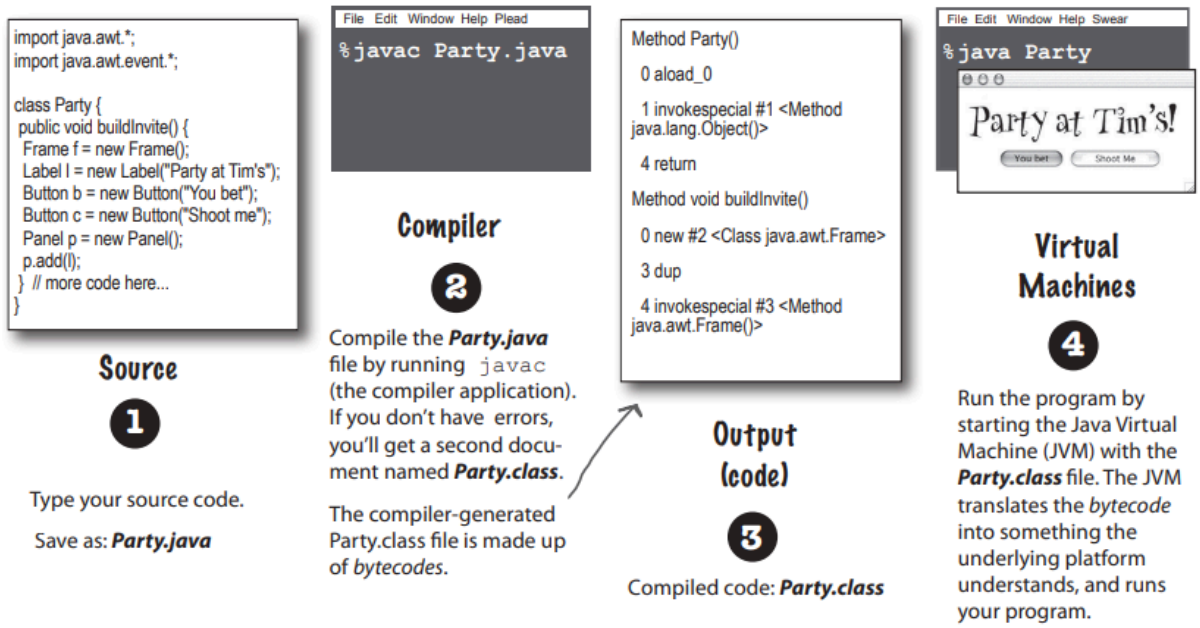
Head First Java Notes

Chapter 1 → Breaking the Surface

The way Java works

You'll have a source code file, compile it using the javac compiler (any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform independent), and then run the compiled bytecode on a Java virtual machine.



History, Speed and Memory usage

Java was initially released, on January 23, 1996 by James Gosling. Java is famous for its backward compatibility, so old code can run quite happily on new JVMs.

When Java was first released, it was slow. But soon after, the HotSpot VM was created, as were other performance enhancers. While it's true that Java isn't the fastest language out there, it's considered to be a very fast language—almost as

fast as languages like C and Rust, and much faster than most other languages out there.
Java has a magic super-power—the JVM. The Java Virtual Machine can optimize your code while it's running, so it's possible to create very fast applications without having to write specialized high-performance code.
Compared to C and Rust,
Java uses a lot of memory.

The Compiler and JVM battle over the question, "Who's more important?"

JVM

What, are you kidding? HELLO. I am Java. I'm the one who actually makes a program run. The compiler just gives you a file, but the file doesn't do anything unless I'm there to run it.

A programmer could just write bytecode by
hand, and I'd take it. You might be out of a job
soon, buddy.

Compiler

Excuse me, but without me, what exactly would you run? There's a reason Java was designed to use a bytecode compiler. If Java
were a purely interpreted language, where—at
runtime—the virtual machine had to translate
straight-from-a-text-editor source code, a Java program would run at a ludicrously glacial pace.

While it is true but a programmer writing bytecode by hand is like painting pictures of your vacation instead of taking photos—sure, it's an art, but most people prefer to use their time differently.

So, what do you actually do?

Remember that Java is a strongly typed language, and that means I can't allow variables to hold data of the wrong type. This is a crucial safety feature, and I'm able to stop the vast majority of violations before they ever get to you.

But some still get through! I can throw ClassCastExceptions and sometimes I get people trying to put the wrong type of thing in an array that was declared to hold something else.

Yes, there are some datatype exceptions that can emerge at runtime, but some of those have to be allowed to support one of Java's other important features—dynamic binding. At runtime, a Java program can include new objects that weren't even known to the original programmer, so I have to allow a certain amount of flexibility. But my job is to top anything that would never—could never—succeed at runtime.

OK. Sure. But what about security?

Excuse me, but I am the first line of defense, as they say. The datatype violations I previously described could wreak havoc in a program if they were allowed to manifest. I am also the one who prevents access violations, such as code trying to invoke a private method, or change a method that—for

Whatever. I have to do that same stuff too, though, just to make sure nobody snuck in after you and changed the bytecode before running it.

security reasons—must never be changed. I stop people from touching code they're not meant to see, including code trying to access another class critical data.

Of course, but as I indicated previously, if I didn't prevent what amounts to perhaps 99% of the potential problems, you would grind to a halt.

Chapter 2 → A Trip to Objectville

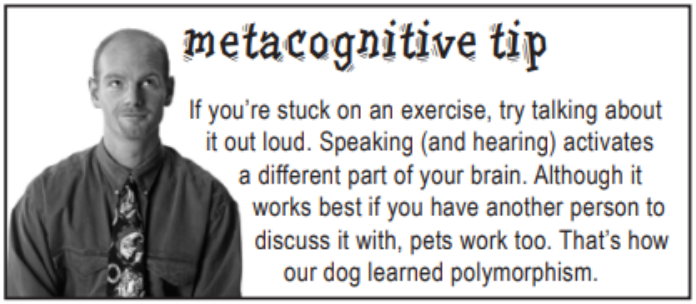
What do you like about OO?

"Object-oriented programming lets you extend a program without having to touch previously tested, working code. (not messing around with code I've already tested, just to add a new feature)"

"It helps me design in a more natural way. Things have a way of evolving."

"I like that the data and the methods that operate on that data are together in one class."

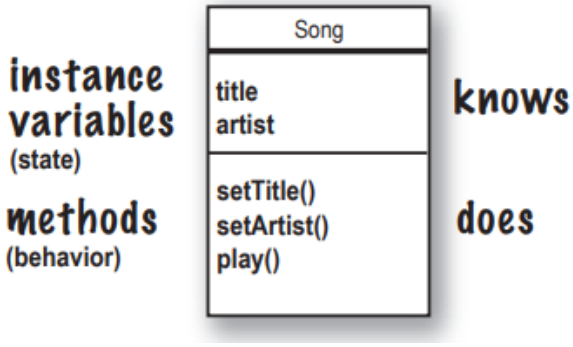
"Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later."



When you design a class, think about the objects that will be created from that class type. Think about:

- things the object knows
- things the object does

Things an object knows about itself are called instance variables. They represent an object's state (the data) and can have unique values for each object of that type.
Things an object can do are called methods. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data.
So objects have instance variables and methods, but those instance variables and methods are designed as part of the class.



What's the difference between a class and an object?

💡 **A class is not an object (but it's used to construct them)**

A class is a blueprint for an object. It tells the virtual machine how to make an object of that particular type. Each object made from that class can have its own values for the instance variables of that class.

Chapter 3 → Know Your Variables

Declaring a variable

Variables come in two flavors: primitive and object reference.
Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating-point numbers.

