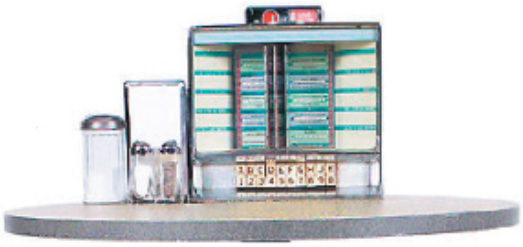
Chapter 11. Collections and Generics: Data Structures

Sorting is a snap in Java. You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). In this chapter, you're

going to get a peek at when Java can save you some typing and figure out the types that you need.

The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've tried to speak with their mic muted on a video call? Sort your pets by number of tricks learned? It's all here...

Tracking song popularity on your jukebox



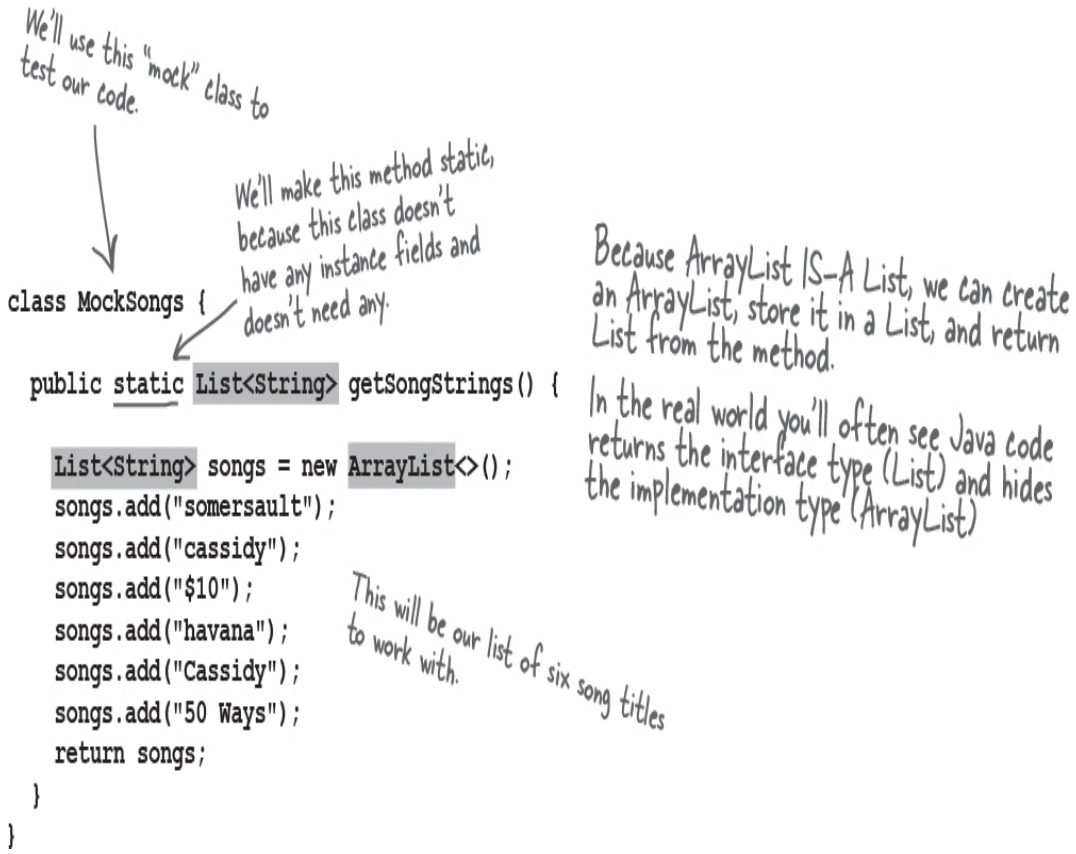
Congratulations on your new job—managing the automated jukebox system at Lou's Diner. There's no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.

Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You're not writing the entire app—some of the other software developers are involved as well, but you're responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. Another programmer will be writing the code to read the song data from a file and put the songs into in List. (In a few chapters you'll learn how to read data from files, and write data to files.) All you're going to get is a List with the song data the jukebox keeps adding to.

Let's not wait for that other programmer to give us the actual file of songs, let's create a small test program to provide us with some sample data we can work with. We've agreed with the other programmer that she'll ultimately

provide a Songs class with a getSongs method we'll use to get the data. Armed with that information, we can write a small class to temporarily "stand in" for the actual code. Code that stands in for other code is often called "**mock**" code.

> **NOTE**
>
> **You'll often want to write some temporary code that stands in for the real code that will come later. This is called "mocking".**



Your first job, sort the songs in alphabetical order

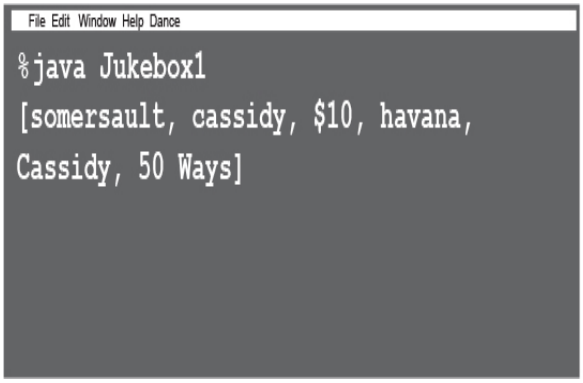
We'll start by creating code that reads in data from the mock Songs class and prints out what it got. Since an ArrayList's elements are placed in the order in which they were added, it's no surprise that the song titles are not yet alphabetized.

```
import java.util.*;

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

  public void go() {
    List<String> songList = MockSongs.getSongStrings();
    System.out.println(songList);
  }
}

// Below is the "mock" code. A stand in for the actual
// I/O code that the other programmer will provide later

class MockSongs {
  public static List<String> getSongStrings() {
    List<String> songs = new ArrayList<>();
    songs.add("somersault");
    songs.add("cassidy");
    songs.add("$10");
    songs.add("havana");
    songs.add("Cassidy");
    songs.add("50 Ways");
    return songs;
  }
}
```

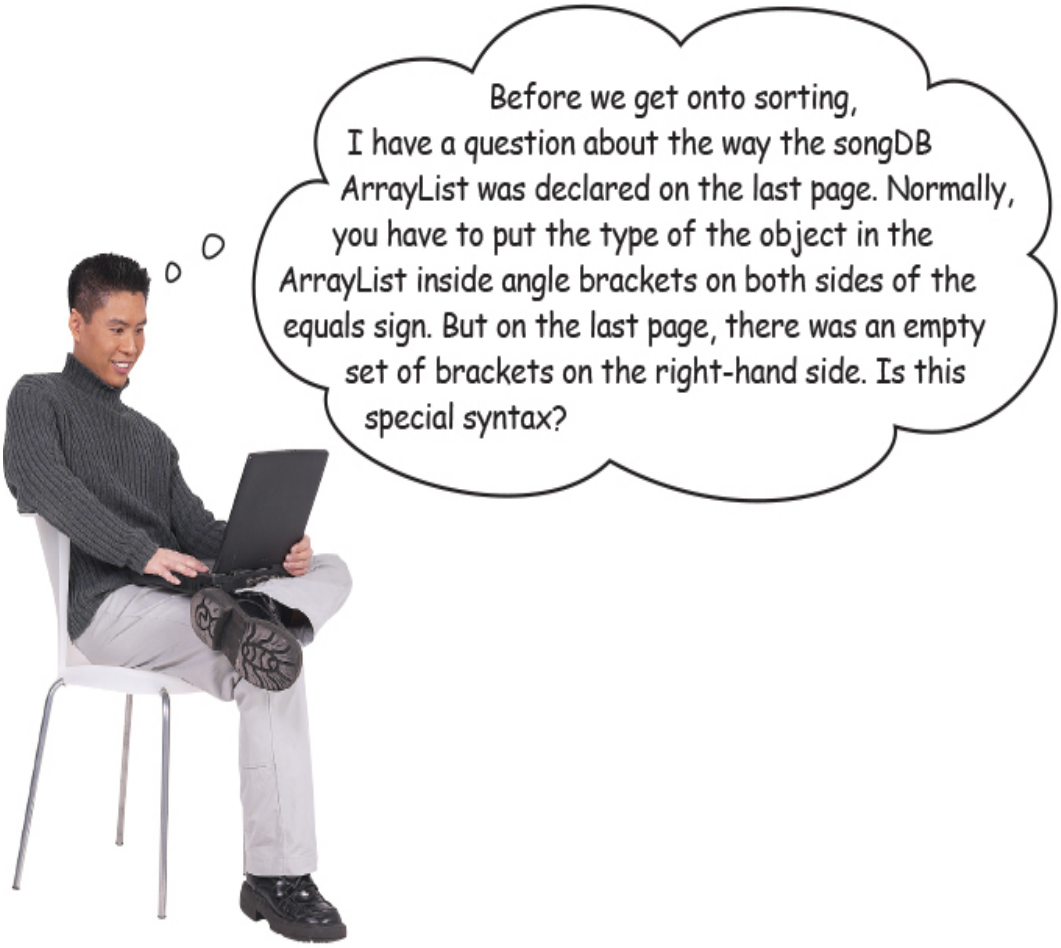
We'll store the song titles in a List of Strings.

Then print the contents of the songList

Nothing special here... just some sample data we can use to work on our sorting code.



Great question! You spotted the diamond operator



So far, we've been declaring our ArrayLists by showing the element type twice:

```
ArrayList<String> songs = new ArrayList<String>();
```

Most of the time, we don't need to say the same thing twice. The compiler can tell from what you wrote on the left-hand side what you probably want on the right-hand side. It uses *type inference* to infer (work out) the type you need.

```
ArrayList<String> songs = new ArrayList<>();
```

No type needed

This syntax is called the diamond operator (because, well, it's diamond-shaped!) and was introduced in Java 7, so it's been around a while and is probably available in your version of Java.

> **NOTE**
>
> **Over time, Java has evolved to remove unnecessary code duplication from its syntax. If the compiler can figure out a type, you don't always need to write it out in full.**

THERE ARE NO DUMB QUESTIONS

Q: Should I be using the diamond operator all the time? Are there any downsides?

A: The diamond operator is "syntactic sugar", which means it's there to make our lives easier as we write (and read) code but it doesn't make any difference to the underlying byte code. So if you're worried about whether using the diamond means something different to using the specific type, don't worry! It's basically the same thing.

However, sometimes you might choose to write out the full type. The main reason you might want to is to help people reading your code. For example, if the variable is declared a long way from where it's initialized, you might want to use the type when you initialize it so you can see clearly what objects go into it.

```
ArrayList<String> songs;
 // lots of code between these lines...
 songs = new ArrayList<String>();
```

Q: Are there any other places the compiler can work out the types for me?

A: Yes! For example, the **`var`** keyword ("Local Variable Type Inference"), which we'll talk about in Appendix B. And another important example, lambda expressions, which we will see later in this chapter.

Q: I saw you were creating an ArrayList but assigning it to a List reference, and that you created an ArrayList but returned a List from the method. Why not just use ArrayList everywhere?

A: One of the advantages of polymorphism is that code doesn't need to know the specific implementation type of an object to work well with it. List is a well known, well understood interface (which we'll see more of in this chapter). Code that is working with an ArrayList doesn't usually

need to know it's an ArrayList. It could be a LinkedList. Or a specialised type of List. Code that's working with the List only needs to know it can call List methods on it (like add(), size() etc). It's usually safer to pass the interface type (i.e. List) around instead of the implementation. That way, other code can't go rooting around inside your object in a way that was never intended.

It also means that should you ever want to change from an ArrayList to a LinkedList, or CopyOnWriteArrayList (see Chapter 18) at a later date, you can without having to change all the places the List is used.

Exploring the java.util API, List and Collections

We know that with an ArrayList, or any List, the elements are kept in the order in which they were added. So we're going to need to sort the elements in the song list. In this chapter, we'll be looking at some of the most important and commonly used collection classes in the java.util package, but for now, let's limit ourselves to two classes: java.util.List and java.util.Collections.

We've been using ArrayList for a while now. Because ArrayList IS-A List, and because many of the methods we're familiar with on ArrayList come from List, we can comfortably transfer most of what we know about working with ArrayLists to List.

The Collections class is known as a "utility" class. It's a class that has a lot of handy methods for working with the various collection types.

Excerpts from the API

java.util.List

sort(**Comparator**) : Sorts this list according to the order induced by the specified **Comparator**.

java.util.Collections

sort(List) : Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

sort(List, **Comparator**) : Sorts the specified list according to the order defined by the **Comparator**.

In the "Real-World"™ there are lots of ways to sort

We don't always want our lists sorted alphabetically. We might want to sort clothes by size, or movies by how many 5-star reviews they get. Java lets you sort the good old fashioned way, alphabetically, and it also lets you create your own custom sorting approaches. Those references you see above to "Comparator" have to do with custom sorting, which we'll get to later this chapter. So for now, let's stick with "natural ordering" (alphabetical).

Since we know we have a List, it looks like we've found the perfect method, **Collections.sort()**.

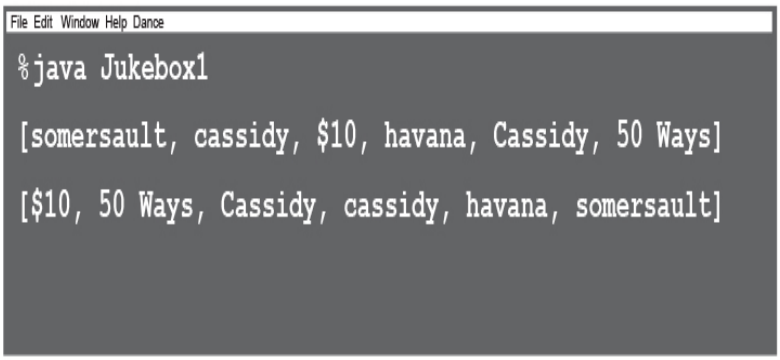
"Natural Ordering", what Java means by alphabetical

Lou wants you to sort songs "alphabetically", but what exactly does that mean? The A-Z part is obvious, but how about lowercase vs uppercase letters? How about numbers and special characters? Well this is another can-

of-worms topic, but Java uses Unicode and for many of us in "the west" that means that numbers sort before upper case letters and uppercase letters sort before lowercase letters, and some special characters sort before numbers and some sort after numbers. That's pretty clear, right? Ha! Well, the upshot is that - by default - sorting in Java happens in what's called "natural order", which is - more or less - alphabetical. Let's take a look at what happens when we sort our list of songs:

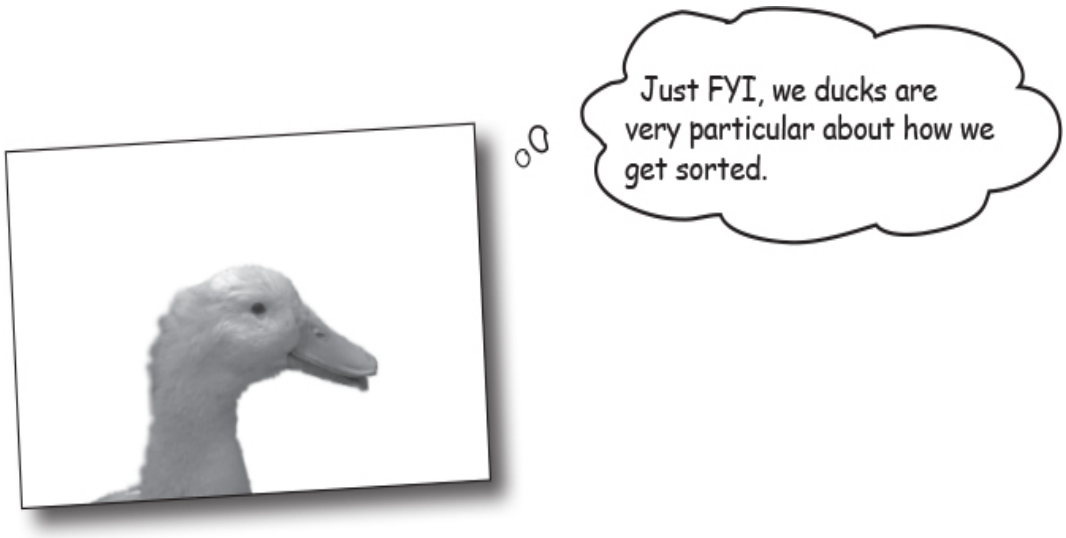
```
  public void go() {
    List<String> songList = MockSongs.getSongStrings();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

Sort our song titles using "natural ordering"



Our songs unsorted, in the order they were added

Our songs sorted. Notice how the special characters, numbers, and uppercase letters got sorted.



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

Now your boss Lou wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so the actual song file will have *three* pieces of information for each song.

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

```
class SongV2 {
  private String title;
  private String artist;
  private int bpm;

  SongV2(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }
  public String getTitle() {
    return title;
  }
  public String getArtist() {
    return artist;
  }
  public int getBpm() {
    return bpm;
  }
  public String toString() {
    return title;
  }
}
```

Three instance variables for the three song attributes in the file.

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

The getter methods for the three attributes.

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

```
class MockSongs {
  public static List<String> getSongStrings() { ... }

  public static List<SongV2> getSongsV2() {
    List<SongV2> songs = new ArrayList<>();
    songs.add(new SongV2("somersault", "zero 7", 147));
    songs.add(new SongV2("cassidy", "grateful dead", 158));
    songs.add(new SongV2("$10", "hitchhiker", 140));

    songs.add(new SongV2("havana", "cabello", 105));
    songs.add(new SongV2("Cassidy", "grateful dead", 158));
    songs.add(new SongV2("50 ways", "simon", 102));
    return songs;
  }
}
```

We made a new method in the MockSongs class to mock the new song data

Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little. The big change is that the List will be of type <SongV2> instead of <String>.

```
import java.util.*;

public class Jukebox2 {
  public static void main(String[] args) {
    new Jukebox2().go();
  }

  public void go() {

    List<SongV2> songList = MockSongs.getSongsV2();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

Change to a List of SongV2 objects instead of Strings.

Call the mock class to load song data into our List of songs.

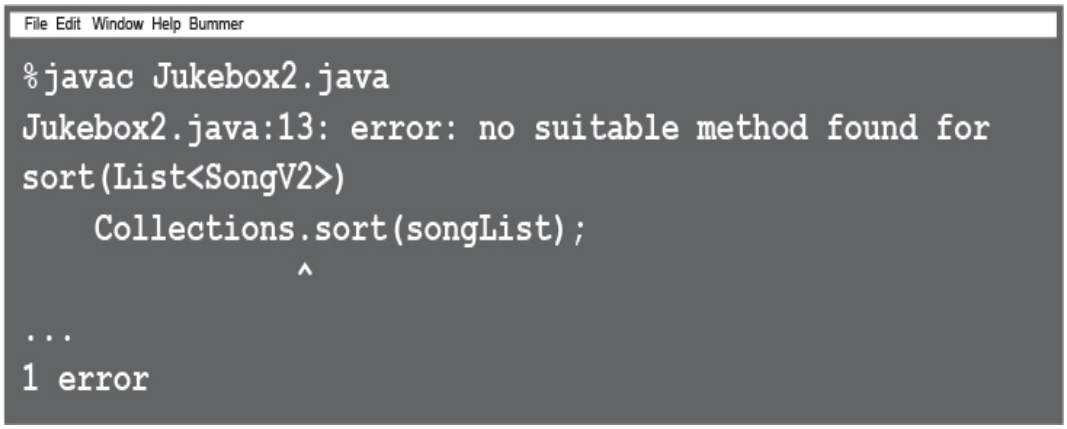
And once again, call the sort method to sort the songs.

It won't compile!

He's right to be curious, something's wrong... the Collections class clearly shows there's a sort() method, that takes a List. It *should* work.

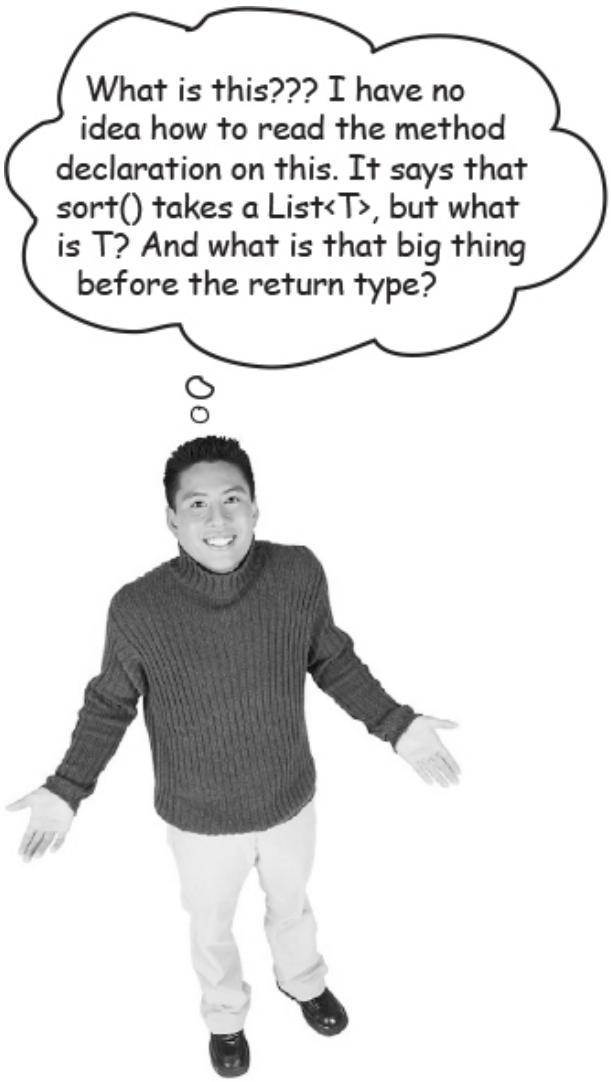
But it doesn't!

The compiler says it can't find a sort method that takes a List<SongV2>, so maybe it doesn't like a List of Song objects? It didn't mind a List<String>, so what's the important difference between Song and String? What's the difference that's making the compiler fail?



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

We'll get into all that a few pages from now, but first, let's find out why the compiler won't even let us pass a SongV2 List to the sort() method.



The sort() method declaration

From the API docs (looking up the java.util.Collections class, and scrolling to the sort() method), it looks like the sort() method is declared... *strangely*.

Or at least different from anything we've seen so far.

That's because the sort() method (along with other things in the whole collection framework in Java) makes heavy use of *generics*. Any time you see something with angle brackets in Java source code or documentation, it means generics—a feature added in Java 5. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort String objects in a List, but not a List of Song objects.

Generics means more type-safety

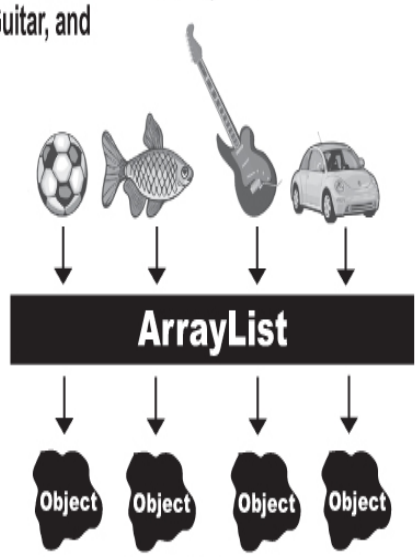
Although generics can be used in other ways, you'll often use generics to write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Without generics the compiler could not care less what you put into a collection, because all collection implementations hold type Object. You could put *anything* in any ArrayList without generics; it's like the ArrayList is declared as ArrayList<Object>.

WITHOUT generics

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

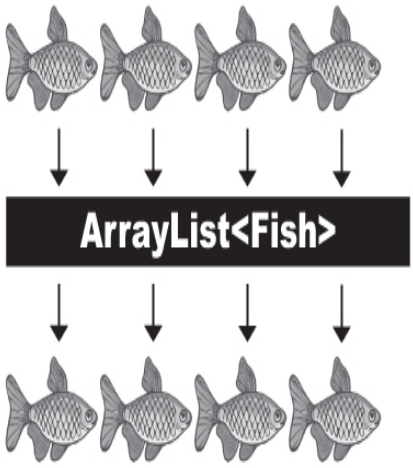
Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects



And come OUT as a reference of type Object

WITH generics

Objects go IN as a reference to only Fish objects



And come out as a reference of type Fish

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

NOTE

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

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

Learning generics

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

1. ① **Creating instances of generic *classes* (like ArrayList)**

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

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

2. ② **Declaring and assigning *variables* of generic types**

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

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

3. ③ **Declaring (and invoking) *methods* that take generic types**

   If you have a method that has as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky

polymorphism issues that are very different from the way you write methods that take plain old arrays.

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

```
void foo(List<Song>list)


x.foo(songList)
```

THERE ARE NO DUMB QUESTIONS

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

A: You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and the things that really need to be generic are collection classes or classes and methods that work on collections. There are some other cases too (like Optional, which we'll see in the next chapter). Generally, generic classes are classes that hold or operate on objects of some other type they don't know about.

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

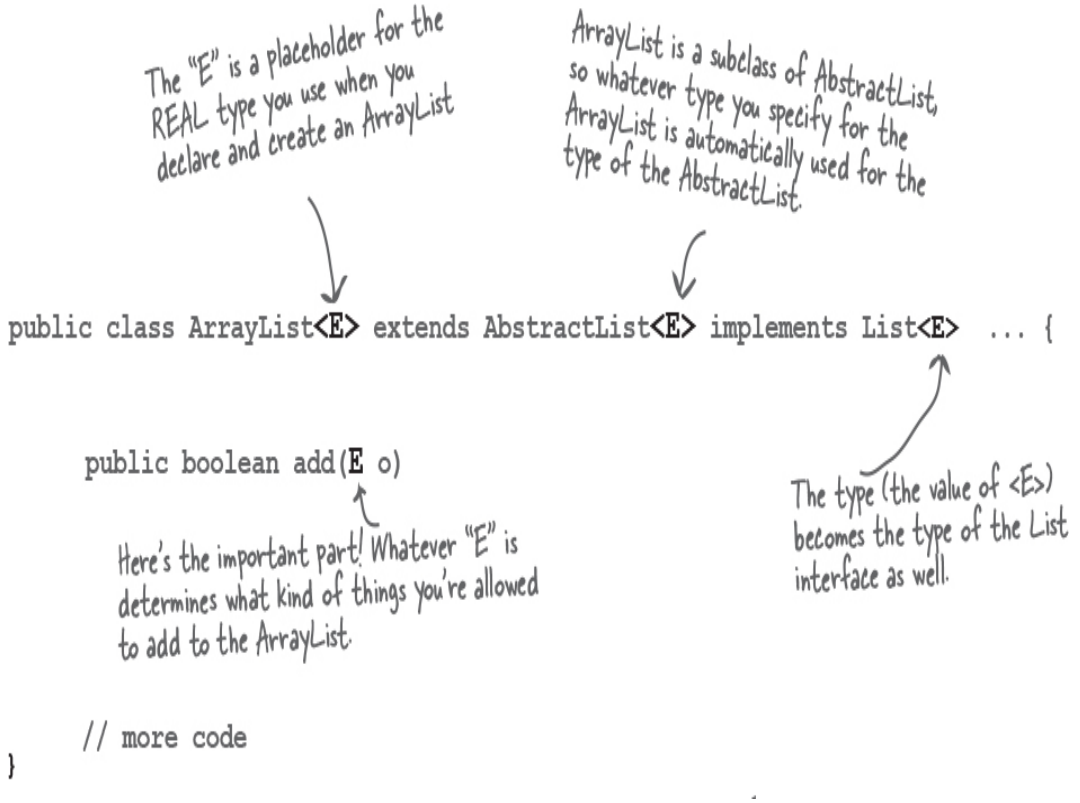
Using generic CLASSES

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

1) The *class* declaration

2) The *method* declarations that let you add elements

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



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

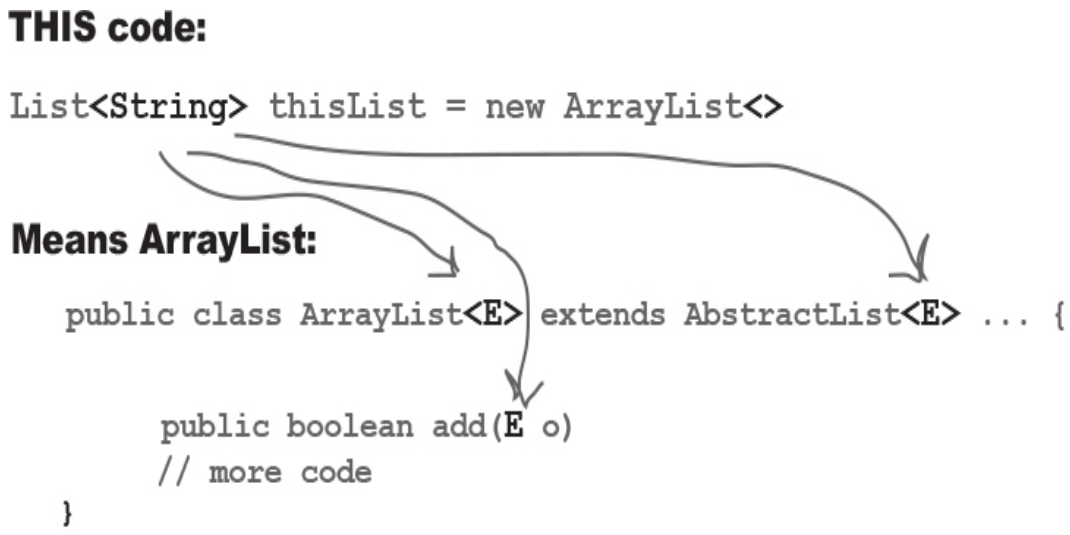
So, new ArrayList<Song> means that "E" becomes "Song", in any method or variable declaration that uses "E".

