

```
File Edit Window Help Visa
% java RyanAndMonicaTest
Ryan is about to spend
Monica is about to spend
Overdrawn!
Ryan finishes spending
Monica finishes spending
```

This code is not deterministic; it doesn't always produce the same result every time. You may need to run it a few times before you see the problem.

This is common with multithreaded code, since it depends upon which threads start first and when each thread gets its time on a CPU core.

Sometimes the code works correctly and they don't go overdrawn. →

```
File Edit Window Help WorksOnMyMachine
% java RyanAndMonicaTest
Ryan is about to spend
Ryan finishes spending
Sorry, not enough for Monica
```

The **goShopping()** method always checks the balance before making a withdrawal, but still we went overdrawn.

Here's one scenario:

Ryan checks the balance, sees that there's enough money, and goes to check out.

Meanwhile, Monica checks the balance. She, too, sees that there's enough money. She has no idea that Ryan is about to pay for something.

Ryan completes his purchase.

Monica completes her purchase. Big Problem! In between the time when she checked the balance and spent the money, Ryan had already spent money!

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

Monica must be stopped from getting into the account until Ryan has finished, and vice versa.

They need a lock for account access!

The lock works like this:

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



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

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



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

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

Now, even if Ryan gets distracted after he checks the balance, he has a guarantee that the balance will be the same when he spends the money, because he kept the key while he was doing something else!



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

We need to check the balance and spend the money as one atomic thing



We need to make sure that once a thread starts a shopping transaction, *it must be allowed to finish* before any other thread changes the bank account.

In other words, we need to make sure that once a thread has checked the account balance, that thread has a guarantee that it can spend the money *before any other thread can check the account balance!*

Use the **synchronized** keyword on a method, or with an object, to lock an object so only one thread can use it at a time.

That's how you protect the bank account! We can put a lock on the bank account inside the method that does the banking transaction. That way, one thread gets to complete the whole transaction, start to finish, even if that thread is taken out of the "running" state by the thread scheduler or another thread is trying to make changes at exactly the same time.

On the next couple of pages we'll look at the different things that we can lock. With the Ryan and Monica example, it's quite simple—we want to wrap our shopping transaction in a block that locks the bank account:

```
private void goShopping(int amount) {  
    synchronized (account) {  
        if (account.getBalance() >= amount) {  
            System.out.println(name + " is about to spend");  
            account.spend(amount);  
            System.out.println(name + " finishes spending");  
        } else {  
            System.out.println("Sorry, not enough for " + name);  
        }  
    }  
}
```

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

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

To protect your data (like the bank account), synchronize the code that acts on that data.

there are no
Dumb Questions

Q: Why don't you just synchronize all the getters and setters from the class with the data you're trying to protect?

A: Synchronizing the getters and setters isn't enough. Remember, the point of synchronization is to make a specific section of code work ATOMICALLY. In other words, it's not just the individual methods we care about; it's methods that require **more than one step to complete!**

Think about it. We added a synchronized block inside the goShopping() method. If getBalance() and spend() were both synchronized instead, it wouldn't help—Ryan (or Monica) would have checked the balance returned the key! The whole point is to keep the key until **both** operations are completed.

Using an object's lock

Every object has a lock. Most of the time, the lock is unlocked, and you can imagine a virtual key sitting with it. Object locks come into play only when there is a **synchronized block** for an object (like in the last page) or a class has **synchronized methods**.

A method is synchronized if it has the synchronized keyword in the method declaration.

When an object has one or more synchronized methods, **a thread can enter a synchronized method only if the thread can get the key to the object's lock!**

The locks are not per *method*, they are per *object*. If an object has two synchronized methods, it doesn't *only* mean two threads can't enter the same method. It means you can't have two threads entering *any* of the synchronized methods. If you have two synchronized methods on the same object, method1() and method2(), if one thread is in method1(), a second thread can't enter method1(), obviously, but it *also can't enter method2()*, or any other synchronized method on that object.

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

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

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

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



Hey, this object's takeMoney() method is synchronized. I need to get this object's key before I can go in...