> Eg. byte x = 7;

Primitive Types

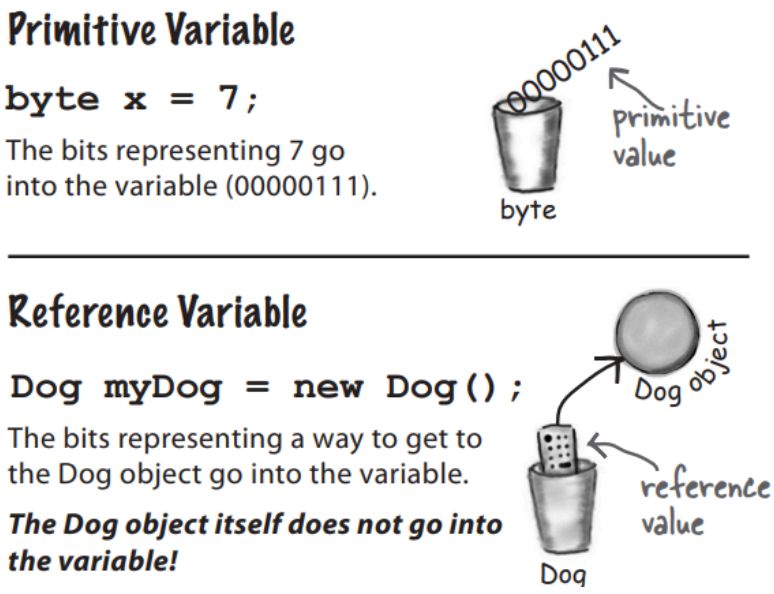
Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32 bits	varies
double	64 bits	varies

An object reference variable holds bits that represent a way to access an object. There is actually no such thing as an object variable. There's only an object reference variable.

Think of a Dog reference variable as a Dog remote control. You use it to get the object to do something (invoke methods).

```
Eg. Dog myDog = new Dog();
```

Although a primitive variable is full of bits representing the actual value of the variable, an object reference variable is full of bits representing a way to get to the object.
You use the dot operator (.) on a reference variable to say, “use the thing before the dot to get me the thing after the dot.” For example:
myDog.bark();
means, “use the object referenced by the variable myDog to invoke the bark() method.” When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.
An object reference is just another variable value. Only this time, the value is a remote control.



You really don’t want to spill that...

Be sure the value can fit into the variable.

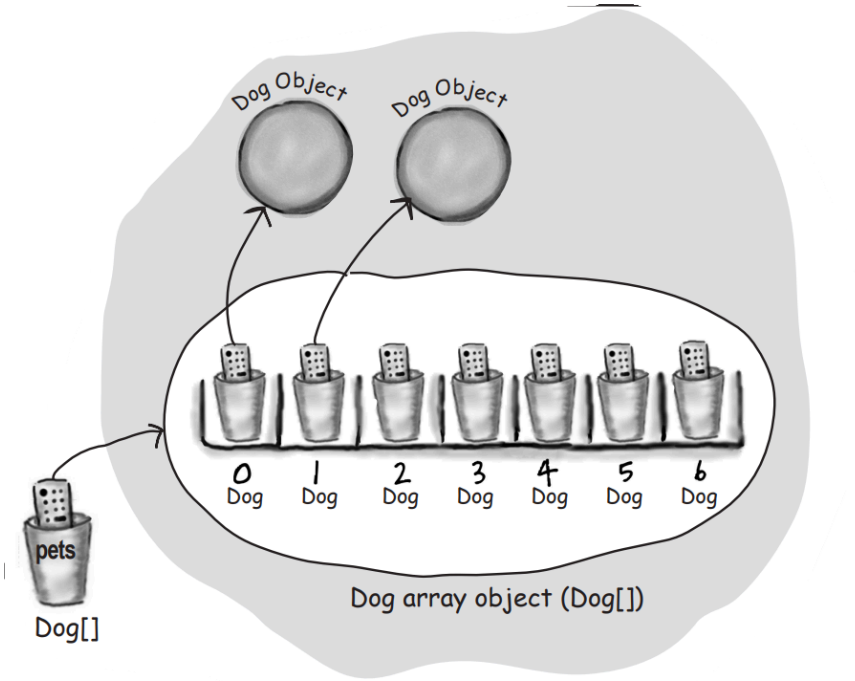
```
int x = 24;
```

```
byte b = x;  //won't work!!
```

Why doesn't this work? After all, the value of x is 24, and 24 is definitely small enough to fit into a byte. You know that, and we know that, but all the compiler cares about is that you're trying to put a big thing into a small thing, and there's the possibility of spilling. Don't expect the compiler to know what the value of x is, even if you happen to be able to see it literally in your code.

Arrays

Arrays are always objects, whether they're declared to hold primitives or object references.



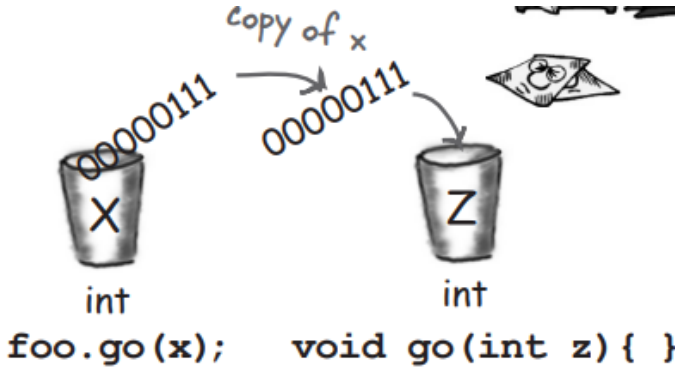
It also behaves same for the array that holds primitives

Strings are a special type of object. You can create and assign them as if they were primitives (even though they're references).

```
Eg. String name = "Avish";
```

Chapter 4 → How Objects Behave

Java is pass-by-value



Java is pass by value means pass by copy. The bits in x are copied, and the copy lands in z. Changing the value of z inside the method doesn't change the value of x!

Encapsulation

Make your instance variable private and provide public getters and setters for access control.

```
class GoodDog {
 private int size;

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

 void bark() {
    if (size > 60) {
       System.out.println("Wooof! Wooof!");
    } else {
       System.out.println("Ruff! Ruff!");
```

```
    }
   }
}
```

Declaring and initializing instance variables & difference b/w instance and local variables

Instance variables are declared inside a class and local variables are declared within a method.

You don’t have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variable get null.

Local variables do not get a default value! The compiler complains if you try to use a local variable before the variable is initialized.

Comparing Variables

Use == to compare two primitives or to see if two references refer to the same object.

```
//primitives
int a = 3;
byte b = 3;
System.out.println(a == b);  //true

// references
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
System.out.println(a == b);  //false
System.out.println(a == c);  //true
```

Use the equals() method to see if two different objects are equal.

```
String s1 = new String("Avish");
String s2 = new String("Avish");
String s3 = "Avish";
String s4 = "Avish";

// Compare using ==
System.out.println(s1 == s2);      // ❌ false (different objects)

System.out.println(s3 == s4);      // ✅ true (same reference from String pool)

// Compare using equals()
System.out.println(s1.equals(s2));  // ✅ true (same content)

System.out.println(s3.equals(s4)); // ✅ true
```

Chapter 5 → Extra-Strength Methods

Test code for SimpleStartup class

```
int [] locationCells
int numOfHits
String checkYourself(int guess)
void setLocationCells(int[] loc)
```

SimpleStartup have the above instance variables and methods. Then ask yourself, “If the checkYourself() method were implemented, what test code could I write that would prove to me the method is working correctly? The test code is given below -

```
public class SimpleStartupTestDrive {
 public static void main(String[] args) {
    SimpleStartup dot = new SimpleStartup();

    int[] locations = {2, 3, 4};
```

```
    dot.setLocationCells(locations);

    int userGuess = 2;
    String result = dot.checkYourself(userGuess);

    String testResult = "failed";
    if (result.equals("hit")) {
       testResult = "passed";
    }
    System.out.println(testResult);
 }
}
```

The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do.
As soon as your implementation code is done, you already have test code just waiting to validate it. Ideally, write a little test code, then write only the implementation code you need in order to pass that test. Then write a little more test code and write only the new implementation code needed to pass that new test. At each test iteration, you run all the previously written tests to prove that your latest code additions don't break previously tested code.

Bullet Points:

▼ Your Java program should start with a high-level design.

▼ Typically you'll write three things when you create a new class:
- prep code
- test code
- real (Java) code

▼ Prep code should describe what to do, not how to do it. Implementation comes later.

▼ Use the prep code to help design the test code. Write test code before you implement the methods.

▼ The concept of writing the test code first is one of the practices of Test-Driven Development (TDD), and it can make it easier (and faster) for you to write your

code.

Chapter 6 → Using the Java Library

Bullet Points:

▼ An ArrayList resizes dynamically to whatever size is needed. It grows when objects are added, and it shrinks when objects are removed.

▼ To put something into an ArrayList, use add(). To remove something from an ArrayList use remove(). To find out where something is (and if it is) in an ArrayList, use indexOf(). To find out if an ArrayList is empty, use isEmpty(). To get the size (number of elements) in an ArrayList, use the size() method.