NOTE

Think of "E" as a stand-in for "the type of element you want this collection to hold and return." (E is for Element.)

Using type parameters with ArrayList

THIS code:



Is treated by the compiler as:

```
public class ArrayList<String> extends AbstractList<String>... {

    public boolean add(String o)
    // more code
}
```

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

THERE ARE NO DUMB QUESTIONS

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

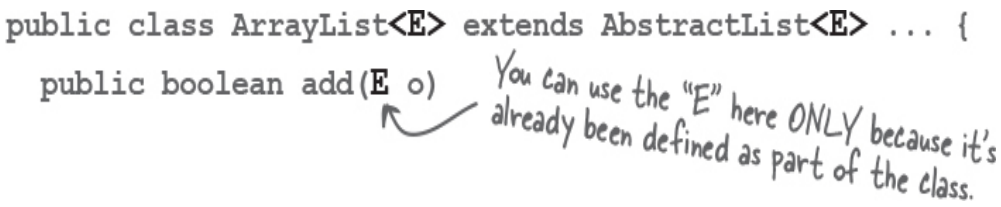
A: You can use anything that's a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But you'll often see single letter used. Another convention is to use "T" ("Type") unless you're specifically writing a collection class, where you'd use "E" to represent the "type of the Element the collection will hold". Sometimes you'll see "R" for "Return type".

Using generic METHODS

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

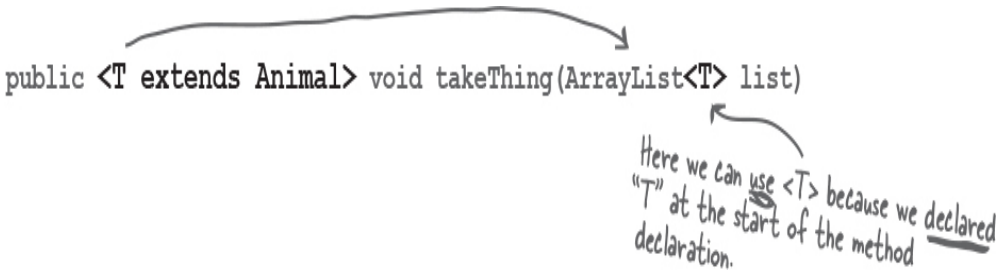
You can use type parameters in a method in several different ways:

1. ① **Using a type parameter defined in the class declaration**



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

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



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

Here's where it gets weird...

This:

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

Is NOT the same as this:

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

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

The first one, where **<T extends Animal>** is part of the method declaration, means that any ArrayList declared of a type that is Animal, or one of

Animal's subtypes (like Dog or Cat), is legal. So you could invoke the top method using an ArrayList<Dog>, ArrayList<Cat>, or ArrayList<Animal>.

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

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

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

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

Remember where we were...

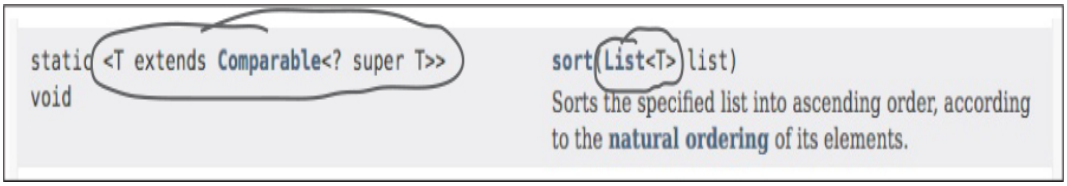
```
File Edit Window Help Bummer
%javac Jukebox2.java
Jukebox2.java:13: error: no suitable method found
for sort(List<SongV2>)
    Collections.sort(songList);
               ^
...
1 error
```

```
import java.util.*;

public class Jukebox2 {
  public static void main(String[] args) {
    new Jukebox2().go();
  }

  public void go() {

    List<SongV2> songList = MockSongs.getSongsV2();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

This is where it breaks! It worked fine when passed in a List<String>, but as soon as we tried to sort a List<SongV2>, it failed.

Revisiting the sort( ) method

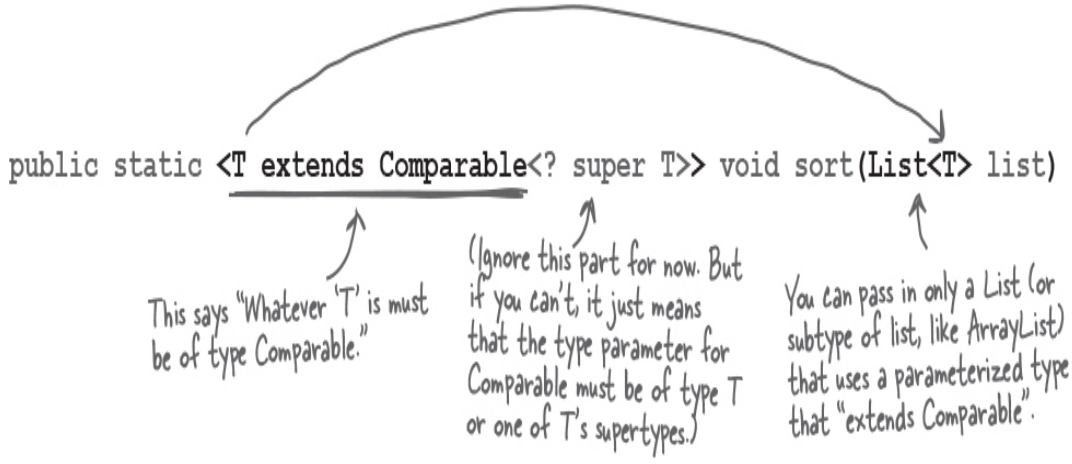
So here we are, trying to read the sort() method docs to find out why it was OK to sort a list of Strings, but not a list of Song objects. And it looks like the answer is...

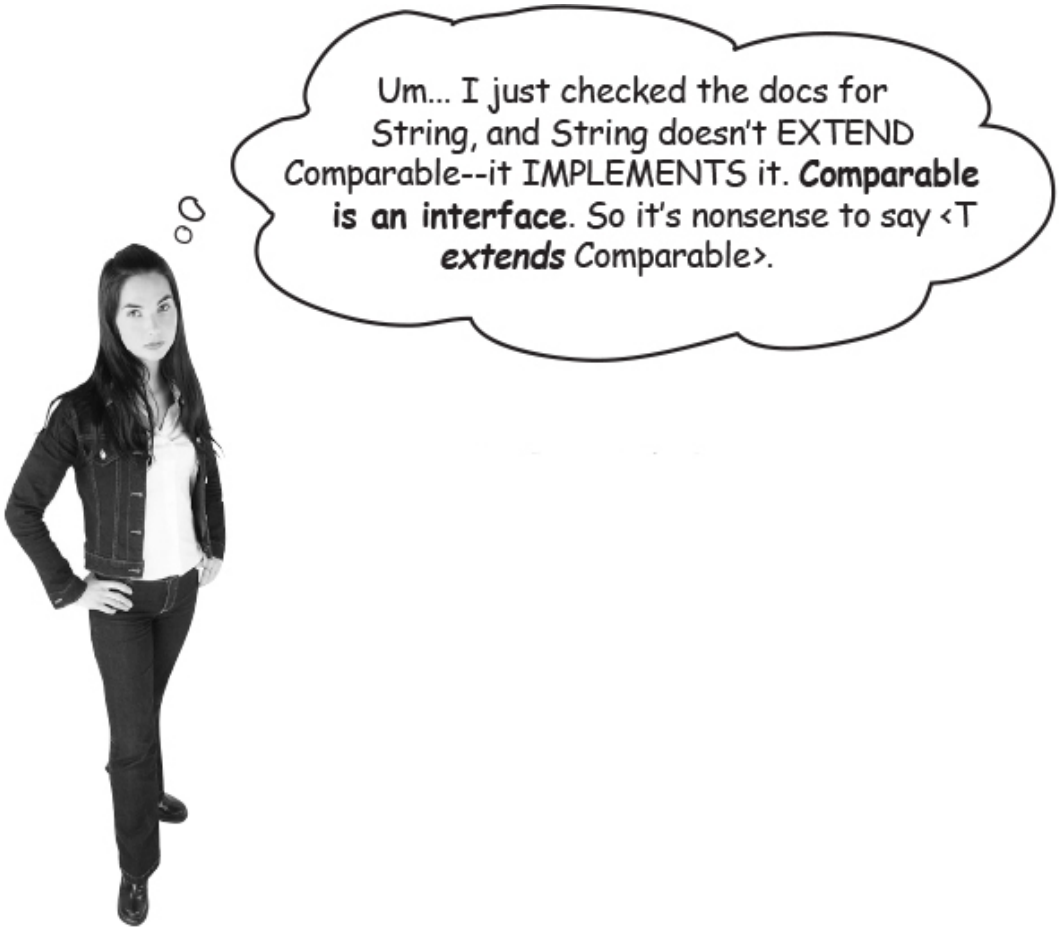


The sort() method can take only lists of Comparable objects.

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

At least not yet...






```
public final class String
    implements java.io.Serializable, Comparable<String>,
```

```
CharSequence {
```

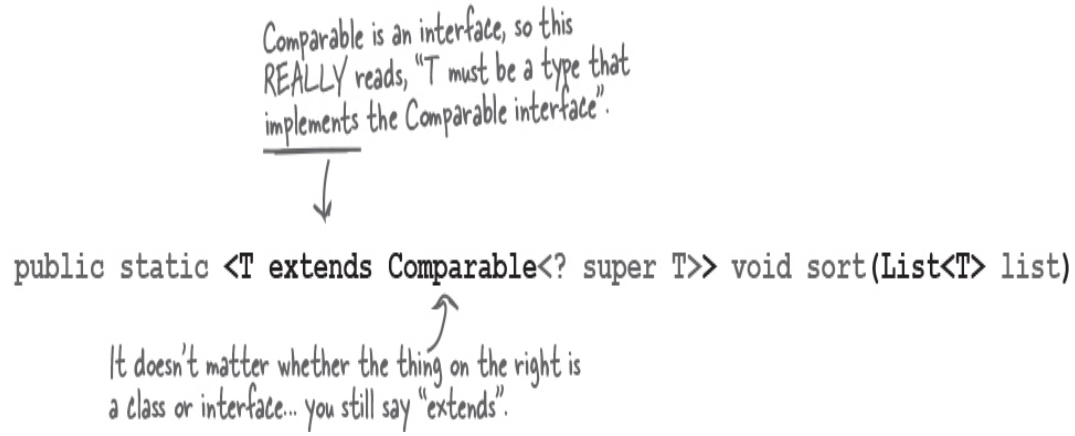
Great point, and one that deserves a full answer! Turn the page...

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

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

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

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



THERE ARE NO DUMB QUESTIONS

Q: Why didn't they just make a new keyword, "is"?

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

In recent years, new "keyword-like" terms have been added to the language, without actually making it a keyword that would trample all over your earlier code. For example, the identifier **`var`**, which we'll talk about in Appendix B, is a *reserved type name*, **not** a keyword. This means any existing code that used var (for example, as a variable name) will not break if it's compiled with a version of Java that supports **`var`**.

Finally we know what's wrong...

The Song class needs to implement Comparable

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

java.lang.Comparable

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

And the method documentation for compareTo() says

```
Returns:
a negative integer if this object is
less than the specified object;
a zero if they're equal;
a positive integer if this object is
greater than the specified object.
```

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

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

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

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

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

SHARPEN YOUR PENCIL

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

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

The new, improved, comparable Song class

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

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

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

This means that SongV3 objects can be compared to other SongV3 objects, for the purpose of sorting.

```
class SongV3 implements Comparable<SongV3> {
  private String title;
  private String artist;
  private int bpm;

  public int compareTo(SongV3 s) {
    return title.compareTo(s.getTitle());
  }

  SongV3(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public int getBpm() {
    return bpm;
  }

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

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

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

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

File Edit Window Help Ambient

```
%java Jukebox3
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
```

We can sort the list, but...

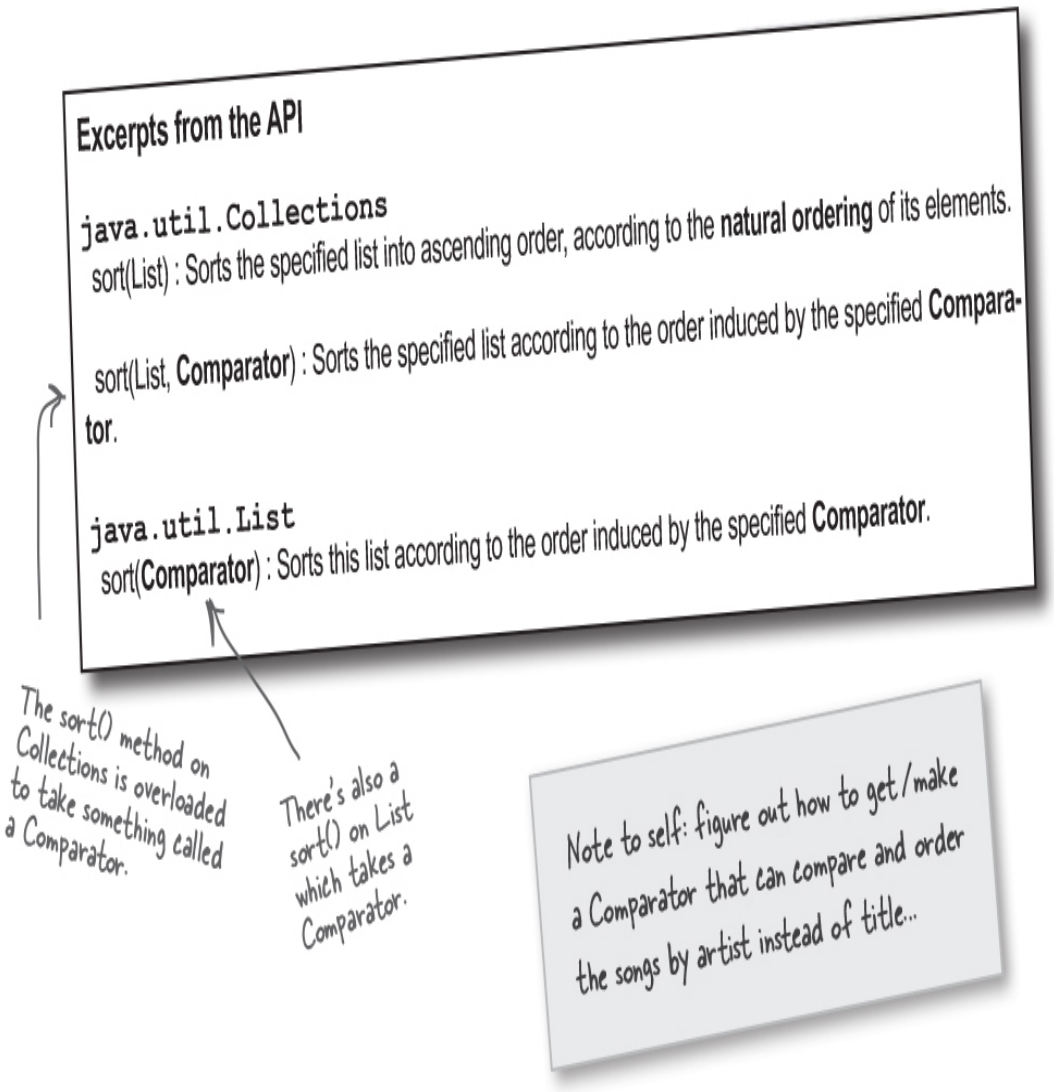
There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

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

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

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

Look at API documentation again. There's a second sort() method on Collections—and it takes a Comparator. There's also a sort method on List that takes a Comparator.



Using a custom Comparator

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

Then all you need to do is call a sort() method that takes a Comparator (Collections.sort or List.sort), which will use this Comparator to put things in order.

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

To summarize, the rules are:

- Invoking the **Collections.sort(List list)** method means the list element's compareTo() method determines the order. The elements in the list MUST implement the **Comparable** interface.
- Invoking **List.sort(Comparator c)** or **Collections.sort(List list, Comparator c)** means the **Comparator**'s compare() method will be used. That means the elements in the list do NOT need to implement the Comparable interface, but if they do, the list element's compareTo() method will NOT be called.

java.util.Comparator

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

NOTE

If you pass a Comparator to the sort() method, the sort order is determined by the Comparator.

If you don't pass a Comparator and the element is Comparable, the sort order is determined by the element's compareTo() method.

THERE ARE NO DUMB QUESTIONS

Q: Why are there two sort methods that take a comparator on two different classes? Which sort method should I use?

A: Both methods that take a comparator, Collections.sort(List, Comparator) and List.sort(Comparator) do the same thing, you can use either and expect exactly the same results.

List.sort was introduced in Java 8, so older code *must* use Collections. sort(List, Comparator).

We use List.sort because it's a bit shorter, and generally you already have the list you want to sort, so it makes sense to call the sort method on that list.

Updating the Jukebox to use a Comparator

We're going to update the Jukebox code in three ways:

1) Create a separate class that implements Comparator (and thus the ***compare()*** method that does the work previously done by ***compareTo()***).

2) Make an instance of the Comparator class.

3) Call the List.sort() method, giving it the instance of the Comparator class.

```
import java.util.*;

public class Jukebox4 {
  public static void main(String[] args) {
    new Jukebox4().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);
    System.out.println(songList);
  }
}

class ArtistCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getArtist().compareTo(two.getArtist());
  }
}
```

Make an instance of the Comparator class.

Invoke sort() on our list, passing it a reference to the new custom Comparator object.

This is a String (the artist)

We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.

```
File Edit Window Help Ambient
%java Jukebox4
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]
```

Unsorted songList

Sorted by title (using the Song's compareTo method)

Sorted by artist name (using ArtistComparator)

SHARPEN YOUR PENCIL

Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the "possible answers" list, to correctly answer the question. Answers are at the end of the chapter.

Possible Answers:

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

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

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

Given the following compilable statements (they both do the same thing):

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

But wait! We're sorting in two different ways!

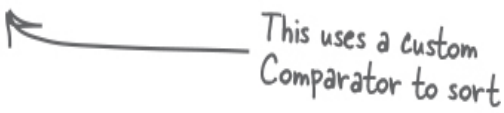
Now we're able to sort the song list two ways:

1) Using Collections.sort(songList), because Song implements **Comparable**

2) Using songLists.sort(artistCompare) because the ArtistCompare class implements **Comparator**.

While our new code allows us to sort songs by title and by artist, it is reminiscent of Frankenstein's monster, cobbled together bit by bit.

```
public void go() {
 List<SongV3> songList = MockSongs.getSongsV3();
 System.out.println(songList);

 Collections.sort(songList);
 System.out.println(songList);

 ArtistCompare artistCompare = new ArtistCompare();
 songList.sort(artistCompare);
 System.out.println(songList);
}
```



NOTE

A better approach would be to handle all of the sorting definitions in classes that implement Comparator

THERE ARE NO DUMB QUESTIONS

Q: So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

A: That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

Q: But why doesn't *every* class implement Comparable?

A: Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And there's no problem if you *don't* implement Comparable, since a programmer can compare anything in any way that they choose using their own custom Comparator.

Sorting using only Comparators

Having Song implement Comparable and creating a custom Comparator for sorting by Artist absolutely works, but it's confusing to rely on two different mechanisms for our sort. It's much clearer if our code uses the same technique to sort, regardless of how Lou wants his songs sorted. The code below has been updated to use Comparators for sorting by both Title and Artist, the new code is in bold.

```
public class Jukebox5 {
  public static void main(String[] args) {
    new Jukebox5().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    TitleCompare titleCompare = new TitleCompare();
    songList.sort(titleCompare);
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);
    System.out.println(songList);
  }
}
class TitleCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
}
class ArtistCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getArtist().compareTo(two.getArtist());
  }
}

// more specialized Comparator classes could go here,
// e.g. BpmCompare
```

Make an instance of the Comparator class and use the sort() method on List

This is the new class that implements Comparator

Just the code that matters

The Jukebox class does have a lot of code that's needed for sorting. Let's zoom in on one of the Comparator classes we wrote for Lou. The first thing to notice is that all we really want to sort our collection is the one line of code in the middle of the class. The rest of the code is just the long-winded syntax that's necessary to let the compiler know what type of class this is and which method it implements.

CODE UP CLOSE

```
class TitleCompare implements Comparator<Song> {
   public int compare(Song one, Song two) {
     return one.getTitle().compareTo(two.getTitle());
   }
}
```

The one line of code that's doing all the work

There's more than one way to declare small pieces of functionality like this. One approach is inner classes, which we'll look at in a later chapter. You can even declare the inner class right where you use it (instead of at the end of your class file), this is sometimes called an "argument-defined anonymous inner class", sounds fun already:

```
songList.sort(new Comparator<SongV3>() {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
});
```

While this lets us declare the sorting logic in exactly the location we need it (where we call the sort method, instead of in a separate class), there's still a lot of code there for saying "sort by title please").

RELAX

We're not going to learn how to write "argument-defined anonymous inner classes"!

We just wanted you to see this example, in case you stumble across it in Real Code.

What do we REALLY need in order to sort?

```
public class Jukebox5 {
  public void go() {
1   List<SongV3> songList = MockSongs.getSongsV3();
    ...
    TitleCompare titleCompare = new TitleCompare();
2   songList.sort(titleCompare);
    ...
  }
}
class TitleCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
}
```

The compiler knows the List contains SongV3 objects

The compiler understands that sort() expects a Comparator for SongV3 objects

Let's take a look at the API documentation for the sort method on List:

default void	sort(Comparator<? super E> c)	Sorts this list according to the order induced by the specified Comparator.

If we were to explain out loud the following line of code:

```
songList.sort(titleCompare);
```

We could say:

NOTE

"Call the sort method on the list of songs (1) and pass it a reference to a Comparator object which is designed specifically to sort Song objects (2)."

If we're honest, we could say all that without even looking at the TitleCompare class. We can work it all out just by looking at the documentation for sort and the type of the List that we're sorting!

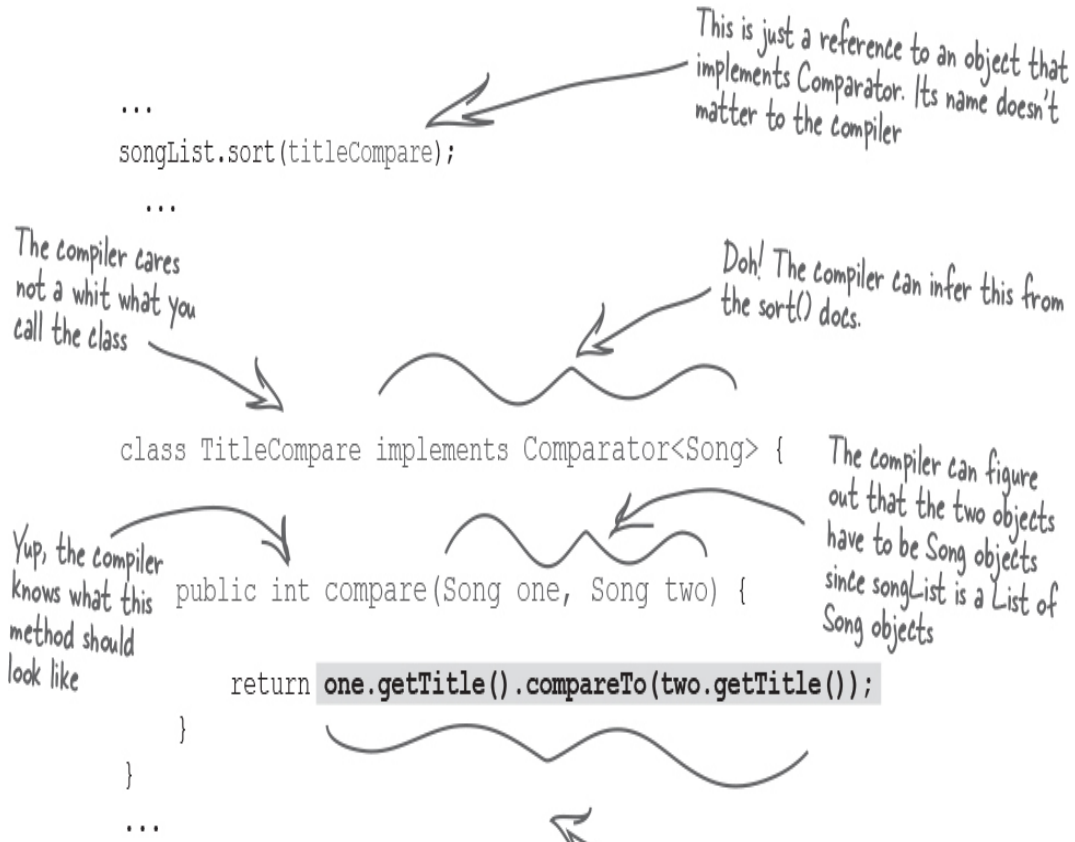
BRAIN BARBELL

Do you think the compiler cares about the name "TitleCompare"? If the class was called "FooBar" instead, would the code still work?

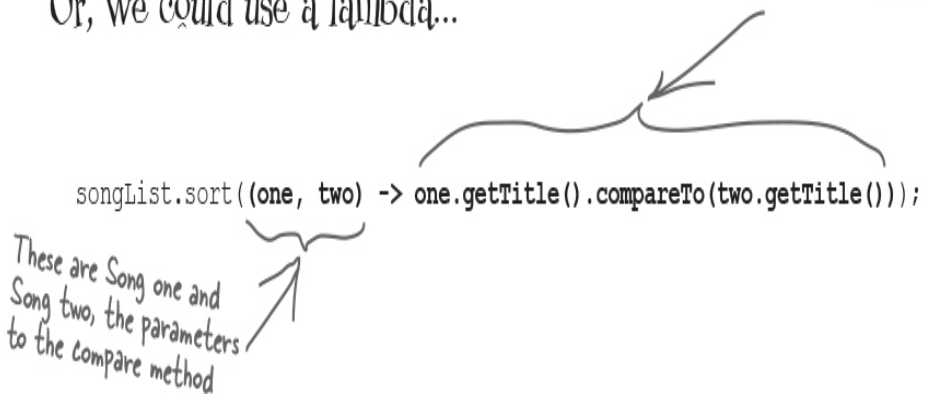
Wouldn't it be wonderful if the code that described HOW you want to sort your Songs wasn't so far away from the sort method? And wouldn't it be great if you didn't have to write a bunch of code the compiler could probably figure out on its own...

Enter lambdas! Leveraging what the compiler can infer

We could write a whole bunch of code to say how to sort a list (like we have been doing)...



Or, we could use a lambda...



BRAIN BARBELL

What do you think would happen if Comparator needed you to implement more than one method? How much could the compiler fill in for you?

Where did all that code go?

To answer this question, let's take a look at the API documentation for the Comparator interface.

Method Summary

int	compare(T o1, T o2) Compares its two arguments for order.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.

Remember from Chapter 8 that every class and interface inherits methods from class Object, and that equals() is implemented in class Object.

Because equals has been implemented by Object, if we create a custom comparator, we know we only *need* to implement the compare method.

We also know exactly the shape of that method - it has to return an int, and it takes two arguments of type T (remember generics?). Our lambda expression implements the compare() method, without having to declare the class or the method, only the details of what goes into the body of the compare() method.

Some interfaces have only ONE method to implement

With interfaces like Comparator, we only have to implement a ***single abstract method***, SAM for short. These interfaces are so important that they have several special names:

SAM Interfaces a.k.a. **Functional Interfaces**

If an interface has only one method that needs to be implemented, that interface can be implemented as a ***lambda expression***. You don't need to create a whole class to implement the interface; the compiler knows what the class and method would look like. What the compiler *doesn't* know is the logic that goes *inside* that method.

RELAX

We'll look at lambda expressions and functional interfaces in much more detail in the next chapter. For now, back to Lou's diner...

Updating the Jukebox code with Lambdas

```
import java.util.*;

public class Jukebox6 {
  public static void main(String[] args) {
    new Jukebox6().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    songList.sort((one, two) -> one.getArtist().compareTo(two.getArtist()));
    System.out.println(songList);
  }
}
```

Here's our lambda expression in action – no need to create a custom Comparator class, just put the sorting logic right inside sort method call

You can tell what the list will be sorted by, just by looking at the field used in the lambda

The output is exactly the same as when we used Comparator classes, but our code was much shorter

```
File Edit Window Help Ambient
%java Jukebox6
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]
```

SHARPEN YOUR PENCIL

How could you sort the songs differently?

Write lambda expressions to sort the songs in these ways (the answers are at the end of the chapter):

- Sort by BPM

- Sort by title in descending order

Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the program will create the output shown.

SHARPEN YOUR PENCIL

```
import ____________________;

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

  public void go() {
    List_________________ 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._______(____->____________);
    System.out.println("by name:\n" + mountains);

    __________._______(____->____________);
    System.out.println("by height:\n" + mountains);
  }
}

class Mountain {
  ____________________;
  ____________________;

  _____________________ {
    ____________________;
    ____________________;
  }
  _____________________________ {
    ____________________________________;
  }
}
```

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]
```

Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great, now we know how to sort on both *title* and *artist*. But there's a new problem we didn't notice with a test sample of the jukebox songs—***the sorted list contains duplicates.***

Unlike the mock code, Lou's real jukebox application appears to just keeps writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The SongListMore.txt jukebox text file is an example. It's a complete record of every song that was played, and might contain the same song multiple times.

