Websites

Java Forum: <https://javaranch.com/>

Head First Java的Java code下載地址：wickedlysmart.com

Learning habit

## Principle

A person’s brain craves novelty

e.g. unusual, interesting, dangerous

## A stimulating learning process

1. Get - and keep – the reader’s attention.
2. Make it (knowledge & skills) visual （視覺刺激）
3. Use a conversational and personalized style （陪伴）
4. Get the learner to think more deeply

* The big picture & layout
* Think（Don’t just read. Stop and think.）
* Practice（Your brain is turned to learn and remember more when you do things than when you read about things.）
* Repetition (刺激multiple senses, the content gets coded into more than one area of your brain) （執行力）

1. Touch learns’ emotions

* curiosity, surprise, fun （過程好玩）
* achievement (when a problem is solved or you have made some progress)（結果有成就感）

1. Rest

* Part of the learning (especially the transfer to long-term memory) happens after you put the book down. Your brain needs time on its own, to do more processing.（消化）
* When your brain is getting overloaded, rest. （放鬆）

Every Java application has to have at least one class, and at least one main method (not one main per class; just one main per application).

About Exercise

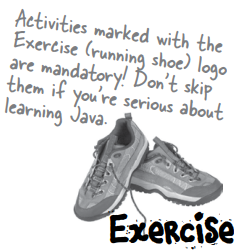
Pxxvii

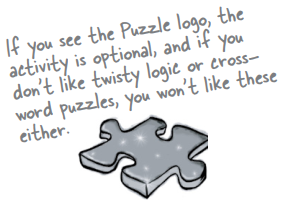
Read the “There are No Dumb Questions”

That means all of them. They are not optional side-bars – they’re part of the core content! Sometimes the questions are more useful than the answers.

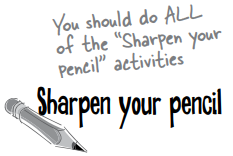
Type and run the code.

Pxxix

The end-of-chapter exercises are mandatory.

Puzzles are optional.

Answers for both are at the end of each chapter.

The ‘Sharpen Your Pencil’ exercises don’t have answers.

Not printed in the book, anyway. For some of them, there is no right answer, and for the others, part of the learning experience for the Sharpen activities is for you to decide if and when your answers are right. (Some of our suggested answers are available on wickedlysmart.com)

Terms

Java Virtual Machine (JVM)

a pair of curly braces { }

a comparison operator: < less than, > greater than, == equality

The equals operator uses two equals signs ==

The assignment operator is one equals sign =

Declare an int variable with a name and a type: int x;

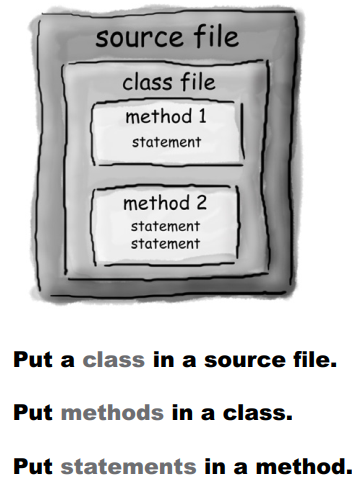
int x = 4; // assign 4 to x

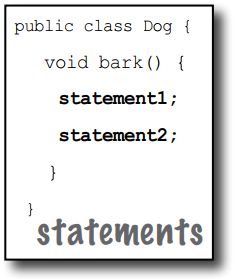
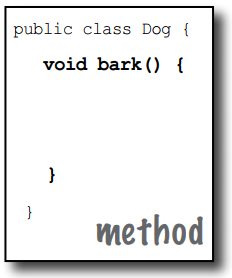
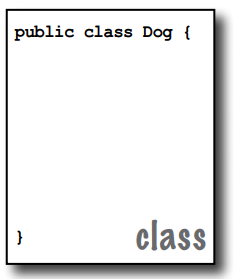
Put a boolean test inside parentheses（小括號）: while (x == 4) { }

Coding

## Chapters 1 & 2

### Java code structure





### While loops

Q: In my other language I can do a boolean test on an integer. In Java, can I say something like: int x = 1; while (x){ } Is this right?

A: No. A boolean and an integer are not compatible types in Java. Since the result of a conditional test must be a boolean, the only variable you can directly test (without using a comparison operator) is a boolean. For example, you can say: boolean isHot = true; while(isHot) { }

### System.out.print vs. System.out.println

System.out.println inserts a newline (think of println as printnewline while System.out.print keeps printing to the same line. If you want each thing you print out to be on its own line, use println. If you want everything to stick together on one line, use print.

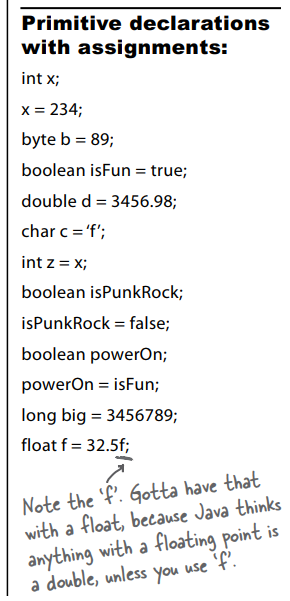
### To make a method or a variable ‘global’

marking a method as public and static makes it behave much like a ‘global’

If you mark a variable as public, static, and final – you have essentially made a globally-available constant.

## Chapter 3 Primitives and References

### primitives



### assign a value to a variable

You can assign a value to a variable in one of several ways including:

* type a literal value after the equals sign (x=12, isGood = true, etc.)
* assign the value of one variable to another (x = y)
* use an expression combining the two (x = y + 43)

### spillage

You can’t put a large value into a small cup.

For example, you can’t pour an int-full of stuff into a byte-sized container, as follows:

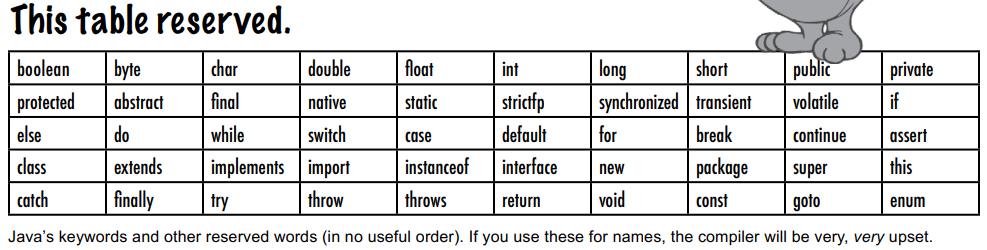
int x = 24;

byte b = x;

//won’t work!!

### names for variables

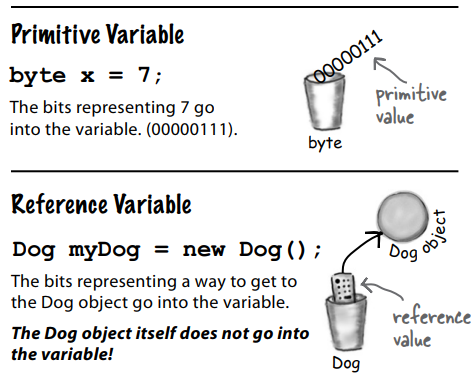
* It must start with a letter, underscore (\_), or dollar sign ($). You can’t start a name with a number.
* After the first character, you can use numbers as well. Just don’t start it with a number.
* It can be anything you like, subject to those two rules, just so long as it isn’t one of Java’s reserved words.



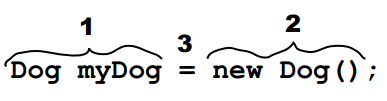
### primitive variables vs object reference variables

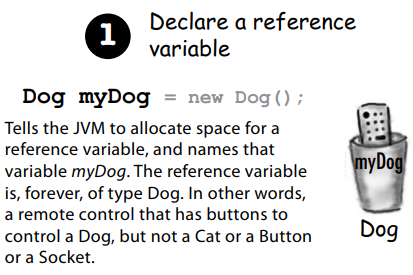
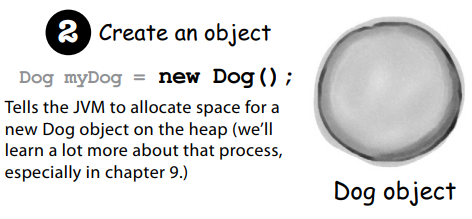
Although a primitive variable is full of bits representing the actual value of the variable, an object reference variable is full of bits representing a way to get to the object. It doesn’t hold the object itself, but it holds something like a pointer. Or an address.

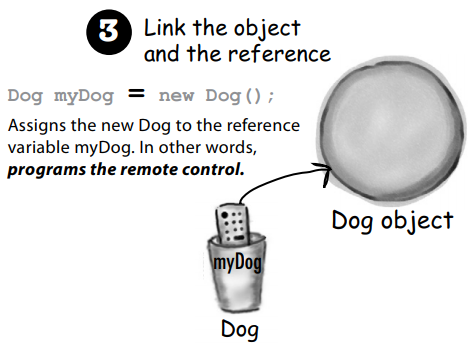
Objects live in one place and one place only—the garbage collectible heap!



