

## Object-killer #1

**Reference goes out of scope, permanently.**



```
public class StackRef {
    public void foof() {
        barf();
    }

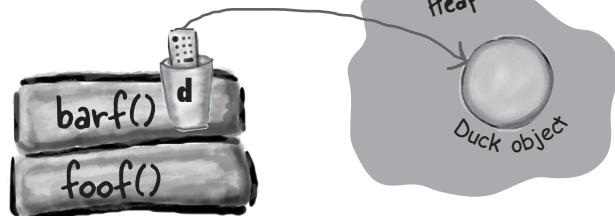
    public void barf() {
        Duck d = new Duck();
    }
}
```



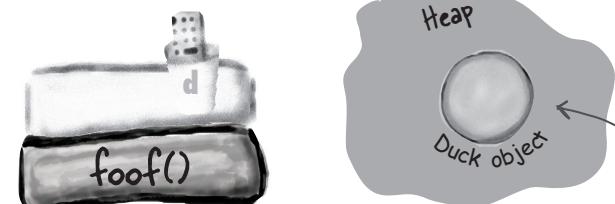
- 1 *foof()* is pushed onto the Stack; no variables are declared.



- 2 *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



- 3 *barf()* completes and pops off the Stack. Its frame disintegrates, so "d" is now dead and gone. Execution returns to *foof()*, but *foof()* can't use "d."



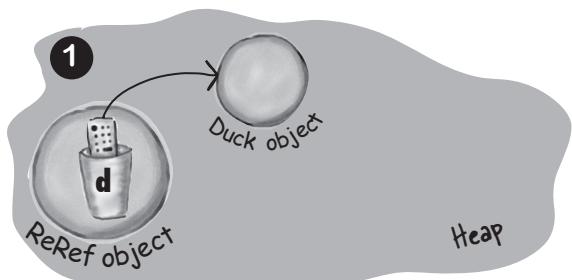
## Object-killer #2

Assign the reference  
to another object



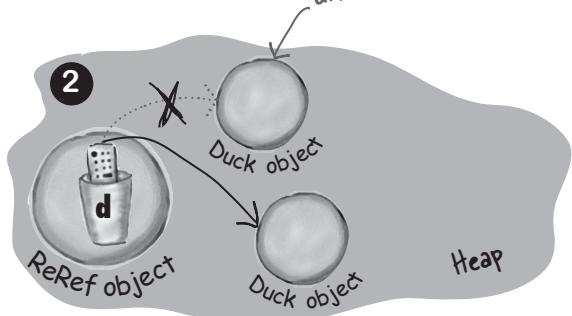
```
public class ReRef {
    Duck d = new Duck();

    public void go() {
        d = new Duck();
    }
}
```



The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...

When someone calls the go() method, this Duck is abandoned. Its only reference has been reprogrammed for a different Duck.



'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is now as good as dead.

Dude,  
all you had to do  
was reset the reference.  
Guess they didn't have  
memory management back  
then.



## Object-killer #3

Explicitly set the reference to null



```
public class ReRef {
    Duck d = new Duck();

    public void go() {
        d = null;
    }
}
```

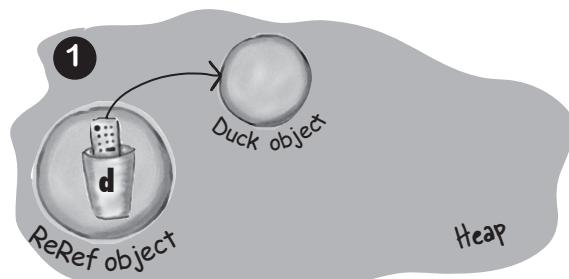
### The meaning of null

When you set a reference to `null`, you're deprogramming the remote control.

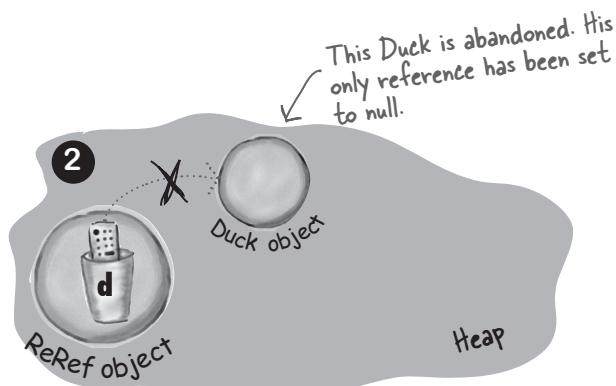
In other words, you've got a remote control, but no TV at the other end. A null reference has bits representing "null" (we don't know or care what those bits are, as long as the JVM knows).

If you have an unprogrammed remote control, in the real world, the buttons don't do anything when you press them. But in Java, you can't press the buttons (i.e., use the dot operator) on a null reference, because the JVM knows (this is a runtime issue, not a compiler error) that you're expecting a bark, but there's no Dog there to do it!

**If you use the dot operator on a null reference, you'll get a `NullPointerException` at runtime.** You'll learn all about exceptions in Chapter 13, *Risky Behavior*.



The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



'd' is set to null, which is just like having a remote control that isn't programmed to anything. You're not even allowed to use the dot operator on 'd' until it's reprogrammed (assigned an object).

## Fireside Chats



Tonight's Talk: **An instance variable and a local variable discuss life and death (with remarkable civility)**

### Instance Variable

I'd like to go first, because I tend to be more important to a program than a local variable. I'm there to support an object, usually throughout the object's entire life. After all, what's an object without state? And what is state? Values kept in *instance variables*.

No, don't get me wrong, I do understand your role in a method; it's just that your life is so short. So temporary. That's why they call you guys "temporary variables."

My apologies. I understand completely.

I never really thought about it like that. What are you doing while the other methods are running and you're waiting for your frame to be the top of the Stack again?

### Local Variable

I appreciate your point of view, and I certainly appreciate the value of object state and all, but I don't want folks to be misled. Local variables are *really* important. To use your phrase, "After all, what's an object without *behavior*?" And what is behavior? Algorithms in methods. And you can bet your bits there'll be some *local variables* in there to make those algorithms work.

Within the local-variable community, the phrase "temporary variable" is considered derogatory. We prefer "local," "stack," "automatic," or "Scope-challenged."

Anyway, it's true that we don't have a long life, and it's not a particularly *good* life either. First, we're shoved into a Stack frame with all the other local variables. And then, if the method we're part of calls another method, another frame is pushed on top of us. And if *that* method calls *another* method...and so on. Sometimes we have to wait forever for all the other methods on top of the Stack to complete so that our method can run again.

Nothing. Nothing at all. It's like being in stasis—that thing they do to people in science-fiction movies when they have to travel long distances. Suspended animation, really. We just sit there on hold. As long as our frame is still there, we're safe and the value we hold is secure, but it's a mixed blessing when our frame gets to run again. On the one hand, we get to be active again. On the other

**Instance Variable**

We saw an educational video about it once. Looks like a pretty brutal ending. I mean, when that method hits its ending curly brace, the frame is literally *blown* off the Stack! Now *that's* gotta hurt.

I live on the Heap, with the objects. Well, not *with* the objects, actually *in* an object. The object whose state I store. I have to admit life can be pretty luxurious on the Heap. A lot of us feel guilty, especially around the holidays.

OK, hypothetically, yes, if I'm an instance variable of the Collar and the Collar gets GC'd, then the Collar's instance variables would indeed be tossed out like so many pizza boxes. But I was told that this almost never happens.

They let us *drink*?

**Local Variable**

hand, the clock starts ticking again on our short lives. The more time our method spends running, the closer we get to the end of the method. We *all* know what happens then.

Tell me about it. In computer science they use the term *popped* as in “the frame was popped off the Stack.” That makes it sound fun, or maybe like an extreme sport. But, well, you saw the footage. So why don't we talk about you? I know what my little Stack frame looks like, but where do *you* live?

But you don't *always* live as long as the object who declared you, right? Say there's a Dog object with a Collar instance variable. Imagine *you're* an instance variable of the Collar object, maybe a reference to a Buckle or something, sitting there all happy inside the Collar object who's all happy inside the Dog object. But...what happens if the Dog wants a new Collar or *nulls* out its Collar instance variable? That makes the Collar object eligible for GC. So...if *you're* an instance variable inside the Collar and the whole Collar is abandoned, what happens to *you*?

And you believed it? That's what they say to keep us motivated and productive. But aren't you forgetting something else? What if you're an instance variable inside an object, and that object is referenced *only* by a *local* variable? If I'm the only reference to the object you're in, when I go, you're coming with me. Like it or not, our fates may be connected. So I say we forget about all this and go get a drink while we still can. Carpe RAM and all that.



## BE the Garbage Collector

Which of the lines of code on the right, if added to the class on the left at point A, would cause exactly one additional object to be eligible for the Garbage Collector? (Assume that point A (`//call more methods`) will execute for a long time, giving the Garbage Collector time to do its stuff.)

```
public class GC {  
    public static GC doStuff() {  
        GC newGC = new GC();  
        doStuff2(newGC);  
        return newGC;  
    }  
  
    public static void main(String[] args) {  
        GC gc1;  
        GC gc2 = new GC();  
        GC gc3 = new GC();  
        GC gc4 = gc3;  
        gc1 = doStuff();  
  
        // call more methods  
    }  
  
    public static void doStuff2(GC copyGC) {  
        GC localGC = copyGC;  
    }  
}
```

→ Answers on page 272.

- 1    `copyGC = null;`
- 2    `gc2 = null;`
- 3    `newGC = gc3;`
- 4    `gc1 = null;`
- 5    `newGC = null;`
- 6    `gc4 = null;`
- 7    `gc3 = gc2;`
- 8    `gc1 = gc4;`
- 9    `gc3 = null;`



# Popular Objects

```

class Bees {
    Honey[] beeHoney;
}

class Raccoon {
    Kit rk;
    Honey rh;
}

class Kit {
    Honey honey;
}

class Bear {
    Honey hunny;
}

public class Honey {
    public static void main(String[] args) {
        Honey honeyPot = new Honey();
        Honey[] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
        Bees bees = new Bees();
        bees.beeHoney = ha;
        Bear[] bears = new Bear[5];
        for (int i = 0; i < 5; i++) {
            bears[i] = new Bear();
            bears[i].hunny = honeyPot;
        }
        Kit kit = new Kit();
        kit.honey = honeyPot;
        Raccoon raccoon = new Raccoon();
        raccoon.rh = honeyPot;
        raccoon.rk = kit;
        kit = null;
    } // end of main
}

```

Here's a new Raccoon object!

Here's its reference variable 'raccoon.'

In this code example, several new objects are created. Your challenge is to find the object that is “most popular,” i.e., the one that has the most reference variables referring to it. Then list how many total references there are for that object, and what they are! We’ll start by pointing out one of the new objects and its reference variable.

Good luck!

→ Answers on page 272.

## puzzle: Five Minute Mystery



# Five-Minute Mystery



“We’ve run the simulation four times, and the main module’s temperature consistently drifts out of nominal toward cold,” Sarah said, exasperated. “We installed the new temp-bots last week. The readings on the radiator bots, designed to cool the living quarters, seem to be within spec, so we’ve focused our analysis on the heat retention bots, the bots that help to warm the quarters.” Tom sighed, at first it had seemed that nanotechnology was going to really put them ahead of schedule. Now, with only five weeks left until launch, some of the orbiter’s key life support systems were still not passing the simulation gauntlet.

“What ratios are you simulating?” Tom asked.

“Well, if I see where you’re going, we already thought of that,” Sarah replied. “Mission control will not sign off on critical systems if we run them out of spec. We are required to run the v3 radiator bot’s SimUnits in a 2:1 ratio with the v2 radiator’s SimUnits,” Sarah continued. “Overall, the ratio of retention bots to radiator bots is supposed to run 4:3.”

“How’s power consumption, Sarah?” Tom asked. Sarah paused, “Well, that’s another thing, power consumption is running higher than anticipated. We’ve got a team tracking that down too, but because the nanos are wireless, it’s been hard to isolate the power consumption of the radiators from the retention bots.” “Overall power consumption ratios,” Sarah continued, “are designed to run 3:2 with the radiators pulling more power from the wireless grid.”

“OK, Sarah,” Tom said. “Let’s take a look at some of the simulation initiation code. We’ve got to find this problem, and find it quick!”

```
import java.util.ArrayList;

class V2Radiator {
    V2Radiator(ArrayList<SimUnit> list) {
        for (int x = 0; x < 5; x++) {
            list.add(new SimUnit("V2Radiator"));
        }
    }
}

class V3Radiator extends V2Radiator {
    V3Radiator(ArrayList<SimUnit> lglist) {
        super(lglist);
        for (int g = 0; g < 10; g++) {
            lglist.add(new SimUnit("V3Radiator"));
        }
    }
}

class RetentionBot {
    RetentionBot(ArrayList<SimUnit> rlist) {
        rlist.add(new SimUnit("Retention"));
    }
}
```

# Five-Minute Mystery continued...

```

import java.util.ArrayList;

public class TestLifeSupportSim {
    public static void main(String[] args) {
        ArrayList<SimUnit> aList = new ArrayList<SimUnit>();
        V2Radiator v2 = new V2Radiator(aList);
        V3Radiator v3 = new V3Radiator(aList);
        for (int z = 0; z < 20; z++) {
            RetentionBot ret = new RetentionBot(aList);
        }
    }
}

class SimUnit {
    String botType;

    SimUnit(String type) {
        botType = type;
    }

    int powerUse() {
        if ("Retention".equals(botType)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

Tom gave the code a quick look, and a small smile crept across his lips. “I think I’ve found the problem, Sarah, and I bet I know by what percentage your power usage readings are off too!”

*What did Tom suspect? How could he guess the power readings errors, and what few lines of code could you add to help debug this program?*

—————> Answers on page 273.



## Exercise Solutions

## Be the Garbage Collector

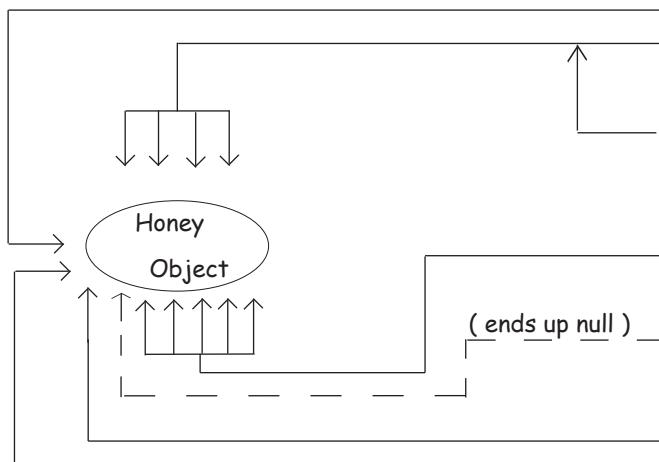
(from page 268)

- 1 copyGC = null; No—this line attempts to access a variable that is out of scope.
- 2 gc2 = null; OK—gc2 was the only reference variable referring to that object.
- 3 newGC = gc3; No—another out of scope variable.
  
- 4 gc1 = null; OK—gc1 had the only reference because newGC is out of scope.
- 5 newGC = null; No—newGC is out of scope.
  
- 6 gc4 = null; No—gc3 is still referring to that object.
  
- 7 gc3 = gc2; No—gc4 is still referring to that object.
  
- 8 gc1 = gc4; OK—Reassigning the only reference to that object.
  
- 9 gc3 = null; No—gc4 is still referring to that object.

## Popular Objects

(from page 269)

It probably wasn't too hard to figure out that the Honey object first referred to by the honeyPot variable is by far the most “popular” object in this class. But maybe it was a little trickier to see that all of the variables that point from the code to the Honey object refer to the **same object!** There are a total of 12 active references to this object right before the main() method completes. The kit.honeyPot variable is valid for a while, but kit gets nulled at the end. Since raccoon.rk still refers to the Kit object, raccoon.kit.honeyPot (although never explicitly declared) refers to the object!