```
File Edit Window Help TooManyNotes
%java Jukebox7

[somersault: zero 7, cassidy: grateful dead, $10: hitchhiker,
havana: cabello, $10: hitchhiker, cassidy: grateful dead, 50
ways: simon]

[$10: hitchhiker, $10: hitchhiker, 50 ways: simon, cassidy:
grateful dead, cassidy: grateful dead, havana: cabello,
somersault: zero 7]

[havana: cabello, cassidy: grateful dead, cassidy: grateful
dead, $10: hitchhiker, $10: hitchhiker, 50 ways: simon,
somersault: zero 7]
```

Before sorting.

After sorting by song title

After sorting by artist name

Note that we changed the Song's toString to output the title and the artist

SongListMore.txt

This is what the actual song data file looks like.

```
somersault, zero 7, 147
cassidy, grateful dead, 158
$10, hitchhiker, 140
havana, cabello, 105
$10, hitchhiker, 140
cassidy, grateful dead, 158
50 ways, simon, 102
```

The SongListMore text file now has duplicates in it, because the jukebox machine is writing every song played, in order.

To get the output above, we wrote a MockMoreSongs class with a getSongs() method that returned a List that has all the same entries as this text file

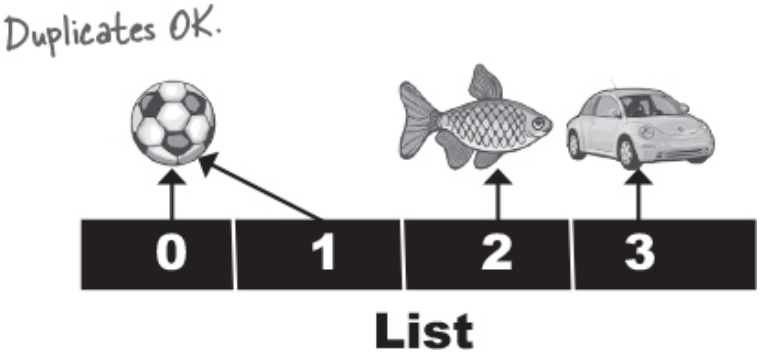
We need a Set instead of a List

From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**. ArrayList is a **List**, but it looks like ***Set*** is exactly what we need.

- **LIST - when *sequence* matters**

  Collections that know about ***index position.***

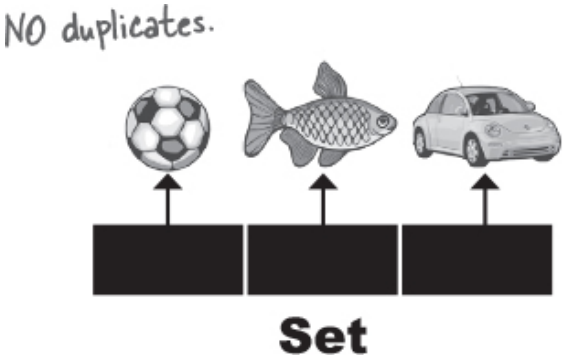
  Lists know where something is in the list. You can have more than one element referencing the same object.



- **SET - when *uniqueness* matters**

  Collections that ***do not allow duplicates.***

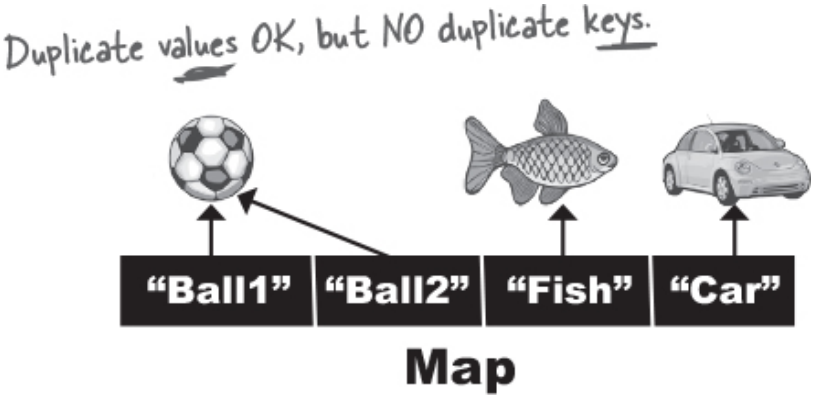
  Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).



- **MAP - when *finding something by key* matters**

  Collections that use ***key-value pairs.***

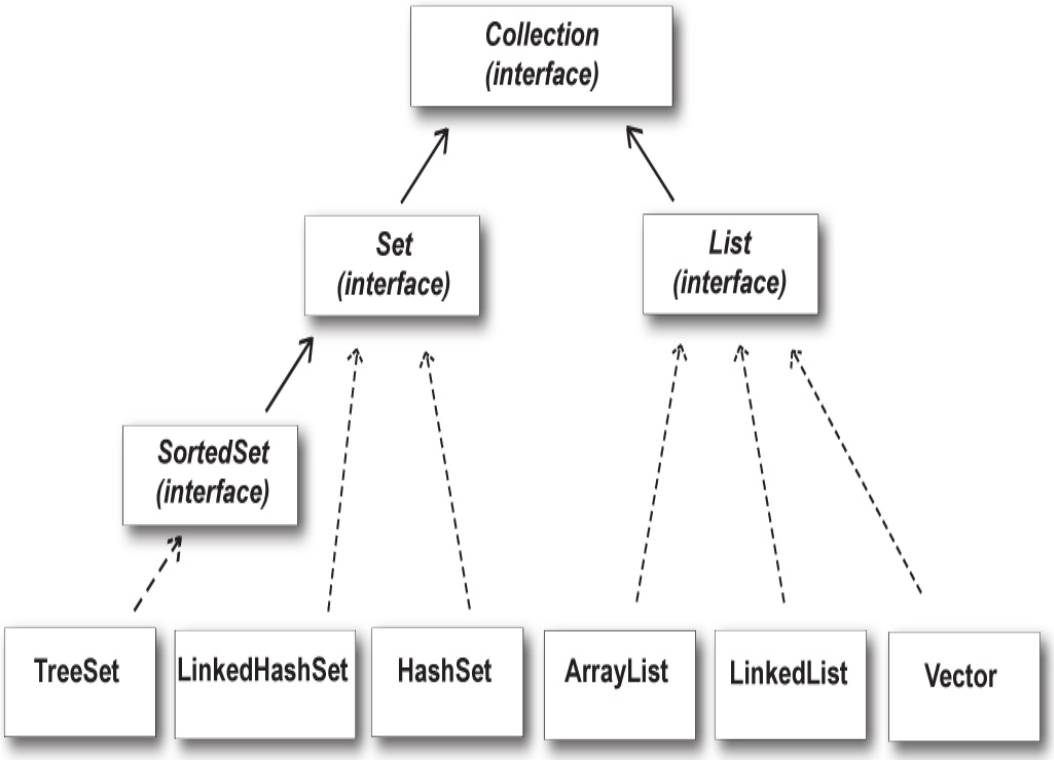
  Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. A key can be any object.

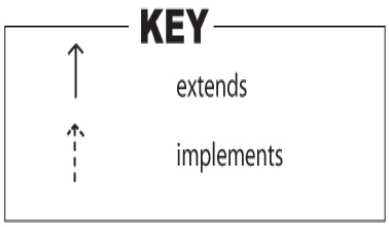


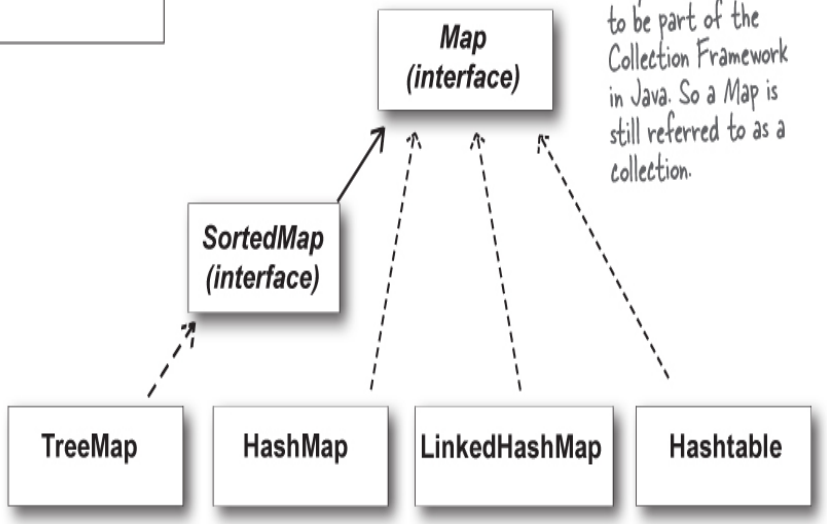
The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework" (also known as the "Collection API"). So Maps are still collections, even though they don't include java.util.Collection in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)







Maps don't extend from java.util.Collection, but they're still considered to be part of the Collection Framework in Java. So a Map is still referred to as a collection.

Using a HashSet instead of ArrayList

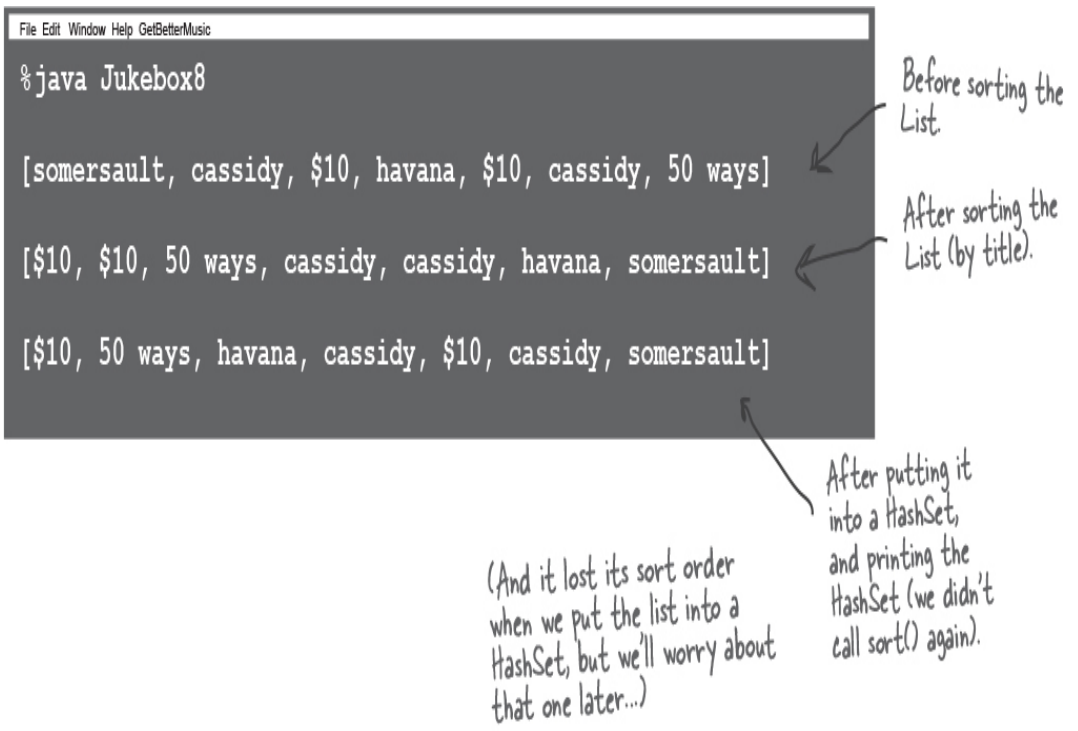
We updated the Jukebox code to put the songs in a HashSet to try to eliminate our duplicate songs. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions.

```
import java.util.*;

public class Jukebox8 {
  public static void main(String[] args) {
    new Jukebox8().go();
  }

  public void go() {
    List<SongV3> songList = MockMoreSongs.getSongsV3();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    Set<SongV3> songSet = new HashSet<>(songList);
    System.out.println(songSet);
  }
}
```

We created a MockMoreSongs class to return a List of SongV3 objects that contain the same values as SongListMore.txt

HashSet has a constructor that takes a Collection, and it will create a set with all the items from that collection

We want the Set to hold SongV3 objects. HashSet IS-A Set, so we can store the HashSet in this Set variable



NOTE

The Set didn't help!!

We still have all the duplicates!

What makes two objects equal?

To figure out why using a Set didn't remove the duplicates, we have to ask—what makes two Song references duplicates? They must be considered ***equal***. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

This brings up a key issue: *reference* equality vs. *object* equality.

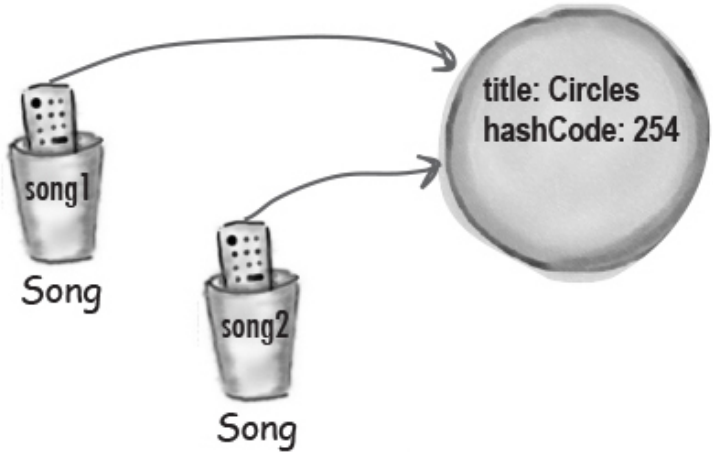
- **Reference equality**

  Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the **hashCode()** method on both references, you'll get the same result. If you don't override the hashCode() method, the default behavior (remember, you inherited this from class Object) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the == operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.

If two objects foo and bar are equal, foo.equals(bar) and *bar.equals(foo)* must be *true*, and both *foo* and *bar* must return the same value from *hash-Code()*. For a Set to treat two objects as duplicates, you must override the hashCode() and equals() methods inherited from class Object, so that you can make two different objects be viewed as equal.




```
if (song1 == song2) {
   // both references are referring
```

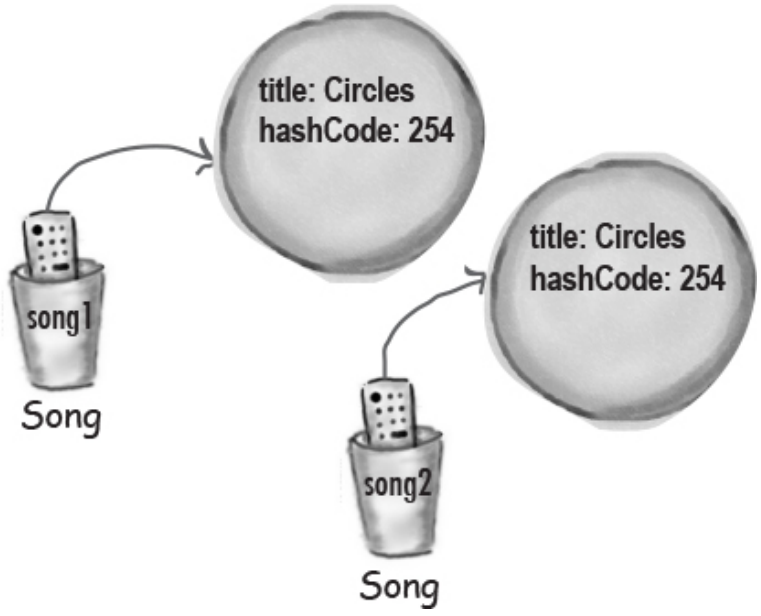
```
   // to the same object on the heap
}
```

- **Object equality**

  Two references, two objects on the heap, but the objects are considered *meaningfully equivalent.*

  If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the **hashCode()** and **equals()** methods inherited from class Object.

  As we said above, if you *don't* override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. So you must override hashCode() to be sure that two equivalent objects return the same hashcode. But you must also override equals() so that if you call it on *either* object, passing in the other object, always returns ***true***.




```
if (song1.equals(song2) && song1.hashCode() == song2.hashCode()) {
   // both references are referring to either a
```

```
    // a single object, or to two objects that are equal
}
```

How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a HashSet, it calls the object's hashCode method to determine where to put the object in the Set. But it also compares the object's hash code to the hash code of all the other objects in the HashSet, and if there's no matching hash code, the HashSet assumes that this new object is *not* a duplicate.

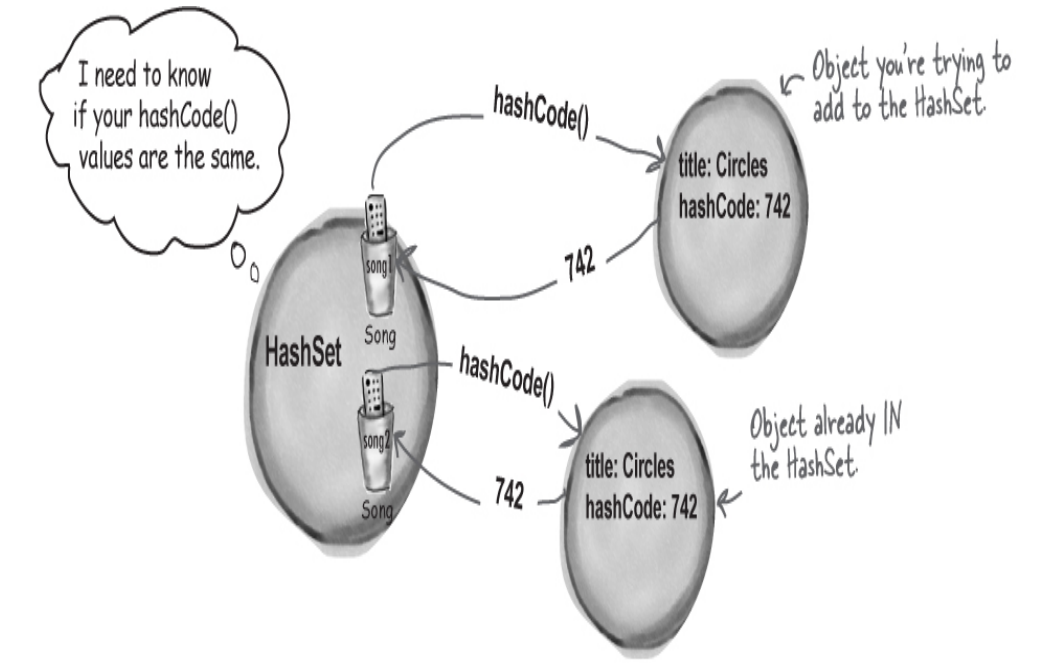
In other words, if the hash codes are different, the HashSet assumes there's no way the objects can be equal!

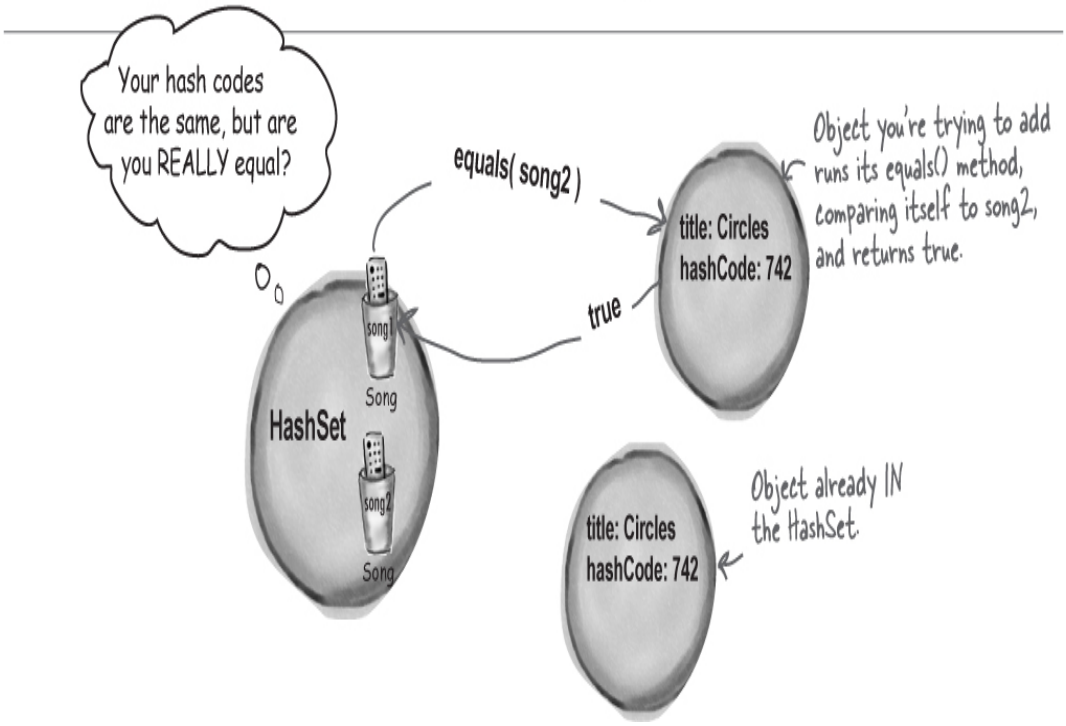
So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hash code might *not* be equal (more on this on the next page), so if the HashSet finds a matching hash code for two objects —one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hash code-matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.


I need to know if your hashCode() values are the same.
hashCode()
Object you're trying to add to the HashSet.
title: Circles
hashCode: 742
742
HashSet
song1
Song
hashCode()
song2
Song
742
title: Circles
hashCode: 742
Object already IN the HashSet.


Your hash codes are the same, but are you REALLY equal?
equals(song2)
Object you're trying to add runs its equals() method, comparing itself to song2, and returns true.
title: Circles
hashCode: 742
true
HashSet
song1
Song
song2
Song
title: Circles
hashCode: 742
Object already IN the HashSet.

The Song class with overridden hashCode() and equals()

```
class SongV4 implements Comparable<SongV4> {
  private String title;
  private String artist;
  private int bpm;

  public boolean equals(Object aSong) {
    SongV4 other = (SongV4) aSong;
    return title.equals(other.getTitle());
  }

  public int hashCode() {
    return title.hashCode();
  }

  public int compareTo(SongV4 s) {
    return title.compareTo(s.getTitle());
  }

  SongV4(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public int getBpm() {
    return bpm;
  }

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

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here... the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable (title).

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.

File Edit Window Help HashingItOut

```
%java Jukebox9

[somersault, cassidy, $10, havana, $10,
cassidy, 50 ways]

[$10, $10, 50 ways, cassidy, cassidy, havana,
somersault]

[havana, $10, 50 ways, cassidy, somersault]
```

Java Object Law For hashCode() and equals()

The API docs for class Object state the rules you MUST follow:

- **If two objects are equal, they MUST have matching hash codes.**
- **If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).**
- **If two objects have the same hash code value, they are NOT required to be equal. But if they're equal, they MUST have the same hash code value.**
- **So, if you override equals(), you MUST override hashCode().**
- **The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.**
- **The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.**

a.equals(b) **must also mean that a.hashCode() == b.hashCode()**

But *a.hashCode() == b.hashCode()* does NOT have to mean *a.equals(b)*

THERE ARE NO DUMB QUESTIONS

Q: How come hash codes can be the same even if objects aren't equal?

A: HashSets use hash codes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hashcode as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one..." the HashSet gets the hashcode value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hashcode #742 is stored...", and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use HashSets effectively. In reality, developing a good hashing algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that hash codes can be the same without necessarily guaranteeing that the objects are equal, because the "hashing algorithm" used in the hashCode() method might happen to return the same value for multiple objects. And yes, that means that multiple objects would all land in the same hash code bucket in the HashSet, but that's not the end of the world. The HashSet might be a little less efficient, because if the HashSet finds more than one object in the same hash code bucket, it has to use the equals() on all those objects to see if there's a perfect match.

If we want the set to stay sorted, we've got TreeSet

TreeSet is similar to HashSet in that it prevents duplicates. But it also *keeps* the list sorted. It works just like the sort() method in that if you make a TreeSet without giving it a Comparator, the TreeSet uses each object's compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead.

The downside to TreeSet is that if you don't *need* sorting, you're still paying for it with a small performance hit. But you'll probably find that the hit is almost impossible to notice for most apps.

```
public class Jukebox10 {
  public static void main(String[] args) {
    new Jukebox10().go();
  }

  public void go() {
    List<SongV4> songList = MockMoreSongs.getSongsV4();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    Set<SongV4> songSet = new TreeSet<>(songList);
    System.out.println(songSet);
  }
}
```

NOTE

Create a TreeSet instead of HashSet. The TreeSet will use SongV4's compareTo() method to sort the items in songList.

If we want the TreeSet to sort on something different (i.e. to NOT use SongV4's compareTo() method), we need to pass in a Comparator (or a lambda) to the TreeSet constructor. Then we'd use songSet.addAll() to add the songList values into the TreeSet.

```
Set<SongV4> songSet = new TreeSet<>((o1, o2) -> o1.getBpm() -
o2.getBpm());
```

```
songSet.addAll(songList);
```

> **NOTE**
>
> Yep, another lambda for sorting. This one sorts by BPM. Remember, this lambda implements Comparator

What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it*.

SHARPEN YOUR PENCIL

Look at this code. Read it carefully, then answer the questions below. (Note: there are no syntax errors in this code.)

```
import java.util.*;

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

  public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");

    Set<Book> tree = new TreeSet<>();
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}

class Book {
  private String title;
  public Book(String t) {
    title = t;
  }
}
```

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

_____

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

_____

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

_____

TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the objects should be sorted. You have to tell the TreeSet *how*.

To use a TreeSet, one of these things must be true:

- **The elements in the list must be of a type that implements *Comparable***

  The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type. The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>() the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and... can't.

  OR

- **You use the TreeSet's overloaded constructor that takes a *Comparator***

  TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom

Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

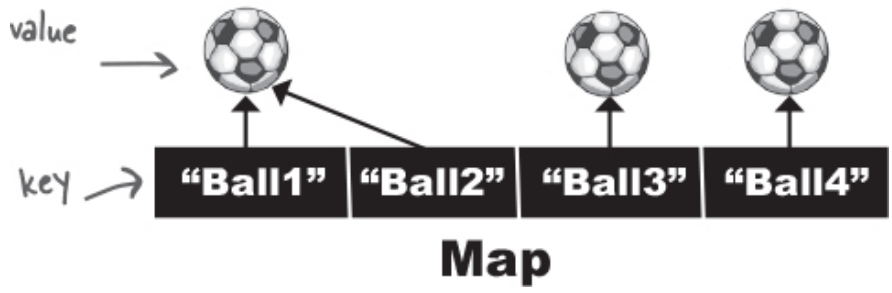
```
class Book implements Comparable<Book> {
  private String title;
  public Book(String t) {
    title = t;
  }

  public int compareTo(Book other) {
    return title.compareTo(other.title);
  }
}

class BookCompare implements Comparator<Book> {
  public int compare(Book one, Book two) {
    return one.title.compareTo(two.title);
  }
}
public class TestTreeComparator {
  public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");
    BookCompare bookCompare = new BookCompare();
    Set<Book> tree = new TreeSet<>(bookCompare);
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}
```

You could use a lambda instead of declaring a new Comparator class

We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Keys can be any Java object (or, through autoboxing, a primitive), but you'll often see String keys (i.e. property names) or Integer keys (representing unique IDs, for example).



Each element in a Map is actually TWO objects—a key and a value. You can have duplicate values, but NOT duplicate keys.

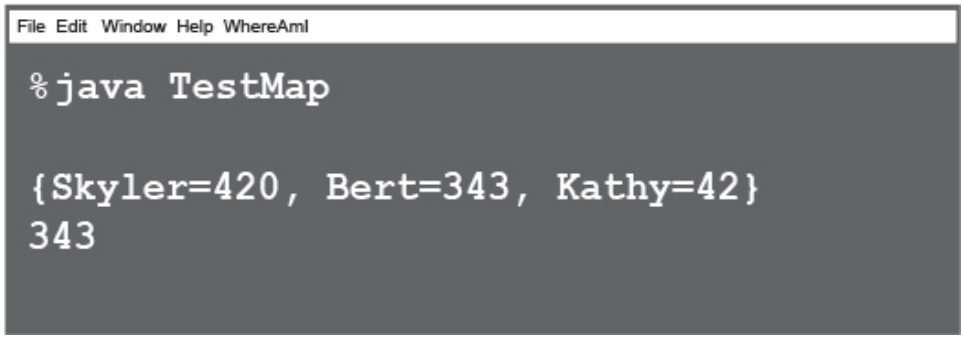
Map example

```
public class TestMap {
  public static void main(String[] args) {
    Map<String, Integer> scores = new HashMap<>();

    scores.put("Kathy", 42);
    scores.put("Bert", 343);
    scores.put("Skyler", 420);

    System.out.println(scores);
    System.out.println(scores.get("Bert"));
  }
}
```

HashMap needs TWO type parameters—one for the key and one for the value.

Use put() instead of add(), and now of course it takes two arguments (key, value).

The get() method takes a key, and returns the value (in this case, an Integer).