### The 3 steps of object declaration, creation and assignment





### ‘re-reference’

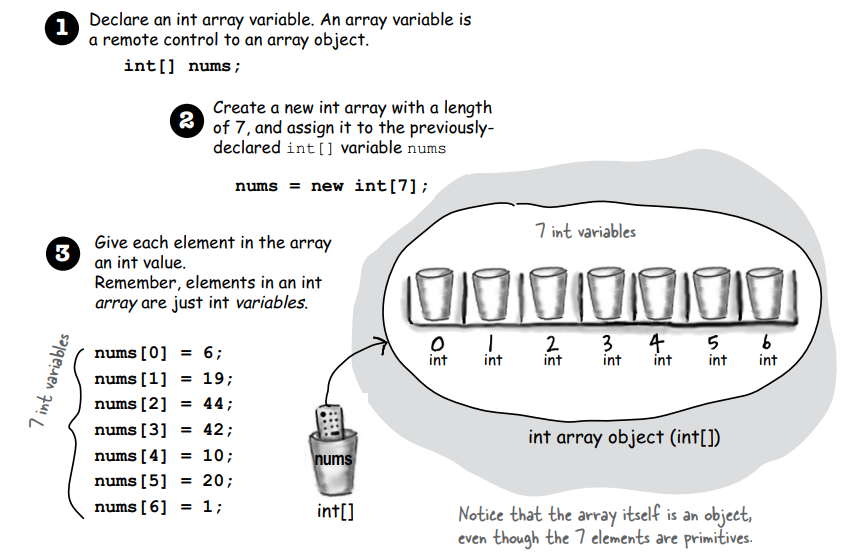
An object reference variable can be referring to one object (e.g. Dog), and then can refer to some other object (e.g. another Dog). But it cannot refer to a Cat.

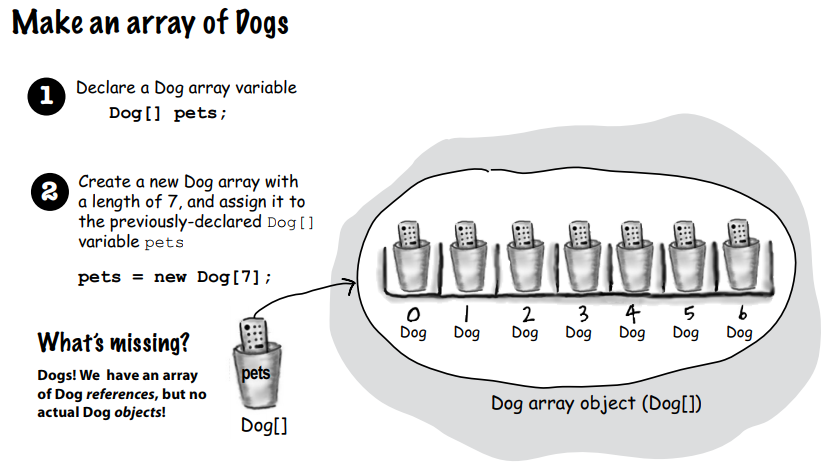
If an object reference variable is marked final, then once it is assigned a Dog, no other object can be assigned to it.

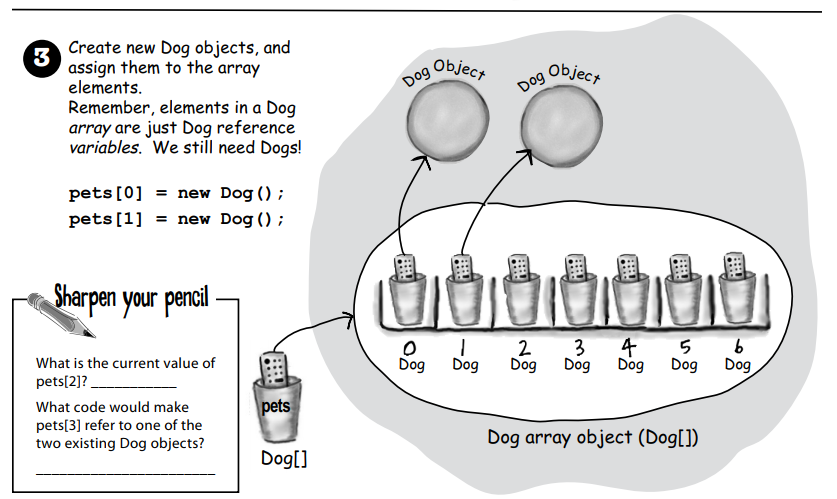
An object reference variable can refer to nothing, meaning the object reference variable is null. Null is a value. The reference is still like a remote control, but it’s like you brought home a new remote control and you don’t have a TV. The remote control is not programmed to control anything.

### arrays

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







## Chapter 4 Methods Use Instance Variables

### parameter VS argument

A method uses parameters.

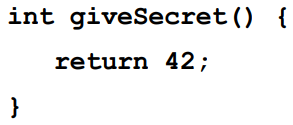
A caller passes arguments.

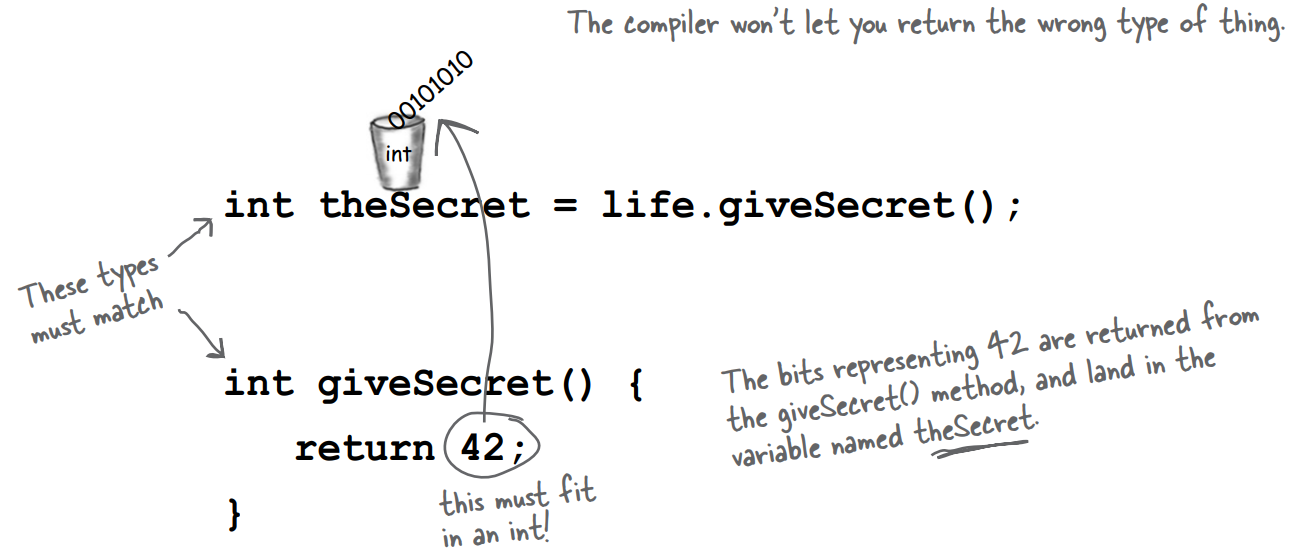
### void method

If a method is made with a void return type, it does not give anything back.

### method returning a specific type

If you declare a method to return a value, you must return a value of the declared type! (Or a value that is compatible with the declared type.)

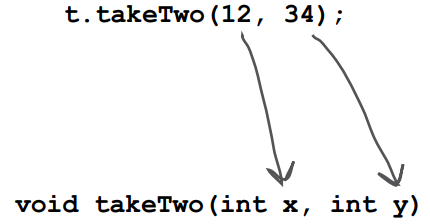




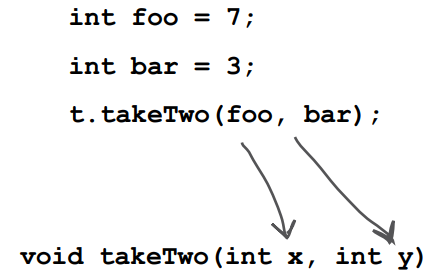
### arguments

If a method has parameters, you must pass arguments of the right type and order.