```
public class Honey {
    public static void main(String[] args) {
        Honey honeyPot = new Honey();
        Honey[] ha = {honeyPot, honeyPot,
                     honeyPot, honeyPot};
        Bees bees = new Bees();
        bees.beeHoney = ha;
        Bear[] bears = new Bear[5];
        for (int i = 0; i < 5; i++) {
            bears[i] = new Bear();
            bears[i].hunny = honeyPot;
        }
        Kit kit = new Kit();
        kit.honey = honeyPot;
        Raccoon raccoon = new Raccoon();

        raccoon.rh = honeyPot;
        raccoon.rk = kit;
        kit = null;
    } // end of main
}
```



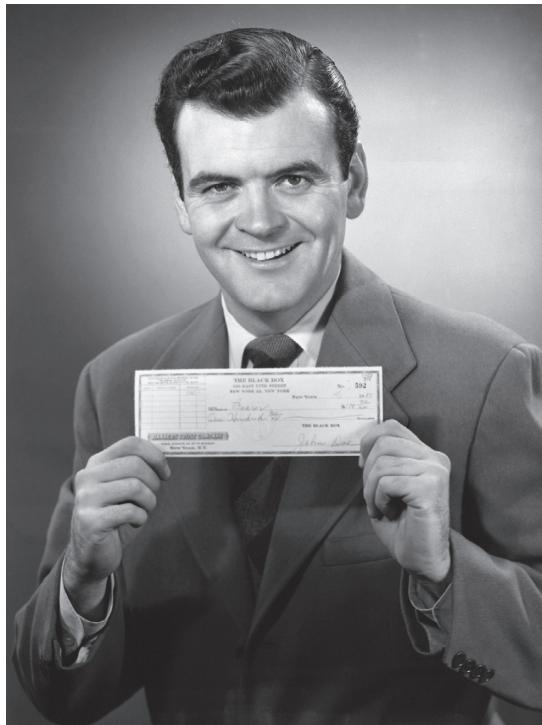
## Five-Minute Mystery (from pages 270–271)

Tom noticed that the constructor for the `V2Radiator` class took an `ArrayList`. That meant that every time the `V3Radiator` constructor was called, it passed an `ArrayList` in its `super()` call to the `V2Radiator` constructor. That meant that an extra five `V2Radiator` `SimUnits` were created. If Tom was right, total power use would have been 120, not the 100 that Sarah's expected ratios predicted.

Since all the Bot classes create `SimUnits`, writing a constructor for the `SimUnit` class, which printed out a line every time a `SimUnit` was created, would have quickly highlighted the problem!



# Numbers Matter



**Do the Math.** But there's more to working with numbers than just doing primitive arithmetic. You might want to get the absolute value of a number, or round a number, or find the larger of two numbers. You might want your numbers to print with exactly two decimal places, or you might want to put commas into your large numbers to make them easier to read. And what about parsing a String into a number? Or turning a number into a String? Someday you're gonna want to put a bunch of numbers into a collection like `ArrayList` that takes only objects. You're in luck. Java and the Java API are full of handy number-tweaking capabilities and methods, ready and easy to use. But most of them are **static**, so we'll start by learning what it means for a variable or method to be static, including constants in Java, also known as static *final* variables.

## MATH methods: as close as you'll ever get to a *global* method

Except there's no global *anything* in Java. But think about this: what if you have a method whose behavior doesn't depend on an instance variable value. Take the round() method in the Math class, for example. It does the same thing every time—rounds a floating-point number (the argument to the method) to the nearest integer. Every time. If you had 10,000 instances of class Math, and ran the round(42.2) method, you'd get an integer value of 42. Every time. In other words, the method acts on the argument but is never affected by an instance variable state. The only value that changes the way the round() method runs is the argument passed to the method!

Doesn't it seem like a waste of perfectly good heap space to make an instance of class Math simply to run the round() method? And what about *other* Math methods like min(), which takes two numerical primitives and returns the smaller of the two? Or max(). Or abs(), which returns the absolute value of a number.

### **These methods never use instance variable values.**

In fact, the Math class doesn't *have* any instance variables. So there's nothing to be gained by making an instance of class Math. So guess what? You don't have to. As a matter of fact, you can't.

### If you try to make an instance of class Math:

```
Math mathObject = new Math();
```

### You'll get this error:

```
File Edit Window Help IwasToldThereWouldBeNoMaths
% javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math
    Math mathObject = new Math();
                           ^
1 error
```

← This error shows that the Math constructor is marked private! That means you can NEVER say 'new' on the Math class to make a new Math object.

Methods in the Math class don't use any instance variable values. And because the methods are "static," you don't need to have an instance of Math. All you need is the Math class.

```
long x = Math.round(42.2);
int y = Math.min(56, 12);
int z = Math.abs(-343);
```

↑  
These methods never use instance variables, so their behavior doesn't need to know about a specific object.

# The difference between regular (non-static) and static methods

Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class. The keyword **static** lets a method run **without any instance of the class**. A static method means “behavior not dependent on an instance variable, so no instance/object is required. Just the class.”

## regular (non-static) method

```
public class Song {  
    String title;
```

Instance variable value affects the behavior of the play() method.

```
    public Song(String t) {  
        title = t;  
    }
```

```
    public void play() {  
        SoundPlayer player = new SoundPlayer();  
        player.playSound(title);  
    }  
}
```

Song
title
play()

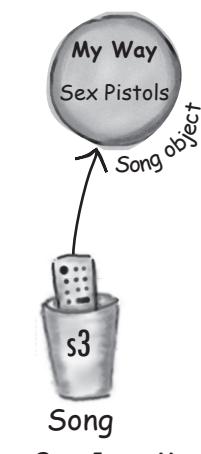
The current value of the 'title' instance variable is the song that plays when you call play().

two instances of class Song



s2.play();

Calling play() on this reference will cause "Politik" to play.



s3.play();

Calling play() on this reference will cause "My Way" to play.

## static method

```
public static int min(int a, int b) {  
    //returns the smallest of a and b  
}
```

Math
min()
max()
abs()
...

No instance variables. The method behavior doesn't change with instance variable state.

Math.min(42, 36);

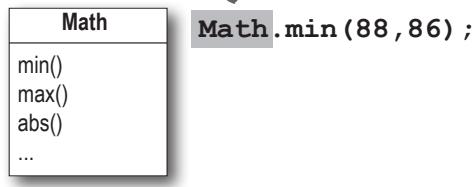
Use the Class name, rather than a reference variable name.



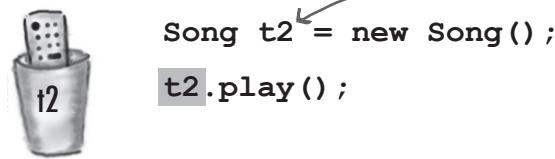
NO OBJECTS!!

Absolutely NO OBJECTS anywhere in this picture!

## Call a static method using a class name



## Call a non-static method using a reference variable name



## What it means to have a class with static methods

Often (although not always), a class with static methods is not meant to be instantiated. In Chapter 8, *Serious Polymorphism*, we talked about abstract classes, and how marking a class with the `abstract` modifier makes it impossible for anyone to say “new” on that class type. In other words, **it’s impossible to instantiate an abstract class.**

But you can restrict other code from instantiating a *non-abstract* class by marking the constructor `private`. Remember, a *method* marked private means that only code from within the class can invoke the method. A *constructor* marked private means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say “new” from *outside* the class. That’s how it works with the `Math` class, for example. The constructor is private; you cannot make a new instance of `Math`. The compiler knows that your code doesn’t have access to that private constructor.

This does *not* mean that a class with one or more static methods should never be instantiated. In fact, every class you put a `main()` method in is a class with a static method in it!

Typically, you make a `main()` method so that you can launch or test another class, nearly always by instantiating a class in `main` and then invoking a method on that new instance.

So you’re free to combine static and non-static methods in a class, although even a single non-static method means there must be *some* way to make an instance of the class. The only ways to get a new object are through “new” or serialization (or something called the Java Reflection API that we don’t go into). No other way. But exactly *who* says new can be an interesting question, and one we’ll look at a little later in this chapter.

# Static methods can't use non-static (instance) variables!

Static methods run without knowing about any particular instance of the static method's class. And as you saw on the previous pages, there might not even *be* any instances of that class. Since a static method is called using the *class* (`Math.random()`) as opposed to an *instance reference* (`t2.play()`), a static method can't refer to any instance variables of the class. The static method doesn't know *which* instance's variable value to use.

## If you try to compile this code:

```
public class Duck {
    private int size;

    public static void main(String[] args) {
        System.out.println("Size of duck is " + size);
    }

    public void setSize(int s) {
        size = s;
    }

    public int getSize() {
        return size;
    }
}
```

*Which Duck?  
Whose size?*

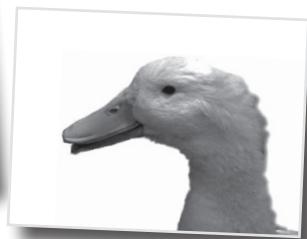
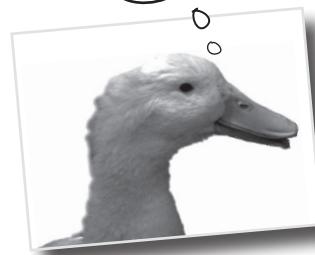
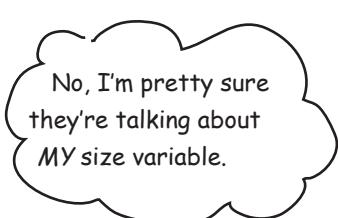
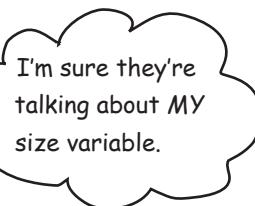
*If there's a Duck on  
the heap somewhere, we  
don't know about it.*

*Static context. Everything  
else in the class is NOT static.*

## You'll get this error:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable
size cannot be referenced from a
static context
        System.out.println("Size
of duck is " + size);
^
```

If you try to use an instance variable from inside a static method, the compiler thinks, “I don’t know which object’s instance variable you’re talking about!” If you have ten Duck objects on the heap, a static method doesn’t know about any of them.



# Static methods can't use non-static methods, either!

What do non-static methods do? **They usually use instance variable state to affect the behavior of the method.** A getName() method returns the value of the name variable. Whose name? The object used to invoke the getName() method.

## This won't compile:

```
public class Duck {
    private int size;

    public static void main(String[] args) {
        System.out.println("Size is " + getSize());
    }

    public void setSize(int s) {
        size = s;
    }

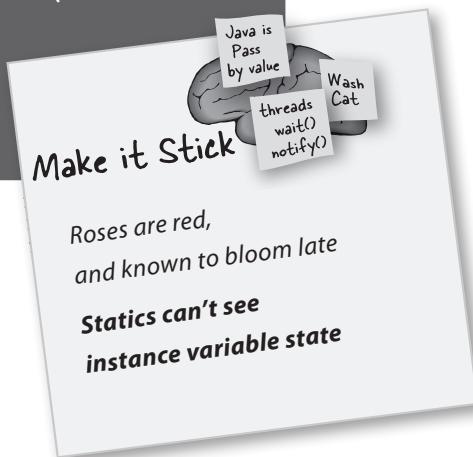
    public int getSize() {
        return size;
    }
}
```

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.

Back to the same problem...  
whose size?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:5: error: non-static
method getSize() cannot be refer-
enced from a static context

    System.out.println("Size is " +
 getSize());
^
1 error
```



there are no  
**Dumb Questions**

**Q:** What if you try to call a non-static method from a static method, but the non-static method doesn't use any instance variables. Will the compiler allow that?

**A:** No. The compiler knows that whether you do or do not use instance variables in a non-static method, you can. And think about the implications...if you were allowed to compile a scenario like that, then what happens if in the future you want to change the implementation of that non-static method so that one day it does use an instance variable? Or worse, what happens if a subclass overrides the method and uses an instance variable in the overriding version?

**Q:** I could swear I've seen code that calls a static method using a reference variable instead of the class name.

**A:** You can do that, but as your mother always told you, "Just because it's legal doesn't mean it's good." Although it works to call a static method using any instance of the class, it makes for misleading (less-readable) code. You can say,

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

This code is legal, but the compiler just resolves it back to the real class anyway ("OK, d is of type Duck, and main() is static, so I'll call the static main() in class Duck"). In other words, using d to invoke main() doesn't imply that main() will have any special knowledge of the object that d is referencing. It's just an alternate way to invoke a static method, but the method is still static!

# Static variable: value is the same for ALL instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
    int duckCount = 0;
    public Duck() {
        duckCount++;
```

this would always set  
duckCount to 1 each time  
a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgy. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance*.

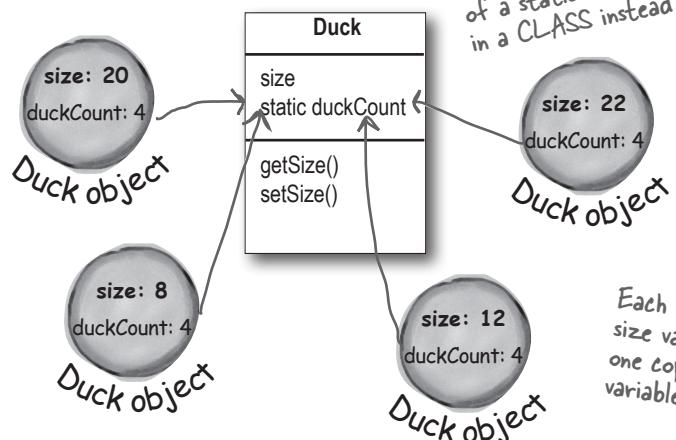
```
public class Duck {
    private int size;
    private static int duckCount = 0;
```

```
public Duck() {
    duckCount++;
```

Now it will keep  
incrementing each time  
the Duck constructor runs,  
because duckCount is static  
and won't be reset to 0.

```
public void setSize(int s) {
    size = s;
```

```
public int getSize() {
    return size;
}
```



A Duck object doesn't keep its own copy  
of duckCount.  
Because duckCount is static, Duck objects  
all share a single copy of it. You can think  
of a static variable as a variable that lives  
in a CLASS instead of in an object.

Each Duck object has its own  
size variable, but there's only  
one copy of the duckCount  
variable—the one in the class.

## static variables



**Static variables are shared.**

All instances of the same class share a single copy of the static variables.

instance variables: 1 per **instance**

static variables: 1 per **class**



Earlier in this chapter, we saw that a private constructor means that the class can't be instantiated from code running outside the class. In other words, only code from within the class can make a new instance of a class with a private constructor. (There's a kind of chicken-and-egg problem here.)

What if you want to write a class in such a way that only ONE instance of it can be created, and anyone who wants to use an instance of the class will always use that one, single instance?

# Initializing a static variable

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

And there are two guarantees about static initialization:

- Static variables in a class are initialized before any *object* of that class can be created.
- Static variables in a class are initialized before any *static method* of the class runs.