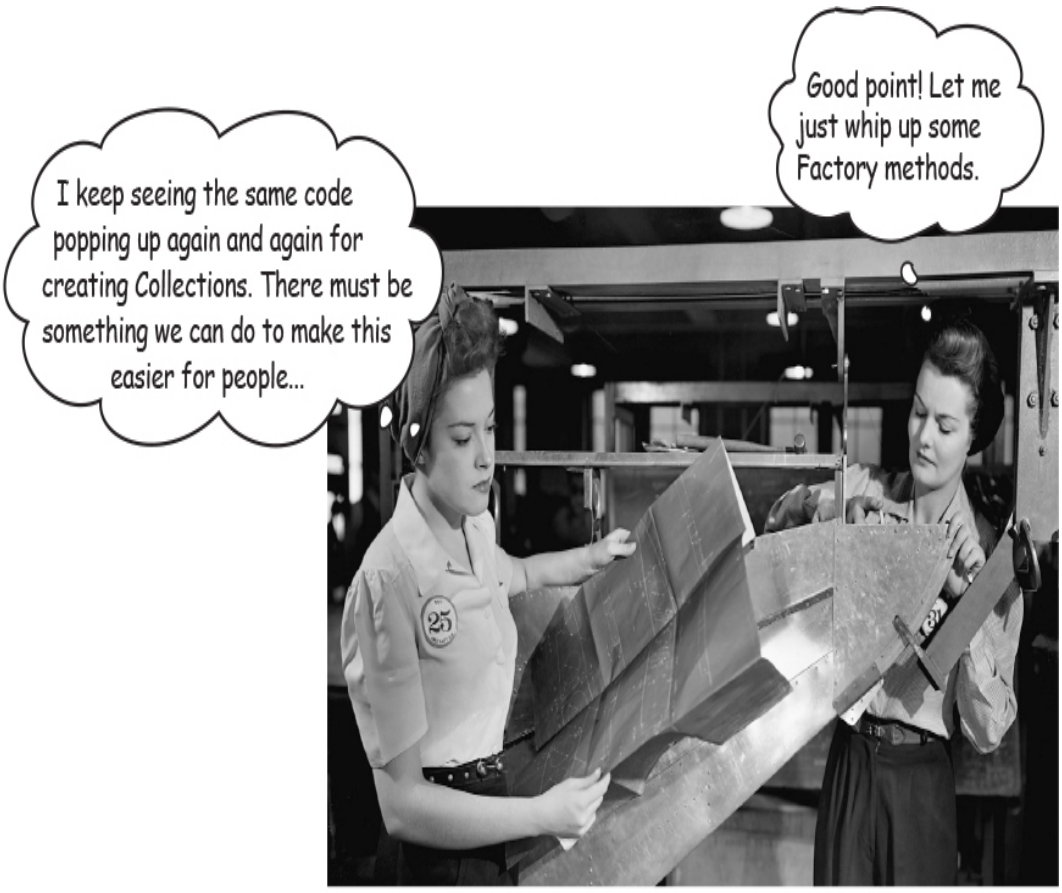


NOTE

When you print a Map, it gives you the key=value pairs, in braces { } instead of the brackets [ ] you see when you print lists and sets.

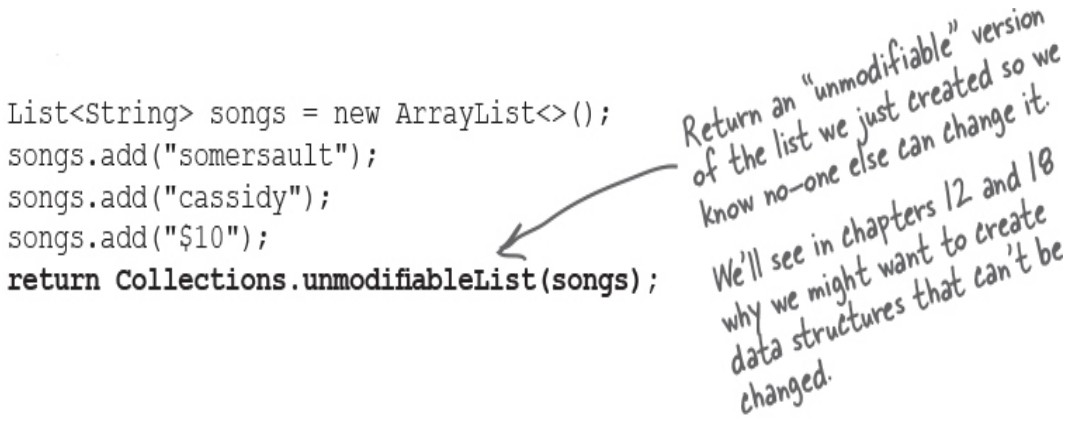
Creating and filling collections

The code for creating, and then filling, a collection crops up again and again. You've already seen code for creating an ArrayList and adding elements to it quite a few times. Code like this:




```
List<String> songs = new ArrayList<>();
songs.add("somersault");
songs.add("cassidy");
songs.add("$10");
```

Whether you're creating a List, a Set, or a Map, it looks pretty similar. What's more, these types of collections are often ones where we know what the data is right at the start, and then we don't intend to change it at all during the lifetime of the collection. If we wanted to really make sure that no-one changed the collection after we'd created it, we'd have to add an extra step.



That's a lot of code! And it's a lot of code for something common that we probably want to do a lot.

Fortunately for us, Java now has "Convenience Factory Methods of Collections" (they were added in Java 9). We can use these methods to create common data structures and fill them with data, with just one method call.

Convenience Factory Methods for Collections

Convenience Factory Methods for Collections allow you to easily create a List, Set or Map that's been pre-filled with known data. There are a couple of things to understand about using them:

1. ① **The resulting collections cannot be changed**. You can't add to them or alter the values, in fact you can't even do the sorting that we've seen in this chapter.
2. ② **The resulting collections are not the standard Collections we've seen**. These are not ArrayList, HashSet, HashMap etc. You can rely on them to behave according to their interface: a List will always preserve the order in which the elements were placed; a Set will never have duplicates. But you can't rely on them being a specific implementation of List, Set or Map.

Convenience Factory Methods are just that - a convenience that will work for most of cases where you want to create a collection pre-filled with data. And for those cases where these factory methods don't suit you, you can still use the Collections constructors and add() or put() methods instead.

- **Creating a List: `List.of()`**

  To create the list of Strings from the last page, we don't need five lines of code, we just need one:

```
List<String> strings = List.of("somersault", "cassidy",
"$10");
```

  If you want to add Song objects instead of simple Strings, it's still short and descriptive

```
List<SongV4> songs = List.of(new SongV4("somersault", "zero
7", 147),
                        new SongV4("cassidy", "grateful dead",
158),
                        new SongV4("$10", "hitchhiker", 140));
```

- **Creating a Set: `Set.of()`**

  Creating a Set uses very similar syntax.

```
Set<Book> books = Set.of(new Book("How Cats Work"),
                     new Book("Remix your Body"),
                     new Book("Finding Emo"));
```

- **Creating a Map: `Map.of()`, `Map.ofEntries()`**

  Maps are different, because they take two objects for each "entry" - a key and a value. If you want to put less than 10 entries into your Map, you can use Map.of, passing in key, value, key, value, etc

```
Map<String, Integer> scores = Map.of("Kathy", 42,
                                     "Bert", 343,
                                     "Skyler", 420);
```

If you have more than 10 entries, or if you want to be clearer about how your keys are paired up to their values, you can use Map.ofEntries instead.

```
Map<String, String> stores = Map.ofEntries(Map.entry("Riley",
"Supersports"),
                                           Map.entry("Brooklyn",
"Camera World"),
                                           Map.entry("Jay",
"Homecase"));
```

To make the line shorter you can use a *static import* on Map.entry (we talked about static imports in Chapter 10).

Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be... *weird*. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story. The examples are going to use a class hierarchy of Animals.

```
abstract class Animal {
  void eat() {
    System.out.println("animal eating");
  }
}
class Dog extends Animal {
  void bark() { }
}
class Cat extends Animal {
  void meow() { }
}
```

The simplified Animal class hierarchy.

Using polymorphic arguments and generics

Generics can be a little... counter-intuitive when it comes to using polymorphism with a generic type (the class inside the angle brackets). Let's create a method that takes a List<Animal> and use this to experiment.

Passing in List<Animal>

```
public class TestGenerics1 {
  public static void main(String[] args) {

    List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
    takeAnimals(animals);
  }

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

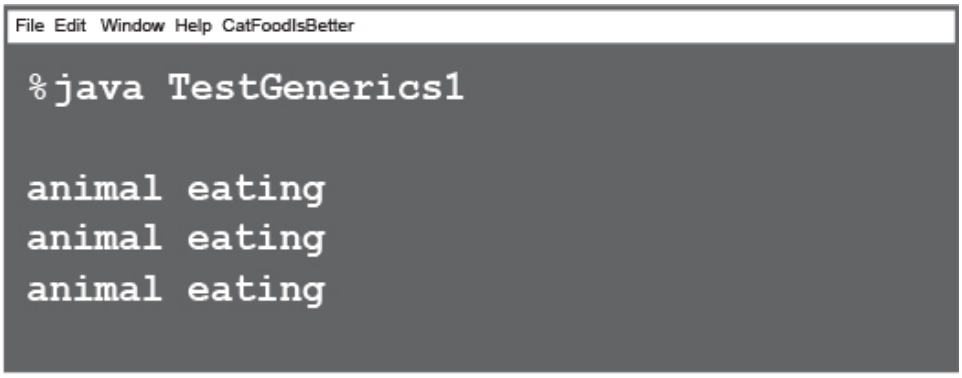
Using the List.of factory method we just looked at

Pass a List<Animal> into our testAnimals method

Method that has a generic class (List) as a parameter

Remember, we can call ONLY the methods declared in type Animal, since the animals parameter is of type List<Animal>.

Compiles and runs just fine



But will it work with List<Dog> ?

So a List<Animal> argument can be passed to a method with a List<Animal> parameter. So the big question is, will the List<Animal> parameter accept a List<Dog>? Isn't that what polymorphism is for?

Passing in List<Dog>

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

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

We know this line worked fine.

Make a Dog List and put a couple dogs in.

Will this work now that we changed from an array to a List?

When we compile it:

NOTE

It looked so right, but went so wrong...

What could happen if it *were* allowed...?

Imagine the compiler let you get away with that. It let you pass a List<Dog> to a method declared as:

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

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the

eat() method), a Dog can do as well. So what's the problem with having the method call eat() on each of the Dog references?

There's nothing wrong with *that* code. But imagine *this* code instead:

```
public void takeAnimals(List<Animal> animals) {
   animals.add(new Cat());
}
```

Yikes!! We just stuck a Cat in what might be a Dogs-only List.

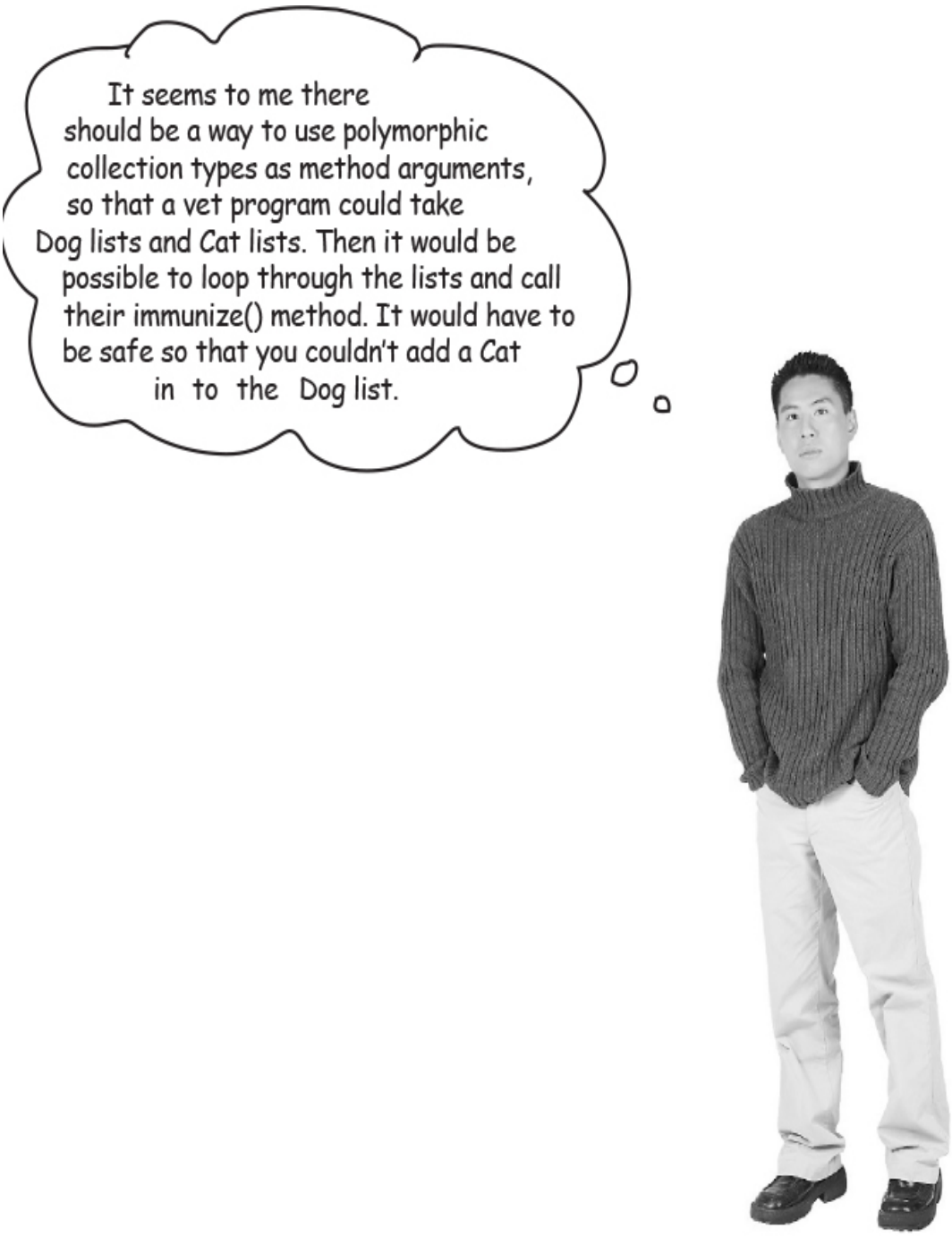
So that's the problem. There's certainly nothing wrong with adding a Cat to a List<Animal>, and that's the whole point of having a List of a supertype like Animal—so that you can put all types of animals in a single Animal List.

But if you passed a Dog List—one meant to hold ONLY Dogs—to this method that takes an Animal List, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog List into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

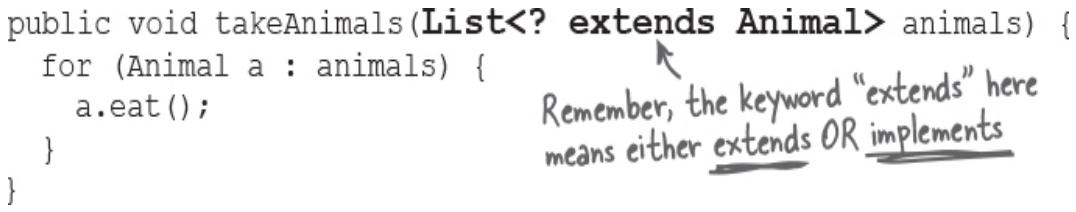
NOTE

If you declare a method to take List<Animal> it can take ONLY a List<Animal>, not List<Dog> or List<Cat>.

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**.



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) { }
```

NOTE

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.

Exercise

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

Possible Answers:

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

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

1. What must the class of the objects stored in `myArrayList` implement? **Comparable**
2. What method must the class of the objects stored in `myArrayList` implement? **compareTo( )**
3. Can the class of the objects stored in `myArrayList` implement both `Comparator` AND `Comparable`? **yes**

Given the following compilable statement:

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

4. Can the class of the objects stored in `myArrayList` implement `Comparable`? **yes**
5. Can the class of the objects stored in `myArrayList` implement `Comparator`? **yes**
6. Must the class of the objects stored in `myArrayList` implement `Comparable`? **no**
7. Must the class of the objects stored in `myArrayList` implement `Comparator`? **no**
8. What must the class of the `myCompare` object implement? **Comparator**
9. What method must the class of the `myCompare` object implement? **compare( )**

Sharpen your pencil Solution

"Reverse Engineer" lambdas exercise

```
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]
```

Sorting with lambdas

```
                                    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]
```

TreeSet exercise

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 can-
not 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 "Finally, a closer look at `finally`")

BE the compiler solution

Compiles?

```
takeAnimals(new ArrayList<Animal>());
```

```
takeDogs(new ArrayList<Animal>());
```

```
takeAnimals(new ArrayList<Dog>());
```

```
takeDogs(new ArrayList<>());
```

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>

```
List<Dog> dogs = new ArrayList<>();
takeDogs(dogs);
```

```
takeSomeAnimals(new ArrayList<Dog>());
```

```
takeSomeAnimals(new ArrayList<>());
```

Here the diamond operator means this is ArrayList<Animal>

```
takeSomeAnimals(new ArrayList<Animal>());
```

```
List<Animal> animals = new ArrayList<>();
takeSomeAnimals(animals);
```

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

```
List<Object> objects = new ArrayList<>();
takeObjects(objects);
```

```
takeObjects(new ArrayList<Dog>());
```

```
takeObjects(new ArrayList<Object>());
```

Chapter 12. Lambdas and Streams: What, not How

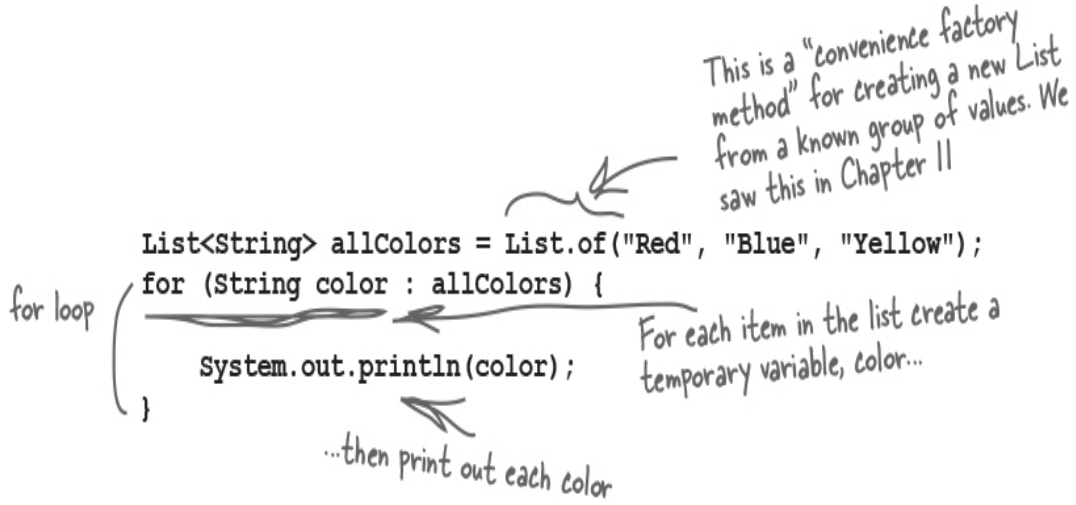
What if... you didn't need to tell the computer HOW to do something? Programming involves a lot of telling the computer how to do something: **while** this is true **do** this thing; **for** all these items **if** it looks like this **then** do this; and so on.

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 last 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

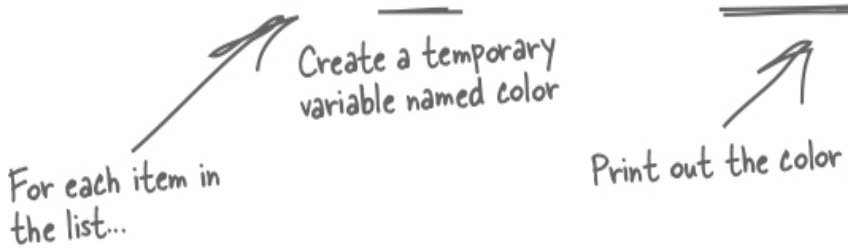
Tell the computer WHAT you want

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.



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));
```




```
File Edit Window Help SingARainbow
%java PrintColors

Red
Blue
Yellow
```

NOTE

The forEach method of a list takes a lambda expression, which we saw for the first time in the last 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

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

forEach()	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.	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.
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.	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.
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.	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 I'm faster. Everyone knows that.	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....
I said you would disappear soon.	

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.

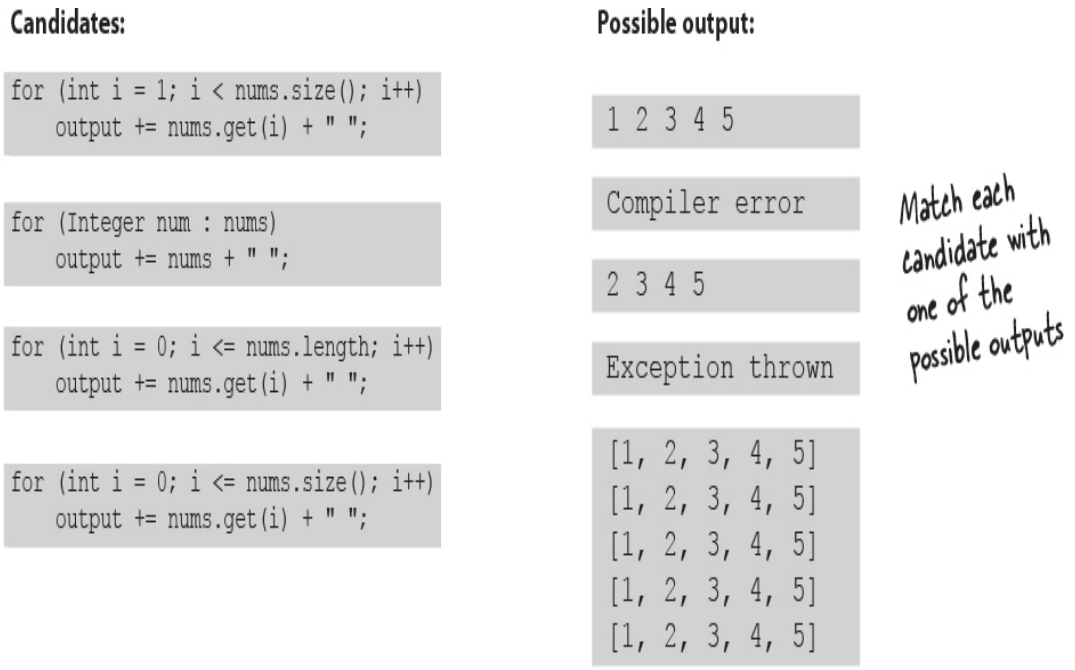
Mixed Messages

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.

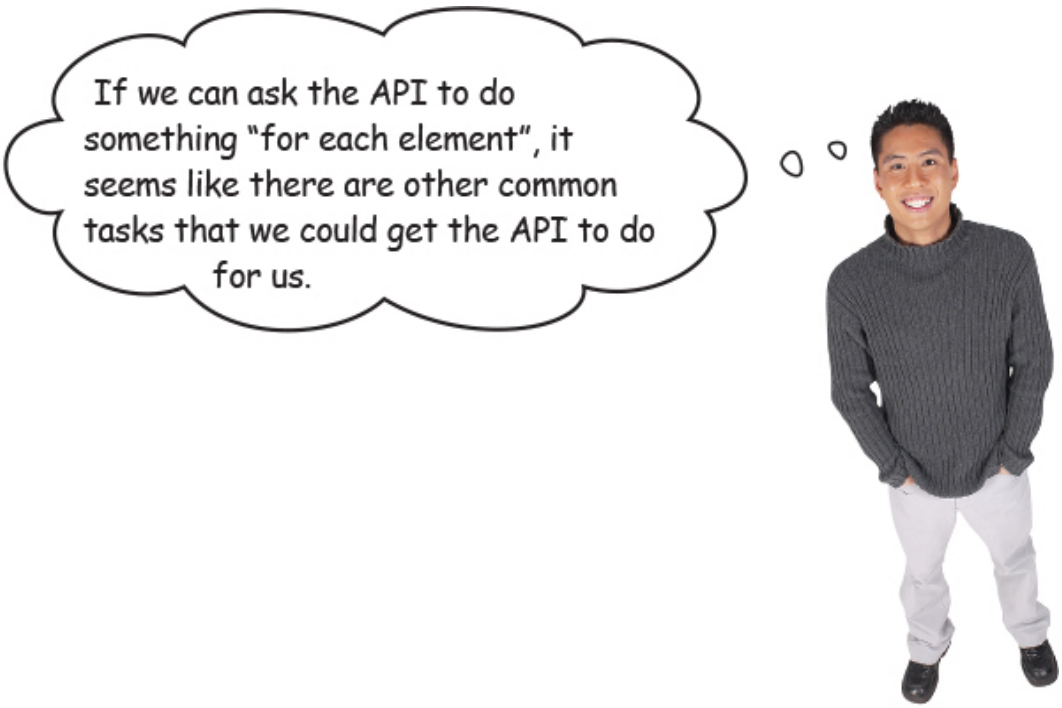
```
class MixForLoops {
  public static void main(String [] args) {
    List<Integer> nums = List.of(1, 2, 3, 4, 5);
    String output = "";
    