▼ You declare the type of the array using a type parameter, which is a type name in angle brackets.
Example: ArrayList<Button> means the ArrayList will be able to hold only objects of type Button (or subclasses of Button).

▼ Although an ArrayList holds objects and not primitives, the compiler will automatically “wrap” (and “unwrap” when you take it out) a primitive into an Object and place that object in the ArrayList instead of the primitive.

▼ Classes are grouped into packages.

▼ A class has a full name, which is a combination of the package name and the class name. Class ArrayList is really java.util.ArrayList.

▼ To use a class in a package other than java.lang, you must tell Java the full name of the class. You get the java.lang package sort of “pre-imported”.

▼ Java 9 (and later versions) introduced the Java Module System, which means that the JDK is now split into modules. These modules group together related packages. This can make it easier to find the classes that interest you, because they’re grouped by function.

▼ Short-Circuit Operators `&&` and `||` evaluate boolean expressions efficiently by checking only what's necessary: `&&` stops if the left side is false, and `||` stops if the left side is true. This helps avoid issues like `NullPointerException` when checking conditions on possibly null references—for example, `if (refVar != null && refVar.isValidType())`.

▼ Non-Short-Circuit Operators `&` and `|` always evaluate both sides, which is useful in bitwise operations but less efficient in boolean logic.

Chapter 7 → Better Living in Objectville

Designing inheritance steps:

1. Look for objects that have common attributes and behaviors.
2. Design a class that represents the common state and behavior.
3. Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.
4. Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.
5. Finish the class hierarchy.

Using IS-A and HAS-A

When you want to know if one thing should extend another, apply the IS-A test.
Triangle IS-A Shape, yeah, that works.
Tub extends Bathroom, sounds reasonable. Until you apply the IS-A test.
To know if you've designed your types correctly, ask, "Does it make sense to say type X IS-A type Y?" If it doesn't, you know there's something wrong with the design, so if we apply the IS-A test, Tub IS-A Bathroom is definitely false.
What if we reverse it to Bathroom extends Tub?
Bathroom IS-A Tub doesn't work. Tub and Bathroom are related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship.
Does it make sense to say "Bathroom HAS-A Tub"? If yes, then it means that Bathroom has a Tub instance variable. In other words, Bathroom has a reference to a Tub, but Bathroom does not extend Tub and vice versa.

```
// All these are seperate class files.

// Bathroom class
Tub bathtub;
Sink theSink;
```

```
// Tub class
int size;
Bubbles b;

// Bubbles class
int radius;
int colorAmt;

// Bathroom HAS-A Tub and Tub HAS-A Bubbles. But nobody inherits from (exter
```

The inheritance IS-A relationship works in only one direction!
Triangle IS-A Shape makes sense, so you can have Triangle extend Shape. But the reverse—Shape IS-A Triangle—does not make sense, so Shape should not extend Triangle.
Remember that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).

If class B extends class A, class B IS-A class A.
If class C extends class B, class C passes the IS-A test for both B and A.
Example:
Canine extends Animal, Wolf extends Canine then Canine IS-A Animal, Wolf IS-A Canine and Wolf IS-A Animal are true.

With polymorphism, the reference type can be a superclass of the actual object type

In other words, anything that extends the declared reference variable type can be assigned to the reference variable.
This lets you do things like make
polymorphic arrays.
Ex -

```
Animal[] animals = new Animal[5]; // Declare an array of type Animal
animals[0] = new Dog();  // you can put ANY subclass of Animal in the Animal arr
animals[1] = new Cat();
animals[2] = new Wolf();
```

```
animals[3] = new Hippo();
animals[4] = new Lion();
for (Animal animal : animals) {
 animal.eat();
 animal.roam();
}
```

You can have polymorphic arguments and return types.

```
// Polymorphic argument example
class Vet {
    public void giveShot(Animal a) { // the 'a' parameter can take any Animal type
        a.makeNoise();
    }
}

class PetOwner {
 public void start() {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Hippo hippo = new Hippo();
    vet.giveShot(dog);
    vet.giveShot(hippo);
 }
}

// This is a polymorphic return type example because it returns a superclass Anir
class PetStore {
   public Animal getPet(boolean wantDog) {
       if (wantDog) {
           return new Dog();    // Dog IS-A Animal
       } else {
           return new Hippo();  // Hippo IS-A Animal
       }
```

```
    }
}
```

If I write my code using polymorphic arguments, where I declare the method parameter as a superclass type, I can pass in any subclass object at runtime. Cool. Because that also means I can write my code, go on vacation, and someone else can add new subclass types to the program and my methods will still work.

Keeping the contract: rules for overriding

Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.
Appliance appliance = new Toaster(); (Appliance is reference type and Toaster is object type)
With an Appliance reference to a Toaster, the
compiler cares only if class Appliance has the method you're invoking on an Appliance reference. But at runtime, the JVM does not look at the reference type (Appliance) but at the actual Toaster object on the heap.
So if the compiler has already approved the method call, the only way it can work is if the overriding method has the same arguments and return types.

Basically reference type (compile time) defines what variables and methods are available to the reference variable and the object type (run time) determines which version of an overridden method is executed.

1. **Arguments must be the same, and return types must be compatible.**

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type or a subclass type.

2. **The method can't be less accessible.**

That means the access level must be the same, or friendlier. You can't, for example, override a public method and make it private. What a shock that would be to the code invoking what it thinks (at compile time) is a public method, if suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!

Overloading a method

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.

1. **The return types can be different.**
2. **You can't change ONLY the return type.**
3. **You can vary the access levels in any direction.**

Bullet Points:

▼ A subclass extends a superclass.

▼ Inherited methods can be overridden; instance variables cannot be overridden (although they can be redefined in the subclass, but that's not the same thing, and there's almost never a need to do it.)

▼ When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (The lowest one wins.)

Chapter 8 → Serious Polymorphism

We know we can say:

```
Wolf aWolf = new Wolf();        // A Wolf reference to a Wolf object.
Animal aHippo = new Hippo();    // Animal reference to a Hippo object.

// But here's where it gets weird:
Animal animal = new Animal();   //Animal reference to an Animal object.
// what the heck does an Animal object look like? So make the class abstract.
```

The compiler won't let you instantiate an abstract class

An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose

of polymorphism (to use it as a polymorphic argument or return type, or to make a polymorphic array).

When you're designing your class inheritance structure, you have to decide which classes are abstract and which are concrete. Concrete classes are those that are specific enough to be instantiated or it just means that it's OK to make objects of that type.

Abstract methods

Besides classes, you can mark methods abstract, too. An abstract class means the class must be extended; an abstract method means the method must be overridden.

Abstract methods don't have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement all abstract methods.
Ex- public abstract void eat();
If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class.

Why not make a class generic enough to take anything?

Every class in Java extends class Object. Any class that doesn't explicitly extend another
class, implicitly extends Object. Simply we can say every class in Java is either a direct or indirect subclass of class Object (java.lang.Object).

The Object class serves two main purposes: to act as a polymorphic type for methods that need to work on any class that you or anyone else makes, and to provide real method code that all objects in Java need at runtime (and putting them in class Object means all other classes inherit them) like toString(), hashcode(), getClass(), equals(Object o) and others.