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

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

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

using synchronized

Using synchronized methods

Can we synchronize the goShopping() method to fix Ryan and Monica's problem?

```
private synchronized void goShopping(int amount) {  
    if (account.getBalance() >= amount) {  
        System.out.println(name + " is about to spend");  
        account.spend(amount);  
        System.out.println(name + " finishes spending");  
    } else {  
        System.out.println("Sorry, not enough for " + name);  
    }  
}
```

It does NOT work!

```
File Edit Window Help WaitWhat  
% java RyanAndMonicaTest  
Ryan is about to spend  
Ryan finishes spending  
Monica is about to spend  
Overdrawn!  
Monica finishes spending
```

The synchronized keyword locks an object. The goShopping() method is in RyanAndMonicaJob. Synchronizing an instance method means “lock *this* RyanAndMonicaJob instance.” However, there are *two* instances of RyanAndMonicaJob; one is “ryan,” and the other is “monica.” If “ryan” is locked, “monica” can still make changes to the bank account; she doesn’t care that the “ryan” job is locked.

The object that needs locking, the object these two threads are fighting over, is the BankAccount. Putting synchronized on a method in RyanAndMonicaJob (and locking a RyanAndMonicaJob instance) isn’t going to solve anything.

It's important to lock the correct object

Since it's the BankAccount object that's shared, you could argue it should be the BankAccount that's in charge of making sure it is safe for multiple threads to use. The spend() method on BankAccount could make sure there's enough money *and* debit the account in a single transaction.

```

class RyanAndMonicaJob implements Runnable {
    // ...rest of the RyanAndMonicaJob class
    // the same as before...

    private void goShopping(int amount) {
        System.out.println(name + " is about to spend");
        account.spend(name, amount);
        System.out.println(name + " finishes spending");
    }
}

class BankAccount {
    // other methods in BankAccount...
    public synchronized void spend(String name, int amount) {
        if (balance >= amount) {
            balance = balance - amount;
            if (balance < 0) {
                System.out.println("Overdrawn!");
            } else {
                System.out.println("Sorry, not enough for " + name);
            }
        } Do the balance check and balance
        decrease in the BankAccount itself. If
        this method is synchronized, it becomes
        an atomic transaction that can be done
        in full by only one thread at a time.
    }
}

```

This would no longer need to check the balance before spending if we know the BankAccount spend() method checks for us.

Locks the BankAccount instance the two threads are using.

Ryan and Monica SHOULDN'T go overdrawn now; this should never be the case.

there are no Dumb Questions

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

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

The dreaded “Lost Update” problem

Here's another classic concurrency problem. Sometimes you'll hear them called “race conditions,” where two or more threads are changing the same data at the same time. It's closely related to the Ryan and Monica story, so we'll use this example to illustrate a few more points.

The lost update revolves around one process:

Step 1: Get the balance in the account

```
int i = balance;
```

Step 2: Add 1 to that balance

```
balance = i + 1;
```

Probably not an atomic process

Even if we used the more common syntax of `balance++`, there is no guarantee that the compiled bytecode will be an “atomic process.” In fact, it probably won't—it's actually multiple operations: a read of the current value and then adding one to that value and setting it back into the original variable.

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

```
public class LostUpdate {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool(6);
        Balance balance = new Balance();
        for (int i = 0; i < 1000; i++) { ← Run 1,000 attempts to update
            pool.execute(() -> balance.increment()); ← the balance, on different threads.
        }
        pool.shutdown();
        if (pool.awaitTermination(1, TimeUnit.MINUTES)) { ← Make sure the pool has finished
            System.out.println("balance = " + balance.balance); ← running all the updates before
        }                                         ← printing the final balance. In
    }                                         ← theory, this should be 1,000. If
}                                         ← it's any less than that, we've lost
                                            ← an update!
```

Create a thread pool to run all the jobs. If you add more threads here, you may see even more missing updates.

```
class Balance {
    int balance = 0;
    public void increment() {
        balance++; ← Here's the crucial part! We increment the balance
    }             ← by adding 1 to whatever the value of balance was AT
}               ← THE TIME WE READ IT (rather than adding 1 to
                  ← whatever the CURRENT value is). You might think
                  ← that “++” is an atomic operation, but it is not.
```

Let's run this code...

① Thread A runs for a while

Reads balance: 0
 Set the value of balance to $0 + 1$.
 Now balance is 1



Reads balance: 1
 Set the value of balance to $1 + 1$.
 Now balance is 2

② Thread B runs for a while



Reads balance: 2
 Set the value of balance to $2 + 1$.
 Now balance is 3

Reads balance: 3
[now thread B is sent back to runnable, before it sets the value of balance to 4]

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

Reads balance: 3
 Set the value of balance to $3 + 1$.
 Now balance is 4



Reads balance: 4
 Set the value of balance to $4 + 1$.
 Now balance is 5

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

Set the value of balance to $3 + 1$.
 Now balance is 4

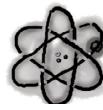


Yikes!!
 Thread A updated it to 5, but now B came back and stepped on top of the update A made, as if A's update never happened.