    
    System.out.println(output);
  }
}
```



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) can also point to a problem in the code. But it can be more tricky to spot code that produces incorrect output just by looking at the code.

Using a method like **`forEach`** that 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 last 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 last 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.

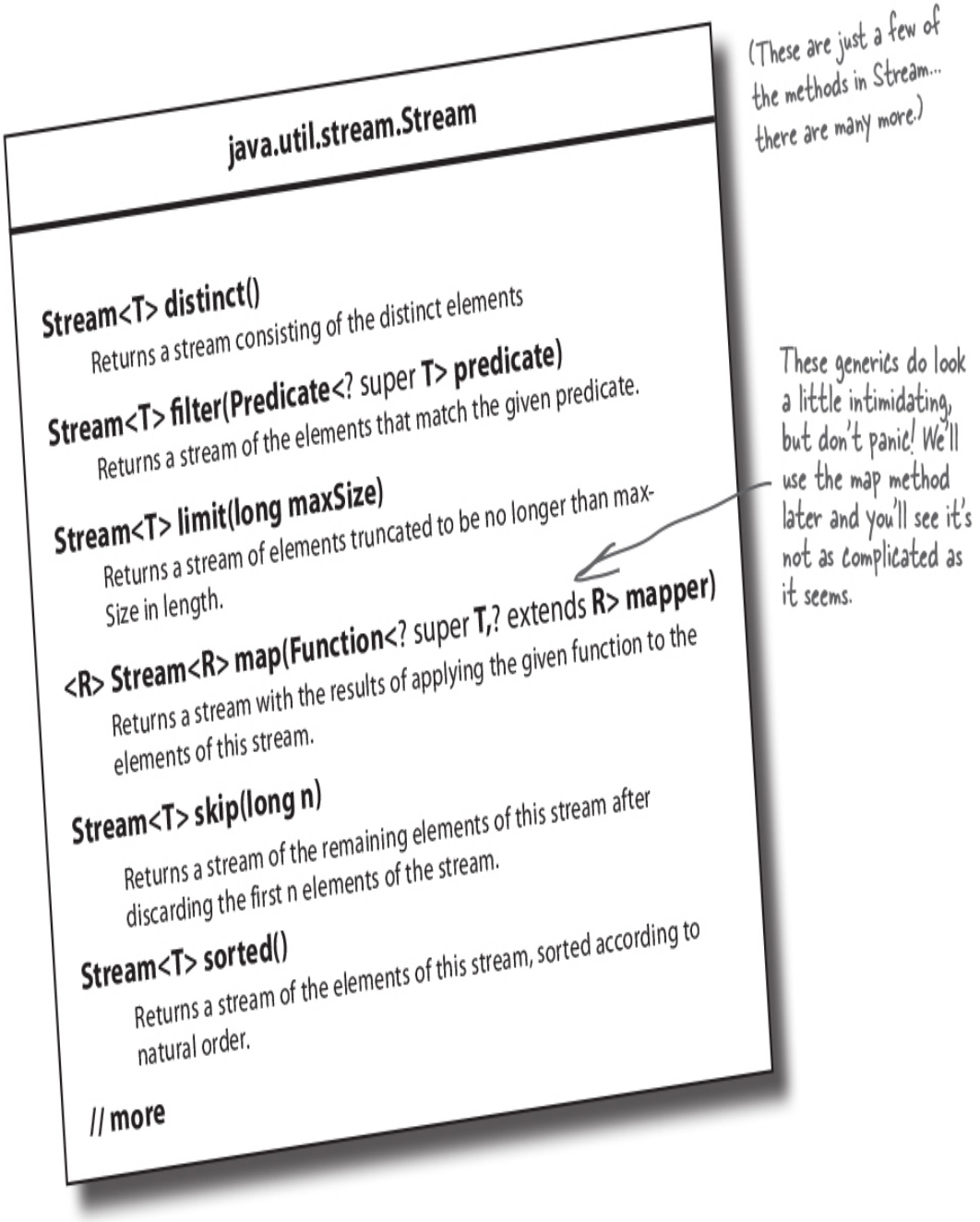
WHO DOES WHAT?

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

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.



NOTE

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

RELAX

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.

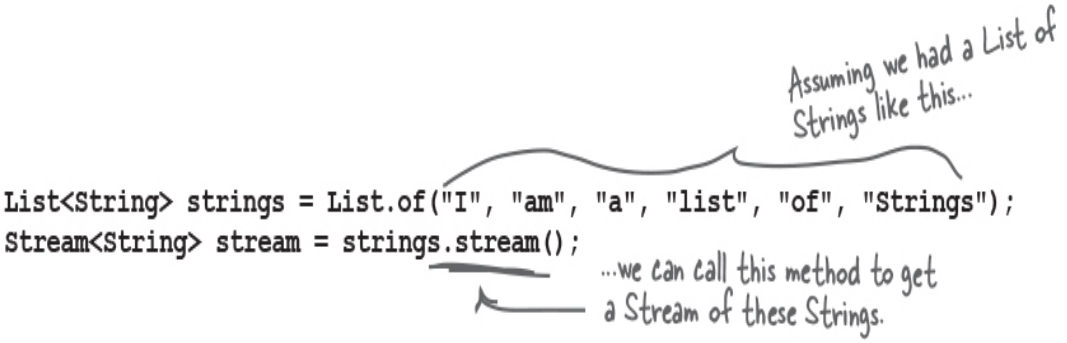
In case you are interested:

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

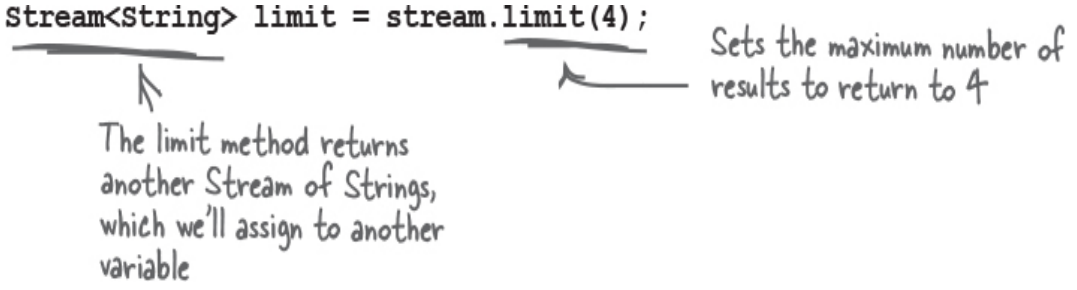
Getting started with Streams

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.

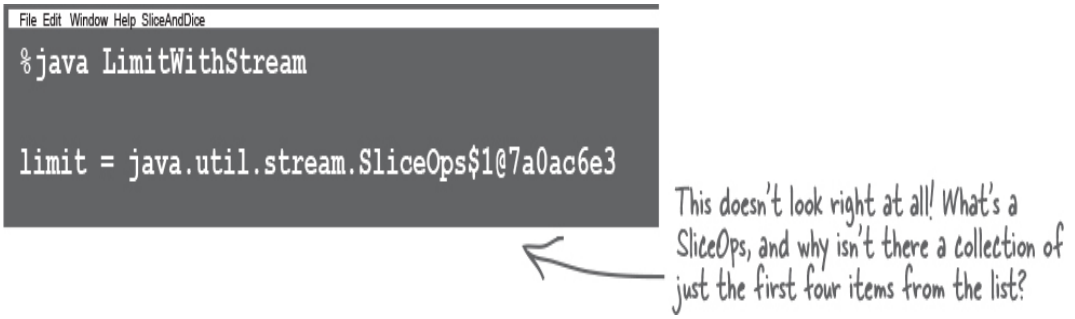


Now we can call the methods of the Streams API. For example, we could use **limit** to say we want a maximum of four elements.



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

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

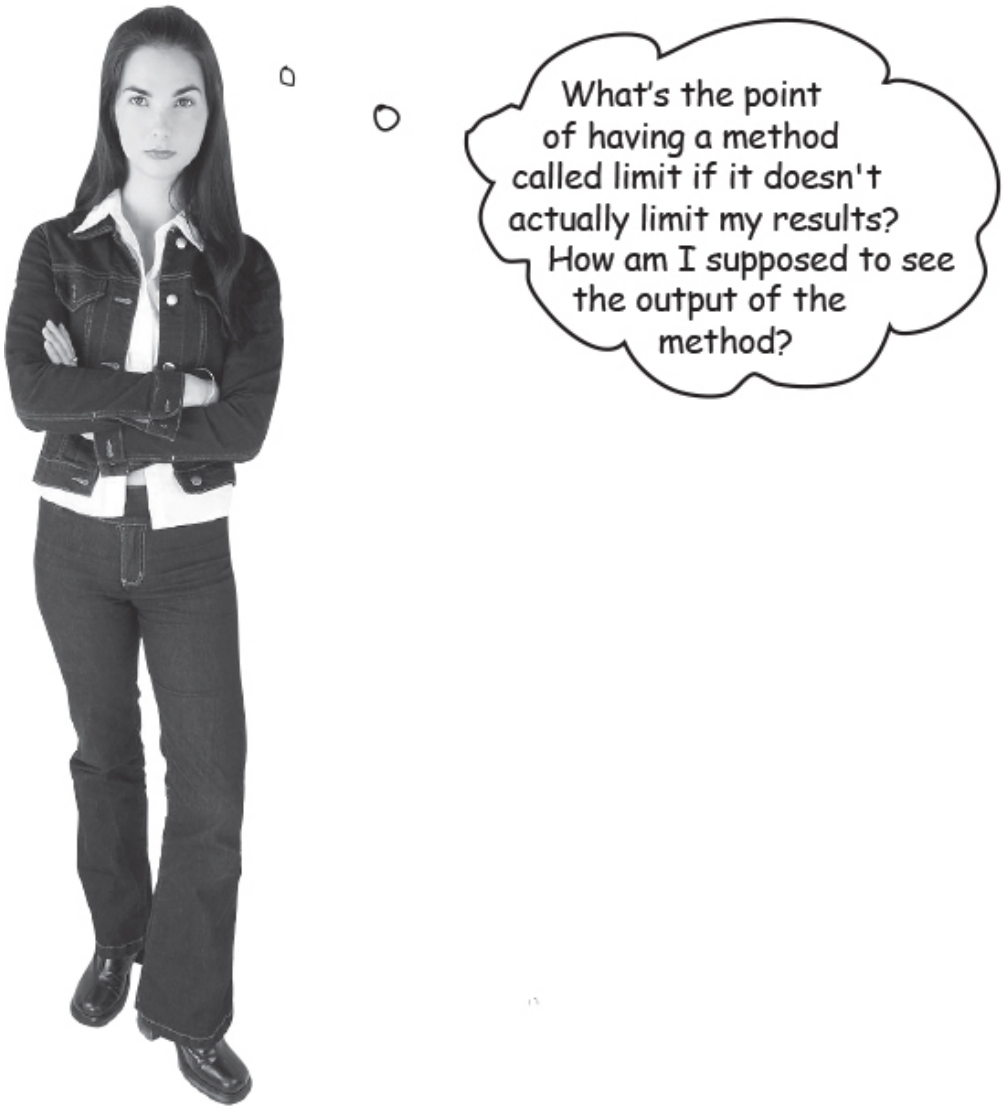


Like everything in Java, the stream variables in the example are Objects. But a stream does ***not*** contain the elements in the collection. It's more like the set of instructions for the operations to perform on the Collection data.

NOTE

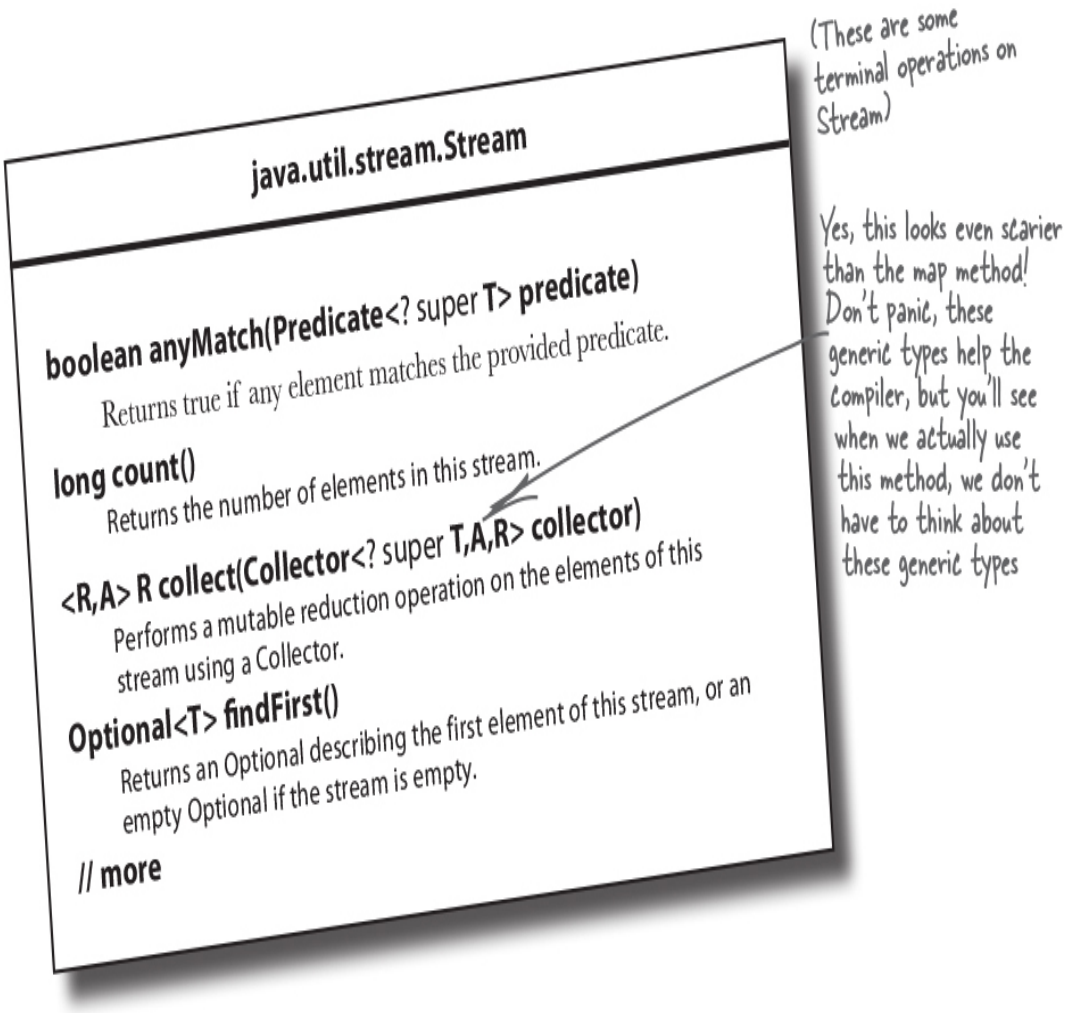
Stream methods that return another Stream are called Intermediate Operations. These are instructions of things to do, but they don't actually perform the operation on their own.

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.



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!

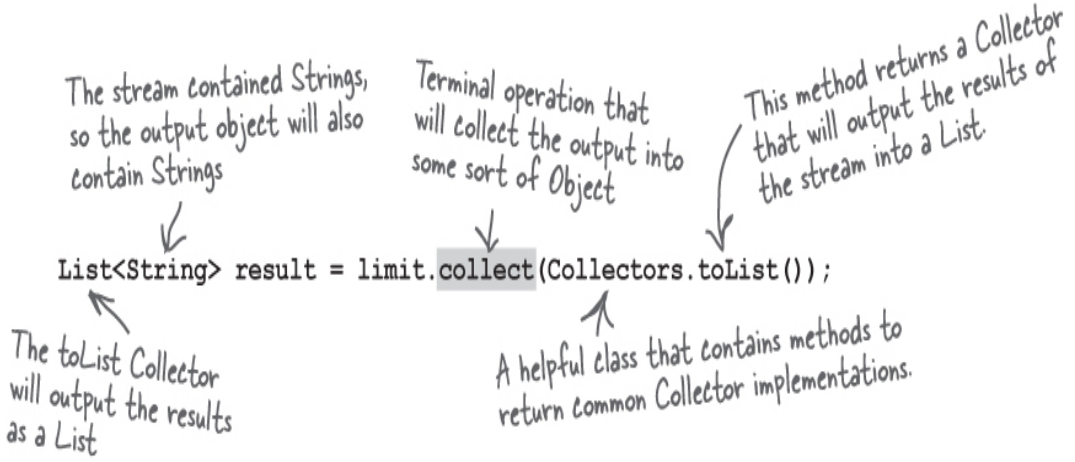
To start to get a feel for what we can do with streams, we going to show code for a simple use of the Streams API. After that, we'll step back and learn more about what we're seeing here.

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

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:




```
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.

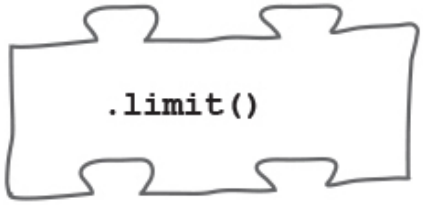
Finally, we have a result which looks like something we would have expected: we had a List of Strings, and we asked to **limit** that list to the first 4 items and then **collect** those four items into a new List.

Stream operations are building blocks

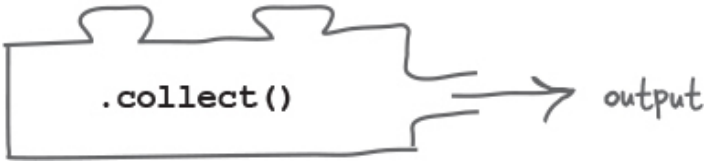
We wrote a lot of code just to output the first four elements in the list. We also introduced a lot of new terminology: streams; intermediate operations; terminal operations. Let's put all this together: you create a **stream pipeline** from three different types of building blocks.

1. ① Get the Stream from a **source** Collection

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



3. 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 re-write the limit-and-collect operation in this way:

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

Get the stream for the collection

```
List<String> result = strings.stream()
```

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

```
                      .limit(4)
                      .collect(Collectors.toList());
```

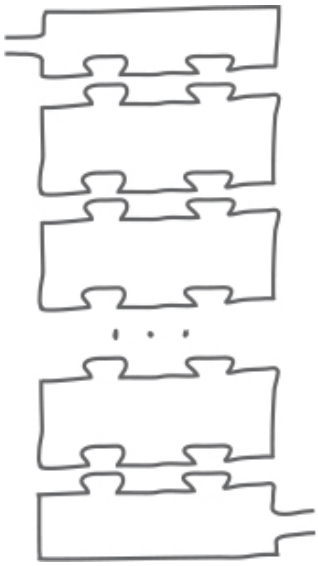
Formatted to align each operation directly underneath the one above, to clearly show each stage.

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.



NOTE

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()
                           .limit(4)
                           .collect(Collectors.toList());

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

Sort what's in the stream (not the original collection), using natural order, before limiting the results

Limit the stream to just four elements

```
File Edit Window Help InChains
%java ChainedStream

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

Natural ordering of Strings will place capitalised Strings ahead of lower case Strings

Customizing the building blocks

We can stack together operations to create a more advanced query on our collection. We can also customise 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 last chapter when we sorted Lou's song list).

Lambda expression that tells the sorted method how to sort the strings in the stream. This lambda expression represents a Comparator, which we talked about in the last chapter. We'll recap lambdas later in this chapter

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

This method from the String class compares the String with another String, in a way that ignores upper or lower case

```
File Edit Window Help IgnoreCaps
%java ChainedStream

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

Create complex pipelines block by block

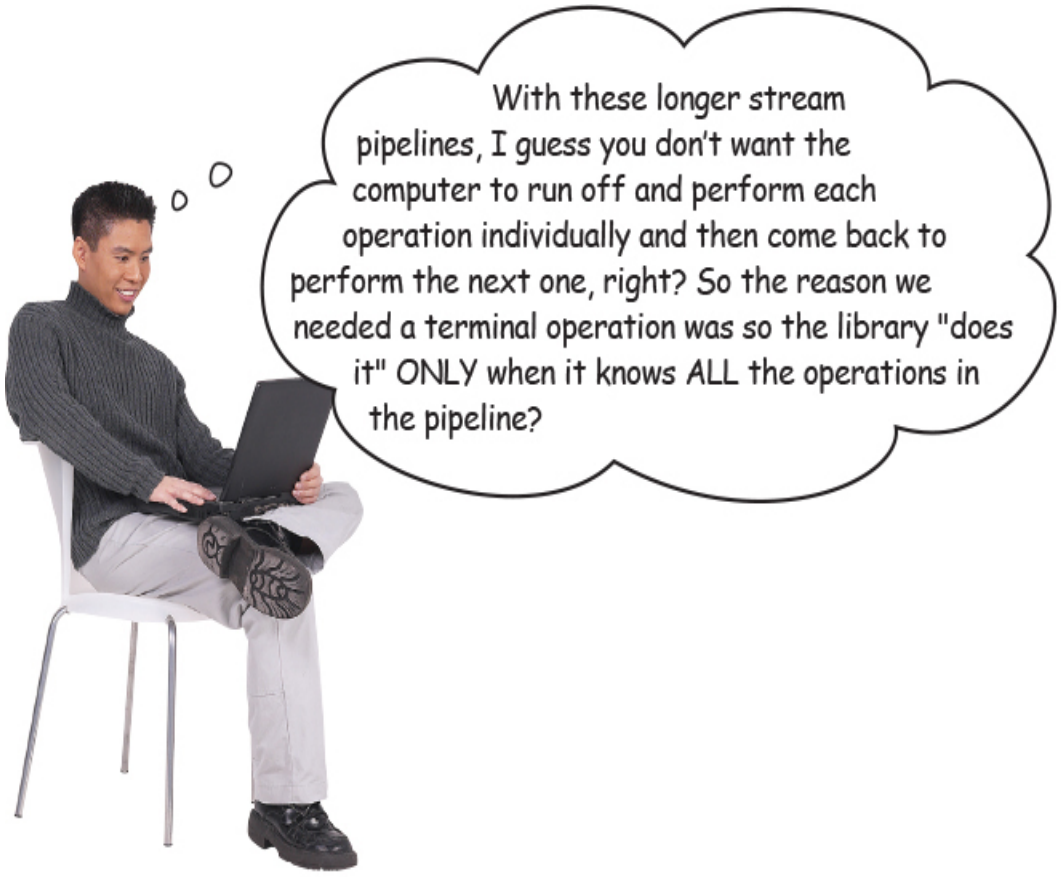
Each new operation you add to the pipeline changes the output from the pipeline. Each operations tell the Streams API *what* it is you want to do.

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

```
File Edit Window Help BoxersDoIt
%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.

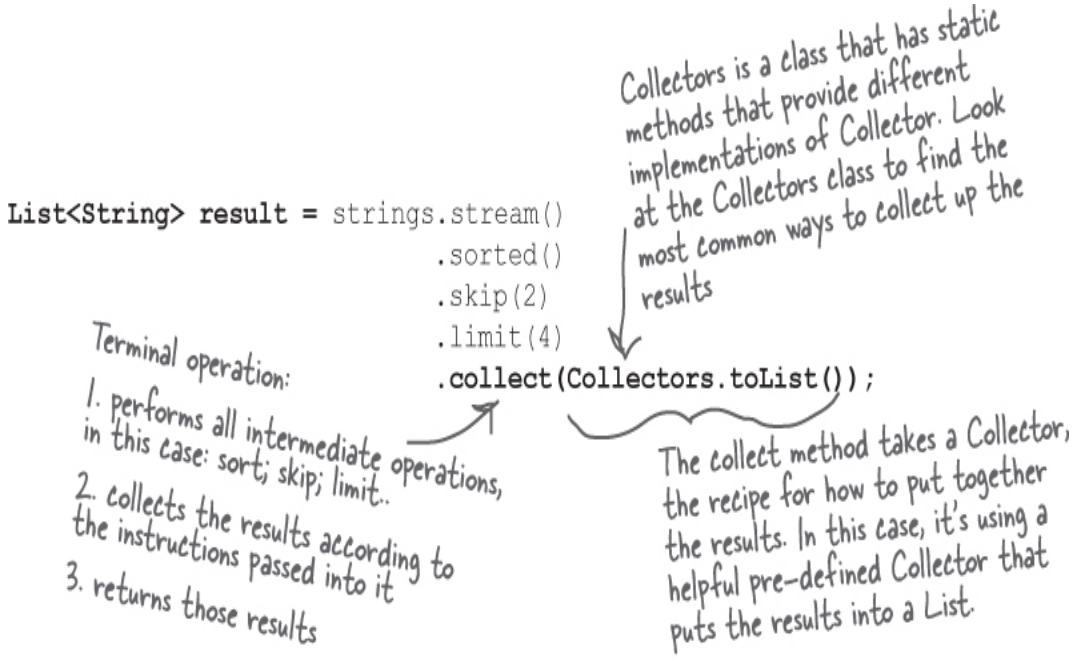
Terminal operations do all the work

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

1. ❶ Perform all the intermediate operations as efficiently as possible. Ideally, just going through the original data once.
2. ❷ 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. ❸ 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.



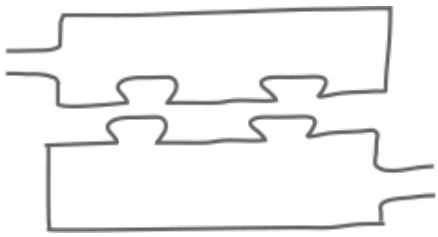
We will look at more Collectors, and other terminal operations, later in the chapter. For now, you know enough to get going with streams.

Guidelines for working with streams

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

1. 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. 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. 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.

Correct! Stream operations <u>don't</u> 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]
```

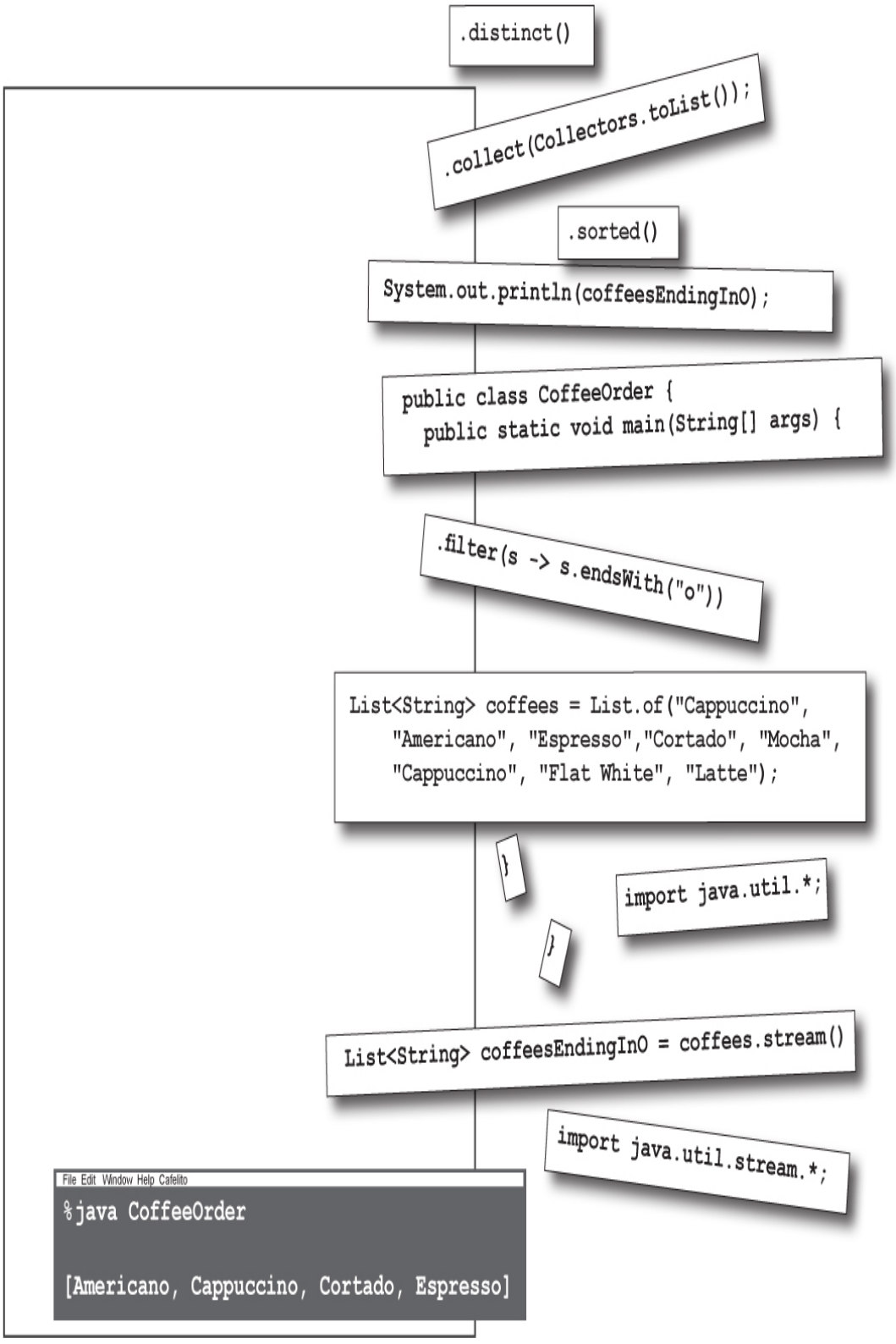
No changes to original collection after the stream operations are run

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

Exercise

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?


.distinct()
.collect(Collectors.toList());
.sorted()
System.out.println(coffeesEndingInO);
public class CoffeeOrder {
public static void main(String[] args) {
.filter(s -> s.endsWith("o"))
List<String> coffees = List.of("Cappuccino",
"Americano", "Espresso","Cortado", "Mocha",
"Cappuccino", "Flat White", "Latte");
}
import java.util.*;
}
List<String> coffeesEndingInO = coffees.stream()
import java.util.stream.*;
File Edit Window Help Cafelito
%java CoffeeOrder
[Americano, Cappuccino, Cortado, Espresso]

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 which outputs the original collection in some new shape which is just right for what you need. Be aware, however, 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 which 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 the chapters on 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 last 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.

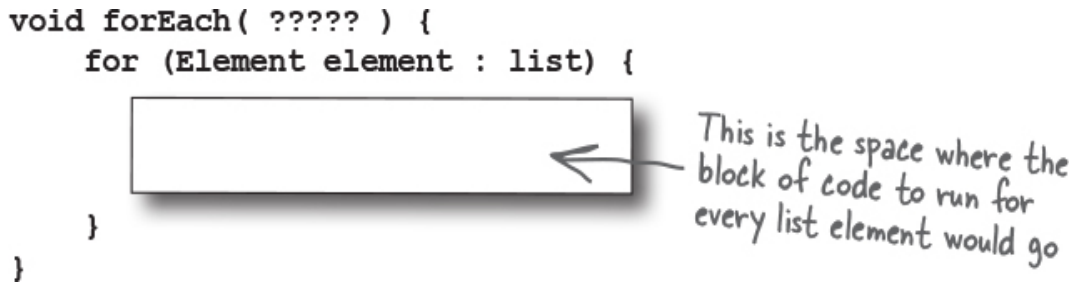
Hello Lambda, my (not so) old friend

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.

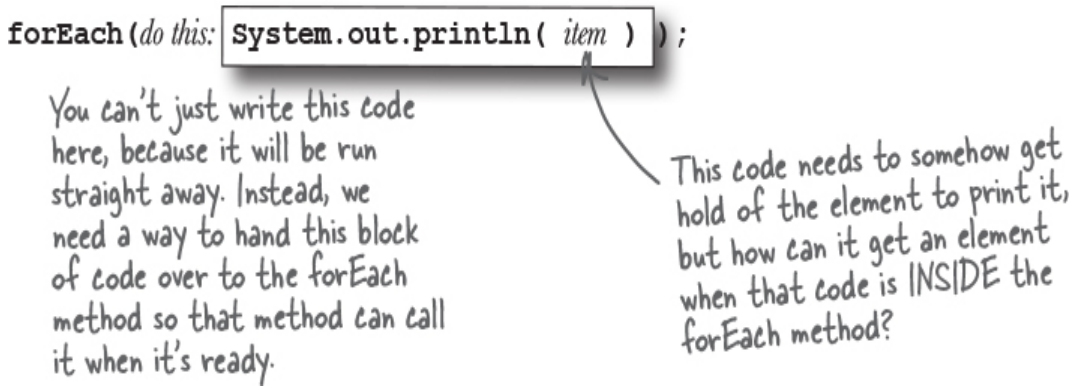
Passing behavior around

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



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.

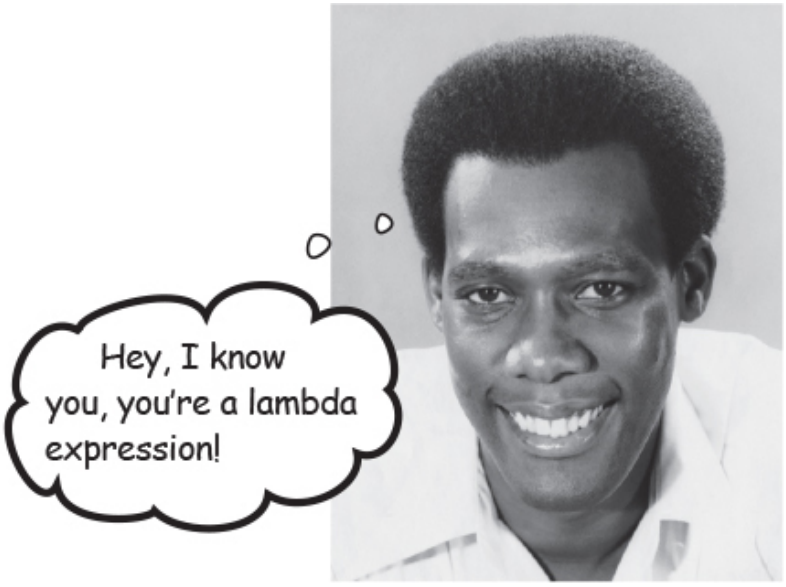
Then you want someone calling the method to be able to say



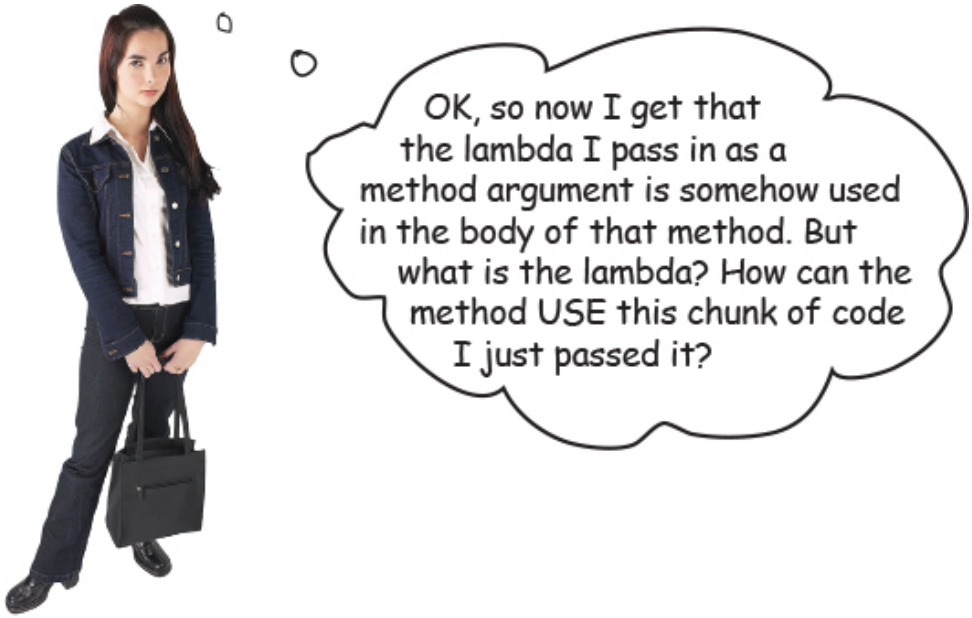
Now, we need to replace the *do this* with some sort of symbol to represent that this code isn't to be run straight away, but instead needs to be passed into the method. We could use, oh, let's see... "**->**" as this symbol.