Q. If it's so good to use polymorphic types, why don't you just make ALL your methods take and return type Object?

For one thing, you would defeat the whole point of "type-safety," one of Java's greatest protection mechanisms for your code. With type-safety, Java guarantees that you won't ask the wrong object to do something you meant to ask of another

object type. Like, ask a Ferrari (which you think is a Toaster) to cook itself.
But the truth is, you don't have to worry about that fiery Ferrari scenario, even if you do use Object references for everything. Because when objects are referred to by an Object reference type, Java thinks it's referring to an instance of type Object. And that means the only methods you're allowed to call on that object are the ones declared in class Object! So if you were to say:

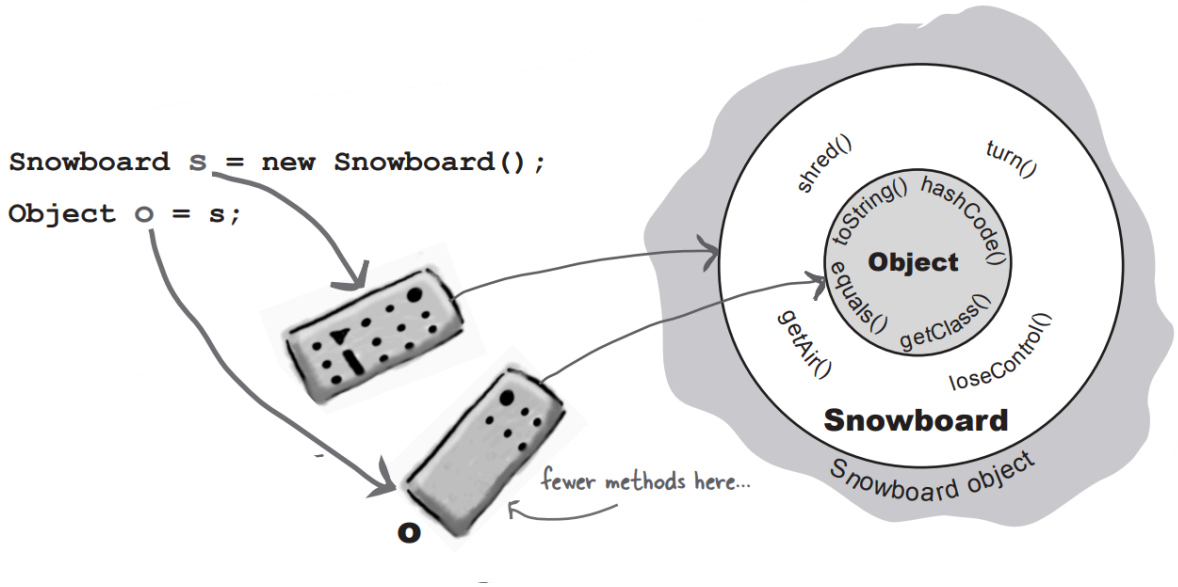
```
Object o = new Ferrari();
o.goFast(); //Not legal!
// You wouldn't even make it past the compiler.
```

Because Java is a strongly typed language, the compiler checks to make sure that you're calling a method on an object that's actually capable of responding.
In other words, you can call a method on an object reference only if the class of the reference type actually has the method.

Using polymorphic references of type Object has a price...

Everything comes out of an ArrayList<Object> as a reference of type Object, regardless of what the actual object is or what the reference type was when you added the object to the list. A cast can be used to assign a reference variable of one type to a reference variable of a subtype.
We can cast the Object, so we can treat it like he really is and call it's method -

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>(); // Object is type p
Dog d = new Dog();
myDogArrayList.add(d);

Object o = myDogArrayList.get(0);
// If you're not sure about object returned is an instance of Dog or Cat use this
if(o instanceOf Dog) {
   Dog dog = (Dog) o;
}
```

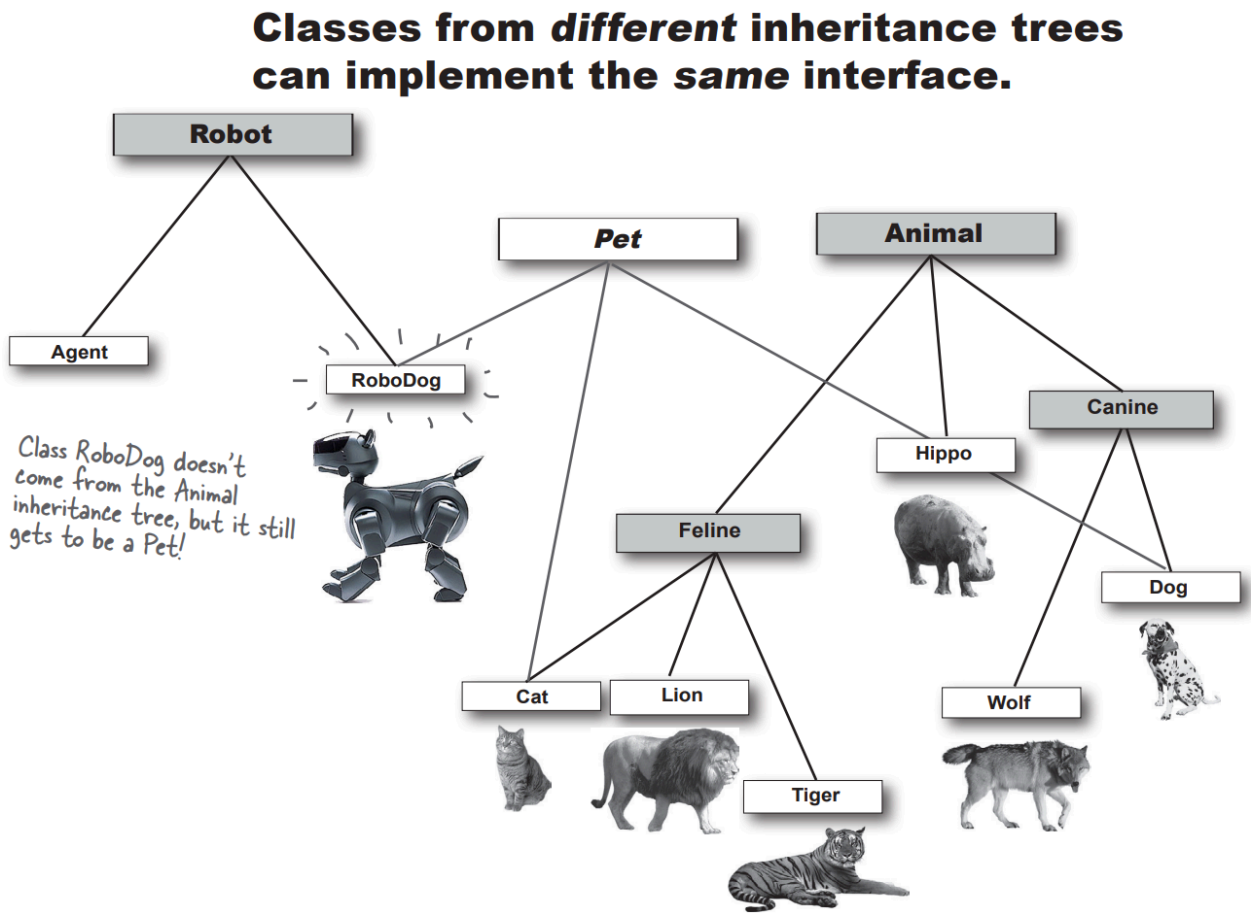


The compiler decides whether you can call a method based on the reference type, not the actual object type.

Interface

A Java interface solves your multiple inheritance problem. A Java class can have only one parent (superclass), and that parent class defines who you are. But you can implement multiple interfaces, and those interfaces define roles you can play.

```
public interface Pet {
 //Interface methods are implicitly public and abstract, so typing it is optional.
 public abstract void beFriendly();
 public abstract void play();
}

//  a class can implement multiple interfaces!
public class Dog extends Animal implements Pet, Saveable, Paintable { ... }
```



Dark colored classes are abstract classes while Pet is an interface. Other are concrete classes.

The main purpose of interfaces is polymorphism, polymorphism, polymorphism. When you use a class as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type. An argument of type Canine can accept a Wolf and a Dog, but not a Cat or a Hippo. But when you use an interface as a polymorphic type (like an array of Pets), the objects can be from anywhere in the inheritance tree. The only requirement is that the objects are from a class that implements the interface. Allowing classes in different inheritance trees to implement a common interface is crucial in the Java API.

How do you know whether to make a class, a subclass, an abstract class, or an interface?

▼ Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.

▼ Make a subclass (in other words, extend a class) only when you need to make a more specific version of a class and need to override or add new behaviors.

▼ Use an abstract class when you want to define a template for a group of subclasses, and you have at least some implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.

▼ Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