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

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

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

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

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

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

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

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

Primitive floating points (float, double): 0.0

boolean: false

object references: null

File Edit Window Help What?

% java PlayerTestDrive

0 ← Before any instances are made

1 ← After an object is created

## static final variables are constants

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

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

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

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

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

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one. **Constant variable names are usually in all caps!**

A **static initializer** is a block of code that runs when a class is loaded, before any other code can use the class, so it's a great place to initialize a static final variable.

```
class ConstantInit1 {  
    final static int X;  
    static {  
        X = 42;  
    }  
}
```

### Initialize a *final* static variable:

- ① At the time you declare it:

```
public class ConstantInit2 {  
    public static final int X_VALUE = 25;  
}  
} Notice the naming convention—static  
final variables are constants, so the  
name should be all uppercase, with an  
underscore separating the words.
```

OR

- ② In a static initializer:

```
public class ConstantInit3 {  
    public static final double VAL;  
  
    static {  
        VAL = Math.random();  
    }  
}
```

This code runs as soon as the class is loaded, before any static method variable can be used.

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

```
public class ConstantInit3 {  
    public static final double VAL;  
}  
} no initialization!
```

The compiler will catch it:

```
File Edit Window Help Init?  
% javac ConstantInit3.java  
ConstantInit3.java:2: error: vari-  
able VAL not initialized in the  
default constructor  
    public static final double VAL;  
                           ^  
1 error
```

# final isn't just for static variables...

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

## non-static final variables

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

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

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

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

## final method

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

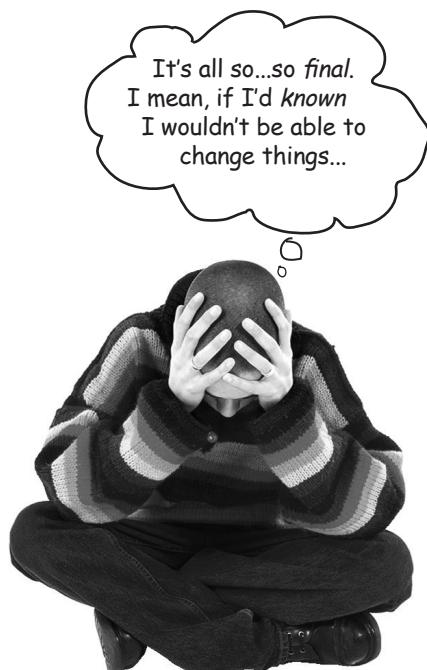
## final class

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

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

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

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



## there are no Dumb Questions

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

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

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

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

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

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

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

### BULLET POINTS

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

```
static {  
    DOG_CODE = 420;  
}
```
- The naming convention for constants (final static variables) is to make the name all uppercase and use underscores (\_) to separate the words.
- A final variable value cannot be changed once it has been assigned.
- Assigning a value to a final *instance* variable must be either at the time it is declared or in the constructor.
- A final method cannot be overridden.
- A final class cannot be extended (subclassed).



## What's Legal?

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



① public class Foo {  
 static int x;

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

② public class Foo2 {
 int x;

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

③ public class Foo3 {
 final int x;

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

④ public class Foo4 {  
 static final int x = 12;

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

⑤ public class Foo5 {  
 static final int x = 12;

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

⑥ public class Foo6 {  
 int x = 12;

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

→ Answers on page 308.

# Math methods

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

All Methods	Static Methods	Concrete Methods	
Modifier and Type	Method	Description	
static double	<code>abs(double a)</code>	Returns the absolute value of a double.	
static float	<code>abs(float a)</code>	Returns the absolute value of a float.	
static int	<code>abs(int a)</code>	Returns the absolute value of an integer.	
static long	<code>abs(long a)</code>	Returns the absolute value of a long.	
static int	<code>absExact(int a)</code>	Returns the mathematical absolute value of an integer. Returns the mathematical absolute value of an integer. ArithmeticException if the result is too large.	
static long	<code>absExact(long a)</code>	Returns the mathematical absolute value of a long. Returns the mathematical absolute value of a long. ArithmeticException if the result is too large.	
static double	<code>acos(double a)</code>	Returns the arc cosine of a value; the result is in radians.	
static int	<code>addExact(int x, int y)</code>	Returns the sum of its arguments.	
static long	<code>addExact(long x, long y)</code>	Returns the sum of its arguments.	
static double	<code>asin(double a)</code>	Returns the arc sine of a value; the result is in radians.	
static double	<code>atan(double a)</code>	Returns the arc tangent of a value; the result is in radians.	
static double	<code>atan2(double y, double x)</code>	Returns the angle theta from the x-axis to the point (x, y).	
static double	<code>cbrt(double a)</code>	Returns the cube root of a double.	
static double	<code>ceil(double a)</code>	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.	
static double	<code>copySign(double magnitude, double sign)</code>	Returns the first floating-point argument with the sign of the second.	
static float	<code>copySign(float magnitude, float sign)</code>	Returns the first floating-point argument with the sign of the second.	
static double	<code>cos(double a)</code>	Returns the trigonometric cosine of an angle.	
static double	<code>cosh(double x)</code>	Returns the hyperbolic cosine of a double.	

## Math.abs()

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

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

## Math.random()

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

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

We've been using this method so far, but there's also `java.util.Random`, which is a bit nicer to use.

## Math.round()

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

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

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

long z = Math.round(24.45); // returns 24L
                           ↙
                           This is a double.
```

## Math.min()

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

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

## Math.max()

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

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

## Math.sqrt()

Returns the positive square root of the argument. The method takes a double, but of course you can pass in anything that fits in a double.

```
double x = Math.sqrt(9); // return 3
double y = Math.sqrt(42.0); // returns 6.48074069840786
```

# Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, collections like ArrayList only work with Objects:

```
ArrayList<???> list;
```

*Can we create an  
ArrayList for ints?*

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

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

**Boolean**

**Character**

**Byte**

**Short**

**Integer**

**Long**

**Float**

**Double**

## wrapping a value

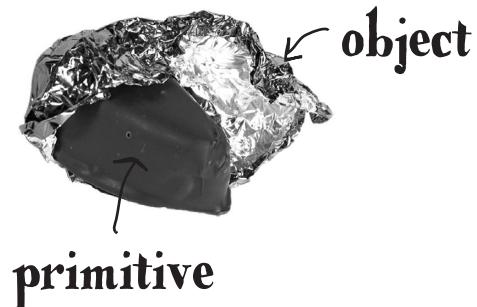
```
int i = 288;
Integer iWrap = new Integer(i);
```

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

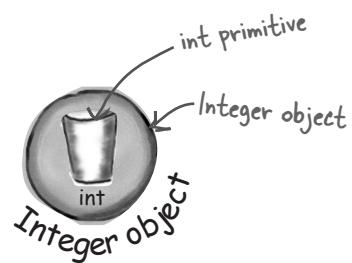
## unwrapping a value

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

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



When you need to treat a primitive like an object, wrap it.



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



This is stupid. You mean I can't just make an ArrayList of ints??? I have to wrap every single frickin' one in a new Integer object and then unwrap it when I try to access that value in the ArrayList? That's a waste of time and an error waiting to happen...

## Java will Autobox primitives for you

In The Olden Days (pre-Java 5), we did have to do all this ourselves, manually wrapping and unwrapping primitives. Fortunately, now it's all done for us *automatically*.

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

### An ArrayList of primitive ints

```
public void autoboxing() {
    int x = 32;
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(x); Just add it!
    int num = list.get(0);
}
```

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

Make an ArrayList of type Integer.



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

*there are no*  
**Dumb Questions**

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

**A:** Because...you can't. At least, not in the versions of Java this book covers (the language is constantly evolving and things may change!). Remember, the rule for generic types is that you can specify only class or interface types, *not primitives*. So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to prevent you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

# Autoboxing works almost everywhere

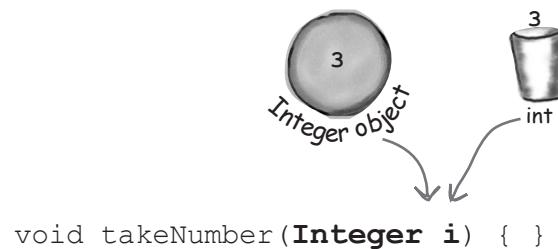
Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection...it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

## Fun with autoboxing

---

### Method arguments

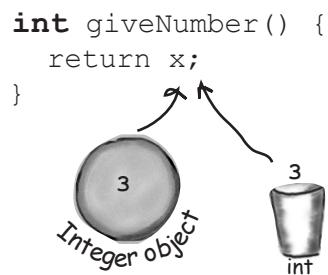
If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.



---

### Return values

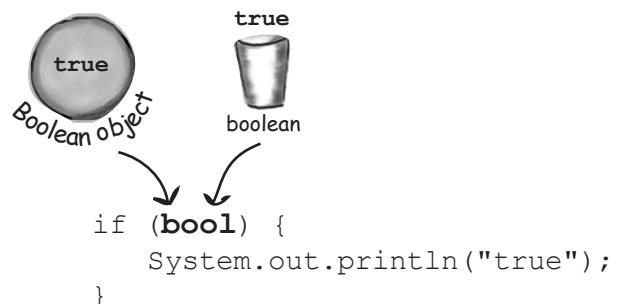
If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.



---

### Boolean expressions

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



## Operations on numbers

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

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

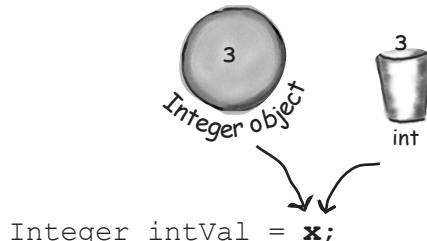
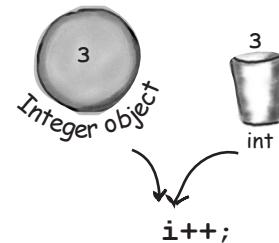
```
Integer i = new Integer(42);
i++;
```

And that means you can also do things like:

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

## Assignments

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



## Sharpen your pencil

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

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

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

→ Yours to solve.

```
public class TestBox {
    private Integer i;
    private int j;

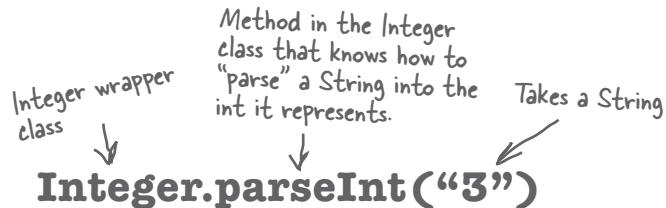
    public static void main(String[] args) {
        TestBox t = new TestBox();
        t.go();
    }

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

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

Besides acting like a normal class, the wrappers have a bunch of really useful static methods.

For example, the *parse* methods take a String and give you back a primitive value.



**Converting a String to a primitive value is easy:**

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");
boolean b = Boolean.parseBoolean("True");
```

No problem to parse "2" into 2.

The `parseBoolean()` method ignores the cases of the characters in the String argument.

**But if you try to do this:**

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

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

**You'll get a runtime exception:**

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at Snippets.badParse(Snippets.java:48)
        at Snippets.main(Snippets.java:54)
```

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

(We'll talk about exceptions in Chapter 13, *Risky Behavior*.)

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

You may want to turn a number into a String, for example when you want to show this number to a user or put it into a message. There are several ways to turn a number into a String. The easiest is to simply concatenate the number to an existing String.

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

Remember the 't' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

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

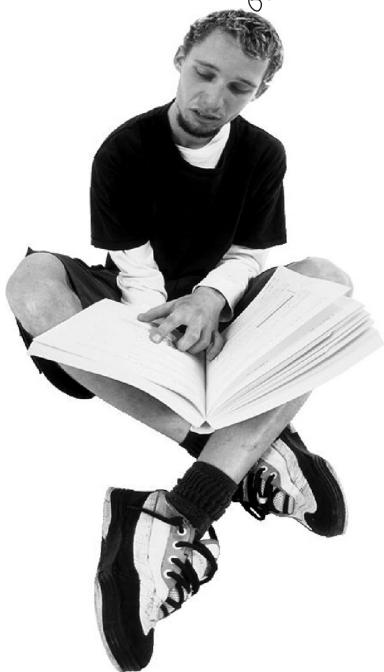
Another way to do it using a static method in class Double.

```
double d = 42.5;
String doubleString = String.valueOf(d);
```

There's also an overloaded a static method "valueOf" on String that will get the String value of pretty much anything.

Yeah,  
but how do I make  
it look like money? With a  
dollar sign and two decimal  
places like \$56.87 or what if I  
want commas like 45,687,890  
or what if I want it in...

Where's my printf  
like I have in C? Is  
number formatting part of  
the I/O classes?



# Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI.

The Java API provides powerful and flexible formatting using the `Formatter` class in `java.util`. But often you don't need to create and call methods on the `Formatter` class yourself, because the Java API has convenience methods in some of the I/O classes (including `printf()`) and the `String` class. So it can be a simple matter of calling a static `String.format()` method and passing it the thing you want formatted along with formatting instructions.

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

We'll start here with a basic example and then look at how it works. (Note: we'll revisit formatting again in Chapter 16, *Saving Objects*.)

## Making big numbers more readable with underscores, a quick detour

Before we get into formatting numbers, let's take a small, useful detour. Sometimes you'll want to declare variables with large initial values. Let's look at three declarations that assign the same large value, a billion, to long primitives:

```
long hardToRead = 1000000000;
long easierToRead = 1_000_000_000; ←
long legalButSilly = 10_0000_0000;
```

*When you're assigning large values, properly located underscores will make your life easier!*

## Formatting a number to use commas

```
public class TestFormats {
    public static void main(String[] args) {
        long myBillion = 1_000_000_000;
        String s = String.format("%,d", myBillion);
        System.out.println(s);
    }
}
```

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

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

1,000,000,000  
*Now we get commas inserted into the number.*

# Formatting deconstructed...

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

## ① Formatting instructions

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

## ② The argument to be formatted.

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

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

Do this...    to this.

①    ②

```
format("%,d", 1_000_000_000);
```

Use these instructions...on this argument.

## What do these instructions actually say?

“Take the second argument to this method, and format it as a **decimal integer** and insert **commas**.”

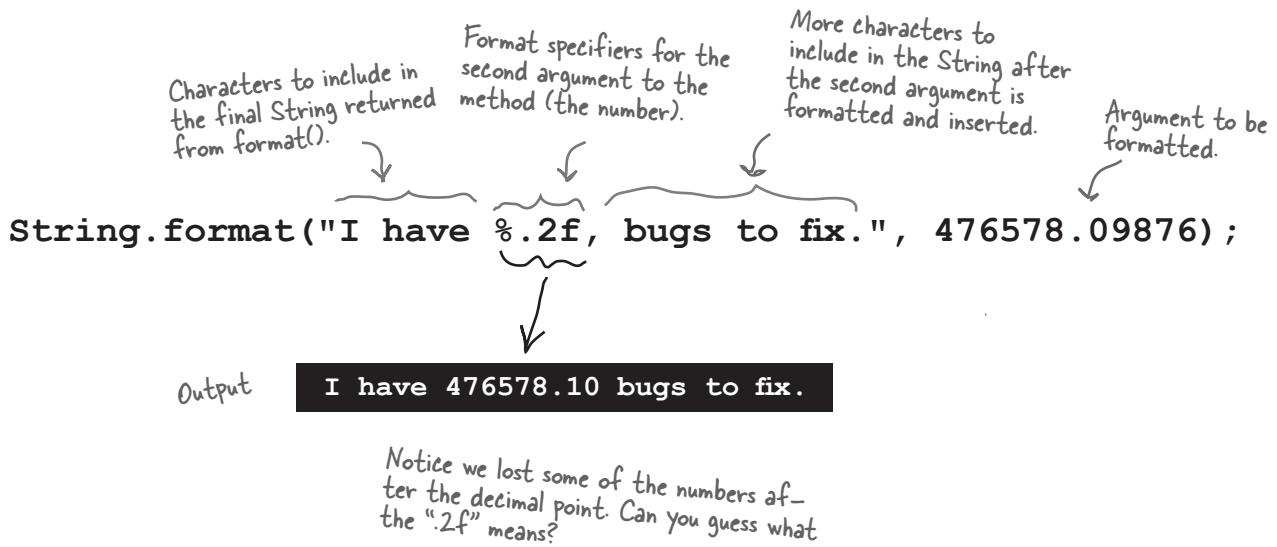
## How do they say that?