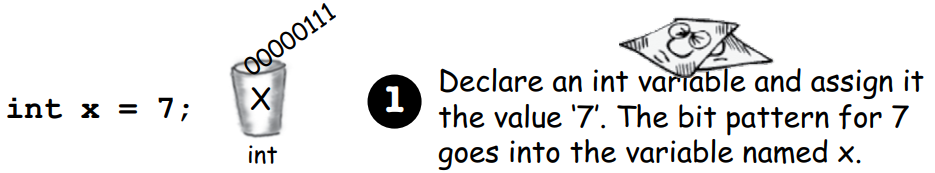




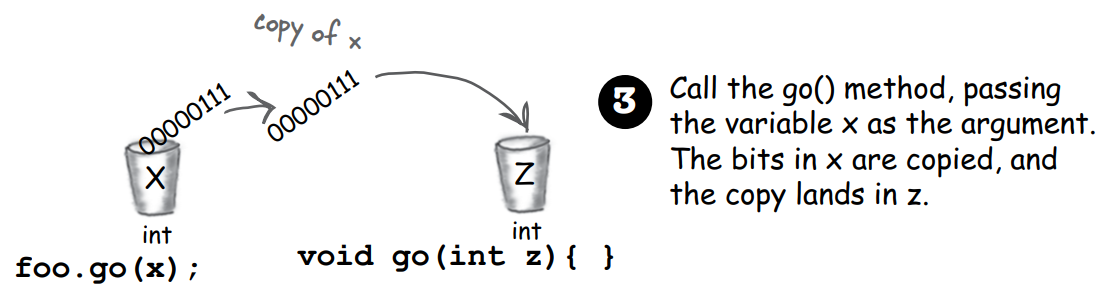


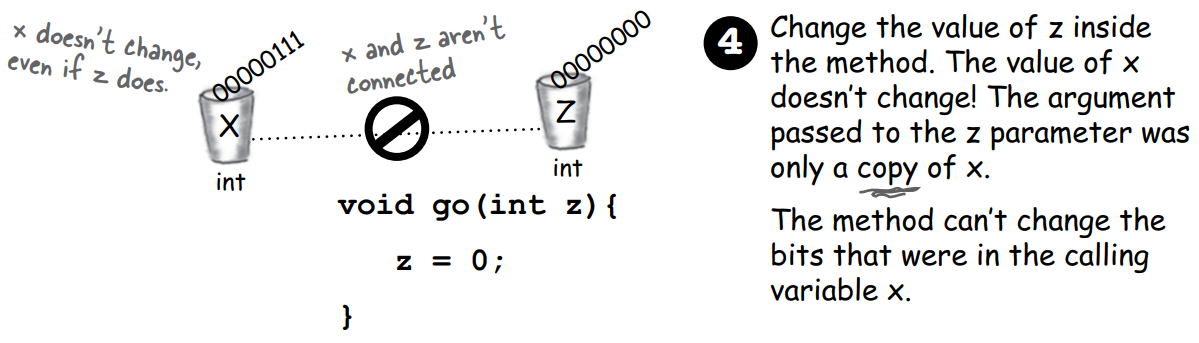
### Java is pass-by-value

Java is pass-by-value. This means pass-by-copy.









It is the same if the argument you want to pass is an object (actually, it should be a reference to an object) instead of a primitive. Java passes everything by value. If you pass a reference to an object into a method, you are passing a copy of the remote control.

### a method declaring multiple return values

A method can declare only one return value. If you want to return, say, three int values, then the declared return type can be an int array. Stuff those ints into the array, and pass it on back.

### Do I have to return the exact type I declared?

You can return anything that can be implicitly promoted to that type. So, you can pass a byte where an int is expected.

When the declared type is smaller than what you’re trying to return you must use an explicit cast.

Same thing with parameters.

### Do I have to do something with the return value of a method?

You can ignore the return value of a method.

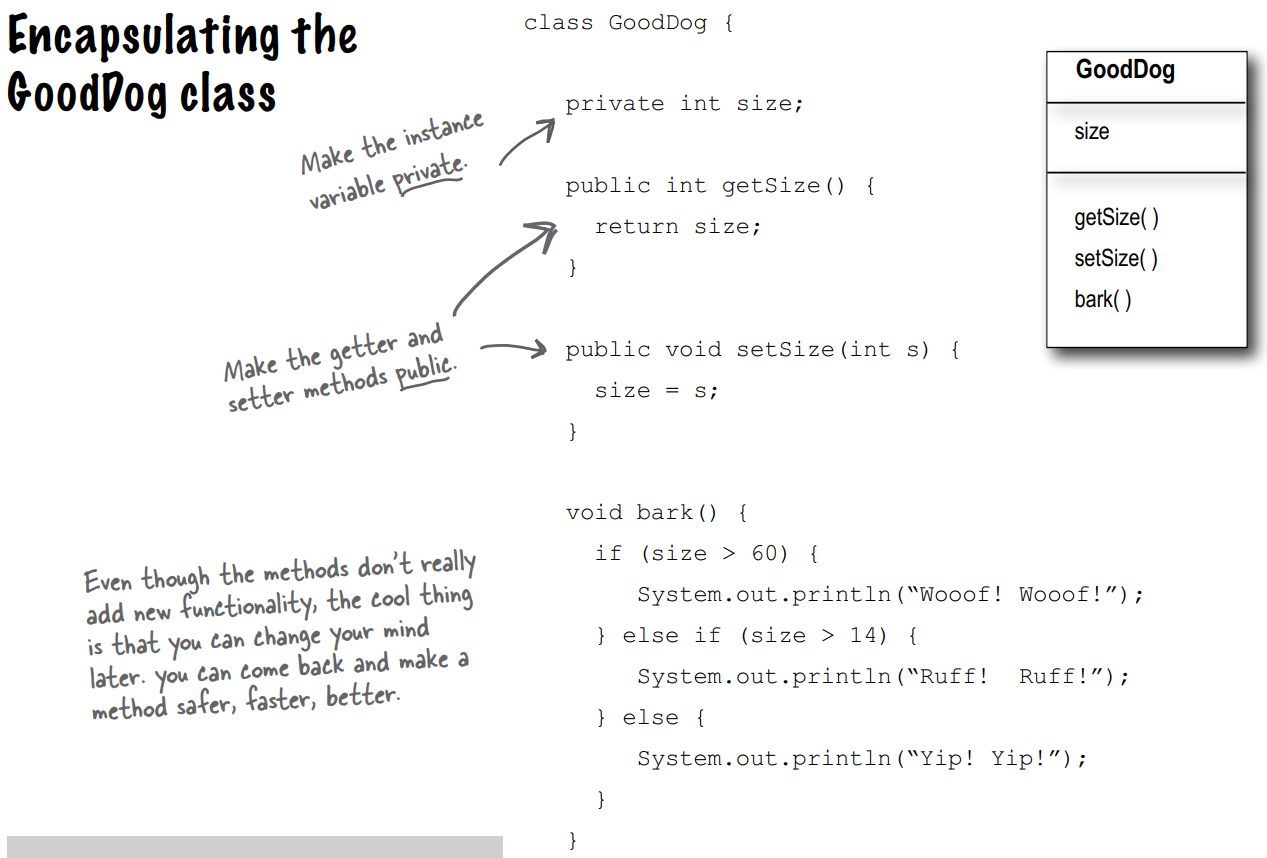
### getters and setters

Getters and Setters (Accessors and Mutators) let you, well, get and set things. Instance variable values, usually.

### hide the data

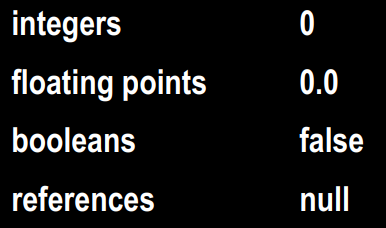
Mark instance variables private.

Mark getters and setters public.

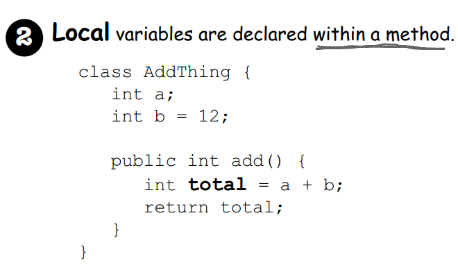
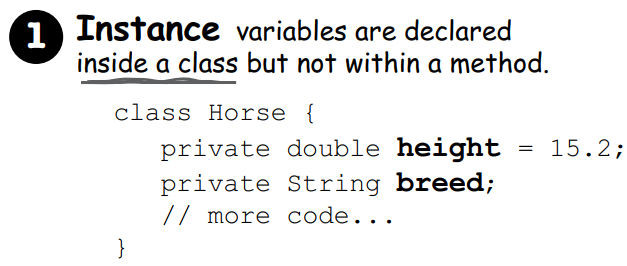


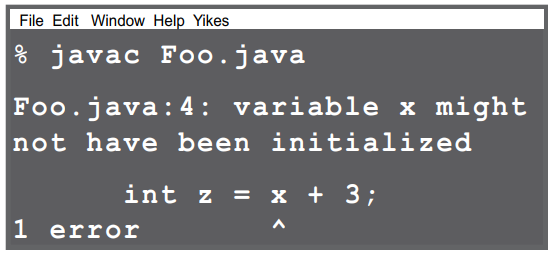
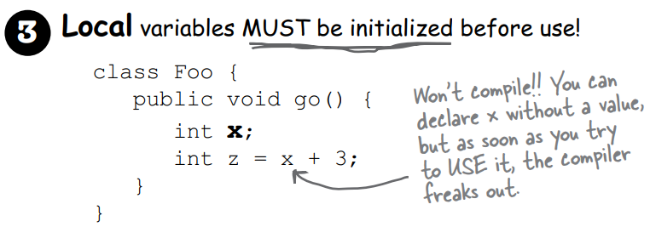
### Instance variables get a default value.

Instance variables always get a default value. If you don’t explicitly assign a value to an instance variable, or you don’t call a setter method, the instance variable still has a value!



### Local variables do not get a default value.





### method parameters – initialized?

Method parameters are virtually the same as local variables—they’re declared inside the method (well, technically they’re declared in the argument list of the method rather than within the body of the method, but they’re still local variables as opposed to instance variables).

But method parameters will never be uninitialized, so you’ll never get a compiler error telling you that a parameter variable might not have been initialized. But that’s because the compiler will give you an error if you try to invoke a method without sending arguments that the method needs.