```
class BuzzwordsReport extends Report {
 void runReport() {
    super.runReport(); // superclass version of method imports stuff that subclass
    printReport();
 }
}
```

Sometimes the concrete code isn't enough to handle all of the subclass-specific work. So the subclass overrides the method and extends it by adding the rest of the code. The super keyword is really a reference to the superclass portion of an object. When subclass code uses super, as in super.runReport(), the superclass version of the method will run.

Chapter 9 → Life and Death of an Object

Bullet Points:

▼ Java has two areas of memory we care about: the Stack and the Heap.

▼ All local variables live on the stack, in the frame corresponding to the method where the variables are declared.

▼ Object reference variables work just like primitive variables—if the reference is declared as a
local variable, it goes on the stack.

▼ All objects live in the heap, regardless of whether the reference is a local or instance variable.

▼ Instance variables live within the object they belong to, on the Heap.

▼ If the instance variable is a reference to an object, both the reference and the object it refers to are on the Heap.

▼ A constructor is the code that runs when you say new on a class type. A constructor must have the same name as the class, and must not have a return type.

▼ If you don't put a constructor in your class, the compiler will put in a default constructor which is always a no-arg constructor.

▼ If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.

▼ All the constructors in an object's inheritance tree must run when you make a new object.

▼ Remember, a subclass might inherit methods that depend on superclass state (instance variables). For an object to be fully formed, all the superclass parts of itself must be fully formed, and that's why the superclass constructor must run first.

▼ When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class Object constructor.

▼ A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo. This all happens in a process called Constructor Chaining.

▼ If you don't provide a constructor, then the compiler puts one that looks like -

```
public ClassName() {
        super();
}
```

▼ If you do provide a constructor but you do not put in the call to super() then the compiler will put a call to super() in each of your overloaded constructors. The compiler inserted call to super() is always a no-arg call.

▼ The call to super() must be the first statement in each constructor! Because the superclass parts of an object have to be fully formed (completely built) before the subclass parts can be constructed.

▼ Use this() to call a constructor from another overloaded constructor in the same class.

▼ The call to this() can be used only in a constructor, and must be the first statement in a constructor.

▼ A constructor can have a call to super() OR this(), but never both!

▼ An object's life depends entirely on the life of references referring to it. An object becomes eligible for GC when its last live reference disappears.

▼ Three ways to get rid of an object's reference:

1. The reference goes out of scope, permanently

```
void go() {
        Life z = new Life();   // reference 'z' dies at end of method.
}
```

2. The reference is assigned another object

```
Life z = new Life();
z = new Life();        // the first object is abandoned when z is
```

'reprogrammed' to a new object.

3. The reference is explicitly set to null

```
Life z = new Life();
z = null;              // the first object is abandoned when z is 'deprogrammed'.
```

Chapter 10 → Numbers Matter

Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, collections like ArrayList only work with Objects.
There's a wrapper class for every primitive type and each one is named after the primitive type it wraps.

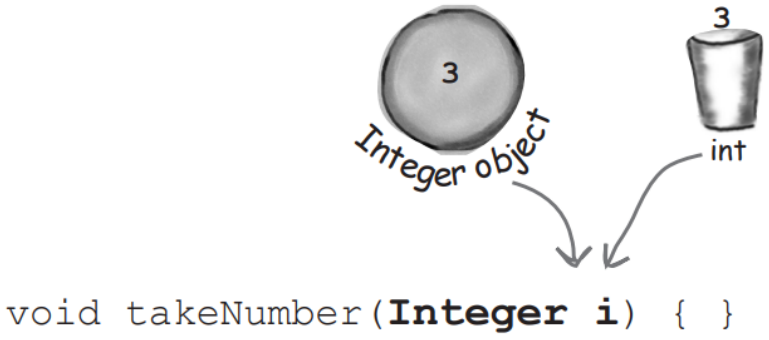
```
// wrapping a value
int i = 288;
Integer iWrap = new Integer(i);    // Give the primitive to the wrapper constructor.

// unwrapping a value
int unWrapped = iWrap.intValue();  // All the wrappers work like this. Boolean has
```

Java will Autobox primitives for you

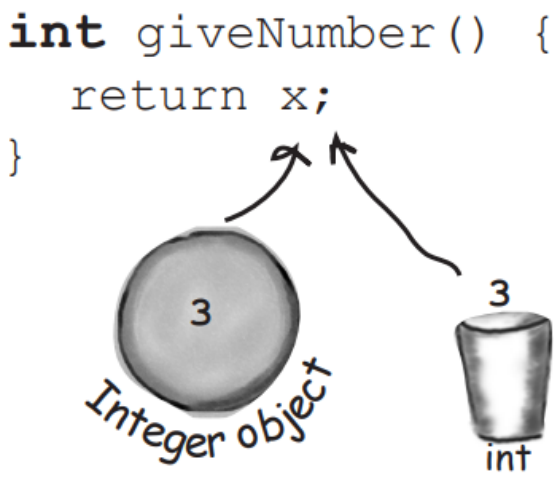
```
public void autoboxing() {
 int x = 32;
 ArrayList<Integer> list = new ArrayList<Integer>();
 list.add(x);  // Although there is NOT a method in ArrayList for add(int), the comp

 int num = list.get(0);  // Compiler automatically unwraps (unboxes) the Integer ol
}
```

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.

Method arguments -



Return values -



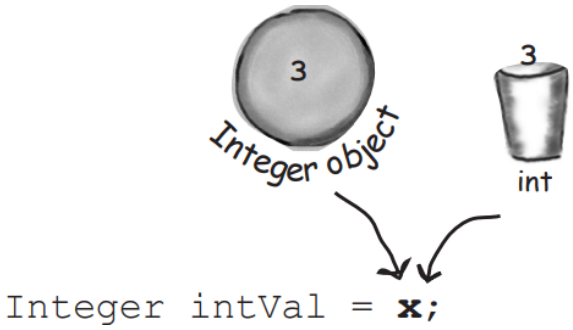
Boolean expressions -

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 -

The compiler simply converts the object to its primitive type before the operation.
Integer i = new Integer(42);
i++;
You can also do things like:
Integer j = new Integer(5);
Integer k = j + 3;

Assignments -



Wrappers have static utility methods too!

Converting a String to primitive value -

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

Converting primitives into a String -

```
double d = 42.5;
String s1 = "" + d; // '+' operator is overloaded in Java (the only overloaded opera
String s2 = Double.toString(d); // using a static method in class Double.
String s3 = String.valueOf(d); // an overloaded static method "valueOf" on String
                                                    // will get the String value of pretty much any
```

Number formatting

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.

The percent (%) says, "insert argument here" (the argument after comma and format it using these instructions).
%,d means "insert commas and format the number as a decimal integer."
%.2f means "format the number as a floating point with a precision of two decimal places."
%,.2f means "insert commas and format the number as a floating point with a precision of two decimal places."

```
long easierToRead = 1_000_000_000;
String s = String.format("%,d", easierToRead);
```

Format specifier pattern → % [argument number] [flags] [width] [.precision] type

("%,6.1f", 42.000); // comma is the flag, 6 is width etc.

Bullet Points:

▼ 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.

▼ The Math method acts on the argument but is never affected by an instance variable state. The only value that changes the way the round() and other methods 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?

▼ A static method should be called using the class name rather than an object reference variable.

▼ 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. Like the Math class.

▼ A static variable is a variable shared by all members of a given class i.e. all instances of the same class share a single copy of the static variables.

▼ A static method can access a static variable.

▼ Static variables are initialized when a class is loaded. 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.

▼ All static variables in a class are initialized before any object of that class can be created or before any static method of the class runs.

▼ 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 the constructor or in a static initializer:

```
static {
    DOG_CODE = 420;
}
```

▼ A static initializer is a block of code that runs when a class is loaded, before any other code can use the class.

▼ 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. A final method cannot be overridden. A final class cannot be extended.

Chapter 11 → Data Structures

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

Java lets you sort the good old-fashioned way, alphabetically, and it also lets you create your own custom sorting approaches (Comparator).
Java uses
Unicode, 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. By default, sorting in Java happens in what's called "natural order," which is more or less alphabetical.

Generics means more type-safety

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects; it's like the ArrayList is declared as ArrayList<Object>.
With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

Using generic CLASSES

A generic class means that the class declaration includes a type parameter. Generally, generic classes are classes that hold or operate on objects of some other type they don't know about.

```
// ArrayList class declaration
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {
    public boolean add(E o){
    }
}

List<String> thisList = new ArrayList<>();
// Is treated by the compiler as:
public class ArrayList<String> extends AbstractList<String> implements List<Str
    public boolean add(String o){
        // more code
    }
}
```

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).
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 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:

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

// 1. Using a type parameter defined in the class declaration
public boolean add(E o)
// you can use E here only because it's already been defined as part of class.

// 2. Using a type parameter that was NOT defined in the class declaration
public <T extends Animal> void takeThing(ArrayList<T> list)
// we can use <T> because we declared “T” at the start of the method declaratio


This:
public <T extends Animal> void takeThing(ArrayList<T> list)
Is NOT the same as this:
public void takeThing(ArrayList<Animal> list)
// 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.
```

Sort method

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

java.util.Collections.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.