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 hand side of the "do this" symbol....

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



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



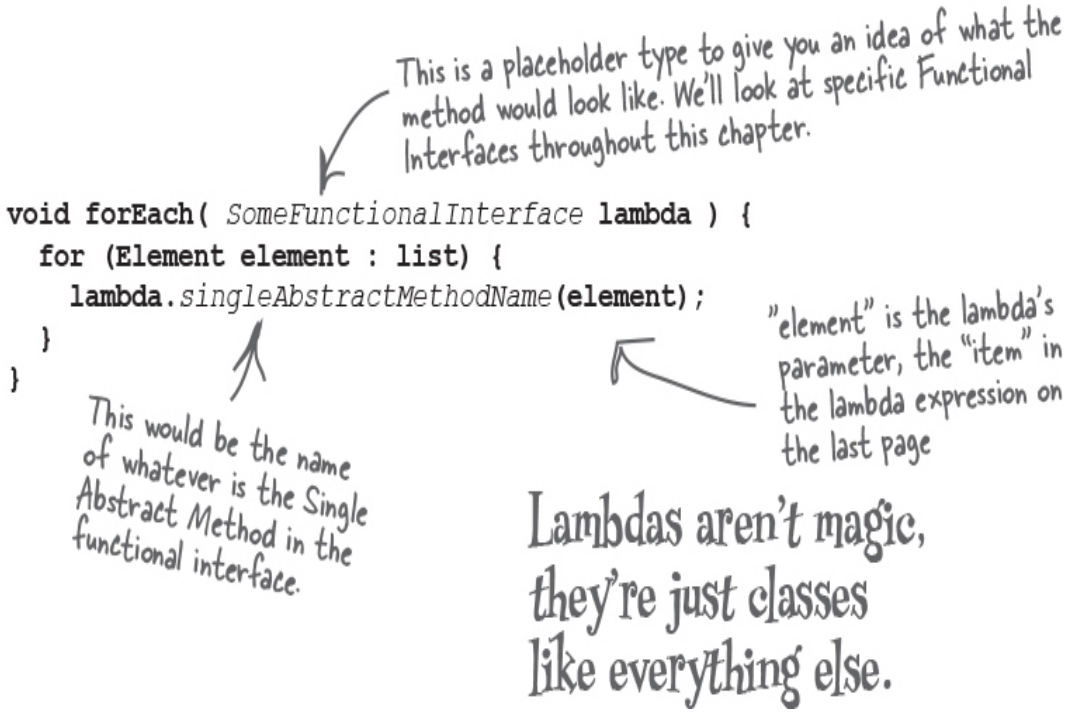
Remember, everything in Java is an Object (well, except for the primitive types), and lambdas are no exception.

> **NOTE**
>
> **A lambda expression implements a Functional Interface.**

Which 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.

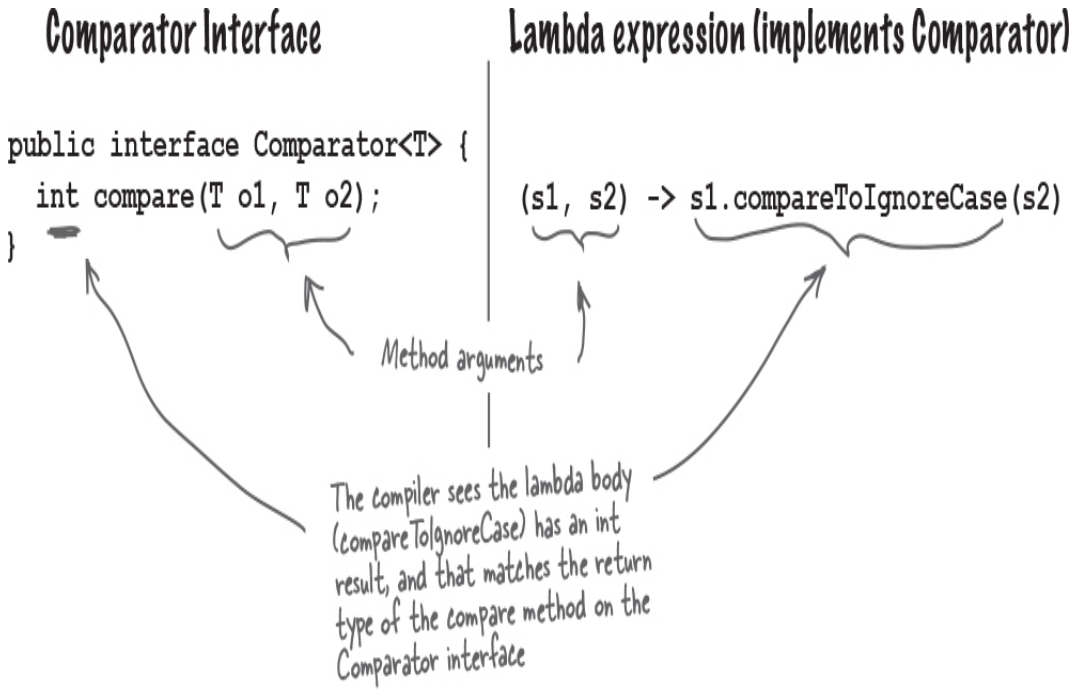
Back to our imaginary **`forEach`** example, our parameter needs to implement a functional interface. We also need to call that lambda expression somehow, passing in the list element.

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.



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 last chapter; and the lambda expression we passed into the **`sorted()`** stream operation on p601. Look at this last example side by side with the Comparator Functional Interface.



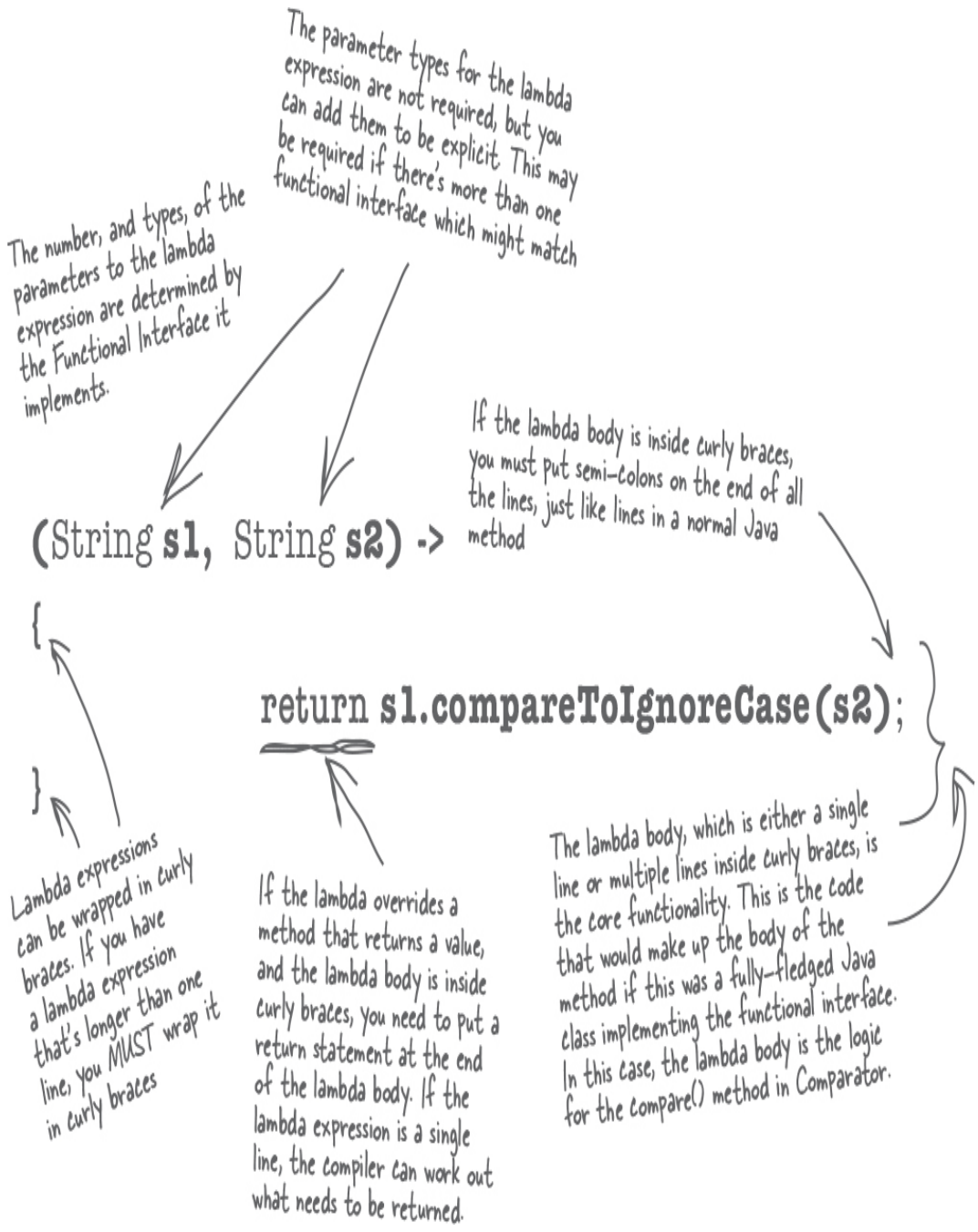
You might be wondering where the **return** keyword is in the lambda expression. The short version is: you don't need it. The longer version is, if the lambda expression is a single line, and if the functional interface's method signature requires a returned value, the compiler just assumes that your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

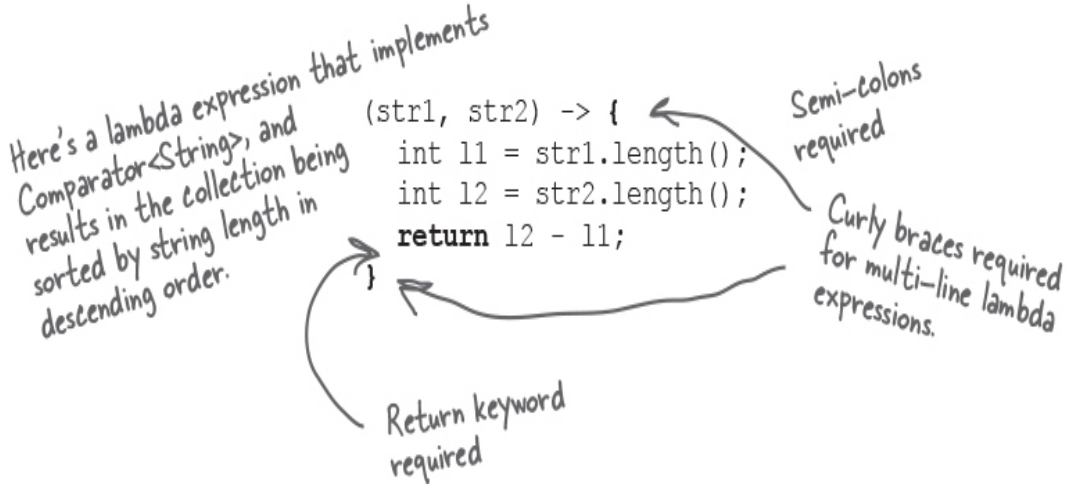
Variety is the spice of life

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

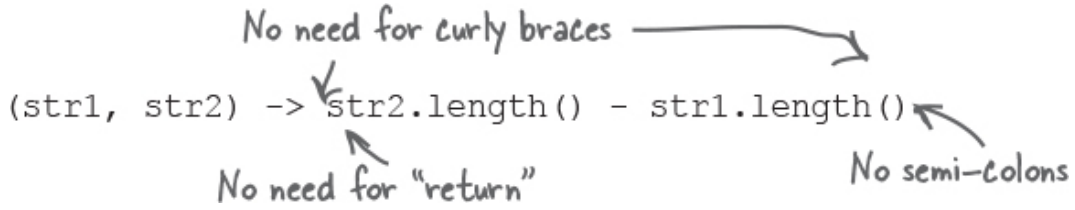
A lambda expression is effectively a method, and can have as many lines as any other method. Multi-line lambda expressions ***must*** be inside curly braces. Then, like any other method code, every line ***must*** end in a semi-colon, and if the method is supposed to return something, the lambda body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

Life would be boring if we all looked the same



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:



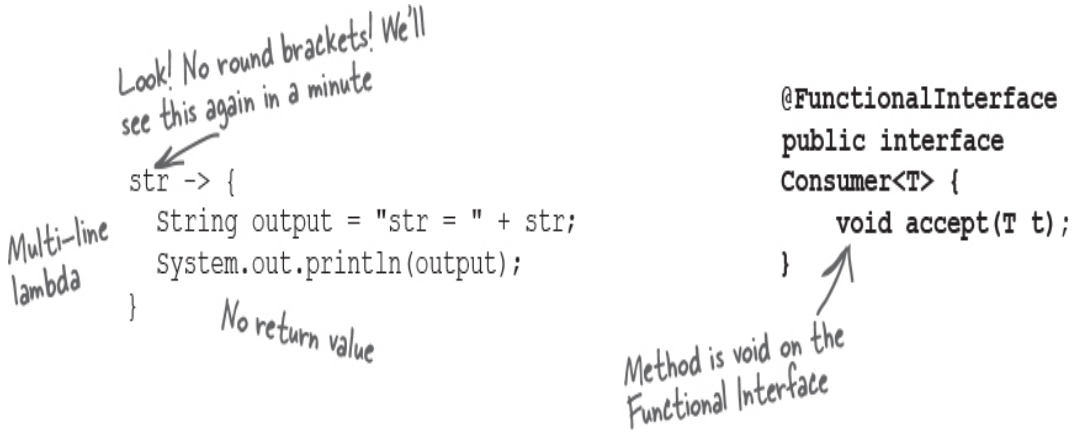
This is the same functional interface (Comparator) and performs the same operation. Whether you use multi-line lambdas or single line lambdas is completely up to you. It will probably depend upon how complicated the logic in the lambda expression is, and how easy you think it is to read - sometimes longer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

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

A lambda might not return anything

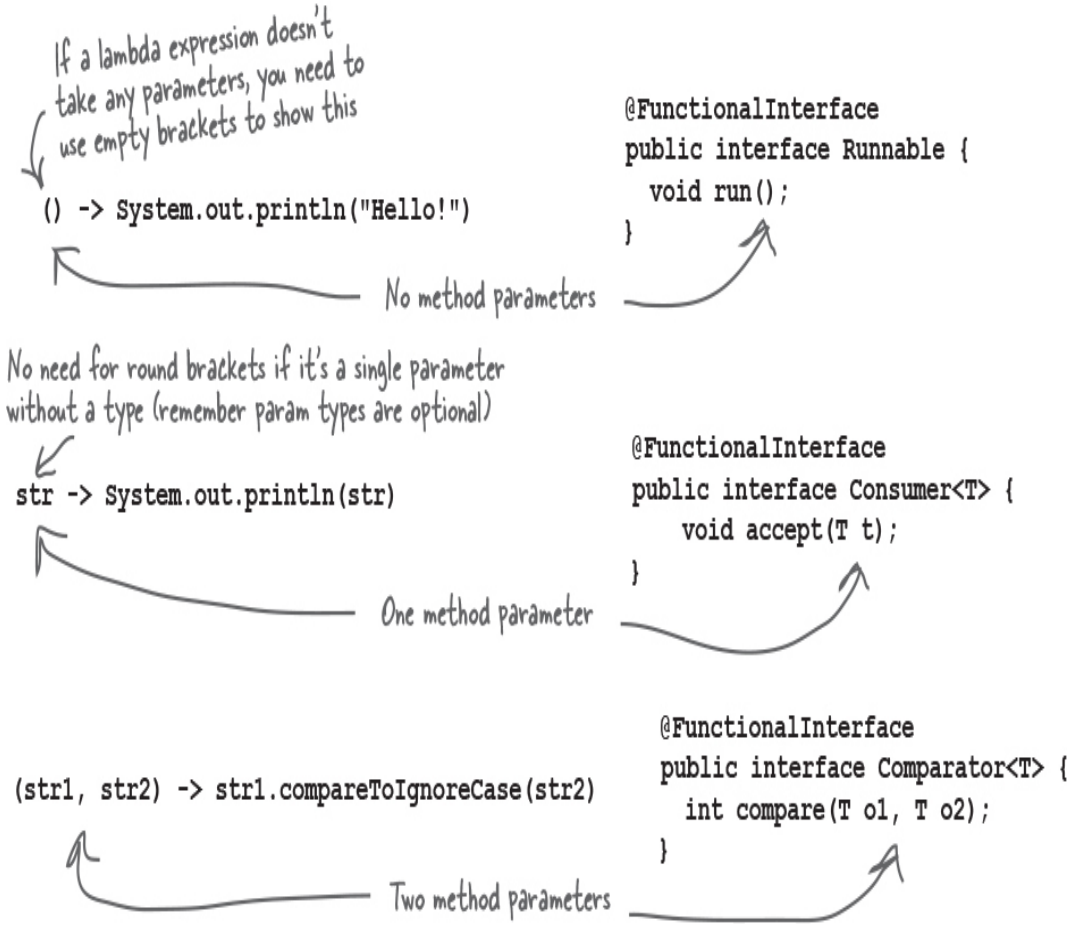
The Functional Interface's method might be declared void i.e. it doesn't return anything. In these cases, the code inside the lambda is simply run, and you don't need to return any values from the lambda body.

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



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.



How can I tell if a method takes a lambda?

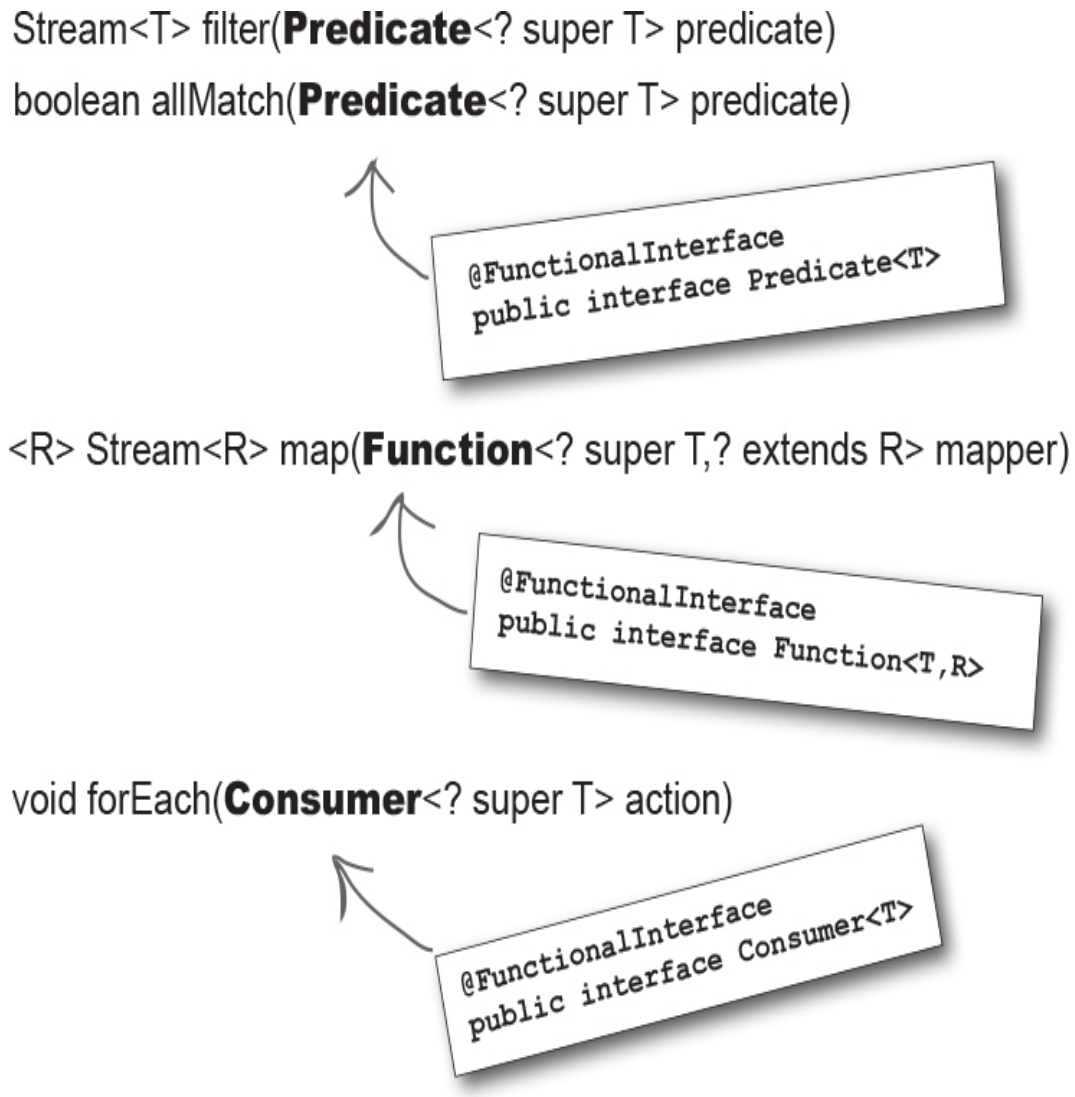
By now you've seen that lambda expressions are implementations of a functional interface - that is, an Interface with a Single Abstract Method. That means the **type** of a lambda expression is this interface.

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:



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.

```
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);
}
```

Check the box if the statement would compile

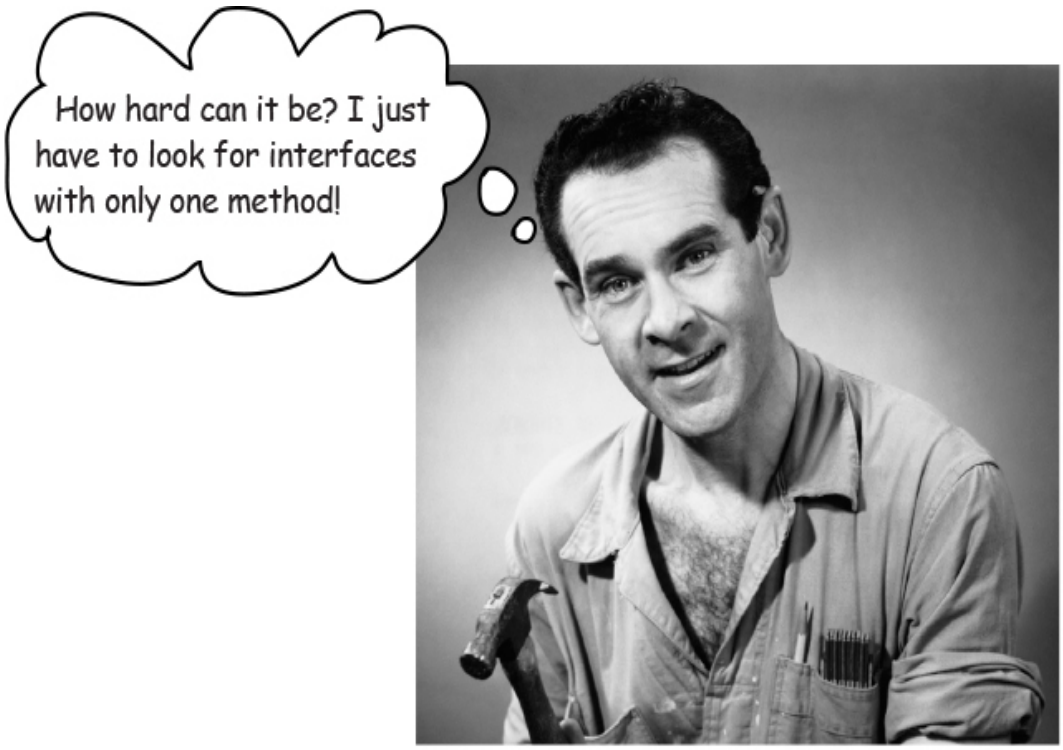
- ☐ `Runnable r = () -> System.out.println("Hi!");`
- ☐ `Consumer<String> c = s -> System.out.println(s);`
- ☐ `Supplier<String> s = () -> System.out.println("Some string");`
- ☐ `Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2);`
- ☐ `Runnable r = (String str) -> System.out.println(str);`
- ☐ `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();`

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, interfaces in Java could only contain **abstract** methods, methods that need to be *overridden* by any class that *implements* this interface. Since Java 8, interfaces can also contain **`default`** and **`static`** methods.

You saw static methods in Chapter 10, 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? 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 the 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 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.

SHARPEN YOUR PENCIL

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

Note: answers are at the end of the chapter.

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)

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,? 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()

SocketOption

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

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, perhaps putting songs, titles or artists into different collections, to find the answers to these questions.

> **NOTE**
>
> **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.

> **EXERCISE**
>
> 
>
> 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?

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.

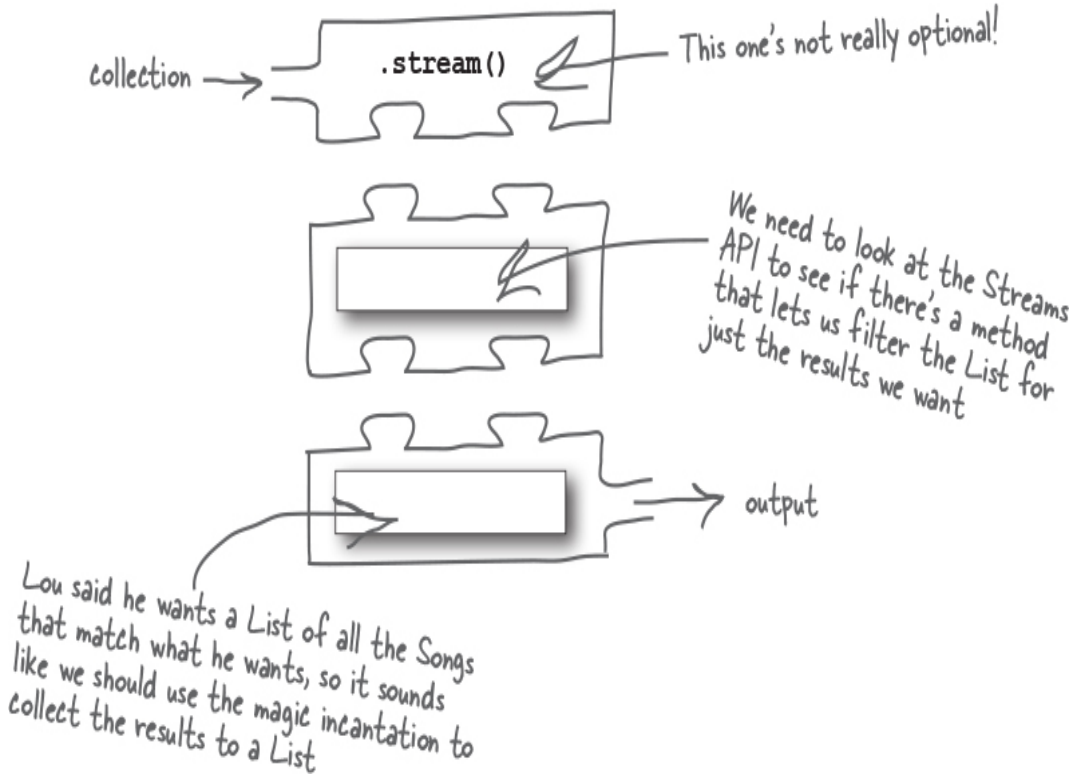
```
class Songs {
  public List<Song> getSongs() {
    return List.of(
      new Song("$10", "Hitchhiker", "Electronic", 2016, 183),
      new Song("Havana", "Camila Cabello", "R&B", 2017, 324),
      new Song("Cassidy", "Grateful Dead", "Rock", 1972, 123),
      new Song("50 ways", "Paul Simon", "Soft 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()
}
```

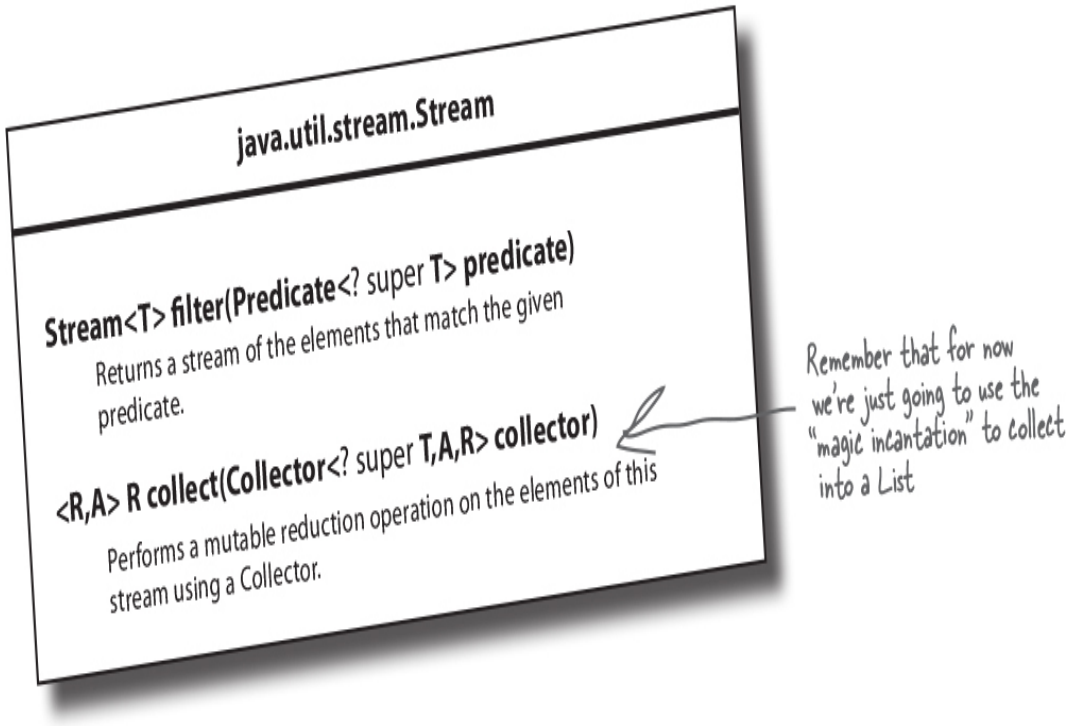
Lou's Challenge #1: Find all the "rock" songs

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.