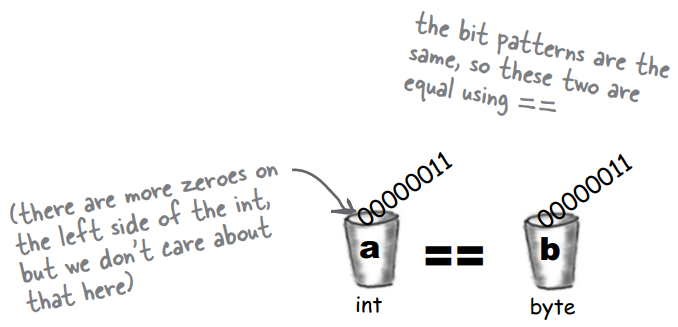
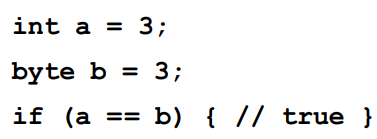
So parameters are ALWAYS initialized, because the compiler guarantees that methods are always called with arguments that match the parameters declared for the method, and the arguments are assigned (automatically) to the parameters.

### comparing variables (primitives and references)

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

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

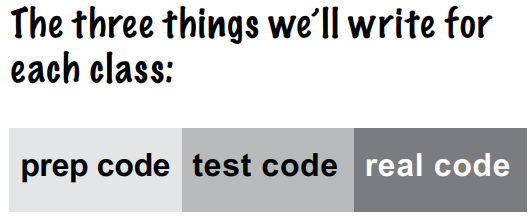
The == operator is used only to compare the bits in two variables. What those bits represent doesn’t matter. The bits are either the same, or they’re not.

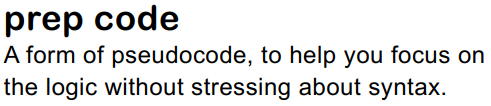


## Chapter 5 Writing a Programme

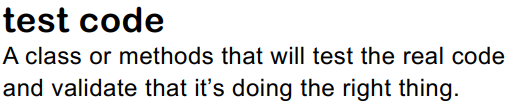
### developing a class

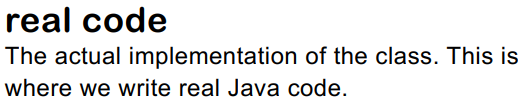
1. Figure out what the class is supposed to do.
2. List the instance variables and methods.
3. Write prepcode for the methods.
4. Write test code for the methods.
5. Implement the class.
6. Test the methods.
7. Debug and reimplement as needed.
8. Express gratitude that we don’t have to test our so-called learning experience app on actual live users.





Most prepcode includes three parts: instance variable declarations, method declarations, method logic.





### Extreme Programming (XP)

Considered by many to be “the way programmers really want to work”, XP emerged in the late 90’s and has been adopted by many companies. The thrust of XP is that the customer gets what he wants, when he wants it, even when the spec changes late in the game.

* Keep it simple.
* Don’t put in anything that’s not in the spec (no matter how tempted you are to put in functionality “for the future”).
* Develop in iteration cycles.
* Make small, but frequent, releases.
* Set realistic schedules, based around small releases.
* No killer schedules; work regular hours.
* Write the test code first.
* Don’t release anything until it passes all the tests.
* Refactor (improve the code) whenever and wherever you notice the opportunity.
* Program in pairs, and move people around so that everybody knows pretty much everything about the code.

### Integer.parseInt()

Integer.parseInt() works only on Strings that represent the ascii values for digits (0,1,2,3,4,5,6,7,8,9). If you try to parse something like “two” or “blurp”, the code will blow up at runtime. (By blow up, we actually mean throw an exception, but we don’t talk about exceptions until the Exceptions chapter. So for now, blow up is close enough.)

### two different styles of for loops

From the first version of Java there has been a single kind of for loop that looks like this:

for (int i = 0; i < 10; i++) {

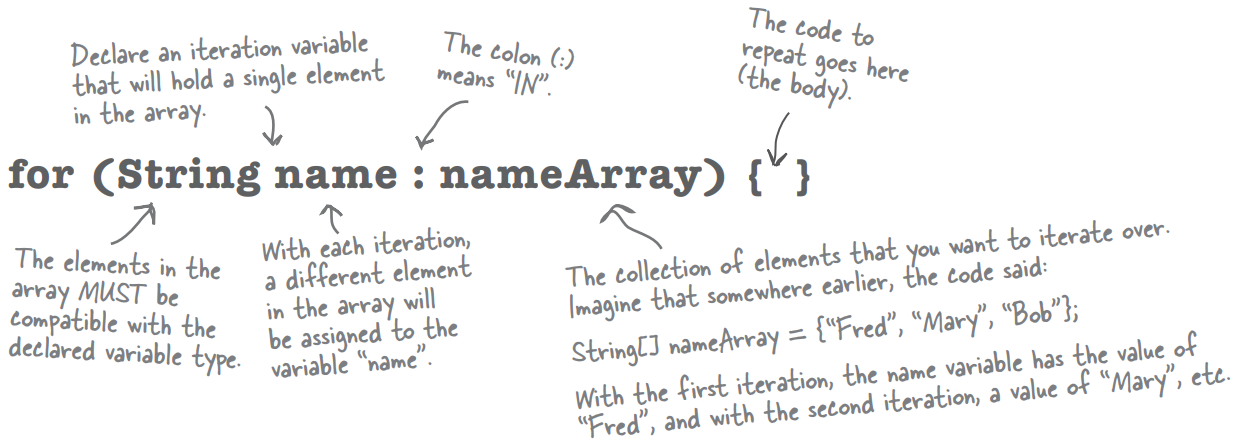
// do something 10 times

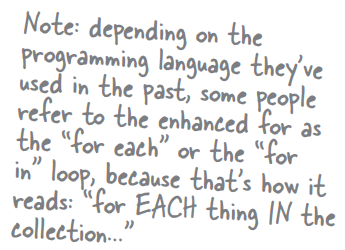
}

You can use this format for any kind of loop you need.

But... beginning with Java 5.0 (Tiger), you can also use the enhanced for loop when your loop needs to iterate over the elements in an array (or another kind of collection).

You can always use the plain old for loop to iterate over an array, but the enhanced for loop makes it easier.





### for loops VS while loops

Choose for loops over while loops when you know how many times you want to repeat the loop code.

### ++x VS x++

Putting the operator before the variable (for example, ++x), means, “first, increment x by 1, and then use this new value of x.” This only matters when the ++x is part of some larger expression rather than just in a single statement.

int x = 0; int z = ++x;

produces: x is 1, z is 1

But putting the ++ after the x give you a different result:

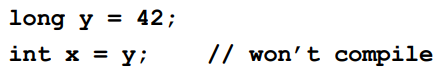
int x = 0; int z = x++;

produces: x is 1, but z is 0!

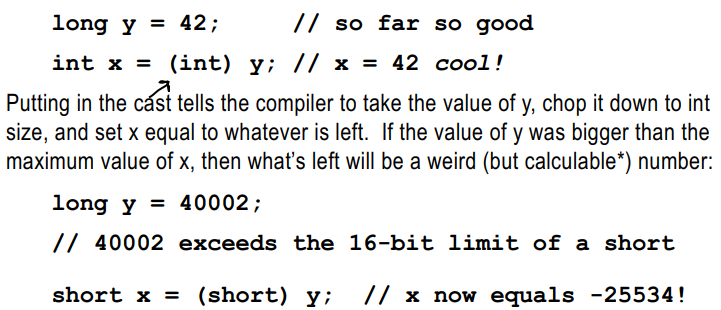
z gets the value of x and then x is incremented.

### casting primitives

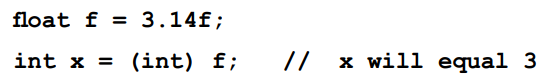
You can’t shove a big thing directly into a small thing.



To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the cast operator. It looks like this:



And let’s say you have a floating point number, and you just want to get at the whole number (int) part of it:

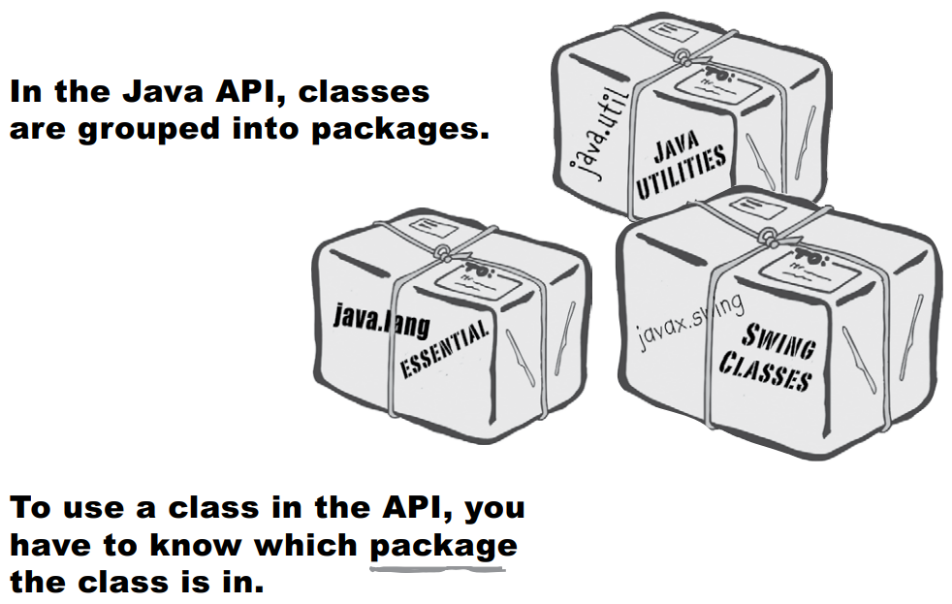


## Chapter 6 Get to Know The Java API

The Java library is known as the Java API.