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

## the format() method

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

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



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

## Adding a comma

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

`I have 476,578.10 bugs to fix.`

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



But how does it even KNOW where the instructions end and the rest of the characters begin? How come it doesn't print out the "f" in "%.**2f**"? Or the "2"? How does it know that the .**2f** was part of the instructions and NOT part of the String?

## The format String uses its own little language syntax

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

You've already seen some examples:

**%,d** means “insert commas and format the number as a decimal integer.”

and

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

and

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

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

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

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

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

# The format specifier

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

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

% [argument number] [flags] [width] [.precision] **type**

We'll get to this later...  
it lets you say WHICH argument if there's more than one. (Don't worry about it just yet.)

These are for special formatting options like inserting commas, putting negative numbers in parentheses, or making the numbers left justified.

This defines the MINIMUM number of characters that will be used. That's \*minimum\* not TOTAL. If the number is longer than the width, it'll still be used in full, but if it's less than the width, it'll be padded with zeros.

You already know this one...it defines the precision. In other words, it sets the number of decimal places. Don't forget to include the “.” in there.

Type is mandatory (see the next page) and will usually be “d” for a decimal integer or “f” for a floating-point number.

% [argument number] [flags] [width] [.precision] **type**

format ("% , 6.1f", 42.000);

There's no “argument number” specified in this format String, but all the other pieces are there.

The value we want to format. Quite important.

# The only required specifier is for TYPE

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

**The TYPE is mandatory, everything else is optional.**

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


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

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

**%f**    **floating point**  
`format("%.3f", 42.000000)`  


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

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

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

**%x**    **hexadecimal**  
`format("%x", 42)`  


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

**%c**    **character**  
`format("%c", 42)`  


The number 42 represents the char "\*".

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

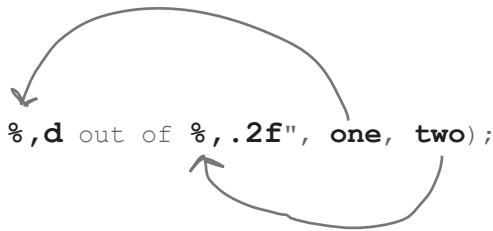
# What happens if I have more than one argument?

Imagine you want a String that looks like this:

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

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

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



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

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

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

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

there are no  
Dumb Questions

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

**A:** Good catch. Yes, there *is* something strange going on, and no there are *not* a bunch of overloaded `format()` methods to take a different number of possible arguments. In order to support this formatting (printf-like) API in Java, the language needed another feature—*variable argument lists* (called *varargs* for short). We’ll talk about varargs more in Appendix B.

# Just one more thing...static imports

Static imports are a real mixed blessing. Some people love this idea, some people hate it. Static imports exist to make your code a little shorter. If you hate to type or hate long lines of code, you might just like this feature. The downside to static imports is that—if you’re not careful—using them can make your code a lot harder to read.

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

## Without static imports:

```
class NoStatic {
    public static void main(String[] args) {
        System.out.println("sqrt " + Math.sqrt(2.0));
        System.out.println("tan " + Math.tan(60));
    }
}
```

## Same code, with static imports:

```
import static java.lang.Math.*;
import static java.lang.System.out;
class WithStatic {
    public static void main(String[] args) {
        out.println("sqrt " + sqrt(2.0));
        out.println("tan " + tan(60));
    }
}
```

*This might be a BAD place to use a static import. Removing the "System" class makes it unclear what this is and where it came from. Also it may lead to naming conflicts; you can't create any other variables called "out" now.*

*The syntax to use when declaring static imports.*

*Static imports in action.*

*You might want to use static imports for these methods. It makes the code shorter, and you don't need the "Math." prefix to understand what these operations are.*

## Use carefully:

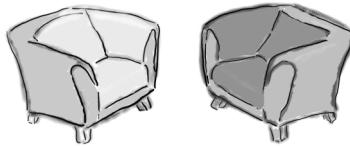
Static imports can make your code confusing to read.

Always re-read your code after using a static import and think: "Will I understand this in six months time?"

## Caveats & Gotchas

- Using a static import removes the information about which class the static came from. We'd advise using static imports only when the static method or variable still means something when it's not prefixed with the class name.
- A big issue with static imports is that it's easy to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use? So it's best not to use a static import if it's possible to create a conflict.
- Notice that you can use wildcards (\*), in your static import declaration.

## Fireside Chats



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

### Instance Variable

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

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

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

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

### Static Variable

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

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

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

Oh, like debugging isn't important?

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

**Instance Variable**

Um, aren't you forgetting something?

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

You're like a global variable, and any programmer  
 worth their sticker-covered laptop knows that's  
 usually a Bad Thing.

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

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

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

Riiiiiight. Whatever you need to tell yourself.

**Static Variable**

What?

What do you mean *un-OO*?

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

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

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

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



## BE the Compiler

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



```
class StaticSuper {  
    static {  
        System.out.println("super static block");  
    }  
  
    StaticSuper () {  
        System.out.println("super constructor");  
    }  
}  
  
public class StaticTests extends StaticSuper {  
    static int rand;  
  
    static {  
        rand = (int) (Math.random() * 6);  
        System.out.println("static block " + rand);  
    }  
  
    StaticTests() {  
        System.out.println("constructor");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("in main");  
        StaticTests st = new StaticTests();  
    }  
}
```

Which of these is the output?

### Possible Output

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

### Possible Output

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

→ Answers on page 308.



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

## TRUE OR FALSE

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

—————> Answers on page 308.

## Exercise Solutions

### Sharpen your pencil (from page 287)

1, 4, 5, and 6 are legal.

2 doesn't compile because the static method references a non-static instance variable.

3 doesn't compile because the instance variable is final but hasn't been initialized.

### BE the Compiler (from page 306)

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

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

Note that this will be a randomly generated number from 0 to 5 inclusive.

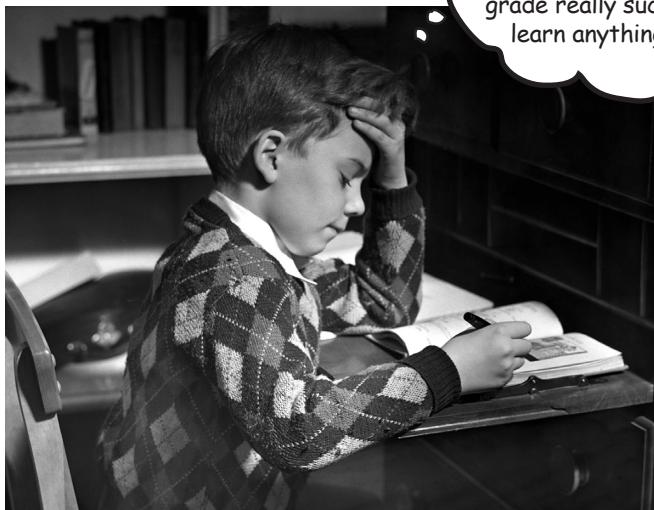
**Output**

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

### TRUE OR FALSE (from page 307)

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

# Data Structures



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

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

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

*We'll use this “mock” class to test our code.*

```

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 make this method static, because this class doesn't have any instance fields and doesn't need any.*

*This will be our list of six song titles to work with.*

Because `ArrayList` IS-A `List`, we can create an `ArrayList`, store it in a `List`, and return `List` from the method.

In the real world you'll often see Java code that returns the interface type (`List`) and hides the implementation type (`ArrayList`).

# 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;
    }
}
```

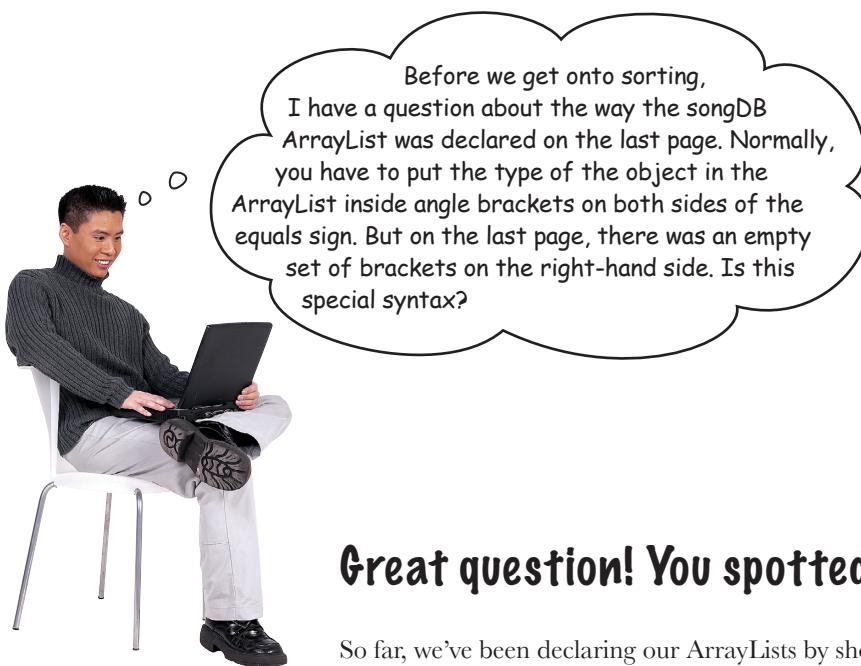
*We'll store the song titles in a List of Strings.*

*Then print the contents of the songList.*

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

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana,
Cassidy, 50 Ways]
```

*This is definitely NOT alphabetical!*



## 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.

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 specialized 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 a `CopyOnWriteArrayList` (see Chapter 18, *Dealing with Concurrency Issues*) 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 five-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 versus 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 uppercase letters, 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); ← Sort our song titles using
    System.out.println(songList);
}
}
```

*“natural ordering.”*

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana, Cassidy, 50 Ways]
[$10, 50 Ways, Cassidy, cassidy, havana, somersault]
```

Our songs unsorted,  
in the order they  
were added.

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



oO

Just FYI, we ducks are  
very particular about how we  
get 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.

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

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.

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?

```
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
```

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

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

Sorts the specified list into ascending order,  
according to the [natural ordering](#) of its elements.

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

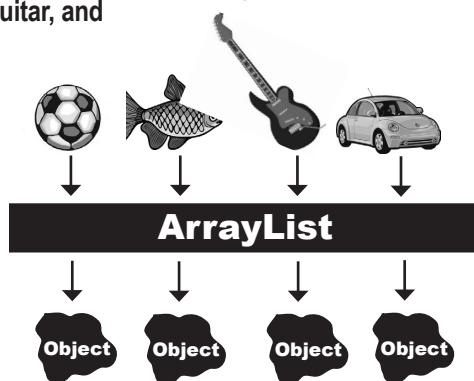
Although generics can be used in other ways, you'll often use generics to write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

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

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

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



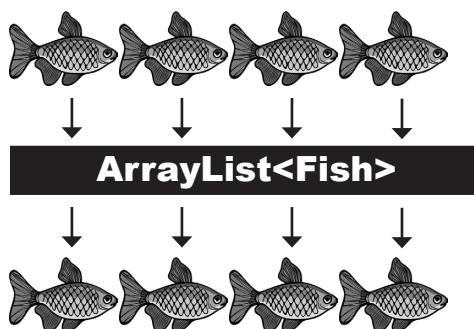
And come OUT as a reference of type Object

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.

### WITH generics

Objects go IN as a reference to only Fish objects



And come out as a reference of type Fish

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

# Learning generics

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

## ① Creating instances of 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.

## ② Declaring and assigning variables of generic types

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

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

If you have a method that 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.)

*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.)

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

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

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

# 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" is a placeholder for the  
REAL type you use when you  
declare and create an ArrayList.  
  
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {  
  
    public boolean add(E o)  
    {  
        Here's the important part! Whatever "E" is  
        determines what kind of things you're allowed  
        to add to the ArrayList.  
  
    } // more code
```

ArrayList is a subclass of AbstractList, so whatever type you specify for the type of the ArrayList, ArrayList is automatically used for the type of the AbstractList.

The type (the value of <E>) becomes the type of the List interface as well.

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

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

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:**

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

**Means ArrayList:**

**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 Using a type parameter defined in the class declaration

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

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

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

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

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

Here we can use <T> because we declared "T" at the start of the method declaration.



Wait...that can't be right. If you can take a list of Animal, why don't you just SAY that? What's wrong with just `takeThing(ArrayList<Animal> list)`?

## 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...



```
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...

```
static <T extends Comparable<? super T>> void sort(List<T> list)
    Sorts the specified list into ascending order,
    according to the natural ordering of its elements.
```

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

This says "Whatever 'T' is must be of type Comparable."

(Ignore this part for now. But if you can't, it just means that the type parameter for Comparable must be of type T or one of T's supertypes.)

You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that "extends Comparable."

Um...I just checked the docs for String, and String doesn't EXTEND Comparable—it IMPLEMENTS it. Comparable is an interface. So it's nonsense to say <T extends Comparable>.



```
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.

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

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

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

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**.

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

# Finally we know what's wrong...

## The Song class needs to implement Comparable

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

`java.lang.Comparable`

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

**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.**

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.

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

 **Sharpen your pencil**

Write in your idea and pseudocode (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 three lines of code!

→ **Yours to solve.**

## The new, improved, comparable Song class

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

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

```
class 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;
    }
}
```

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.

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 and then calls `sort()`, which puts the songs in alphabetical order by title.

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

## 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.**



### Excerpts from the API

#### `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 induced by the specified **Comparator**.

#### `java.util.List`

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

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

There's also a sort() on List, which takes a Comparator.

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

# 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);
}
```

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());
    }
} This is a String (the artist)
```

*Make an instance of the Comparator class.*

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

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

```
% 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)*

**exercise: 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.

### 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? \_\_\_\_\_

—————> Answers on page 364.

# 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);           ← This uses Comparable to sort
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);          ← This uses a custom
    System.out.println(songList);          Comparator to sort
}
```

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.

D  
That's an awful lot of code for sorting our songs in just two different orders. Isn't there a better way?