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

Make the increment() method atomic.

Synchronize it!



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

```
public synchronized void increment() {
    balance++;
}
```

Classic concurrency gotcha: this looks like a single operation, but it's actually more than one—it's a read of the balance, an increment, and an update to the balance.

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

there are no Dumb Questions

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

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

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

Third, and most frightening, synchronized methods can lead to deadlock! (We'll see this in a couple of pages.)

A good rule of thumb is to synchronize only the bare minimum that should be synchronized. And in fact, you can synchronize at a granularity that's even smaller than a method. Remember, you can use the synchronized keyword to synchronize at the more fine-grained level of one or more statements, rather than at the whole-method level (we used this in our first solution to Ryan and Monica's problem).

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

```
public void go() {
    doStuff();

    synchronized(this) {
        criticalStuff();
        moreCriticalStuff();
    }
}
```



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

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

① Thread A runs for a while

Attempt to enter the increment() method.

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

Reads balance: 0

Set the value of balance to $0 + 1$.

Now balance is 1

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

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

Reads balance: 1



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

② Thread B is selected to run



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

The key is not available.

[now thread B is sent into an “object lock not available” lounge]

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

Set the value of balance $1 + 1$.

Now balance is 2

Return the key.

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



④ Thread B is selected to run



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

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

Reads balance: 2

[continues to run...]

Deadlock, a deadly side of synchronization

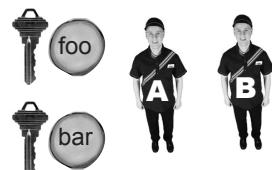
Synchronization saved Ryan and Monica from using their bank account at the same time, and has saved us from losing updates. But we also mentioned that we shouldn't synchronize everything, one reason being that synchronization can slow your program down.

There's another important consideration: we need to be careful using synchronized code, because nothing will bring your program to its knees like thread deadlock. Thread deadlock happens when you have two threads, both of which are holding a key the other thread wants. There's no way out of this scenario, so the two threads will simply sit and wait. And wait. And wait.

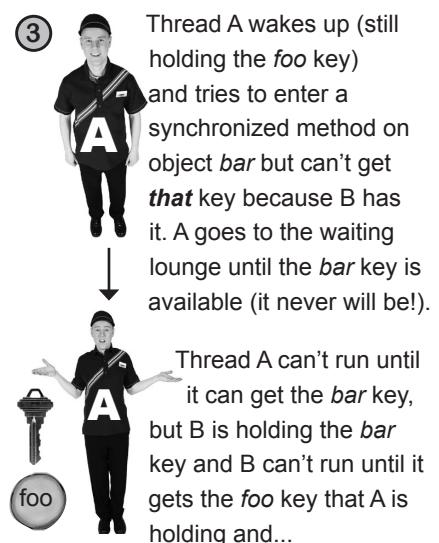
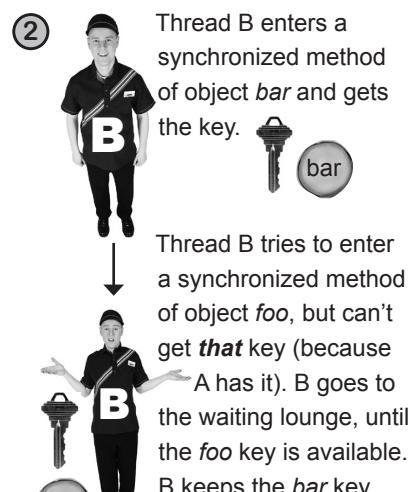
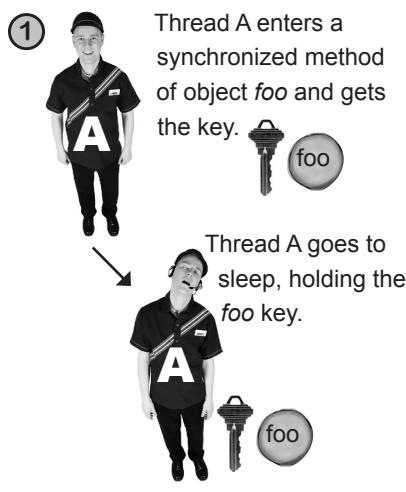
If you're familiar with databases or other application servers, you might recognize the problem; databases often have a locking mechanism somewhat like synchronization. But a real transaction management system can sometimes deal with deadlock. It might assume, for example, that deadlock might have occurred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

Java has no mechanism to handle deadlock. It won't even *know* deadlock occurred. So it's up to you to design carefully. We're not going to go into more detail about deadlock than you see on this page, so if you find yourself writing multithreaded code, you might want to study *Java Concurrency in Practice* by Brian Goetz, et al. It goes into a lot of detail about the sorts of problems you can face with concurrency (like deadlock), and approaches to address these problems.