The core Java library is a giant pile of classes just waiting for you to use like building blocks.

A Java API/ library includes packages. A package includes classes(Classes are grouped into packages.).



### Short Circuit Operators ( && , || )

The operators we’ve looked at so far, && and ||, are known as short circuit operators.

In the case of &&, the expression will be true only if both sides of the && are true. So if the JVM sees that the left side of a && expression is false, it stops right there! Doesn’t even bother to look at the right side.

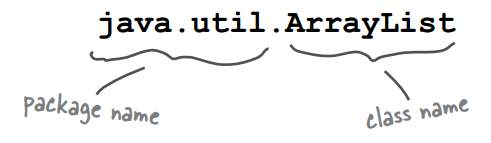
Similarly, with ||, the expression will be true if either side is true, so if the JVM sees that the left side is true, it declares the entire statement to be true and doesn’t bother to check the right side.

### Non Short Circuit Operators ( & , | )

When used in boolean expressions, the & and | operators act like their && and || counterparts, except that they **force the JVM to always check both sides of the expression**. Typically, & and | are used in another context, for manipulating bits.

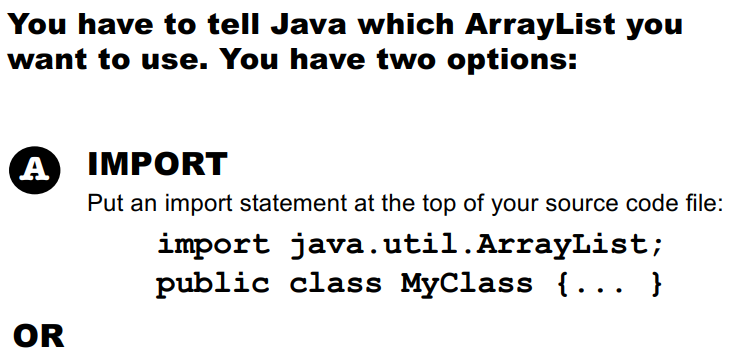
### full name of the class

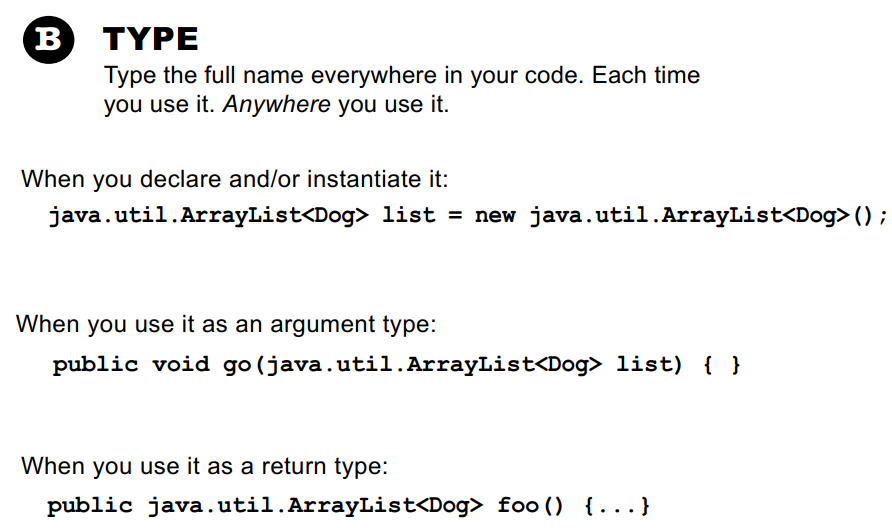
You have to indicate the full name of the library class you want to use, and that means package name + class name.



You have to know the full name\* of the class you want to use in your code. Unless the class is in the java.lang package.

The class java.lang is the one you don’t have to import.





### purposes of a package

First, they help the **overall organization** of a project or library. Rather than just having one horrendously large pile of classes, they’re all grouped into packages for specific kinds of functionality (like GUI, or data structures, or database stuff, etc.)

Second, packages give you **a name-scoping**, to help prevent collisions if you and 12 other programmers in your company all decide to make a class with the same name.

Third, packages provide a level of **security**, because you can restrict the code you write so that only other classes in the same package can access it.

### when a package starts with javax

Packages in the standard library don’t have that “x”, and only extensions have the “x”. These classes which were known as extensions were other packages not included in the standard library.

The mother of all standard extensions was the Swing library. It included several packages, all of which began with javax.swing.

When you see a package in the library that begins with javax, you know it started life as an extension, and then got a promotion (to become a part of a standard library).

(同一級別：The Java library is known as the Java API. an extension)

### declare the type of the array

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 as you’ll learn in the next couple of chapters)

### an ArrayList wrapping and unwrapping primitives

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.

## Chapter 7 Inheritance and Polymorphism

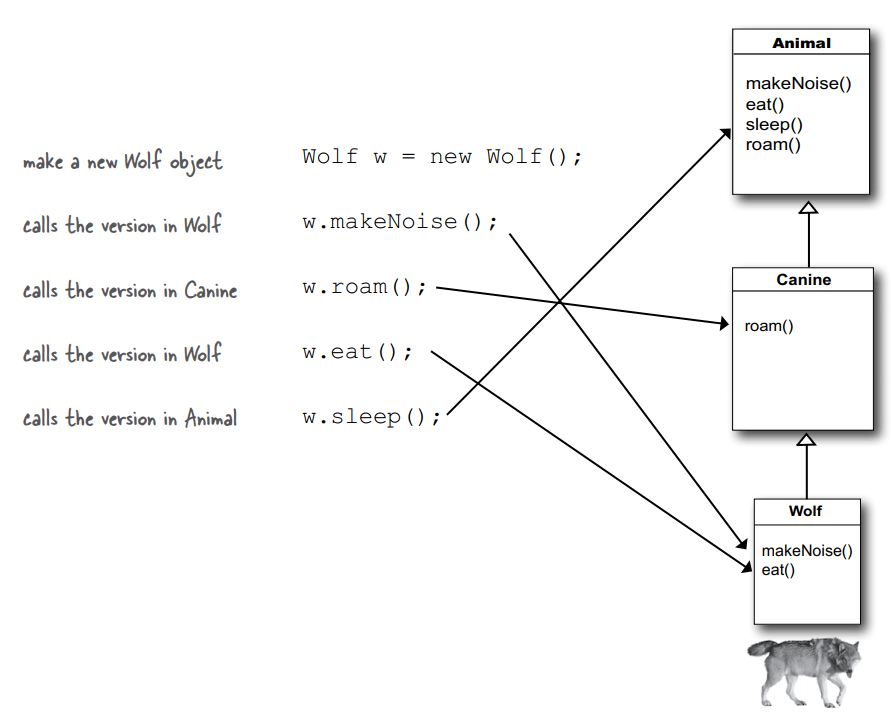
### add and override

A subclass can **add new methods and instance variables** of its own, and it can **override** the **methods** it inherits from the superclass.

Instance variables are not overridden because they don’t need to be. They don’t define any special behavior, so a subclass can give an inherited instance variable any value it chooses.

### calling the most specific version of the method

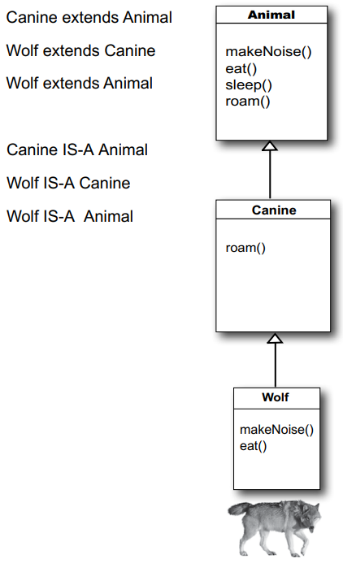
When you call a method on an object reference, you’re calling the most specific version of the method for that object type. In other words, the lowest one wins! “Lowest” meaning lowest on the inheritance tree. Canine is lower than Animal, and Wolf is lower than Canine, so invoking a method on a reference to a Wolf object means the JVM starts looking first in the Wolf class. If the JVM doesn’t find a version of the method in the Wolf class, it starts walking back up the inheritance hierarchy until it finds a match.



### using is-a

When one class **inherits from** another, we say that the subclass **extends** the superclass. When you want to know if one thing should extend another, apply the **IS-A** test. Triangle IS-A Shape, yeah, that works. Cat IS-A Feline, that works too. Surgeon IS-A Doctor, still good.

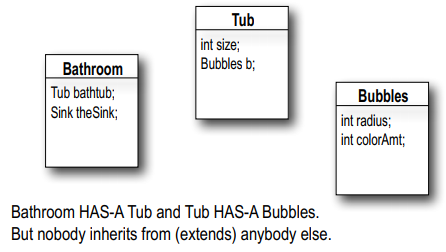
This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.



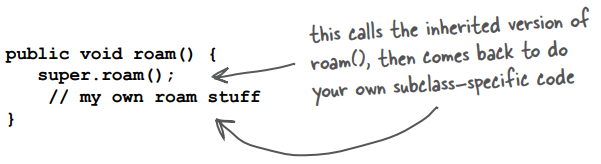
Inheritance IS-A relationship works in only one direction! Triangle IS-A Shape makes sense, so you can have Triangle extend Shape. But the reverse—Shape IS-A Triangle—does not make sense.