## 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.

the compiler already knows

## 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:

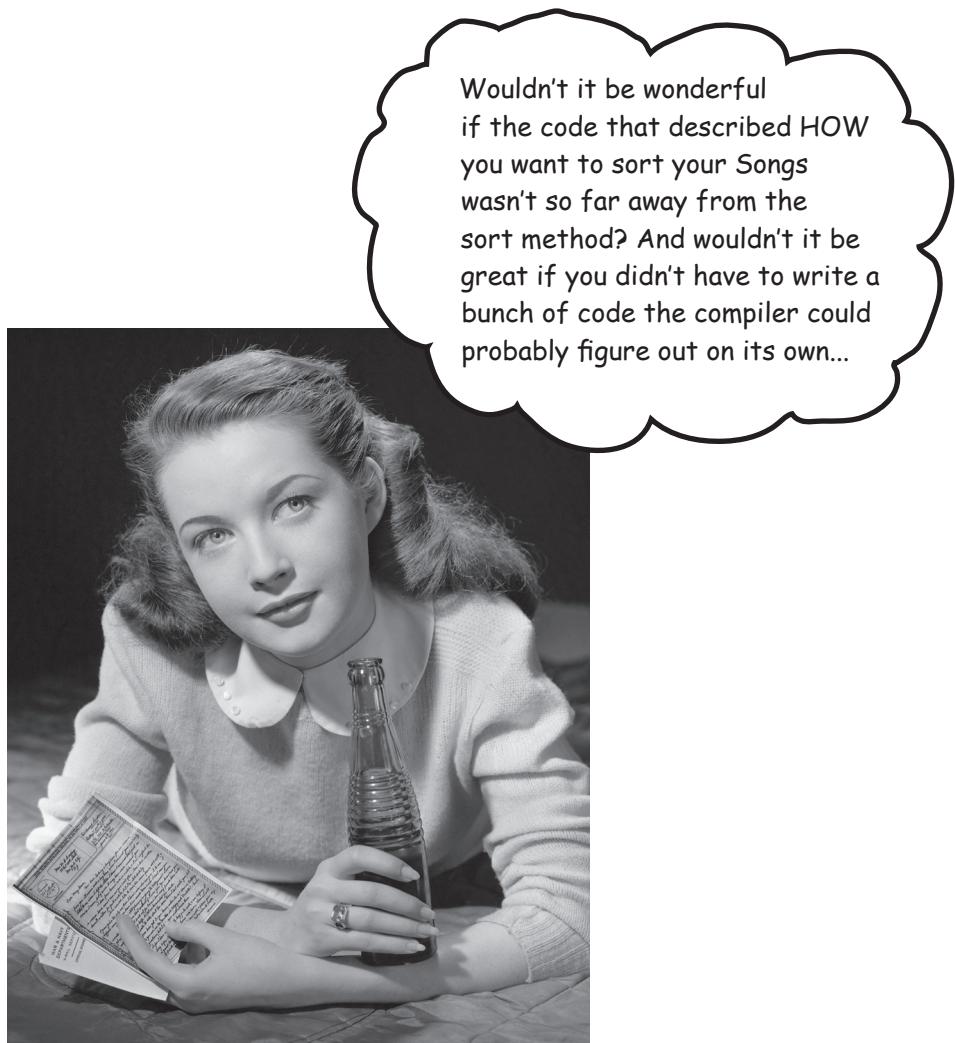
**“Call the sort method on the list of songs ① and pass it a reference to a Comparator object, which is designed specifically to sort Song objects ②.”**

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)...

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

The compiler cares  
not a whit what you  
call the class.

This is just a reference to an object that  
implements Comparator. Its name doesn't  
matter to the compiler.

```
class TitleCompare implements Comparator<Song> {
```

Yup, the compiler  
knows what this  
method should  
look like.

```
    public int compare(Song one, Song two) {
```

Doh! The compiler can infer this from  
the sort() docs.

The compiler can figure  
out that the two objects  
have to be Song objects  
since songList is a List of  
Song objects.

```
        return one.getTitle().compareTo(two.getTitle());
```

```
}
```

```
...
```

Or, we could use a lambda...

This is ALL THE COMPILER NEEDS.  
Just tell it HOW to do the sort!

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

These are Song one and  
Song two, the parameters  
to the compare method.



## 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

Modifier and Type	Method	Description
int	<code>compare(T o1, T o2)</code>	Compares its two arguments for order.
boolean	<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this comparator.

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.

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



## 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.



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);
    }
}

```

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.

```

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

```

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]

```



## 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

→ Answers on  
page 366.



## 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.

```

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]
  
```

→ Answers on page 365.

## 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 keep 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]
```

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

This is what the actual song data file looks like. →

**SongListMore.txt**

```
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

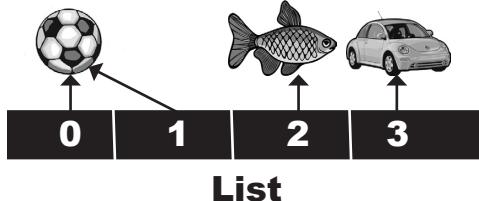
From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**.  
ArrayList is a **List**, but it looks like **Set** is exactly what we need.

## ► 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.

Duplicates OK.

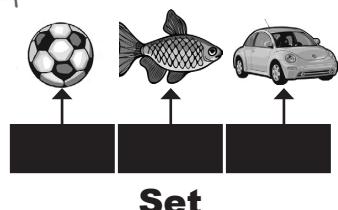


## ► SET - when uniqueness matters

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).

NO duplicates.

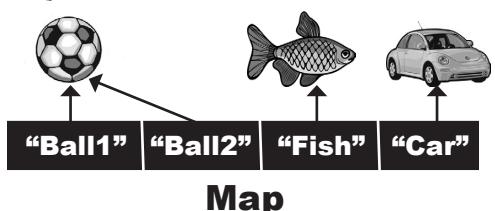


## ► MAP - when *finding something by key* matters

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.

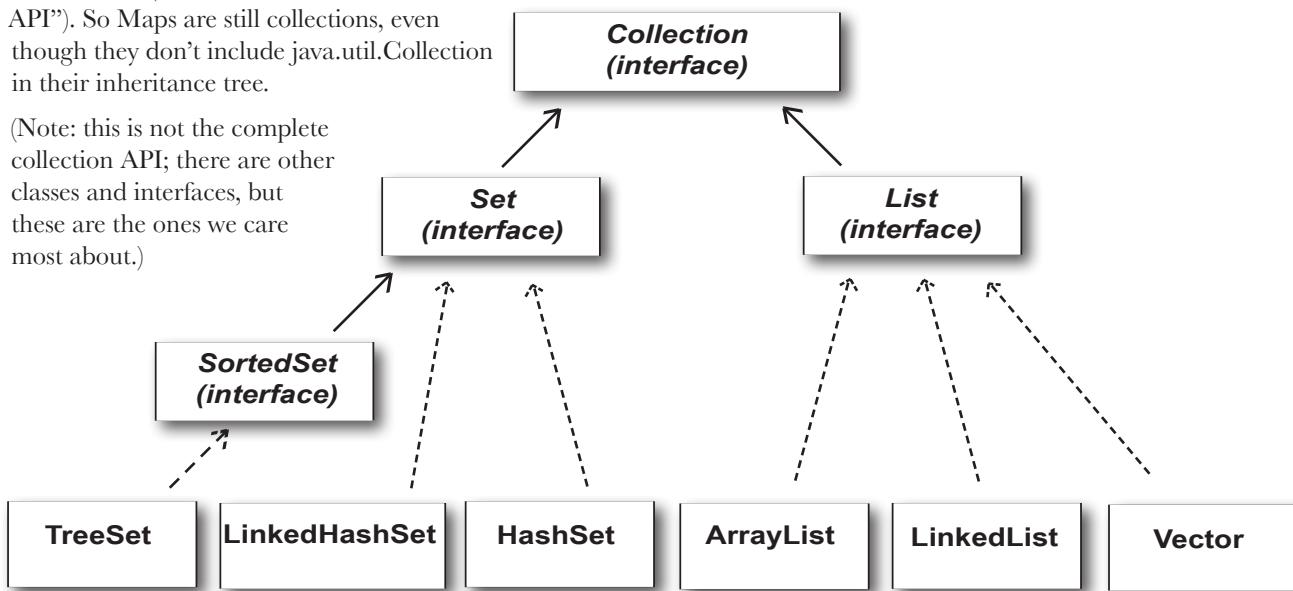
Duplicate values OK, but NO duplicate keys.



# 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.)

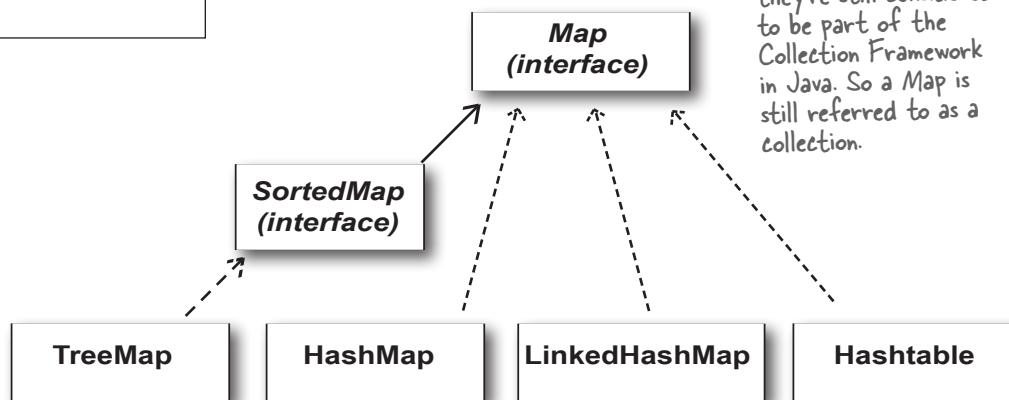


## KEY

↑  
extends

↑  
implements

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

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.

```
File Edit Window Help GetBetterMusic
%java Jukebox8
[somersault, cassidy, $10, havana, $10, cassidy, 50 ways]
[$10, $10, 50 ways, cassidy, cassidy, havana, somersault]
[$10, 50 ways, havana, cassidy, $10, cassidy, somersault]
```

The Set didn't help!!

We still have all the duplicates!

(And it lost its sort order when we put the list into a HashSet, but we'll worry about that one later...)

After putting it into a HashSet and printing the HashSet (we didn't call sort() again).

# 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.

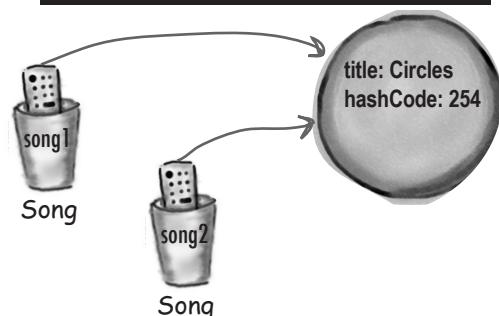
## ► Reference equality

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 `hashCode()`. For a Set to treat two objects as duplicates, you must override the `hashCode()` and `equals()` methods inherited from class `Object` so that you can make two different objects be viewed as equal.



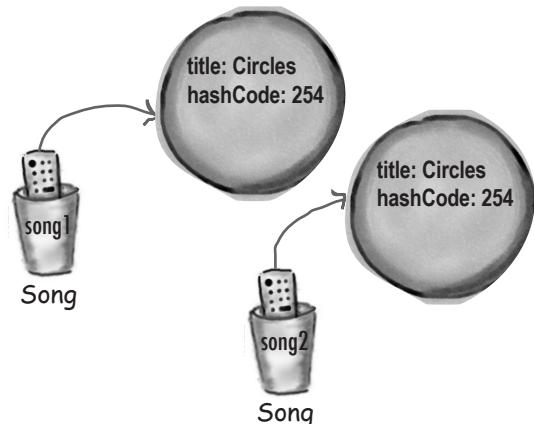
```
if (song1 == song2) {
    // both references are referring
    // to the same object on the heap
}
```

## ► 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!

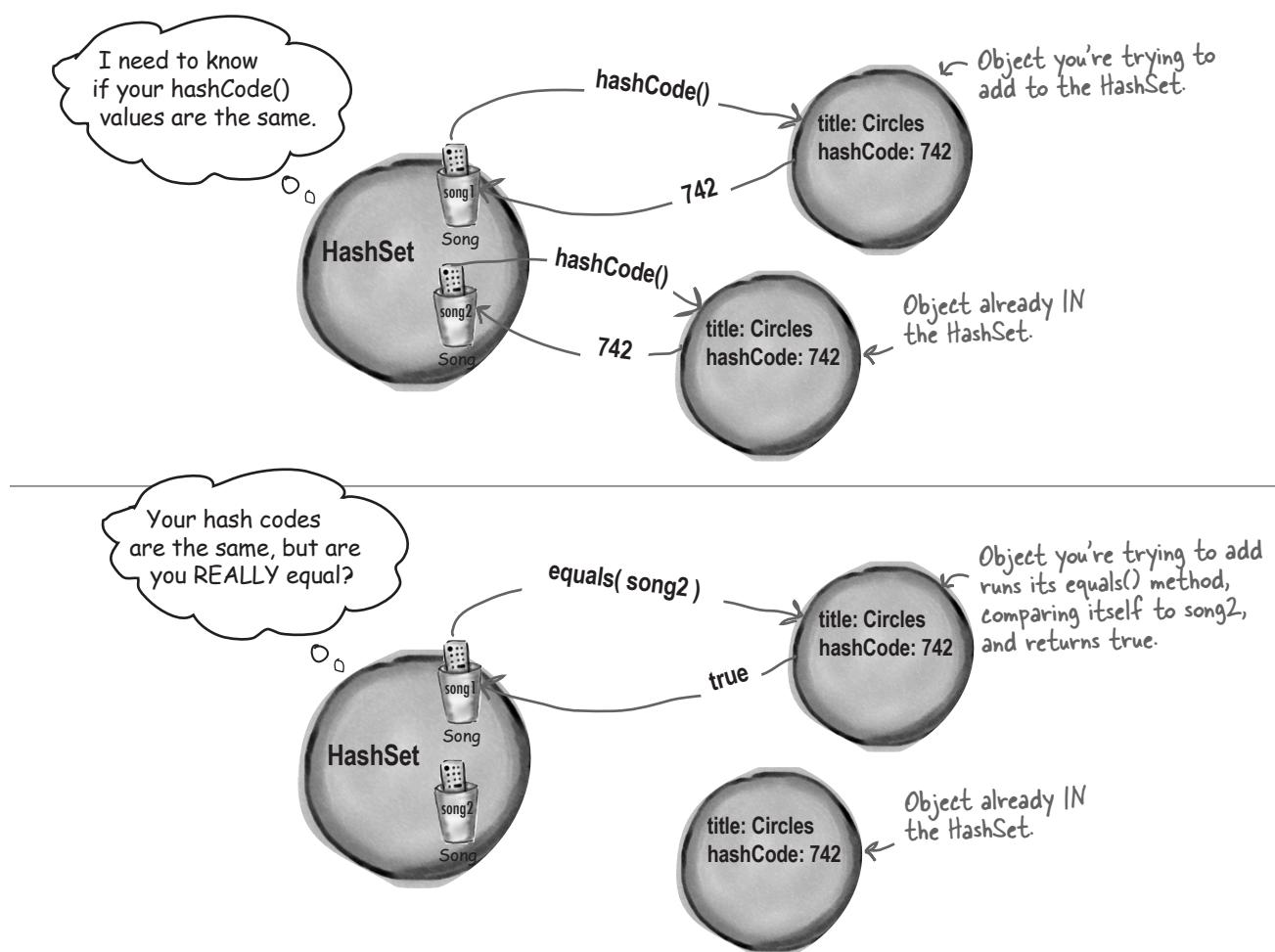
So you must override hashCode() to make sure the objects have the same value.

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.



# 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 hash code 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 hash code 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 hash code #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.

## TreeSets and sorting

# 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);  
    }  
}
```

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);
```

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.*

→ Answers on page 366.