Both methods that take a comparator, Collections.sort(List, Comparator) and List.sort(Comparator), do the same thing.
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.

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

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

// T must be of type Comparable, ? super T means that the type parameter for C
// must be of type T or one of T's supertypes.
```

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

The Song class needs to implement java.lang.Comparable

The big question is: what makes one song less than, equal to, or greater than another song? And if you use the songList like Collections.sort(songList) this will

throw an error. Because it doesn't know how to sort the Song objects.
So, you can't implement the Comparable interface until you make that decision.

The method compareTo() returns:
a
negative integer if this object is less than the passed object;
a
zero if they're equal;
a
positive integer if this object is greater than the passed object.

```
class SongV3 implements Comparable<SongV3> { //we are specifying the type t
    private String title; // implementing class can be compared against as <SongV
    private String artist;
    private int bpm;
    public int compareTo(SongV3 s) {
        return title.compareTo(s.getTitle()); // calling Strings compareTo() method.
    }
    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;
    }
    // We override toString(), because when you do a System.out.println(aSongOb
    // calls the toString() of EACH element in the list(SongV3) and we'll see title
    public String toString() {
        return title;
```

```
    }
}

class MockSongs {
    public static List<SongV3> getSongsV3() {
    List<SongV3> songs = new ArrayList<>();
    songs.add(new SongV3("somersault", "zero 7", 147));
    songs.add(new SongV3("cassidy", "grateful dead", 158));
    songs.add(new SongV3("$10", "hitchhiker", 140));
    return songs;
    }
}
// then we can call the Collections.sort(MockSongs.getSongsV3())
```

Using java.util.Comparator

What if I want two different views (sort) of the song list, one by song tile 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?

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.

```
// Updating the Jukebox to use a Comparator
// 1. Create a separate class that implements Comparator.
// 2. Make an instance of the Comparator class.
// 3. Call the List.sort() method, giving it the instance of the Comparator class.

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

```
    }
    public void go() {
        List<SongV3> songList = MockSongs.getSongsV3();
        System.out.println(songList); // unsorted list
        Collections.sort(songList);  // sort by title
        System.out.println(songList);

        ArtistCompare artistCompare = new ArtistCompare();
        songList.sort(artistCompare); // sort by artist
        System.out.println(songList);
    }
}
class ArtistCompare implements Comparator<SongV3> {
    public int compare(SongV3 one, SongV3 two) {
            return one.getArtist().compareTo(two.getArtist()); // calling String compa
            // method, since Strings already know how to alphabetize themselves.
    }
}
```

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

In previous codes, we were using Collections.sort(songList), because SongV3 implements Comparable and songList.sort(artistCompare) because the ArtistCompare class implements Comparator.

Q. But why doesn't every class implement Comparable?
Do you really believe that everything can be ordered? It would be misleading If you have element types that just don't lend themselves to any kind of natural ordering. 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.

A better approach would be to handle all of the sorting definitions in classes that implement 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 either by title or artist.

```
// you can add this Comparator in above Comparator code to sort using title.
// there is a better way ahead for both Comparator :)
 TitleCompare titleCompare = new TitleCompare();
 songList.sort(titleCompare);

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

The return line is doing all the work and the rest is necessary syntax for Comparator. There's more than one way to declare small pieces of functionality like this(for return line). One approach is inner classes. 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".

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

Enter lambdas! Leveraging what the compiler can infer

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.

```
// no need to create custom Comparator class, just put the sorting logic inside m
songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
songList.sort((one, two) -> one.getArtist().compareTo(two.getArtist()));
```

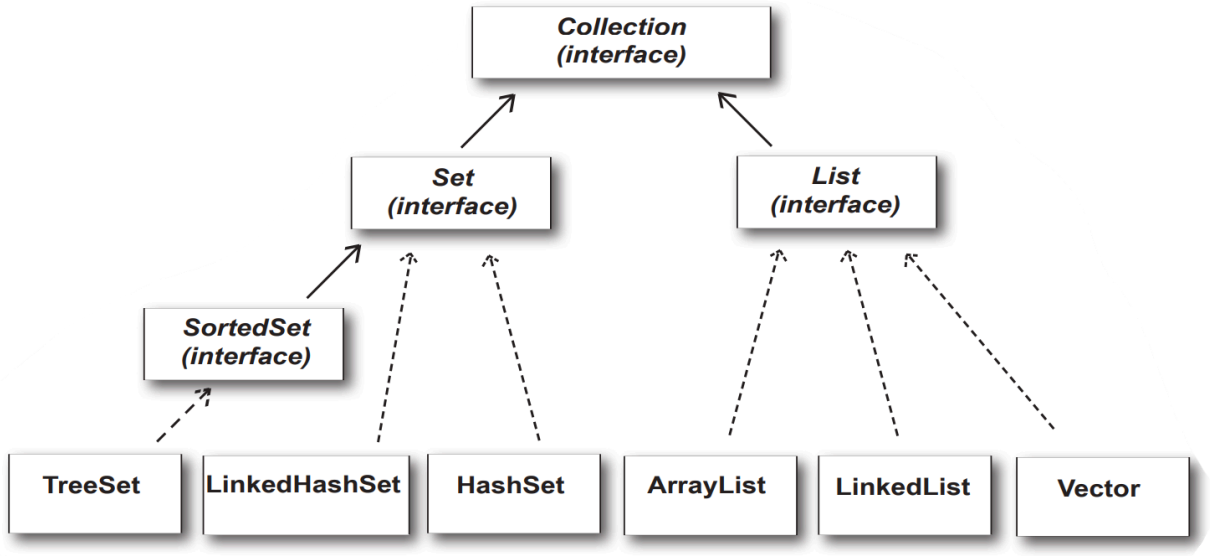
The Collection API (part of it)

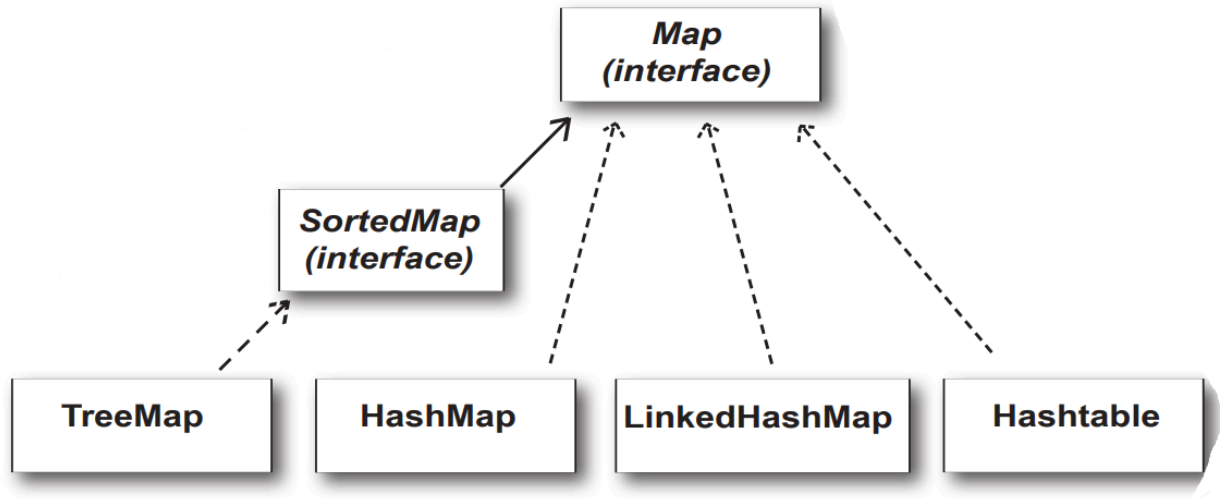
The Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework". So Maps are still collections, even though they don't include java.util.Collection in their inheritance tree.
From the Collection API, we find three main interfaces, List, Set, and Map.

LIST - when sequence matters
Collections that know about index position. Lists know where something is in the list. You
can have more than one element referencing the same object.

SET - when uniqueness matters
Collections that do not allow duplicates. Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal).

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.





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.

If the HashSet finds a matching hash code for two objects then the HashSet will call object's equals() methods to see if these hash code–matched objects really are equal. HashSet don't preserve sort order.

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.

Java Object Law for hashCode() and equals()

▼ If two objects foo and bar are equal, foo.equals(bar) and bar.equals(foo) must be true, and both foo and bar must have same hash codes.

▼ If two objects have the same hash code value, they are NOT required to be equal.

▼ 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)

```
// class SongV3 ... {       this class is created above, so we are just adding to it
// if we don't specify these method then hashset just add all duplicates song obje
        public boolean equals(Object aSong) {
        SongV3 other = (SongV3) aSong;
           return title.equals(other.getTitle());
        }
        public int hashCode() {
           return title.hashCode();
        }
// }

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

```
// class Jukebox4 ... {
        // ...
        Set<SongV3> songSet = new HashSet<>(songList);
    }
```

TreeSet

It keeps the list sorted. It works just like the sort() method in that if you make a TreeSet without giving 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 elements in the list must be of a type that implements Comparable i.e. Son
Set<SongV3> songSet = new TreeSet<>(songList); // TreeSet will use SongV3's
// method to sort the items in songList.