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



A simple deadlock scenario:



You don't always have to use synchronized

Since synchronization can come with some costs (like performance and potential deadlocks), you should know about other ways to manage data that's shared between threads. The `java.util.concurrent` package has lots of classes and utilities for working with multithreaded code.

Atomic variables

If the shared data is an int, long, or boolean, we might be able to replace it with an *atomic variable*. These classes provide methods that are atomic, i.e., can safely be used by a thread without worrying about another thread changing the object's values at the same time.

There are few types of atomic variable, e.g., **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, and **AtomicReference**.

We can fix our Lost Update problem with an `AtomicInteger`, instead of synchronizing the increment method.

```
class Balance {
    AtomicInteger balance = new AtomicInteger(0);
    public void increment() {
        balance.incrementAndGet();
    }
}
```

No need to add "synchronized" when you're using atomic operations.

Use an `AtomicInteger` initialized to zero, instead of an int value.

 if it's used by multiple threads, it will safely increase the value by one in a single operation. The `incrementAndGet` method returns the new, updated value, but we don't need that for our example; we're not going to use the returned value.

So I can use `AtomicInteger` as long as all I want to do is a simple increment. How does this help me if I want to do normal things like complex calculations?



Atomic variables get more interesting when you use their *compare-and-swap* (**CAS**) operations. CAS is yet another way to make an atomic change to a value. You can use CAS on atomic variables by using the **compareAndSet** method. Yes, it's a slightly different name! Gotta love programming, where naming is always the hardest problem to solve.

The `compareAndSet` method takes a value, which is what you *expect* the atomic variable to be, compares it to the *current* value, and if that matches, *then* the operation will complete.

In fact, we can use this to fix our Ryan and Monica problem, instead of locking the whole bank account with **synchronized**.

Compare-and-swap with atomic variables

How could we make use of atomic variables, and CAS (via `compareAndSet`), to solve Ryan and Monica's problem?

Since Ryan and Monica were both trying to access an int value, the account balance, we could use an AtomicInteger to store that balance. We could then use `compareAndSet` to update the balance when someone wants to spend money.

```
private AtomicInteger balance = new AtomicInteger(100);
```

...

```
boolean success = balance.compareAndSet(expectedValue, newValue)
```

True if the balance was updated to the new value. If this is false, the balance wasn't changed and YOU decide what you need to do next.

This is the value you THINK the balance is.

If the current balance is the same as the expected value, update it to the new value.

This is the value you want the balance to have.



In plain English:

“Set the balance to this new value only if the current balance is the same as this expected value, and tell me if the balance was actually changed.”

Compare-and-swap uses *optimistic locking*. Optimistic locking means you don't stop all threads from getting to the object; you *try* to make the change, but you embrace the fact that the change **might not happen**. If it doesn't succeed, you decide what to do. You might decide to try again, or to send a message letting the user know it didn't work.

This may be more work than simply locking all other threads out from the object, but it can be faster than locking everything. For example, when the chances of multiple writes happening at the same time are very low or if you have a lot of threads reading and not so many writing, then you may not want to pay the price of a lock on every write.

When you're using CAS operations, you have to deal with the times when the operation does NOT succeed.

Ryan and Monica, going atomic

Let's see the whole thing in action in Ryan and Monica's bank account. We'll put the balance in an AtomicInteger and use compareAndSet to make an *atomic* change to the balance.

```

import java.util.concurrent.atomic.AtomicInteger;
class BankAccount {
    private final AtomicInteger balance = new AtomicInteger(100);

    public int getBalance() {
        return balance.get();
    }
    Not synchronized
    public void spend(String name, int amount) {
        int initialBalance = balance.get(); } Like before, check if there's enough money. This
        if (initialBalance >= amount) { time, keep a record of the balance.

        boolean success = balance.compareAndSet(initialBalance, initialBalance - amount);
        The balance will NOT be changed if
        the initial balance does not match
        the actual balance right now.
        Pass in the balance from
        when we checked if there
        was enough money.
        If success was false, the
        money was NOT spent.
        Tell Ryan or Monica it
        didn't work and they can
        decide what to do.
    }
}

```

```

File Edit Window Help SorryMonica
% java RyanAndMonicaTest
Ryan is about to spend
Monica is about to spend
Ryan finishes spending
Sorry Monica, you can't buy this
Monica finishes spending

```

Monica was able to start her shopping, but by the time she came to pay, the bank said no. At least they didn't go overdrawn!