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);  
    }  
}
```

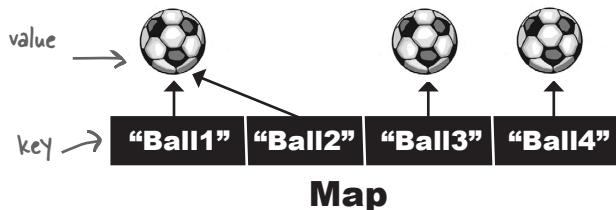
```
class BookCompare implements Comparator<Book> {  
    public int compare(Book one, Book two) {  
        return one.title.compareTo(two.title);  
    }  
}  
public class TestTreeComparator {  
    public void go() {  
        Book b1 = new Book("How Cats Work");  
        Book b2 = new Book("Remix 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).

```

File Edit Window Help WhereAmI
% java TestMap
{Skyler=420, Bert=343, Kathy=42}
343
  
```

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

I keep seeing the same code popping up again and again for creating collections. There must be something we can do to make this easier for people.

Good point! Let me just whip up some Factory methods.

## 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:

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

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.

Return an “unmodifiable” version of the list we just created so we know no one else can change it.

We'll see in Chapters 12 and 18 why we might want to create data structures that can't be changed.



# Convenience Factory Methods for Collections

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

- 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 **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 the cases where you want to create a collection prefilled 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...counterintuitive 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); // Pass a List<Animal> into our testAnimals method
    }

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

Using the List.of factory method we just looked at

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

**Compiles and runs just fine**

```
File Edit Window Help CatFoodIsBetter
% java TestGenerics1

animal eating
animal eating
animal eating
```

# But will it work with List<Dog>?

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); ← We know this line worked fine.

    List<Dog> dogs = List.of(new Dog(), new Dog());
    takeAnimals(dogs); ← Will this work now that we changed
}                                     from an array to a List?

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

Make a Dog List and  
put a couple dogs in.

## When we compile it:

```
File Edit Window Help CatsAreSmarter
% javac TestGenerics2.java

TestGenerics2.java:20: error: incompatible types:
List<Dog> cannot be converted to List<Animal>
    takeAnimals(dogs);
               ^
1 error
```

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



And I'm supposed to be OK with this? That totally screws my animal simulation where the veterinary program takes a list of any type of animal so that a dog kennel can send a list of dogs, and a cat kennel can send a list of cats...now you're saying I can't do that?

## 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.

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

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

## We can do this with wildcards

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

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

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

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

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

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

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

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



## there are no Dumb Questions

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

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

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

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

## Using the method's generic type parameter

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

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

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

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

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

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

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

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

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

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

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



## BE the Compiler, advanced

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

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

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

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

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

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

### Compiles?

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

**Exercise Solution**

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

**Possible Answers:**

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

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

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

**Comparable**

**compareTo( )**

**yes**

Given the following compilable statement:

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

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

**yes**

**yes**

**no**

**no**

**Comparator**

**compare( )**

**Solution****“Reverse Engineer” lambdas exercise**

(from page 343)

```

import java.util.*;

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

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

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

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

class Mountain {
    String name;
    int height;

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

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

```

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

**Output:**

```

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

```

## Solution



### Sorting with lambdas (from page 342)

Sort by BPM ascending

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

Sort by title descending

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

#### Output:

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

## Solution



### TreeSet exercise

(from page 353)

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

**It compiles correctly**

---

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

**It throws an exception:**

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

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

**Make Book implement Comparable, or pass the TreeSet a Comparator**

---

See page 574



# BE the Compiler solution

(from page 363)

## Compiles?

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

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

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

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



# Lambdas and Streams: What, Not How



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

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

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

# Tell the computer **WHAT** you want

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

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

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

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

...then print out each color.

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

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

For each item in the list...

Create a temporary variable named color

Print out the color

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

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

## Fireside Chats



### for loop

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

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

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

Well I'm faster. Everyone knows that.

I said you would disappear soon.

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

### forEach()

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

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

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

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

## When for loops go wrong

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



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

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

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

### Candidates:

```
for (int i = 1; i < nums.size(); i++)  
    output += nums.get(i) + " ";  
  
for (Integer num : nums)  
    output += num + " ";  
  
for (int i = 0; i <= nums.length; i++)  
    output += nums.get(i) + " ";  
  
for (int i = 0; i <= nums.size(); i++)  
    output += nums.get(i) + " ";
```

### Possible output:

1 2 3 4 5

Compiler error

2 3 4 5

Exception thrown

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

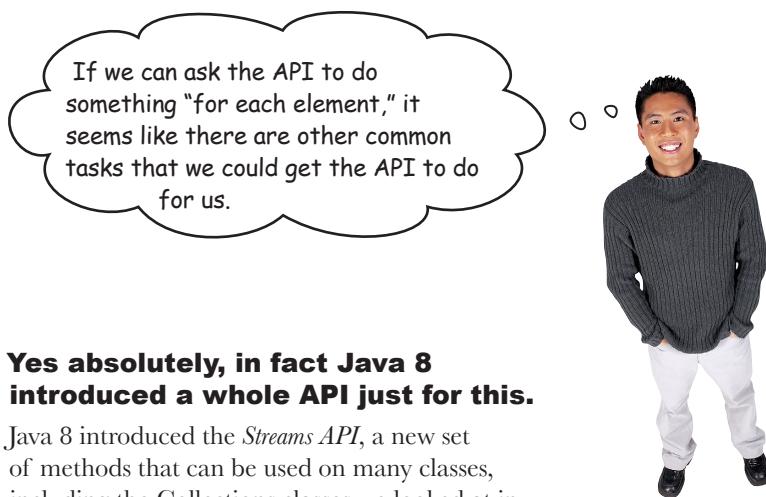
Match each candidate with one of the possible outputs

→ Answers on page 417.

## Small errors in common code can be hard to spot

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

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



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

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

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

**Brain Barbell**

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

## Building blocks of common operations

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

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

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



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

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

—————> Answers on page 417.

# Introducing the Streams API

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

**java.util.stream.Stream**

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

**// more**

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

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

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



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

In case you are interested:

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

# Getting started with Streams

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

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

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

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

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

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

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

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

*Sets the maximum number of results to return to 4*

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

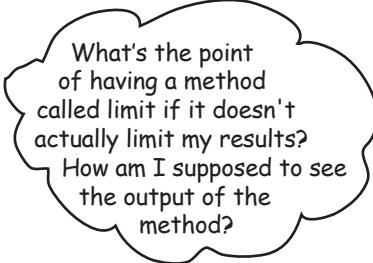
```
System.out.println("limit = " + limit);
```

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

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

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

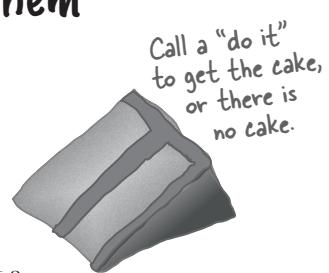
**Stream methods that return another Stream are called Intermediate Operations. These are instructions of things to do, but they don't actually perform the operation on their own.**



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

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

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



(These are some terminal operations on Stream.)

### java.util.stream.Stream

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

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

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

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

// more

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

# Getting a result from a Stream

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

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

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

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

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

```
File Edit Window Help WellDuh
%java LimitWithStream

result = 4
```

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

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

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

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

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

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

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

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

```
File Edit Window Help FinallyAResult
%java LimitWithStream

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



**Relax**

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

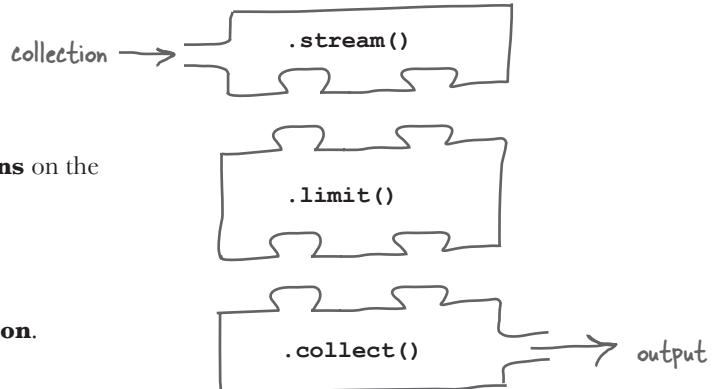
For now, `collect(Collectors.toList())` is a magic incantation to get the output of the stream pipeline in a List.

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

# Stream operations are building blocks

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

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



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

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

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

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

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

```

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

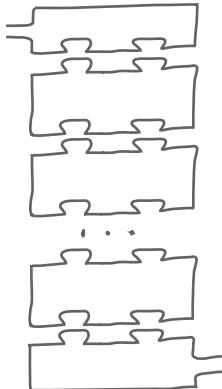
Formatted to align each operation directly underneath the one above, to clearly show each stage.

Get the stream for the collection  
Set a limit to return a maximum of 4 results from the stream  
Returns the results of the operation as a List

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

## Building blocks can be stacked and combined

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



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

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

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

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

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

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

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

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

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

## Customizing the building blocks

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

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

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

File Edit Window Help IgnoreCaps

```
%java ChainedStream
```

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

## Create complex pipelines block by block

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

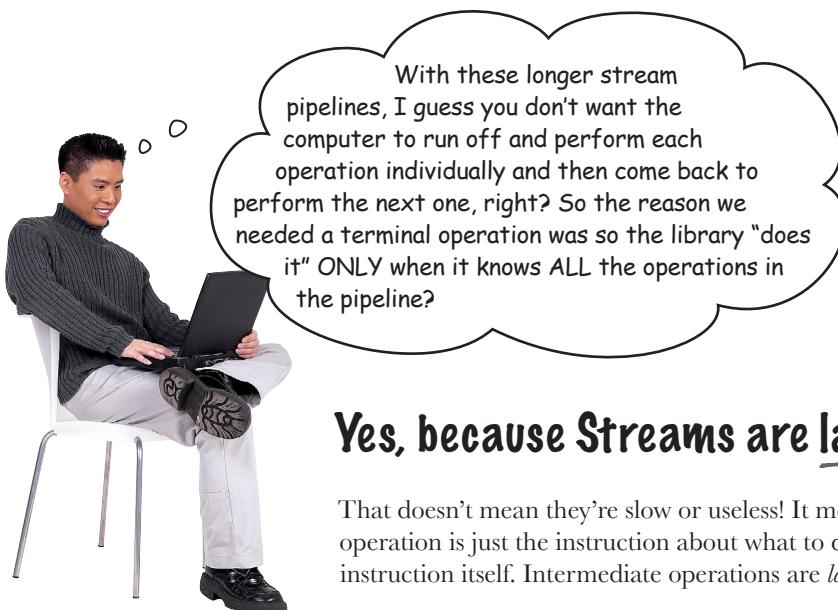
```
List<String> result = strings.stream()
    .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
    .skip(2)
    .limit(4)
    .collect(Collectors.toList());
```

```
File Edit Window Help BoxersDolt
```

```
%java ChainedStream
```

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

The stream skipped over the first two elements.



## Yes, because Streams are lazy

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

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

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



# Terminal operations do all the work

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

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

## Collecting to a List

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

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

Terminal operation:

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

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

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

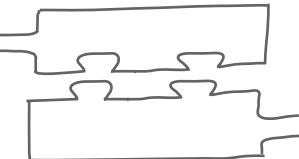
We will look at more Collectors, and other terminal operations, later in the chapter. For now, you know enough to get going with Streams.

## Guidelines for working with streams

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

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

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



### ② You can't reuse Streams.

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

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

```
File Edit Window Help ClosingTime
%java LimitWithStream

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

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

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





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

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

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

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

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

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

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

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

```
File Edit Window Help Untouchable
%java LimitWithStream

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

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

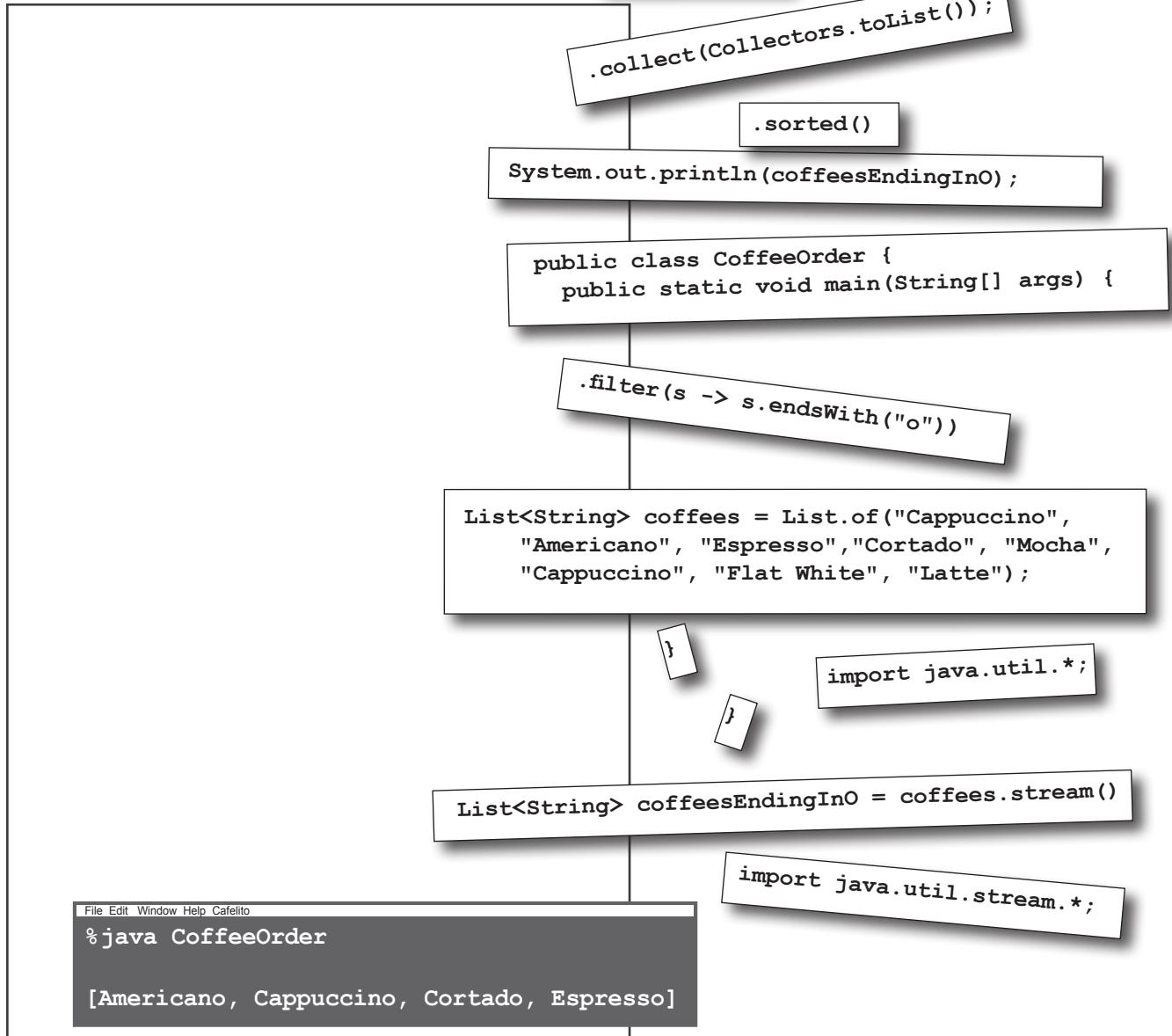
Only the output object has the results of the query. This is a brand new List.