// Passing Comparator to TreeSet constructor
Set<SongV4> songSet = new TreeSet<>((o1, o2) -> o1.getBpm() - o2.getBpm());
songSet.addAll(songList);  // to add the songList values in TreeSet
```

Map

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Kathy", 42);
scores.put("Bert", 343);

System.out.println(scores.get("Bert"));
// When you print a Map, it gives you the key=value pairs, in braces { } instead of
// brackets [ ] you see when you print lists and sets.
System.out.println(scores);

// If we wanted to make sure that no-one change the collection after we'd created
return Collections.unmodifiableList(scores);
```

Convenience Factory Methods for Collections

They allows 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.
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 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()
List<String> strings = List.of("somersault", "cassidy", "$10");
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()
 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()
// If you want to put less than 10 entries into your Map, you can use Map.of:
 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 ke
// 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", "Came
                                                    Map.entry("Jay", "Homecase"
```

Finally, back to generics

```
public class TestGenerics1 {
    public static void main(String[] args) {
    List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
    takeAnimals(animals);
    }
    // If you declare a method to take List<Animal>, it can take ONLY a List<Anima
    // not List<Dog> or List<Cat>.
    public static void takeAnimals(List<Animal> animals) {
        for (Animal a : animals) {
            a.eat(); // we can call ONLY the methods declared in type Animal, since t
                        // animals parameter is of type List<Animal>.
        }
        animals.add(new Cat());
    }
}

// But will it work with List<Dog>? No it'll not work!
List<Dog> dogs = List.of(new Dog(), new Dog());
takeAnimals(dogs);

// What could happen if it were allowed...?
// There’s certainly nothing wrong with adding a Cat to a List<Animal>, and that’s
// whole point of having a List of a supertype like Animal—so that you can put all
// of animals in a single Animal List.

// If you passed a Dog List—one meant to hold ONLY Dogs—to takeAnimals that
// Animal List, then suddenly you’d end up with a Cat in the Dog list. The compile
// 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 t
// the risk.
```

We can do this with wildcards

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) {}
// 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?"
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!

Using the method's generic type parameter

```
// When you call the method, we're going to get the same type back as you put ir
public <T extends Animal> List<T> takeAnimals(List<T> list) { }

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);

// If the method used the wildcard for both method parameter and return type, th
// nothing to guarantee they're the same type. In fact, anything calling the metho
// almost no idea what's going to be in the collection, other than "some sort of an
public List<? extends Animal> takeAnimals(List<? extends Animal> animals) { }
List<? extends Animal> vaccinatedSomethings = takeAnimals(dogs);
```

Bullet Points:

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

▼ ArrayList<String> songs = new ArrayList<>();
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. So no type needed in right <>.

▼ Q. Why use List instead of ArrayList as the reference type?
Using the List interface instead of ArrayList implementation provides flexibility through polymorphism. Code only needs to know it's working with a List interface and its methods (add(), size(), etc.), not the specific implementation. This makes it safer since code can't access implementation-specific details, and allows easy switching between different List types (ArrayList, LinkedList, etc.) without changing code throughout the project.

▼ 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.

Chapter 12 → Lambdas and Streams: What, Not How

Doing something to every item in a list is a really common thing. So, instead of writing a for loop every time we want to process each item in a list, we can use the forEach method from the Iterable interface, which List implements.

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

There are other common tasks that we could get the API to do for us and so Java 8 introduced the Streams API, which provides methods for working with Collections and other classes. Rather than just offering helpful methods, Streams API represents a new approach to data processing. It allows developers to create specifications for what they want to learn about their data.

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.

To use the Streams methods, we need a Stream object. 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();

// limit method returns another Stream of Strings, which we'll assign to another v
Stream<String> limit = stream.limit(4);

// Like everything in Java, the stream variables in the example are Objects. But a
// does not contain the elements in the collection.
System.out.println("limit = " + limit); // limit = java.util.stream.SliceOps$1@7a0ac6
```

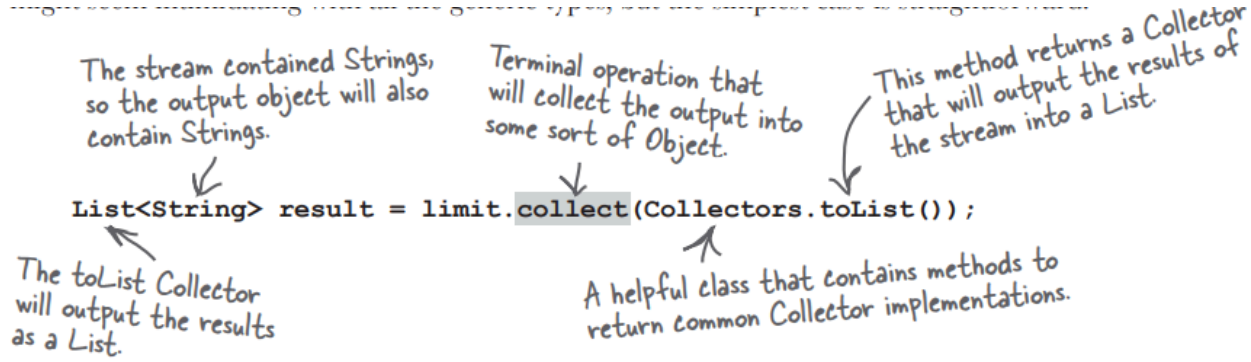
Stream methods that return another Stream are called Intermediate Operations. Streams are
lazy. That doesn't mean they're slow or useless! It means that each intermediate operation is just the instruction about what to do; it doesn't perform the instruction itself. Intermediate operations are lazily evaluated.

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

A recipe only tells how to cook. Opening the recipe doesn't automatically present you with a cake. I need to gather ingredients (Collections are not ingredients) and follow the instructions to get the result I want.
I need to call Streams "do it" methods, to get the result I want. These methods are called
Terminal Operations and they will actually return something to me.
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.

```
long result = limit.count(); // 'count' terminal operator
System.out.println("result = " + result);
```

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.



Terminal operation that is, .collect():
performs all intermediate operations, collects the results according to the instructions passed into it and then returns those results

Building blocks can be stacked and combined

The source (stream from a collection), the intermediate operation(s), and the terminal operation all combine to form a **Stream Pipeline**. This pipeline represents a query on the original collection.
Every intermediate operation acts on a Stream and returns a Stream. That means you can chain together as many of these operations as you want, before calling a terminal operation to output the result.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
List<String> result = strings.stream()
                                    .sorted()//Sort what's in the stream (not origina
                                    .limit(4)
                                    .collect(Collectors.toList());