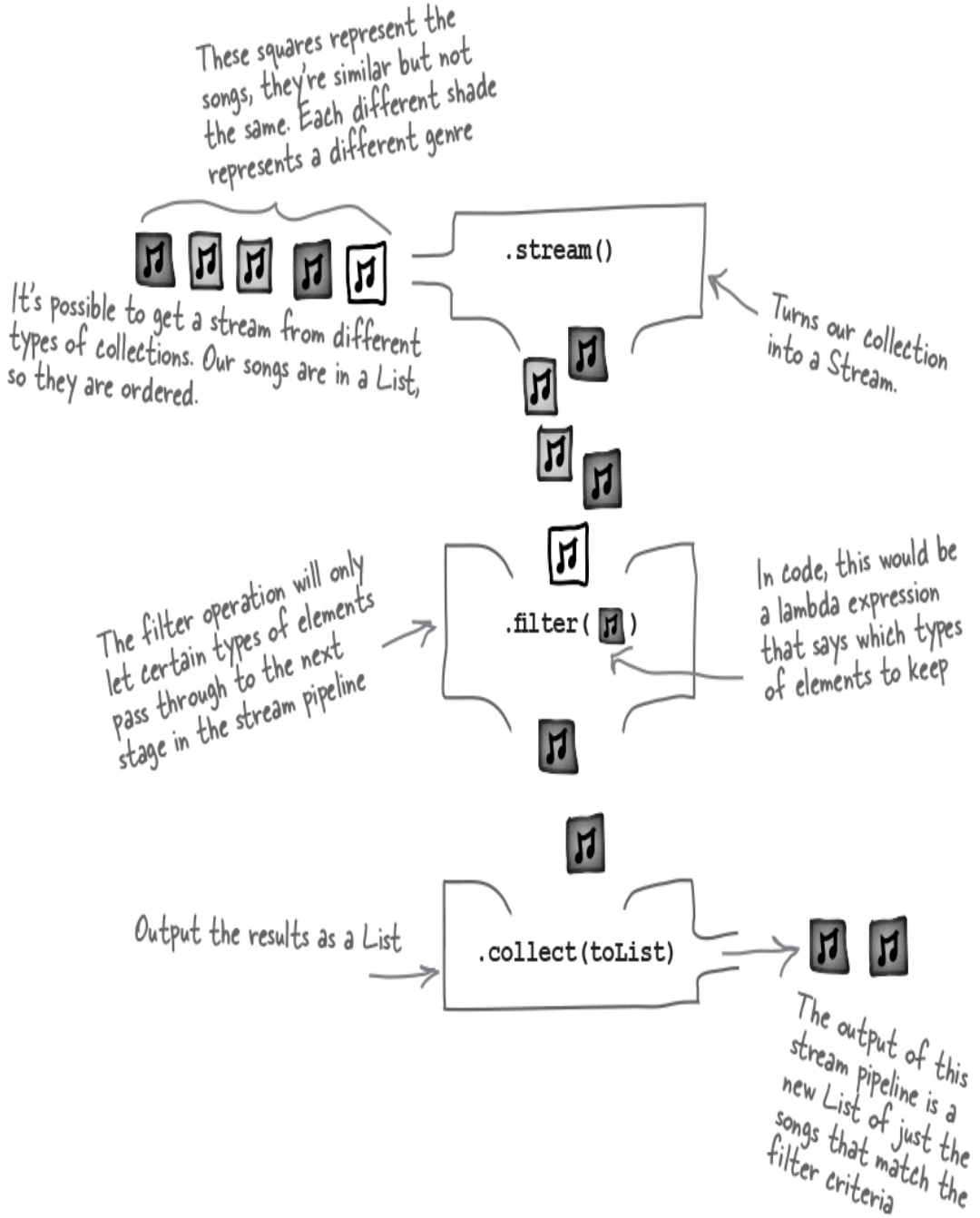


Fortunately, there are hints about how to create a Streams API call based on the requirements Lou gave us: he wants to **`filter`** for just the Songs with a particular genre, and he wants to **`collect`** them into a new List.



Filter a stream to keep certain elements

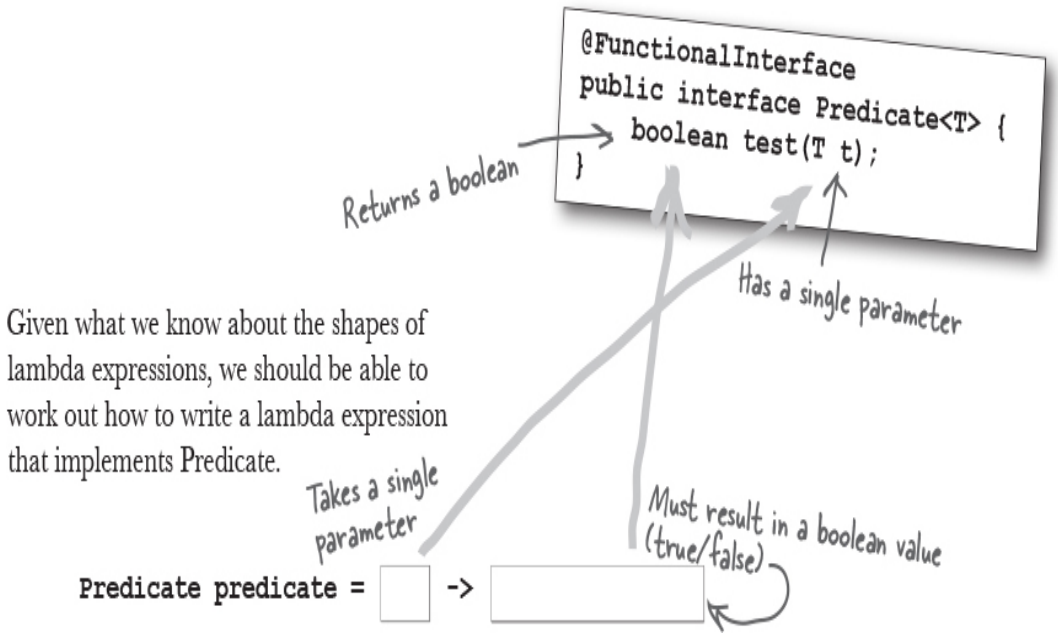
Let's see how a filter operation might work on the list of songs.


These squares represent the songs, they're similar but not the same. Each different shade represents a different genre
It's possible to get a stream from different types of collections. Our songs are in a List, so they are ordered.
.stream()
Turns our collection into a Stream.
The filter operation will only let certain types of elements pass through to the next stage in the stream pipeline
.filter( )
In code, this would be a lambda expression that says which types of elements to keep
Output the results as a List
.collect(toList)
The output of this stream pipeline is a new List of just the songs that match the filter criteria

Let's Rock!

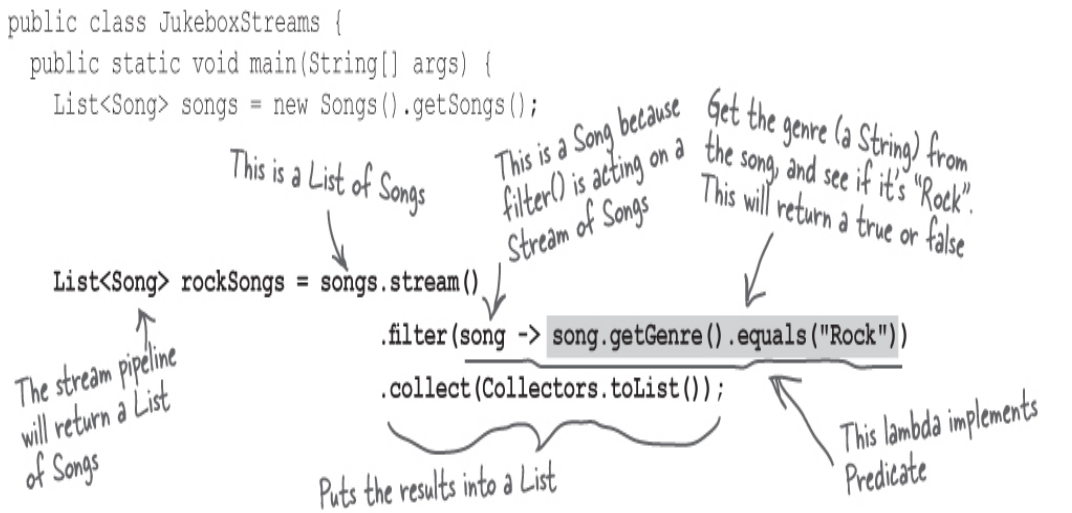
So adding a **`filter`** operation filters out elements that we don't want and the stream continues with just the elements that meet our criteria. It should come as no surprise to find that you can use a lambda expression to state which elements we want to keep in the stream.

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.




```
File Edit Window Help StonefaceVimes
%java JukeboxStreams

[Cassidy, Grateful Dead, Rock
 With a Little Help from My Friends, The Beatles, Rock,
 Come Together, Ike & Tina Turner, Rock,
 With a Little Help from My Friends, Joe Cocker, Rock,
 Immigrant song, Led Zeppelin, Rock]
```

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:

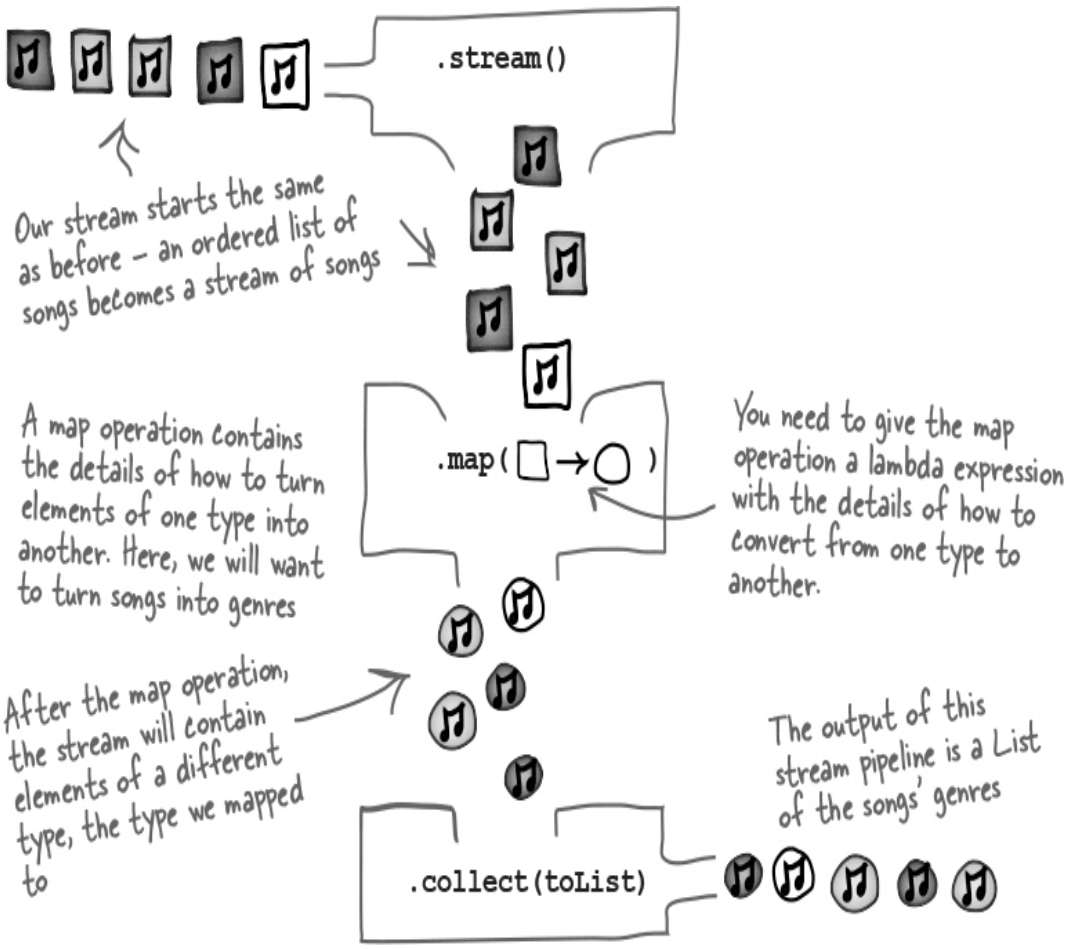
- By The Beatles
- That start with "H"
- More recent than 1995

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.

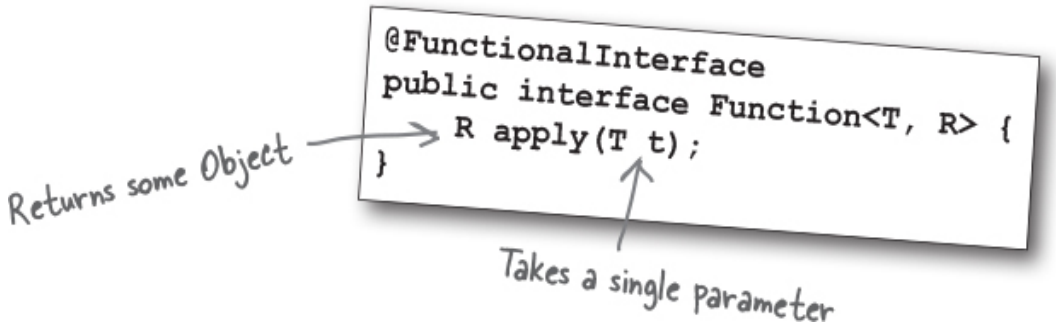
So far, all of our streams have returned the same types that they started with. The earlier examples were Streams of Strings, and returned Lists of Strings. Lou's previous challenge started with a List of Songs and ended up with a (smaller) List of Songs.

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.

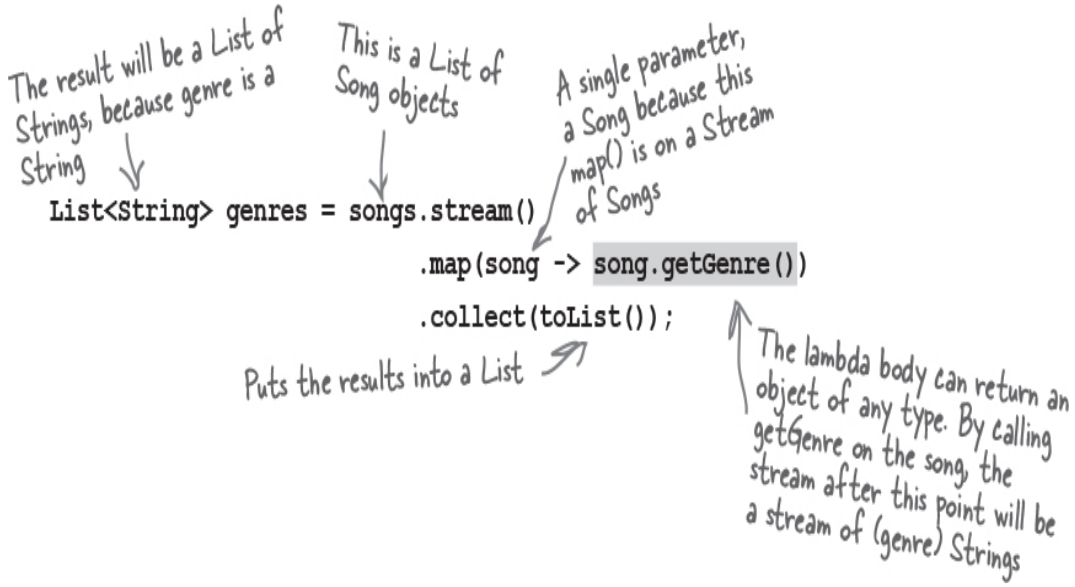


Mapping from one type to another

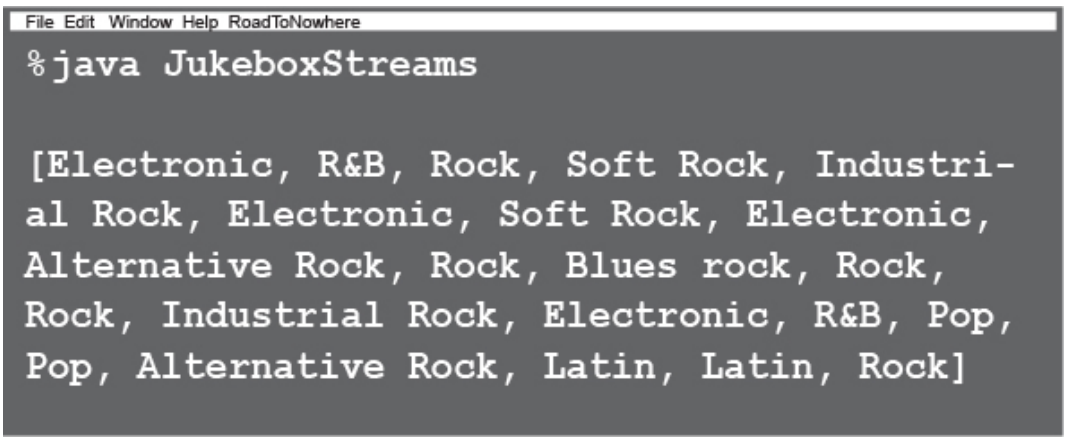
The map method takes a **`Function`**. The generics by definition are a bit vague, which makes it a little tricky to understand, but Functions do one thing: they take something of one type and return something of a different type. Exactly what's needed for mapping from one type to another.



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

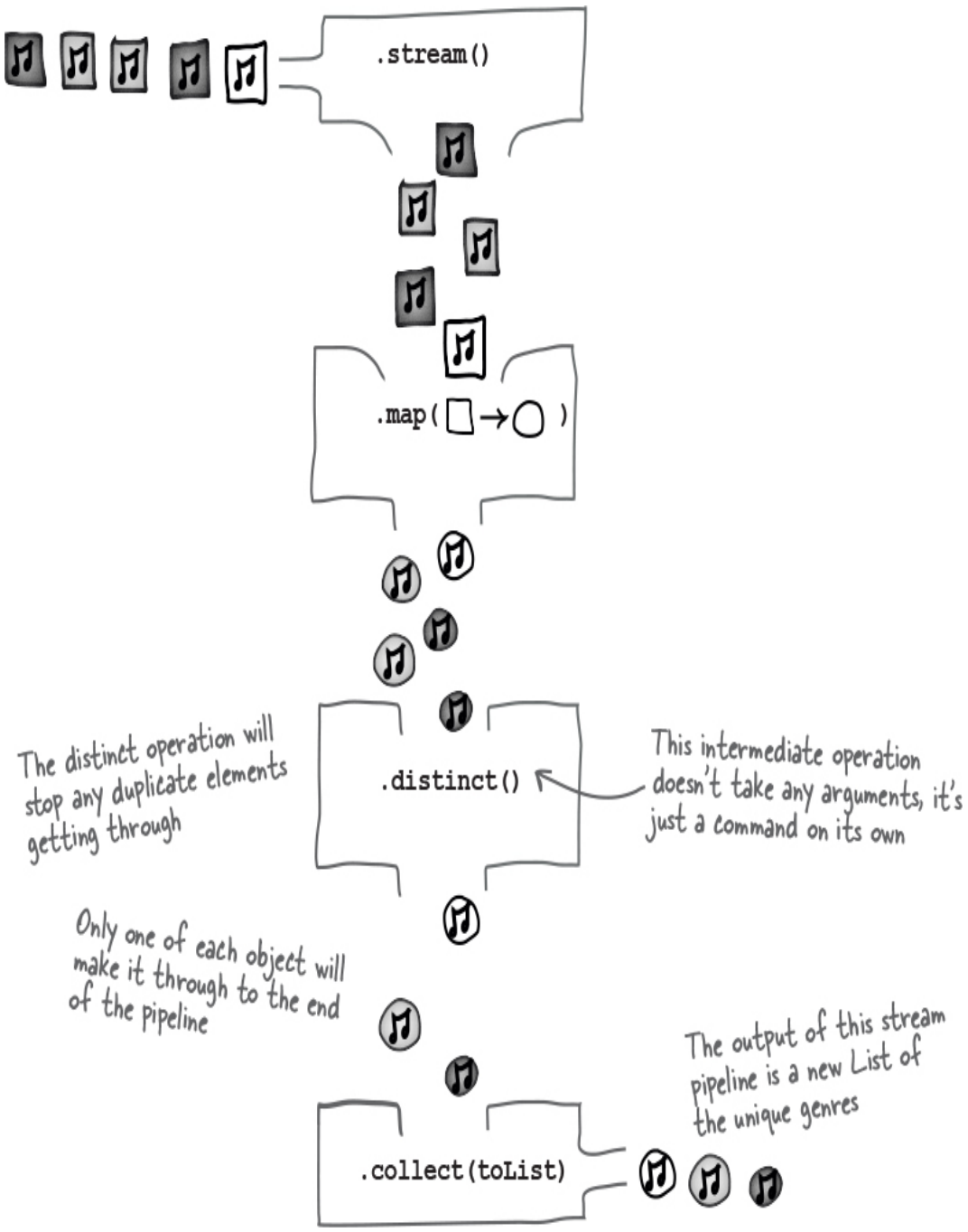


The map's lambda expression is similar to the one for filter, it takes a song and turns it into something else. Instead of returning a boolean, it returns some other object, in this case a String containing the song's genre.



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

```
File Edit Window Help UniqueIsGood
%java JukeboxStreams

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

Outputs a much more readable list of all the genres

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?

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.

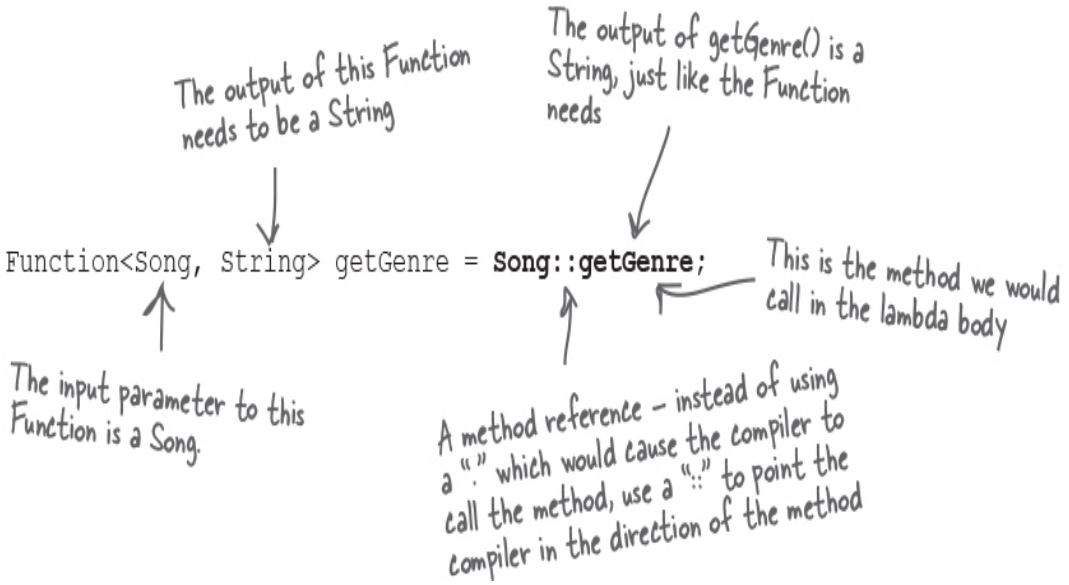
```
Function<Song, String> getGenre = song -> song.getGenre();
```

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**.

NOTE

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

Sometimes method references make the code easier to understand.



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());
```

RELAX

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

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());
```

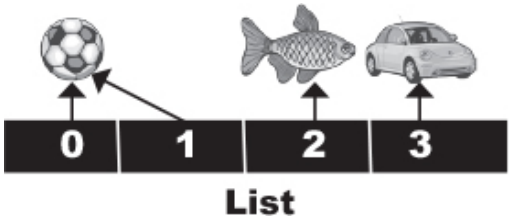
Store the results in a Set of Strings, not a List. Sets cannot contain duplicates

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

Collectors.toList and Collectors.toUnmodifiableList

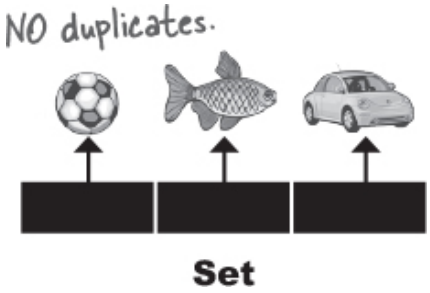
You've already seen **`toList`**. Alternatively, you can get a List that can't be changed (no elements can be added, replaced or removed) by using

`Collectors.toUnmodifableList` instead. This is only available from Java 10 onwards.



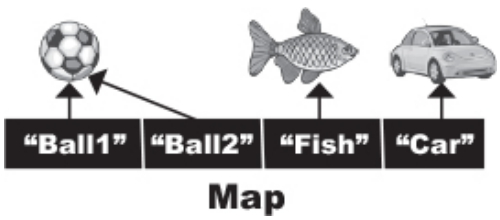
Collectors.toSet and Collectors.toUnmodifiableSet

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 onwards.



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 which 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();
```

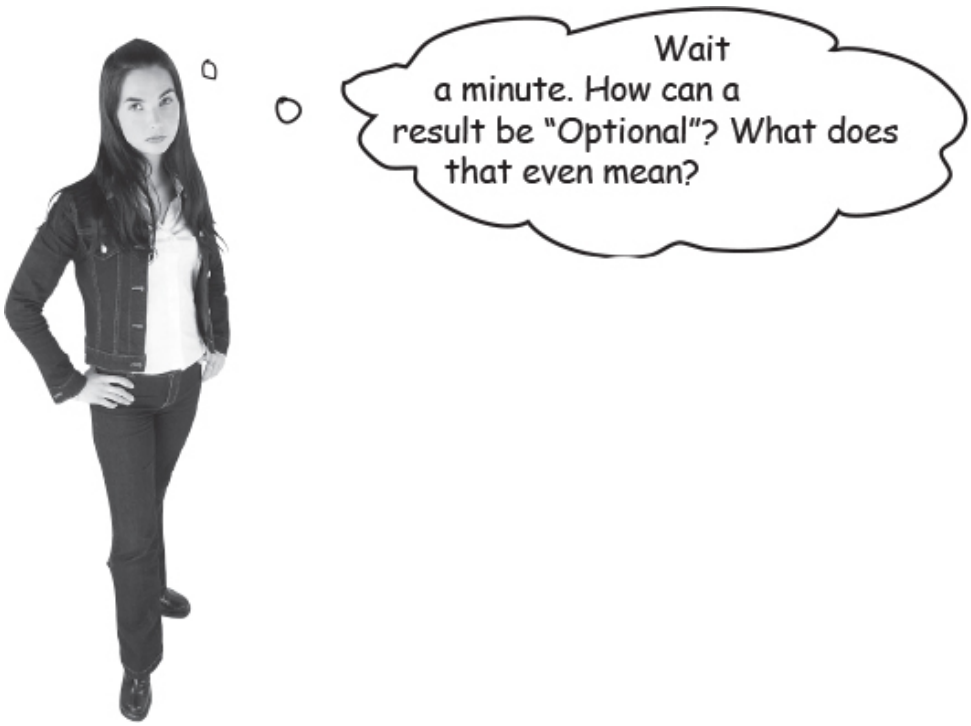
```
long count();
```

NOTE

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.

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.