# Code Magnets



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



→ Answers on page 418.

there are no  
Dumb Questions

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

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

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

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

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

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

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

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

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

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

### BULLET POINTS

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

## Hello Lambda, my (not so) old friend

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

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

### Passing behavior around

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

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

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

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

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

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

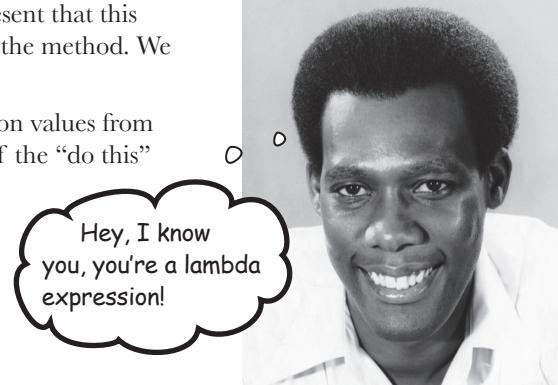
You can't just write this code here, because it will be run straightaway. Instead, we need a way to hand this block of code over to the `forEach` method so that method can call it when it's ready.

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

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

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

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





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

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

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

## A lambda expression implements a Functional Interface.

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

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

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

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

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

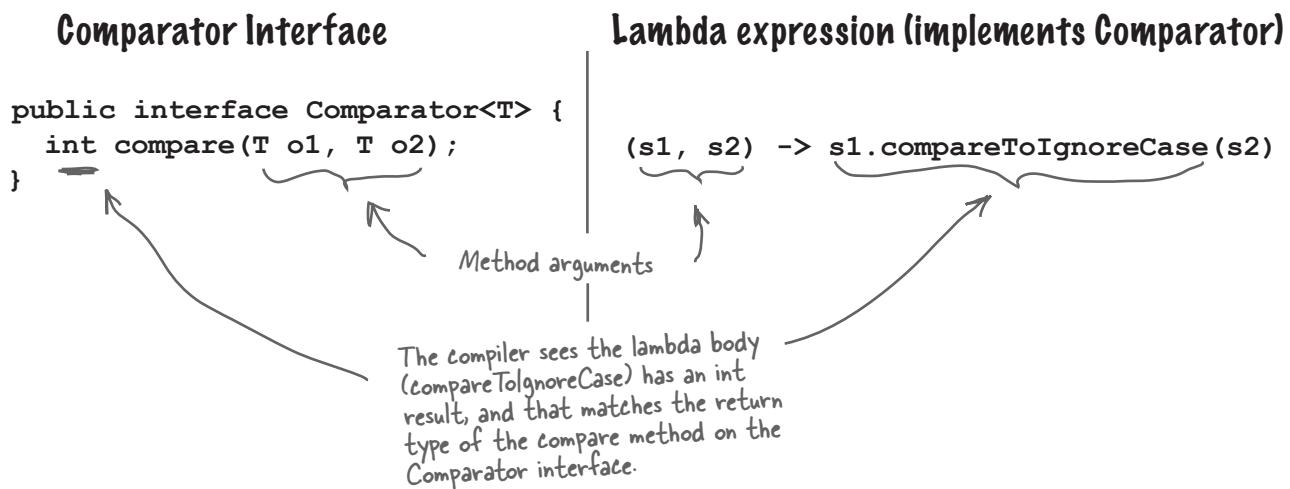
*This is a placeholder type to give you an idea of what the method would look like. We'll look at specific Functional Interfaces throughout this chapter.*

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

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

# The shape of lambda expressions

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



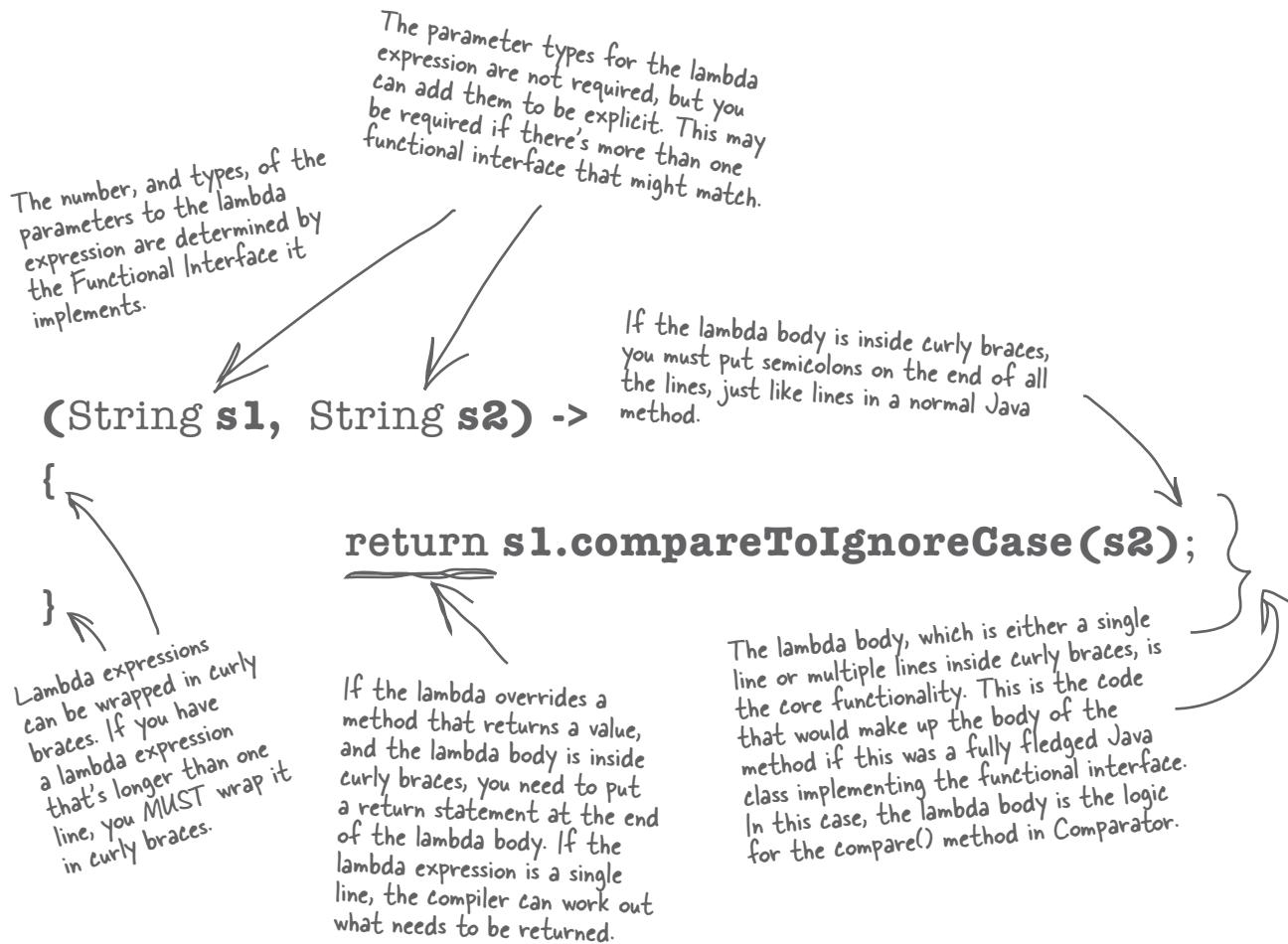
You might be wondering where the **return** keyword is in the lambda expression. The short version is: you don't need it. The longer version is, if the lambda expression is a single line, and if the functional interface's method signature requires a returned value, the compiler just assumes that your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

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

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

# Anatomy of a lambda expression

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



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

# Variety is the spice of life

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

## A lambda might have more than one line

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

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

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

*Semicolons required.*

*Curly braces for multiline lambda expressions.*

*Return keyword required.*



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:

No need for curly braces →  
 $(\text{str1}, \text{str2}) -> \text{str2.length()} - \text{str1.length()}$  ← No semicolons  
 ↕ No need for "return"

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

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

## A lambda might not return anything

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

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

Multiline lambda

```

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

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

Method is void on the Functional Interface

## A lambda might have zero, one, or many parameters

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

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

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

No method parameters

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

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

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

One method parameter

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

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

Two method parameters

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

## How can I tell if a method takes a lambda?

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

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

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

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

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

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

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

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

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

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

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



Exercise

## BE the Compiler, advanced

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

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

Check the box if the statement would compile.

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

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

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

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

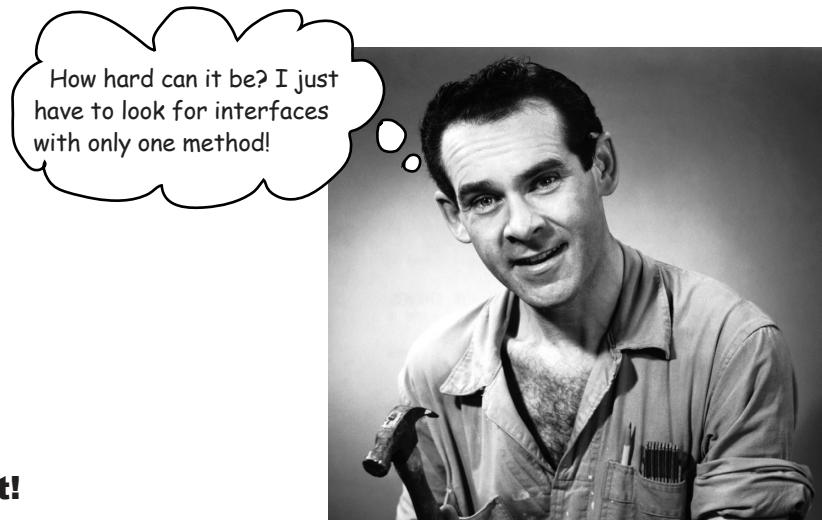
```
public interface Function<T, R> {
    R apply(T t);
}
```

—————> Answers on page 418.

## Spotting Functional Interfaces

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

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



### Not so fast!

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

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

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

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

# Functional interfaces in the wild

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

Here it is! This is our Single Abstract Method.

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

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



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

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

ActionListener	
Modifier and Type	Method
void	actionPerformed(ActionEvent e)

## Iterator

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

## Function

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

## SocketOption

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

→ Answers on page 419.

## Lou's back!

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

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

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

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

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

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



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



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

What do you expect the output to look like?



## Ready-Bake Code

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