// Customizing the building blocks i.e. Lambda expression that tells the sorted me
// how to sort the strings in the stream
```

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

Guidelines for working with streams

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.**
   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.

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

   Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

3. **Don't change the underlying collection while the stream is operating.**
   Streams API is a way to query a collection, but it doesn't make changes to the collection itself.

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().

Collecting results in different ways

If you're using Java 10 or higher, you can use Collectors. toUnmodifiableList, instead of using Collectors.toList, when you call collect, if you want to make sure your results aren't changed by anything.
The Collectors class has convenience methods for collecting toList, toSet, and toMap, as well
as (since Java 10) toUnmodifiableList, toUnmodifiableSet and toUnmodifiableMap.

We can collect stream into a Map of key/value pairs. We'll need to provide some functions to tell the collector what will be the key and what will be the value.

Collectors.joining
We 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.

Lambda

Passing behavior around:
forEach(do this: System.out.println(item));
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, "→" 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....

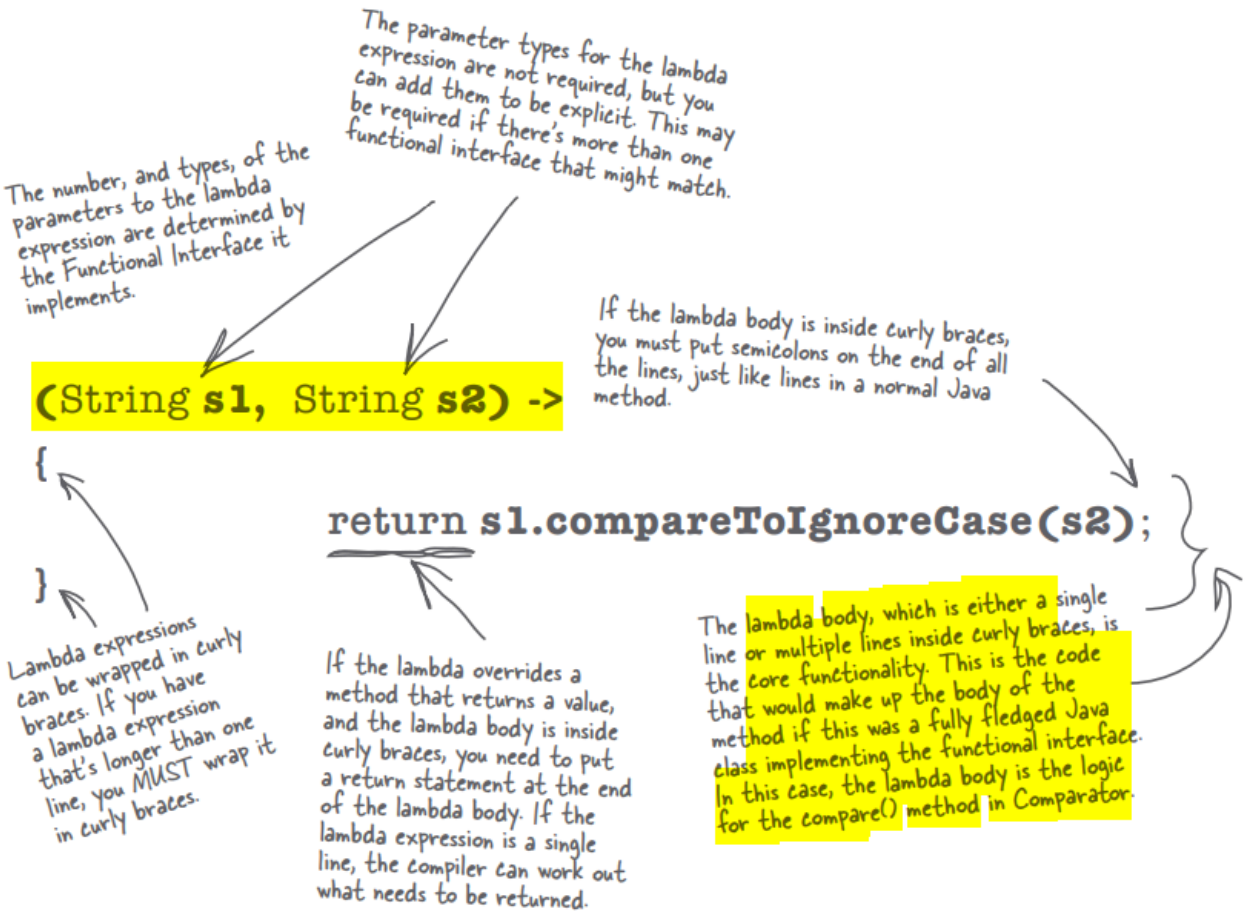
```
// The item is the input parameter, and the code after → is the action that gets
// executed when the forEach method is called.
forEach( item → System.out.println(item) // (this is the lambda's body) );
```

Lambda expressions are objects, and you run them by calling their Single Abstract Method.
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.
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);
        //element is the lambda's parameter, "item" in this case (above lambda e
    }
}
```

Anatomy of a lambda expression



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 of Lambda

A lambda might have more than one line

```
// a lambda expression that implements Comparator<String>
(str1, str2) -> {
    int l1 = str1.length();
    int l2 = str2.length();
    return l2 - l1;
}
```

Single-line lambdas don't need ceremony

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

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.

A lambda might not return anything

The Functional Interface's method might be declared void. This is the case for lambda expression in a forEach method.

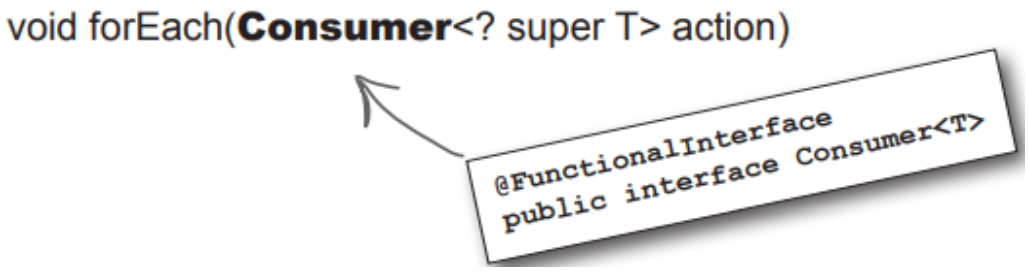
A lambda might have zero, one, or many parameters

The number of parameters the lambda expression needs is dependent upon the number of parameters the Functional Interface's method takes. The parameter types ("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.

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

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

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
str -> System.out.println(str)

@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
(str1, str2) -> str1.compareToIgnoreCase(str2)
```

How can I tell if a method takes a lambda?



We've seen that lambda expressions are implementations of a functional interface. 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 (except primitive types) . 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.

Spotting Functional Interfaces

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.
Both
default (will be inherited by subclasses) and static methods (often used as helper 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.

Map and filter example:

```
// We'll know what the type of the (song) parameter is when we plug it into the St
// operation, since the input type to the lambda will be determined by the types ir
// stream. So song is a Song because filter() is acting on Stream of Song List.
List<Song> rockSongs = songs.stream()
                                    .filter(song -> song.getGenre().contains("Rock"))
                                     .collect(Collectors.toList());

// We now wants a list of genres, which means we need to somehow turn the sor
// in the stream into genre (String) elements. This is what map is for. The map
// operation states how to map(convert) from one type to another type.
// The map method takes a Function.
Function<Song, String> getGenre = song -> song.getGenre(); // use this in map()
// input parameter is Song, return String.

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

Sometimes you don't even need a lambda expression

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

// Instead of spelling this whole thing out, you can point the compiler to a methoc
```

```
// does the operation we want, using a method reference "::"

// A method reference—instead of using a “.” that would cause the compiler to ca
// method, use a “::” to point the compiler in the direction of the method.
// getGenre is the method we would call in lambda body.
Function<Song, String> getGenre = Song::getGenre;

// Method references can replace lambda expressions, but you don’t have to use
// Sometimes method references make the code easier to understand. Eg.

// It let you see which value is being used for sorting and in which direction, to or
// the songs from oldest to newest.
List<Song> result = allSongs.stream()
                                    .sorted((o1, o2) → o1.getYear() - o2.getYear())
                                    .collect(toList());

                          to
// By default, Comparator.comparingInt(Song::getYear) sorts the Song objects in
// order based on their year
List<Song> result = allSongs.stream()
                                    .sorted(Comparator.comparingInt(Song::getYear
                                     .collect(toList());
```

Some more terminal operations

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

// Checking if something exists
boolean result = songs.stream()
                              .anyMatch(s → s.getGenre().equals("R&B"));

Optional<T> findAny();
Optional<T> findFirst();
```

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

// Find a specific thing
Optional<Song> result = songs.stream()
                                    .filter(s -> s.getYear() == 1995)
                                    .findFirst();

// Count the items
long result = songs.stream()
                        .map(Song::getArtist)
                        .distinct()
                        .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.

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.

```
Optional<IceCream> optional = getIceCream("Strawberry");
if (optional.isPresent()) {
} else {
    System.out.println("No ice cream for you!");
}
```