### using has-a

Tub IS-A Bathroom is definitely false. Bathroom IS-A Tub doesn’t work either. Tub and Bathroom are related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship.



### use BOTH the superclass version and my overriding subclass version of a method



### the level of access

A superclass can choose whether or not it wants a subclass to inherit a particular member by the level of access the particular member is given. There are four access levels that we’ll cover in this book. Moving from most restrictive to least, the four access levels are:



Access levels control who sees what.



### when a subclass inherits a member

When a subclass inherits a member, it is **as if the subclass defined the member itself**.

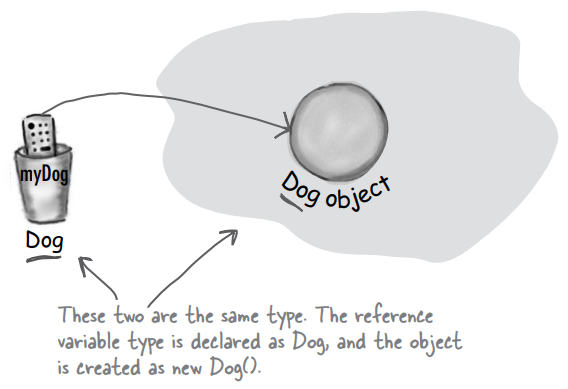
In the Shape example, Square inherited the rotate() and playSound() methods and to the outside world (other code) the Square class simply has a rotate() and playSound() method.

The members of a class include the variables and methods **defined** in the class plus anything **inherited** from a superclass.

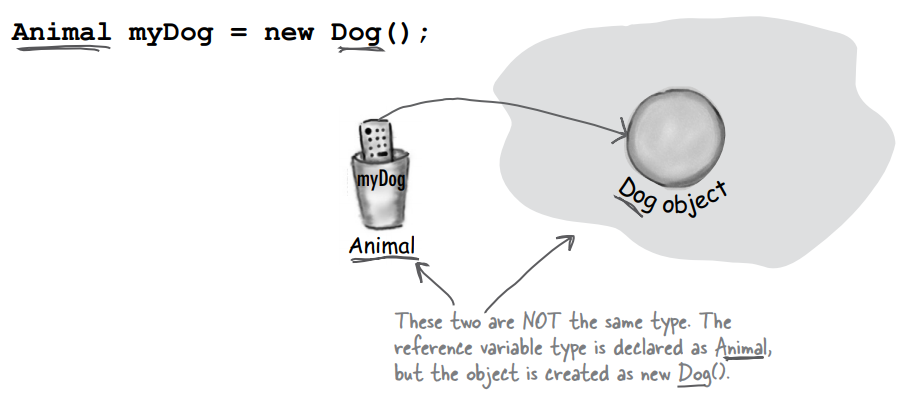
### polymorphism

Before we learn polymorphism, we know that:

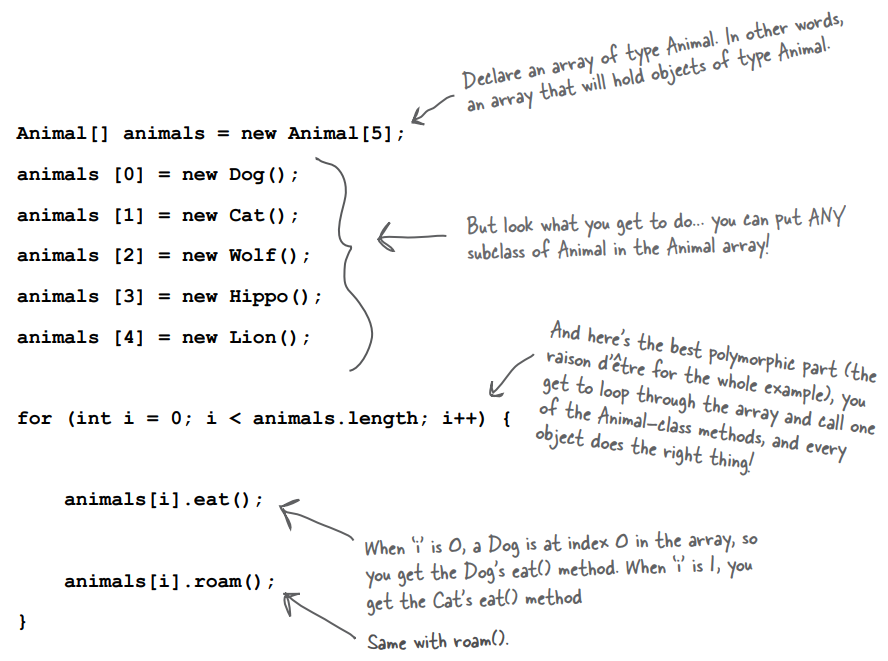
Dog myDog = new Dog();



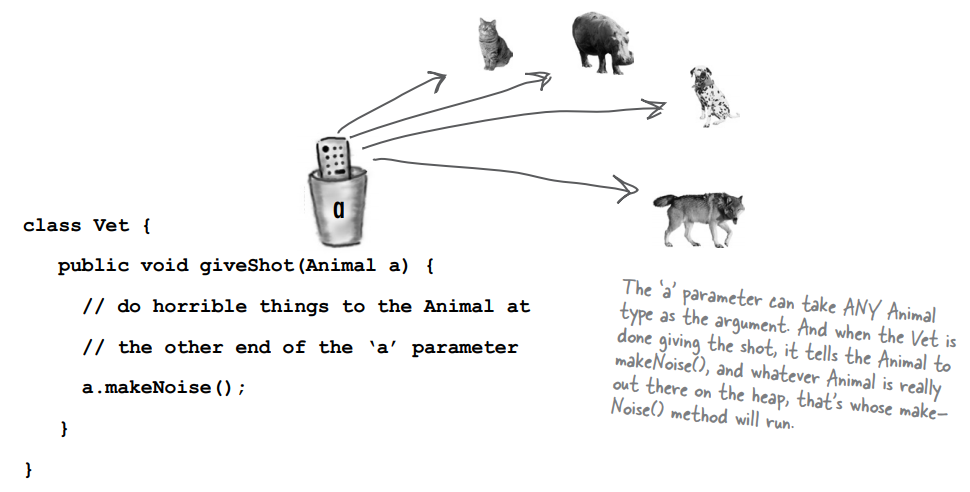
With polymorphism, the reference type can be a superclass of the actual object type.

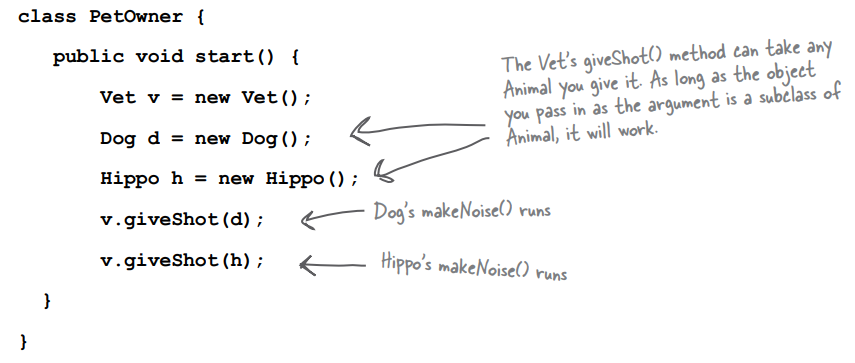


Another example of polymorphism:



————————————





Polymorphism: What you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type.

### Are there any practical limits on the levels of subclassing? How deep can you go?

If you look in the Java API, you’ll see that most inheritance hierarchies are wide but not deep. Most are no more than one or two levels deep, although there are exceptions (especially in the GUI classes). You’ll come to realize that it usually makes more sense to keep your inheritance trees shallow, but there isn’t a hard limit (well, not one that you’d ever run into).

### If you don’t have access to the source code for a class, but you want to change the way a method of that class works, could you use subclassing to do that? To extend the “bad” class and override the method with your own better code?

Yep. That’s one cool feature of OO, and sometimes it saves you from having to rewrite the class from scratch, or track down the programmer who hid the source code.

### a class cannot be subclassed

There’s no such thing as a private class, except in a very special case called an inner class, that we haven’t looked at yet.

But there are three things that can prevent a class from being subclassed.

The first is access control. Even though a class can’t be marked private, a class can be non-public (what you get if you don’t declare the class as public). A non-public class can be subclassed only by classes in the same package as the class. Classes in a different package won’t be able to subclass (or even use, for that matter) the non-public class.

The second thing that stops a class from being subclassed is the keyword modifier final. A final class means that it’s the end of the inheritance line. Nobody, ever, can extend a final class.

The third issue is that if a class has only private constructors (we’ll look at constructors in chapter 9), it can’t be subclassed.

### why making a final class?

Typically, you won’t make your classes final. But if you need security — the security of knowing that the methods will always work the way that you wrote them (because they can’t be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The String class, for example, is final because, well, imagine the havoc if somebody came along and changed the way Strings behave!

### Can you make a method final, without making the whole class final?

If you want to protect a specific method from being overridden, mark the method with the final modifier. Mark the whole class as final if you want to guarantee that none of the methods in that class will ever be overridden.