```

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

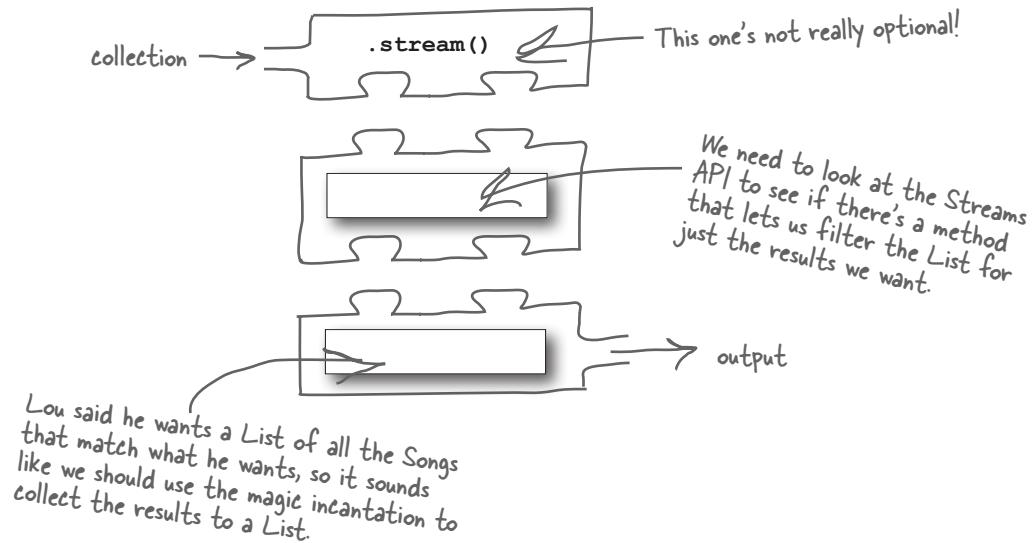
public class Song {
    private final String title;
    private final String artist;
    private final String genre;
    private final int year;
    private final int timesPlayed;
    // Practice for you! Create a constructor, all the getters and a toString()
}

```

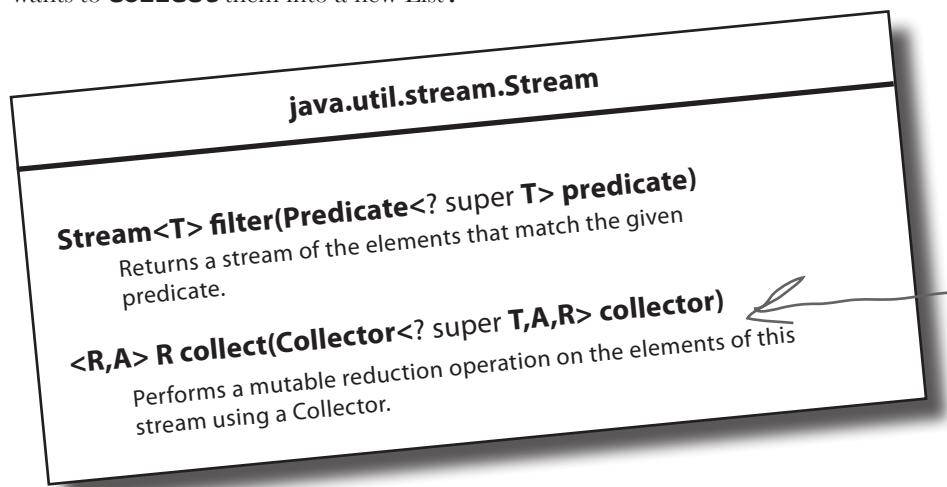
## Lou's Challenge #1: Find all the "rock" songs

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

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



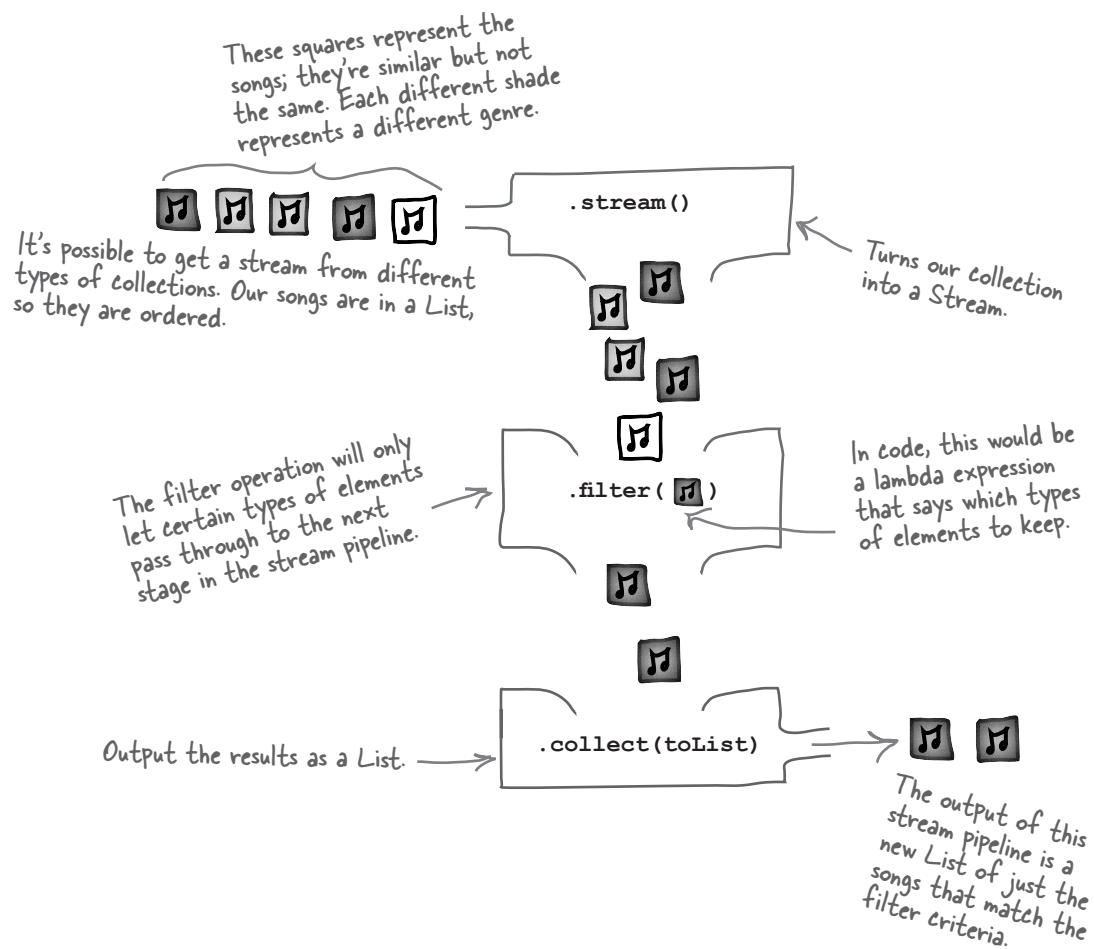
Fortunately, there are hints about how to create a Streams API call based on the requirements Lou gave us: he wants to **filter** for just the Songs with a particular genre, and he wants to **collect** them into a new List.



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

# Filter a stream to keep certain elements

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

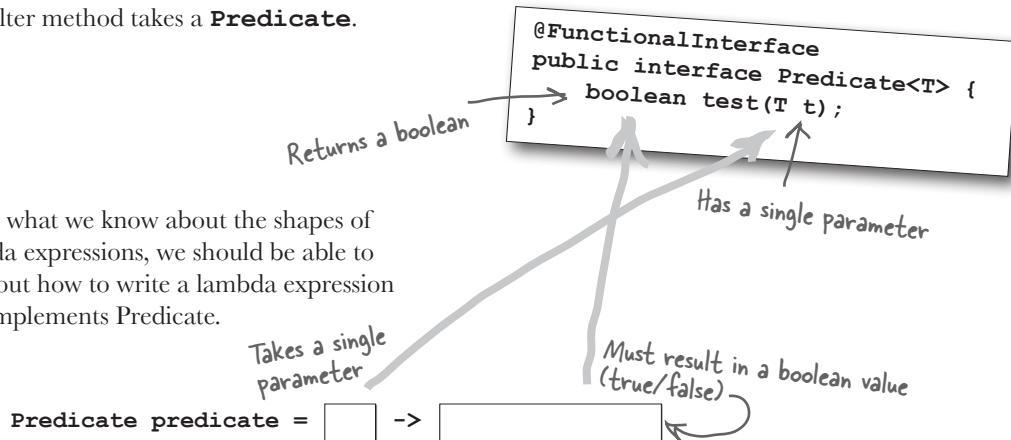


# Let's Rock!

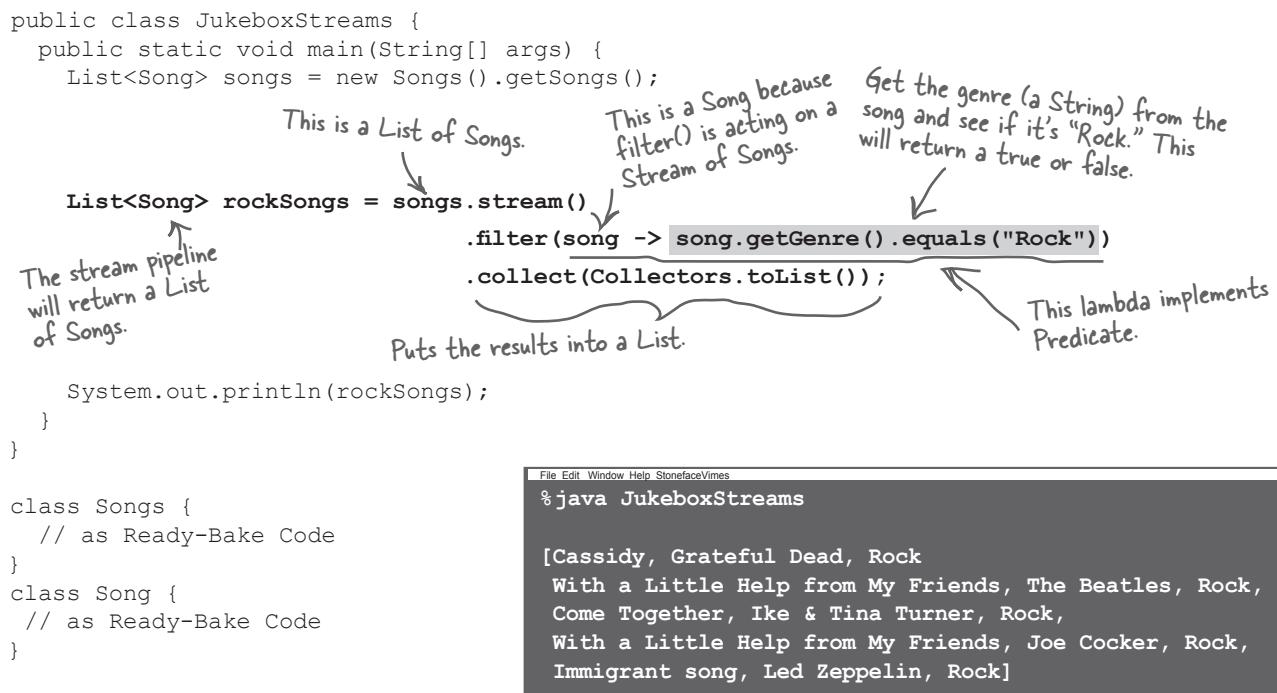
So adding a **filter** operation filters out elements that we don't want, and the stream continues with just the elements that meet our criteria. It should come as no surprise to find that you can use a lambda expression to state which elements we want to keep in the stream.

The filter method takes a **Predicate**.

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



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



# Getting clever with filters

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

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

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

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

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

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

```
File Edit Window Help YouRock
%java JukeboxStreams

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

```

Now the stream returns different  
types of rock songs.

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



**Brain Barbell**

Can you write a filter operation that can select songs:

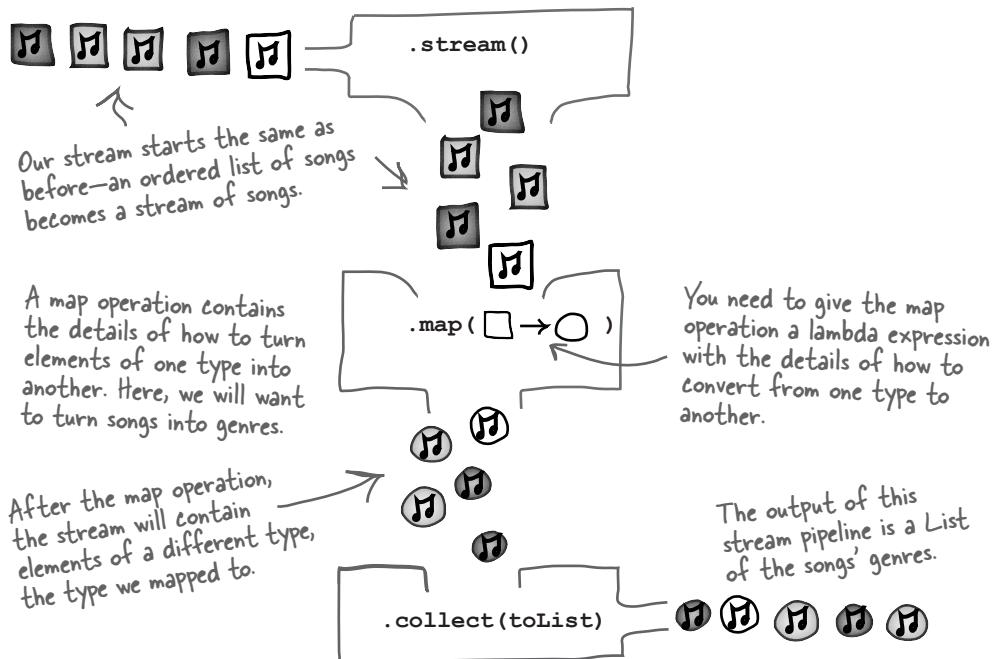
- By The Beatles
- That start with “H”
- More recent than 1995

## Lou's Challenge #2: List all the genres

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

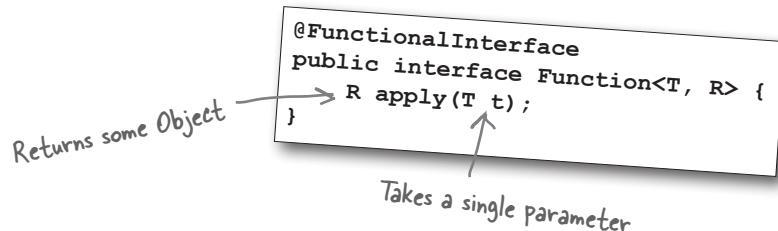
So far, all of our streams have returned the same types that they started with. The earlier examples were Streams of Strings, and returned Lists of Strings. Lou's previous challenge started with a List of Songs and ended up with a (smaller) List of Songs.

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



# Mapping from one type to another

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



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

```

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

Annotations on the code:

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

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

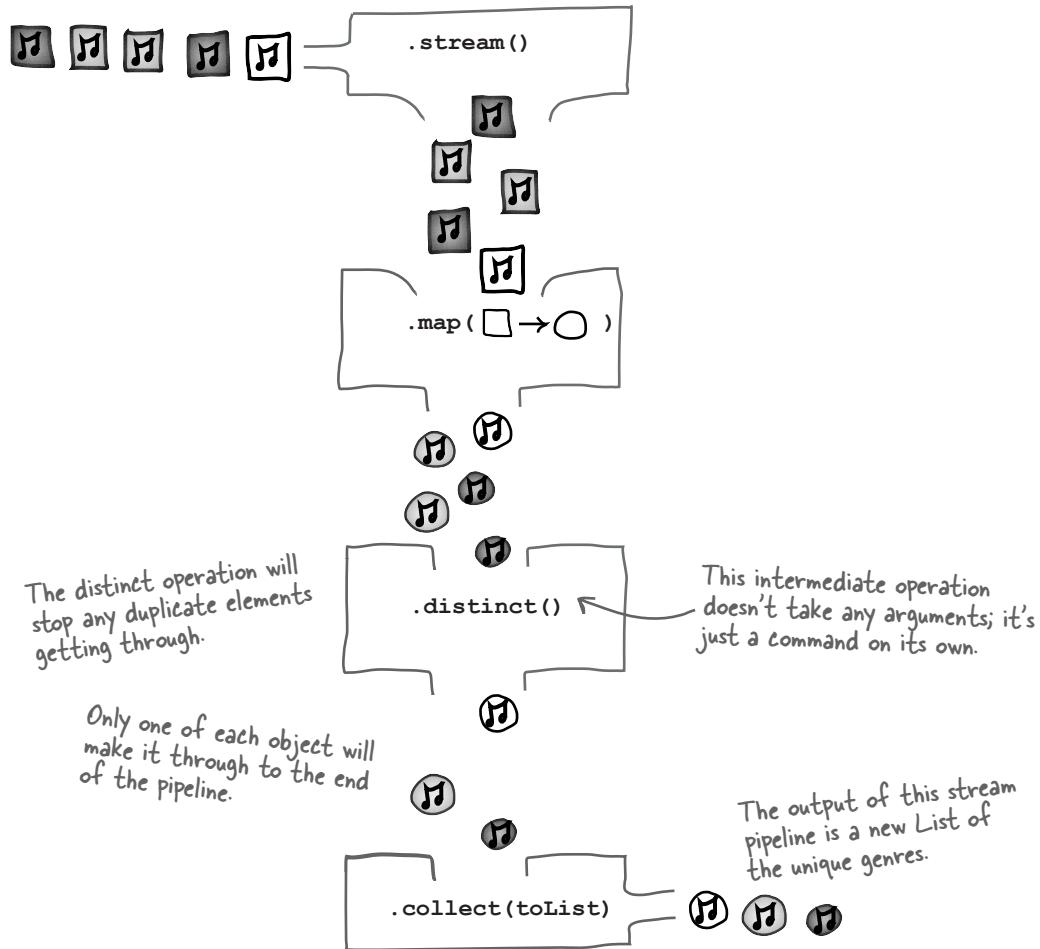
```

File Edit Window Help RoadToNowhere
%java JukeboxStreams

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

## Removing duplicates

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



# Only one of every genre

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

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

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

Outputs a much more readable list of all the genres

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

# Just keep building!

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

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



## Sharpen your pencil

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

→ Yours to solve.

# Sometimes you don't even need a lambda expression

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

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

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

The output of this Function needs to be a String.

The input parameter to this Function is a Song.

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

This is the method call in the lambda body.

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

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

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

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

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

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

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



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

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

Sometimes method references make the code easier to understand.

# Collecting results in different ways

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

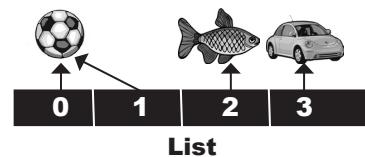
```
Set<String> genres = songs.stream()
    .map(song -> song.getGenre())
    .collect(Collectors.toSet());
```

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

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

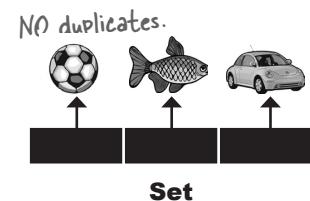
## Collectors.toList and Collectors.toUnmodifiableList

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



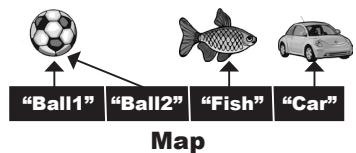
## Collectors.toSet and Collectors.toUnmodifiableSet

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



## Collectors.toMap and Collectors.toUnmodifiableMap

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



## Collectors.joining

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

## But wait, there's more!

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

### Checking if something exists

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

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

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

### Find a specific thing

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

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

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

### Count the items

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

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

```
long count();
```

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

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



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

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

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

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

Strawberry ice cream, please!

Here you go!

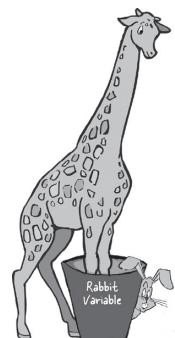
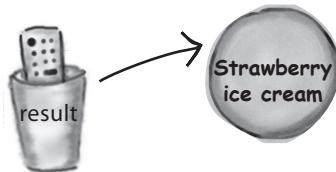
```
IceCream iceCream =  
    getIceCream("Strawberry");
```

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

We don't have any, sorry.

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

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



optional

## Optional is a wrapper

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