`java.util.concurrent` has lots of useful classes and utilities for working with multithreaded code. Take a look at what's there!



So if all these problems
are caused by writing to a shared object, what
if we stopped threads from changing the data in the
shared objects? Is there a way to do that?

Make an object immutable if you're going to share it between threads and you don't want the threads to change its data.

The very best way to know *for sure* that another thread isn't changing your data is to make it impossible to change the data in the object. When an object's data cannot be changed, we call it an **immutable object**.

Writing a class for immutable data

All fields should be FINAL. The value will be set once, in the field declaration or constructor, and cannot be changed afterward.

```
public final class ImmutableData {  
    private final String name;  
    private final int value;  
  
    public ImmutableData(String name, int value) {  
        this.name = name;  
        this.value = value;  
    }  
  
    public String getName() { return name; }  
  
    public int getValue() { return value; }  
}
```

We don't want to allow subclasses that might add mutable values, so make this immutable class final.

All fields need to be initialized once, usually in the constructor.

Immutable objects may have getters, but no setters. The values inside the object should not be changed in any method.



Brain Barbell

There are times when adding the final keyword isn't enough to prevent changes. When do you think that might be the case? We'll give you a clue....



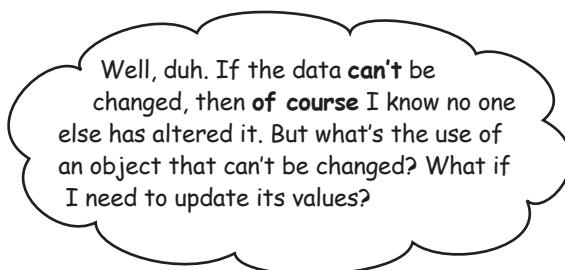
Using immutable objects

It is terribly convenient to be able to change data on a shared object and assume that all the other threads will be able to see these changes.

However, we've also seen that while it's *convenient*, it's not very *safe*.

On the other hand, when a thread is working with an object that cannot be changed, it can make assumptions about the data in that object; e.g., once the thread has read a value from the object, it knows that data can't change.

We don't need to use synchronization or other mechanisms to control who changes the data because it can't change.



Working with immutable objects means thinking in a different way.

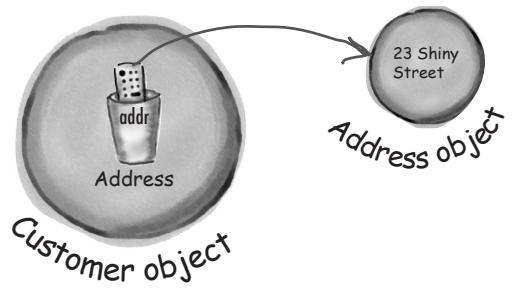
Instead of making changes to the *same* object, we *replace* the old object with a new one. The new object has the updated values, and any threads that need the new values need to use the new object.

What happens to the old object? Well, if it's still being used by something (and it might be—it's perfectly valid sometimes to work with older data), it will hang around on the heap. If it's not being used, it'll be garbage collected, and we don't have to worry about it anymore.

Changing immutable data

Imagine a system that has customers, and that each Customer object has an Address that represents the street address of a customer. If the customer's Address is an immutable object (all its fields are final and the data cannot be changed), how do you change the customer's address when they move?

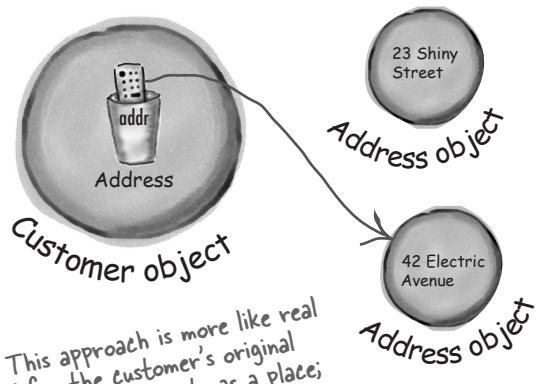
- 1 The Customer has a reference to the original Address object containing the customer's street address data.