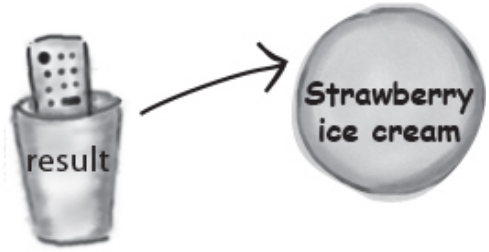
Imagine you're at an ice-cream stand, and you ask for strawberry ice cream.


```
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 flavour".

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?



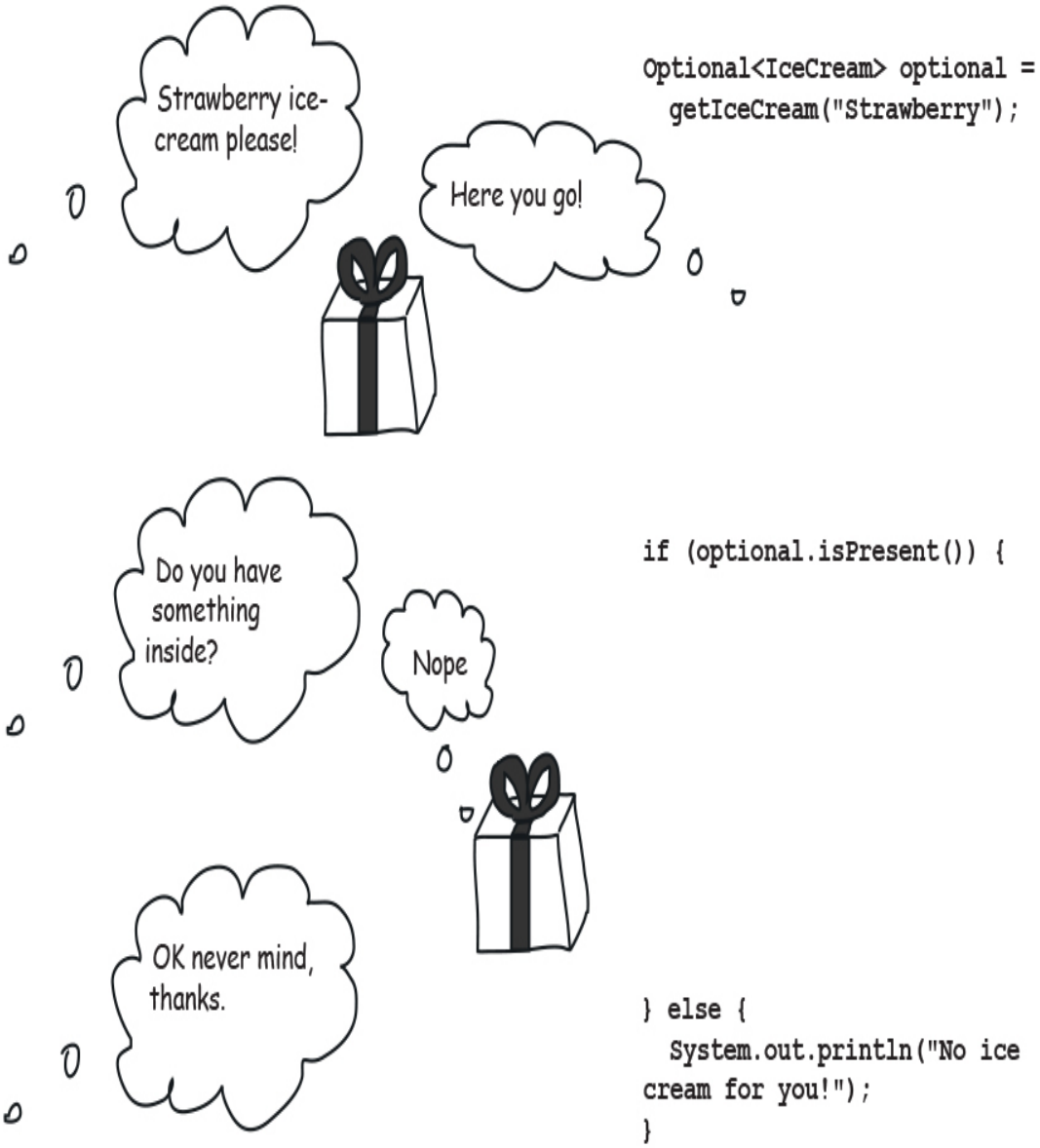
Optional is a wrapper

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 which *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.


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

if (optional.isPresent()) {

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

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 make their own decision about what to do if there isn't one.

Don’t forget to talk to the Optional wrapper

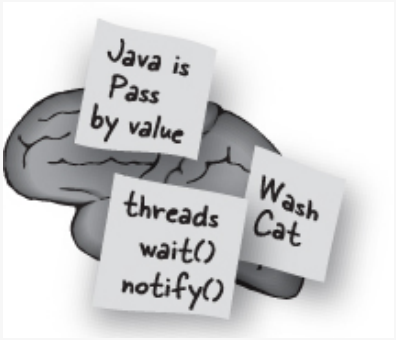
The important thing about Optional results is that **they can be empty**. If you don’t check first to see if there’s a value present and the result is empty, you will get an exception.


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

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

```
File Edit Window Help Boom
%java OptionalExamples

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

MAKE IT STICK



Roses are red, violets are blue,

If you don't call isPresent()

It's going to go BOOM!

Optional objects need to be asked if they contain something before you unwrap them, otherwise you'll get an exception if there's no result.

Five-Minute Mystery

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.

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 ?

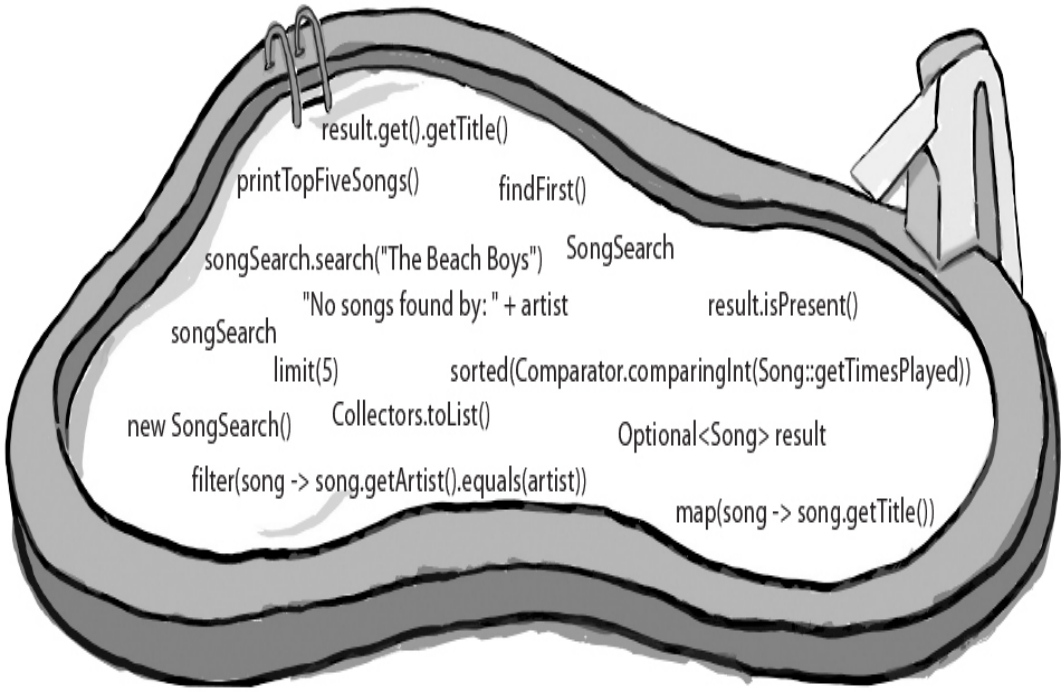
Pool Puzzle

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
```

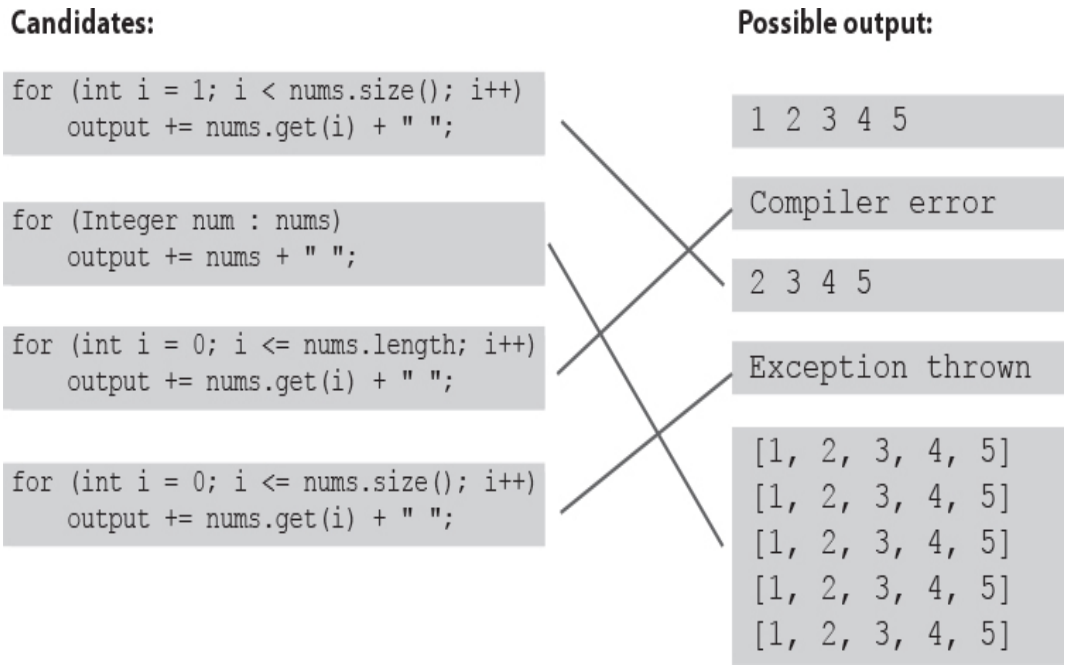
Note: each thing from the pool can only be used once!



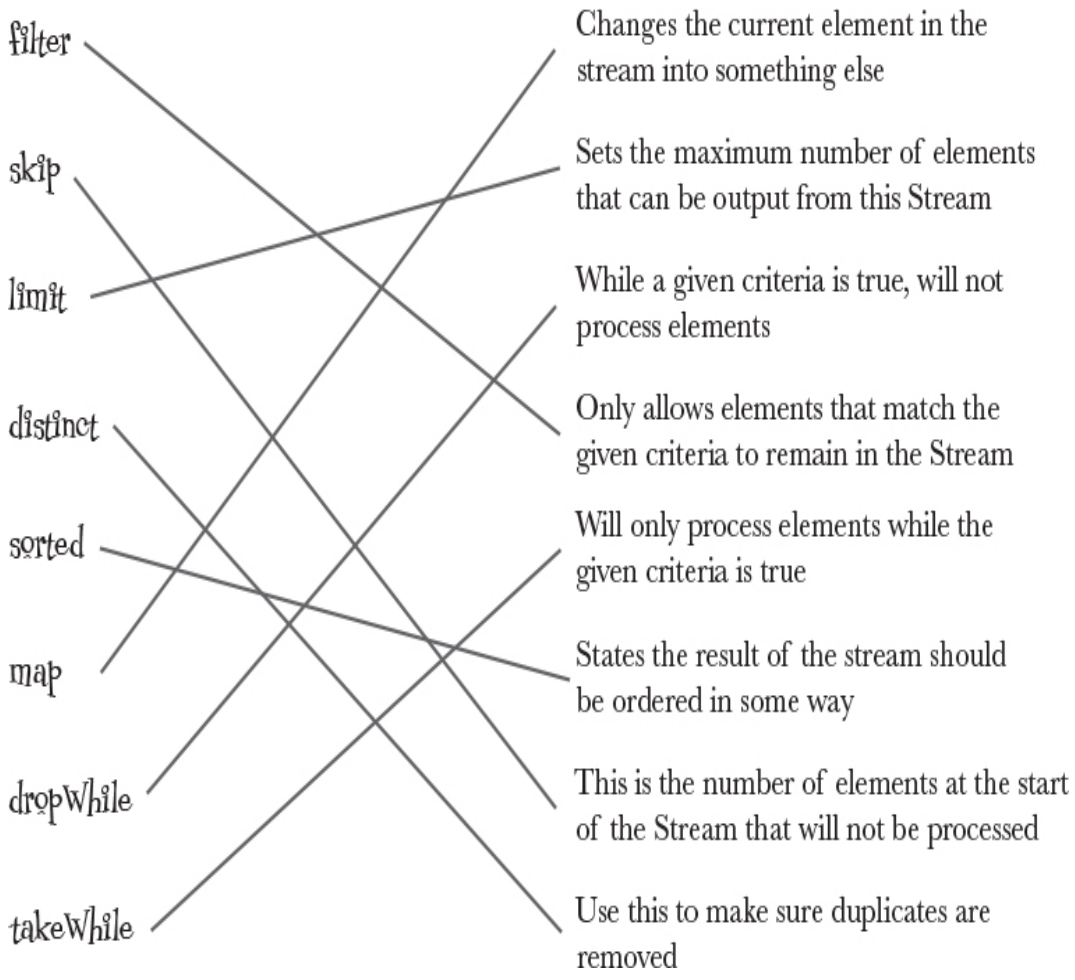

```
public class StreamPuzzle {

  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 Solution



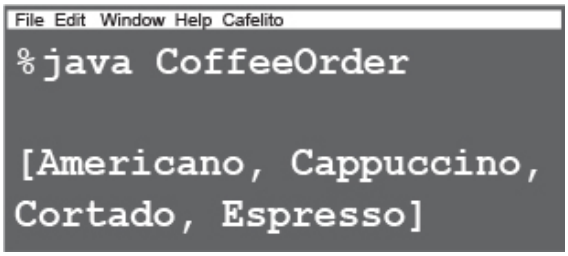
Who Does What? Solution



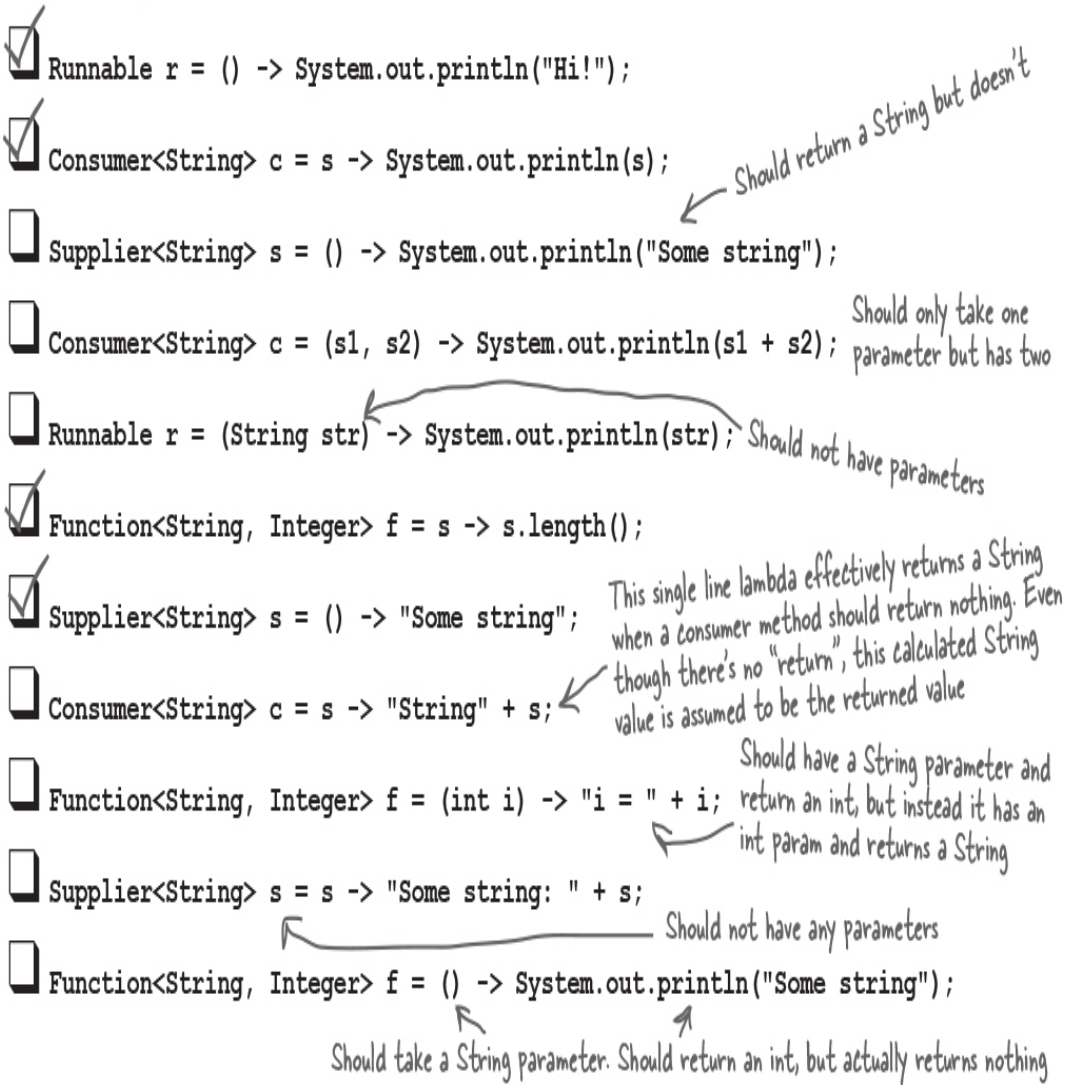
Code Magnets Solution

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

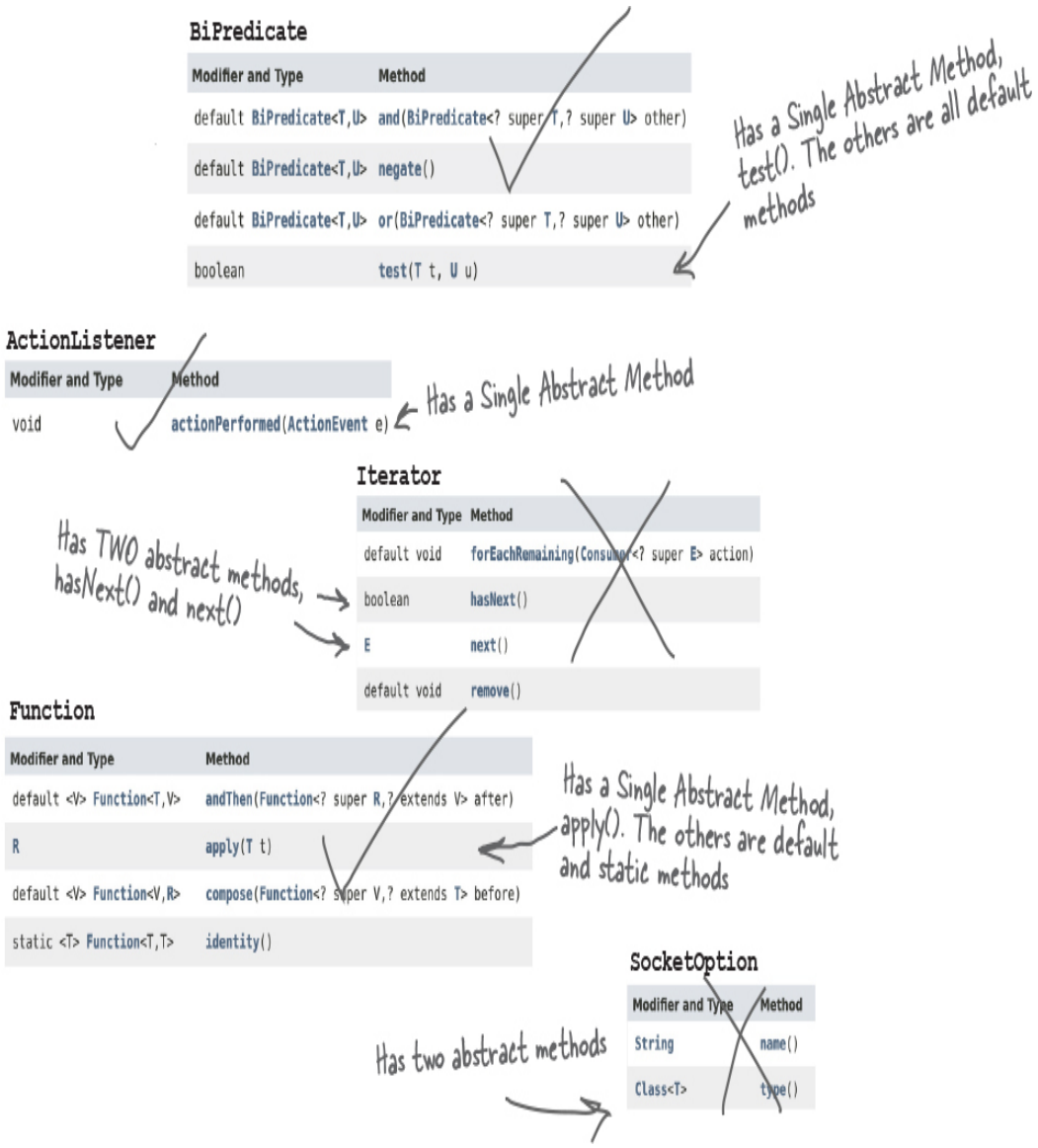
```
import java.util.*;
import java.util.stream.*;

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

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



Be the compiler Solution



Sharpen your pencil Solution



Five-Minute Mystery Solution

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

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

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

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

Pool Puzzle Solution

```
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);
    }
  }
}
```

Chapter 13. Exception Handling: 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

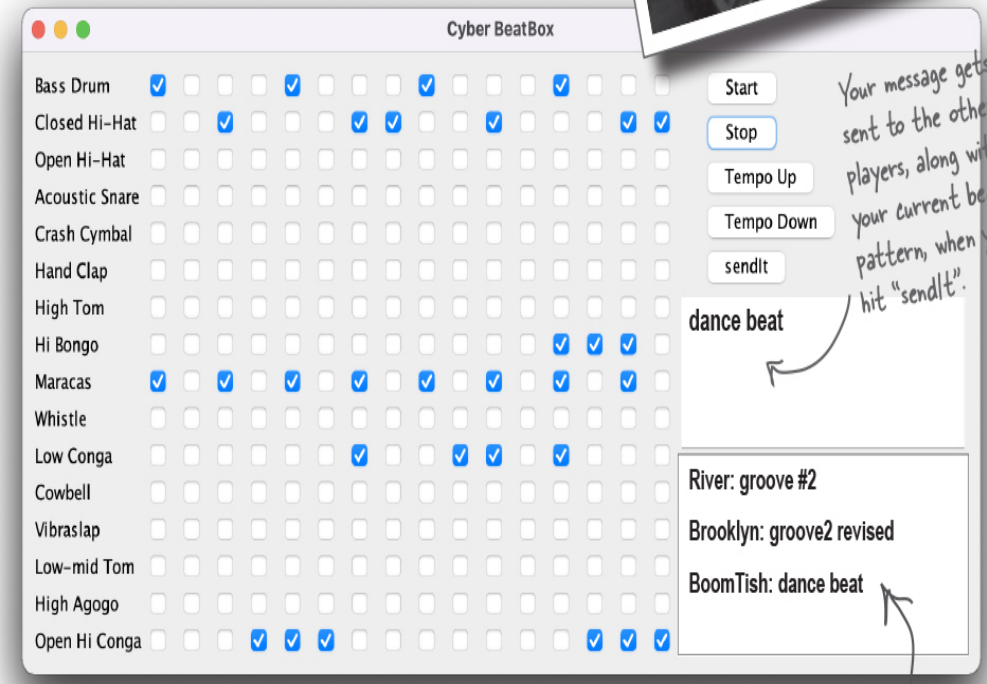
risky? And where do you put the code to *handle* the ***exceptional*** situation? So far in this book, we haven't *really* taken any risks. We've certainly had things go wrong at runtime, but the problems were mostly flaws in our own code. Bugs. And those we should fix at development time. No, the problem-handling 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 multi-player 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.

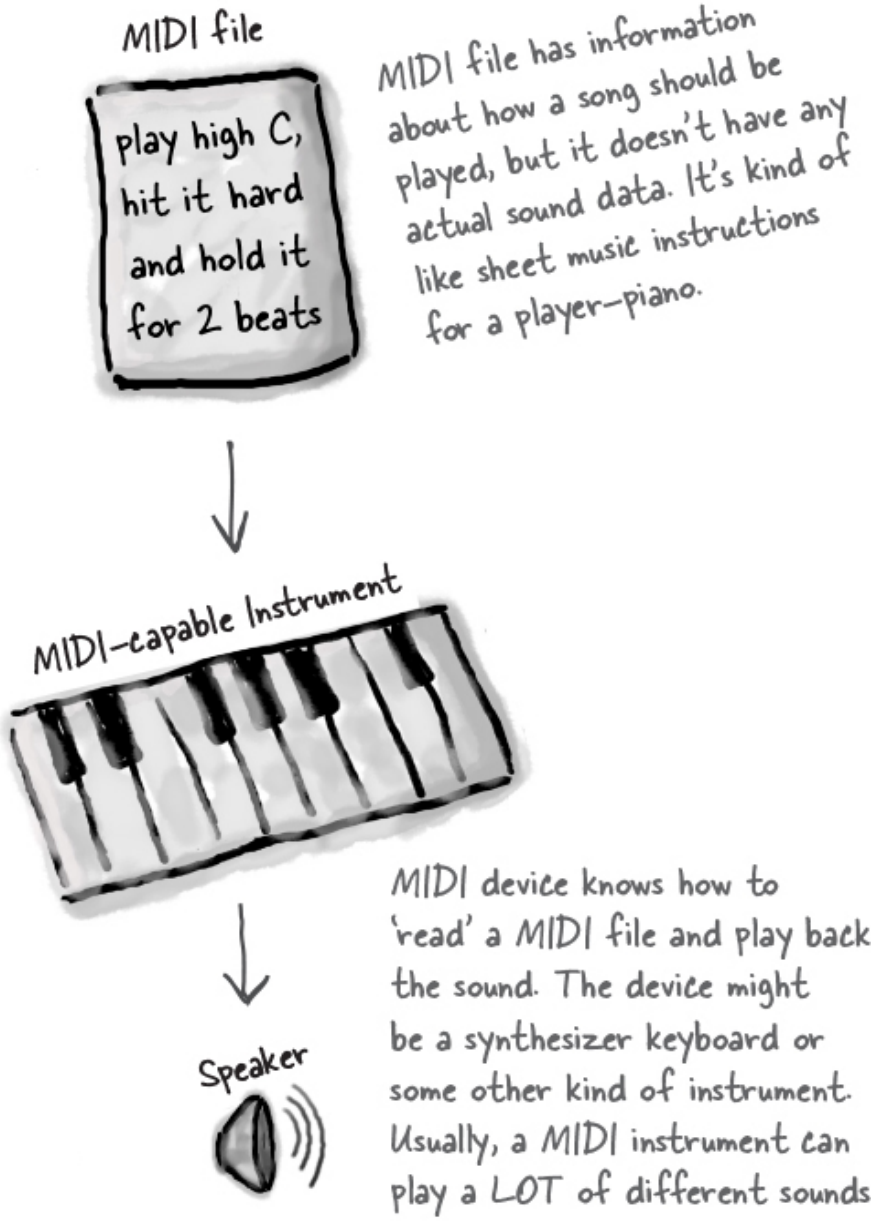
The JavaSound API

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*.



MIDI file has information about how a song should be played, but it doesn't have any actual sound data. It's kind of like sheet music instructions for a player-piano.

MIDI device knows how to 'read' a MIDI file and play back the sound. The device might be a synthesizer keyboard or some other kind of instrument. Usually, a MIDI instrument can play a LOT of different sounds (piano, drums, violin, etc.), and all at the same time. So a MIDI file isn't like sheet music for just one musician in the band -- it can hold the parts for ALL the musicians playing a particular song.

First we need a Sequencer

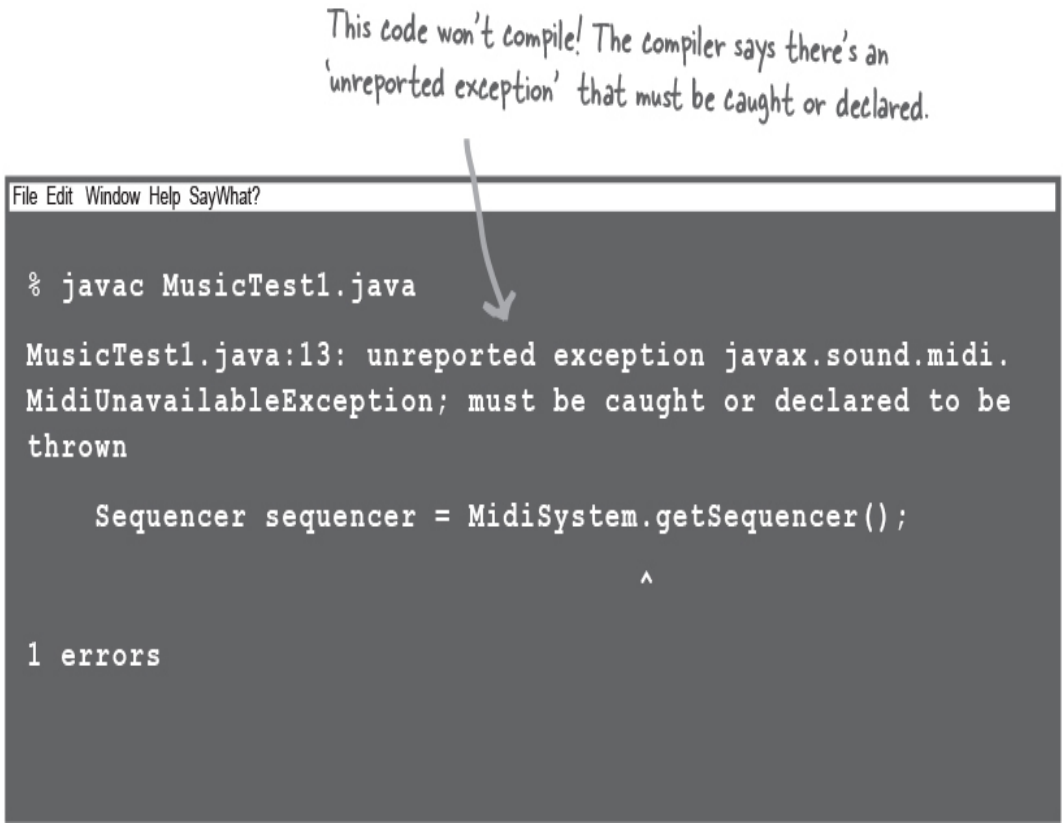
Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback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

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

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

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

import the javax.sound.midi package

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

Something's wrong!



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

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