- 2 When the customer moves, a brand new Address object is created with the new street address for the customer.



- 3 The Customer object's reference to their address is changed to point to the new Address object.



there are no
Dumb Questions

Q: What happens if other parts of the program have a reference to the old Address?

A: Actually sometimes we want this. Imagine the customer placed an order to be delivered to their original address. We still want the details of that order to have the original address; we don't want it changed to contain the details of the new address.

Once the customer changes their address (and the Customer contains a reference to the new address object), *then* we want new orders to use the new Address object.

This approach is more like real life—the customer's original address still exists as a place; it's just not the place that our customer lives at anymore.



- o o

Wait just a minute! The `Address` object is immutable and doesn't change, but the `Customer` object still has to change.

Absolutely right. If your system has data that changes, those changes do have to happen somewhere. The key idea to take away from this discussion is that not all of the classes in your application have to have data that changes. In fact, we'd argue for minimizing the places where things change. Then, there are far fewer places where you have to think about what happens if multiple threads are making changes at the same time.

There are a number of techniques for working effectively with immutable data classes; we've just scratched the surface here. It is interesting to note that Java 16 introduced *records*, which are immutable data classes provided directly by the language.

Use immutable data classes where you can.
Limit the number of places where data can be changed by multiple threads.

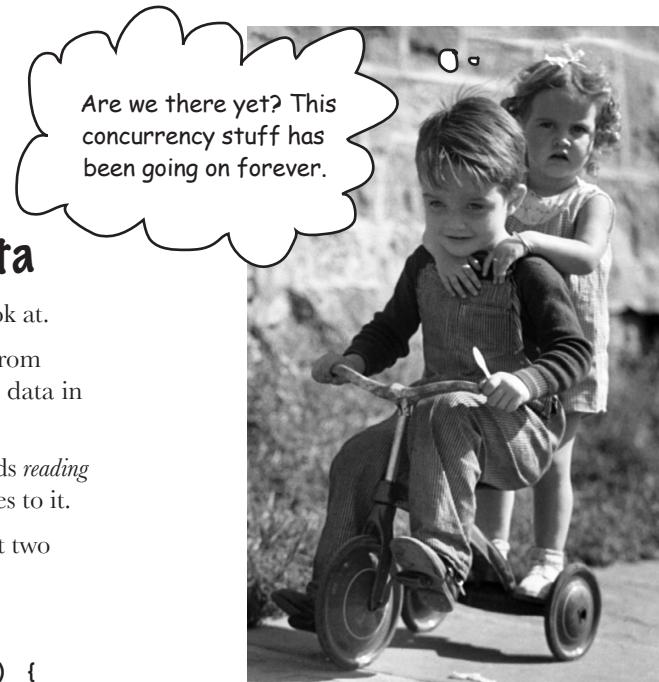
More problems with shared data

We're nearly there, we promise! Just one last thing to look at.

So far we've seen all sorts of problems that can come from many threads writing to the same data. This applies to data in Collections too.

We can even have problems when we have lots of threads *reading* the same data, even if only one thread is making changes to it.

This code has just one thread writing to a collection, but two threads reading it.



```
public class ConcurrentReaders {
    public static void main(String[] args) {
        List<Chat> chatHistory = new ArrayList<>(); ← Stores the Chat
        ExecutorService executor = Executors.newFixedThreadPool(3); objects in an ArrayList,
        for (int i = 0; i < 5; i++) { which is NOT thread
            executor.execute(() -> chatHistory.add(new Chat("Hi there!")));
            executor.execute(() -> System.out.println(chatHistory));
            executor.execute(() -> System.out.println(chatHistory));
        }
        executor.shutdown();
    }

    final class Chat {
        private final String message;
        private final LocalDateTime timestamp; ← Create a writing thread
        public Chat(String message) { that adds to the List, and
            this.message = message;
            timestamp = LocalDateTime.now(); two threads that read
        }
        ← Instances of Chat are immutable.
        public String toString() {
            String time = timestamp.format(ofLocalizedTime(MEDIUM));
            return time + " " + message;
        }
    }
}
```

Making an Object field "final" doesn't guarantee that the data inside that object won't change, just that the reference won't change. String and LocalDateTime are immutable, so this is safe.