### rules for overriding

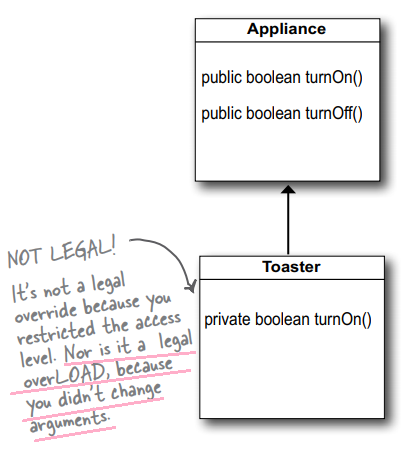
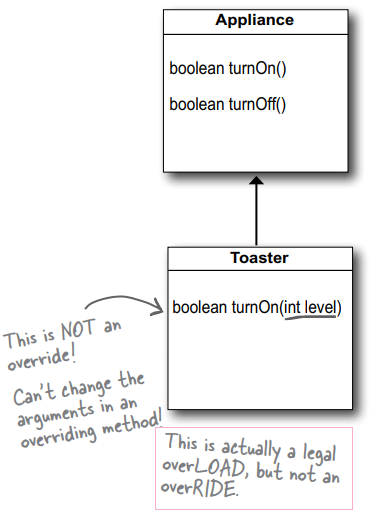
**The methods *are* the contract.**

① The arguments and return types of your overriding method must look to the outside world exactly like the overridden method in the superclass.

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

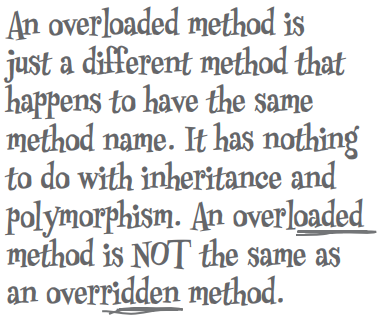
Whatever the superclass declares as a return type, the overriding method must declare either the same type, or a subclass type.

② The method can’t be less accessible. That means the access level must be the same, or friendlier. That means you can’t, for example, override a public method and make it private.



### overloading a method

Method overloading is nothing more than having two methods with the **same name** but **different argument lists**.



The return types can be different.

You can’t change ONLY the return type.

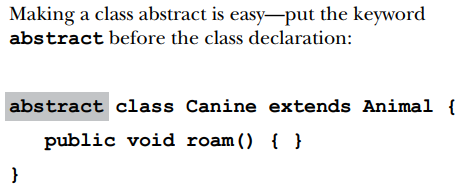
You can vary the access levels in any direction.

## Chapter 8 Interfaces and Abstract Classes

### abstract classes

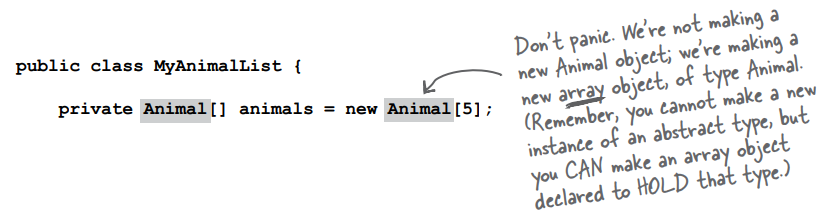
We need an Animal class, for inheritance and polymorphism. But we want programmers to instantiate only the less abstract subclasses of class Animal, not Animal itself. We want Tiger objects and Lion objects, not Animal objects.

By marking the class as abstract, the compiler will stop any code, anywhere, from ever creating an instance of that type.



An **abstract class** has virtually\* no use, no value, no purpose in life, unless it is **extended**.

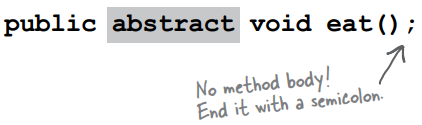
(\*There is an exception to this—an abstract class can have **static members** (see chapter 10).)



### abstract methods

An **abstract class** means the class must be **extended**; an **abstract method** means the method must be **overridden**.

An abstract method has no body!



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. If a class has even one abstract method, the class must be marked abstract.

You can mix both abstract and nonabstract methods in the abstract class.

### You MUST implement all abstract methods

When we say “you must implement the abstract method”, that means you must **provide a body**. That means you must create a non-abstract method in your class with the same method signature (name and arguments) and a return type that is compatible with the declared return type of the abstract method. What you put in that method is up to you. All Java cares about is that the method is there, in your concrete subclass.

Implementing an abstract method is just like overriding a method.

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**.

### Every class in Java extends class Object ((java.lang.Object).

Object is a non-abstract class because it’s got method implementation code that all classes can inherit and use out-of-the-box, without having to override the methods.

Sometimes you just want a generic object to use as, well, as an object. A lightweight object. By far, the most common use of an instance of type Object is for thread synchronization (which you’ll learn about in chapter 15). For now, just stick that on the back burner and assume that you will rarely make objects of type Object, even though you can.

You can override some of the methods in Object. But some of them are marked final, which means you can’t override them. You’re encouraged (strongly) to override hashCode(), equals(), and toString() in your own classes.

ArrayList<DotCom> means I was restricting the ArrayList to hold only DotCom objects. Prior to Java 5.0, ArrayLists couldn’t be restricted. They were all essentially what you get in Java 5.0 today if you write ArrayList<Object>.

### The Object class serves two main purposes.

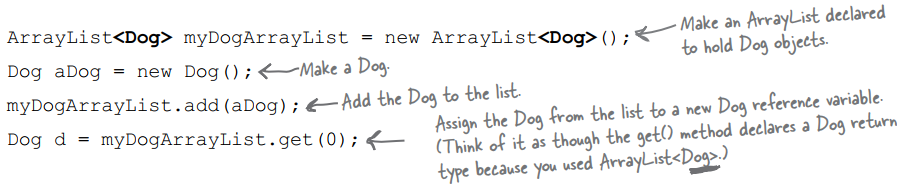
1. to act as a polymorphic type for methods that need to work on any class that you or anyone else makes,

2. 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).

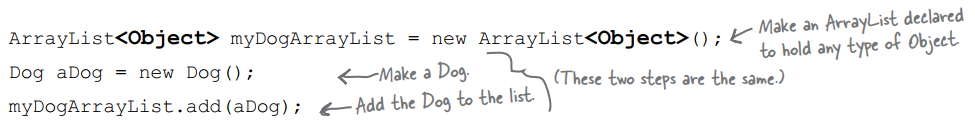
Some of the most important methods in Object are related to threads, and we’ll see those later in the book.

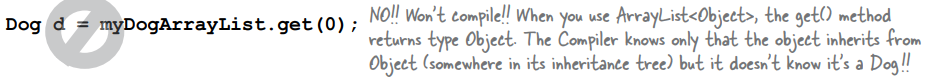
### Using polymorphic references of type Object has a price...

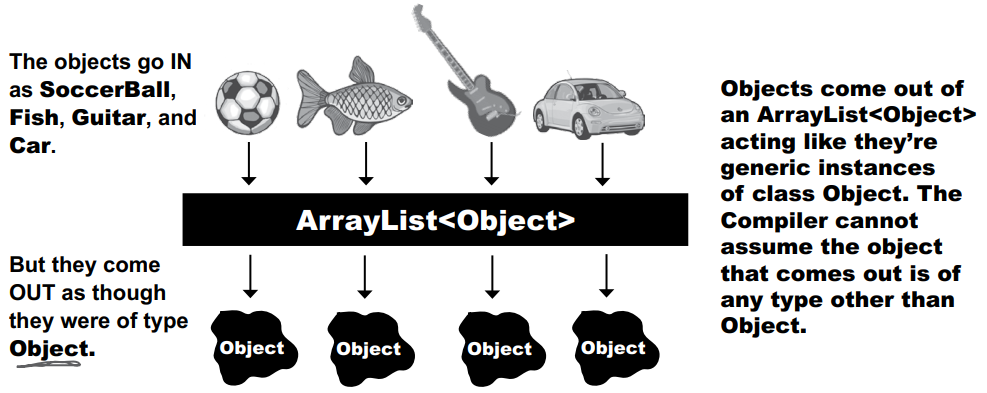
When you put an object into an ArrayList<Dog>, it goes in as a Dog, and comes out as a Dog:



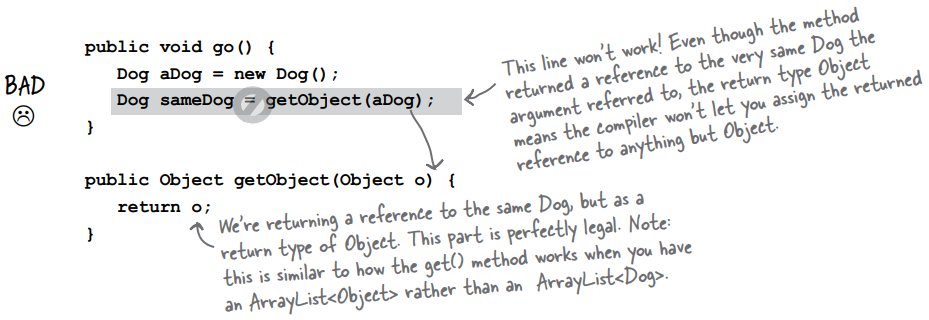
When you declare it as ArrayList<Object>, it goes in as a Dog, and comes out as an Object:

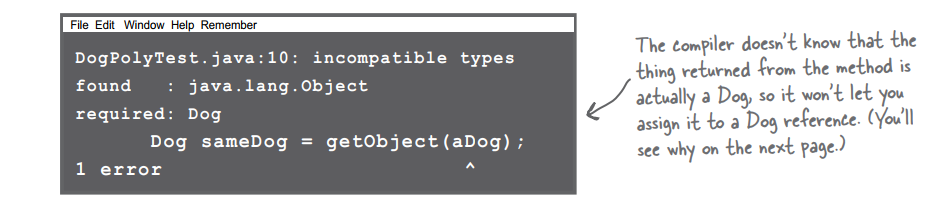


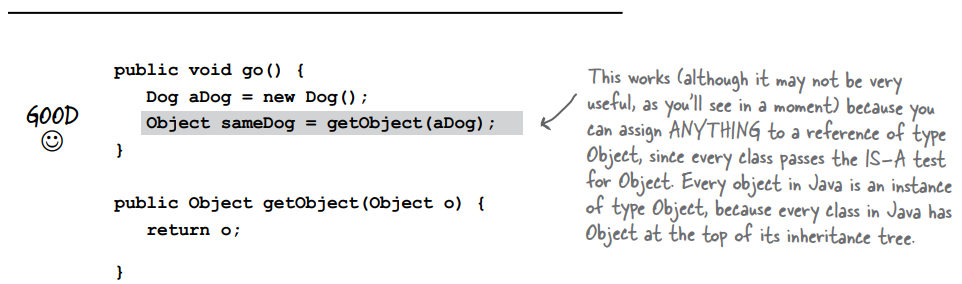




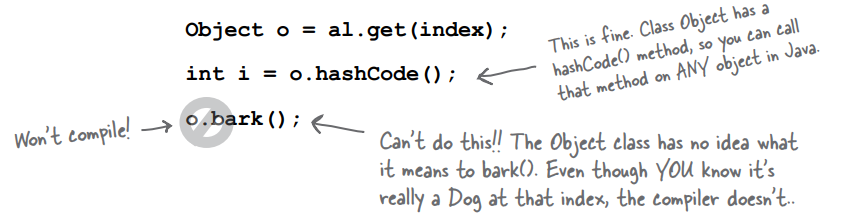
————————————



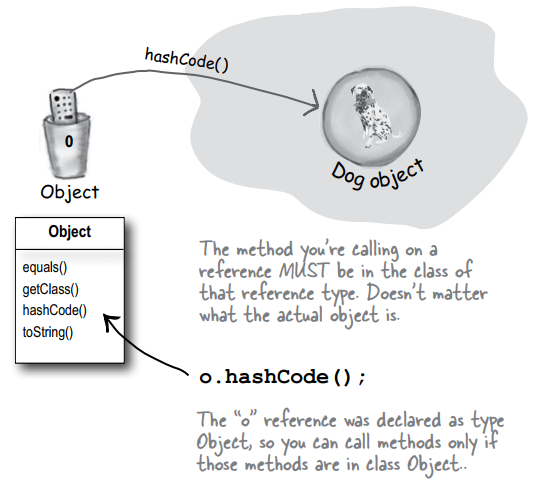




### Objects don’t bark.





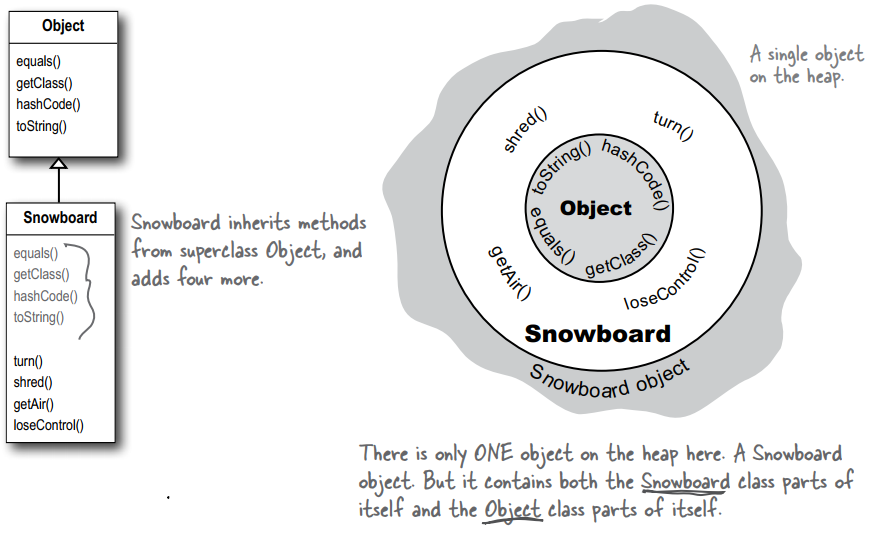


Just remember that the compiler checks the class of the reference variable, not the class of the actual object at the other end of the reference.

You can call methods on an object only if the methods are in the **class (or interface)** used as the reference variable type, regardless of the actual object type.

### objects are Objects

When you say new Snowboard(), you get a single object on the heap—a Snowboard object—but that Snowboard wraps itself around an inner core representing the Object (capital “O”) portion of itself.



### put an object in VS get a reference from an ArrayList<Object>

When you put an object in an ArrayList<Object>, you can treat it only as an Object, regardless of the type it was when you put it in.

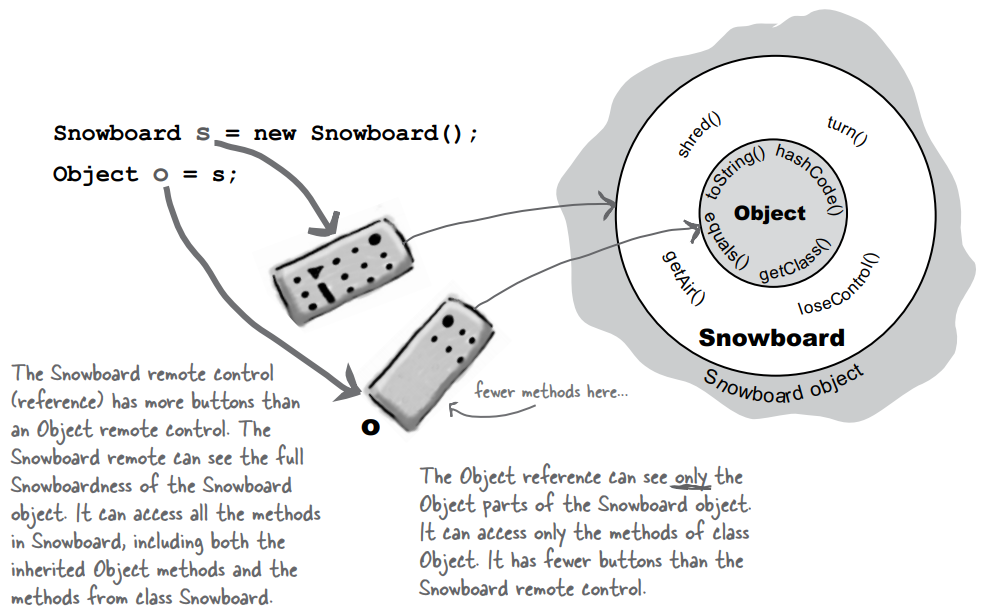
When you get a reference from an ArrayList<Object>, the reference is always of type Object. That means you get an Object remote control.

### ‘Polymorphism’ means ‘many forms’.

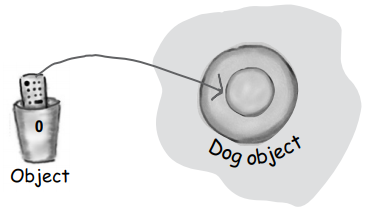
You can treat a Snowboard as a Snowboard or as an Object.

If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all the buttons from class Object, plus any new buttons (for new methods) of class Snowboard. The more specific the class, the more buttons it may have.

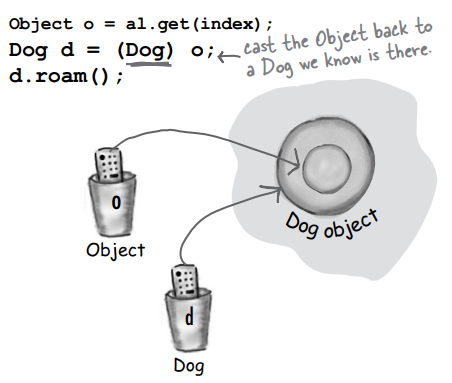
Of course, that’s not always true; a subclass might not add any new methods, but simply override the methods of its superclass. The key point is that even if the object is of type Snowboard, an Object reference to the Snowboard object can’t see the Snowboard-specific methods.



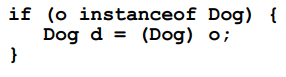
### casting objects: Casting an object reference back to its real type.



It’s really still a Dog object, but if you want to call Dog-specific methods, you need a reference declared as type Dog. If you’re sure\* the object is really a Dog, you can make a new Dog reference to it by copying the Object reference, and forcing that copy to go into a Dog reference variable, using a cast (Dog). You can use the new Dog reference to call Dog methods.



\*If you’re not sure it’s a Dog, you can use the **instanceof** operator to check. Because if you’re wrong when you do the cast, you’ll get a ClassCastException at runtime and come to a grinding halt.



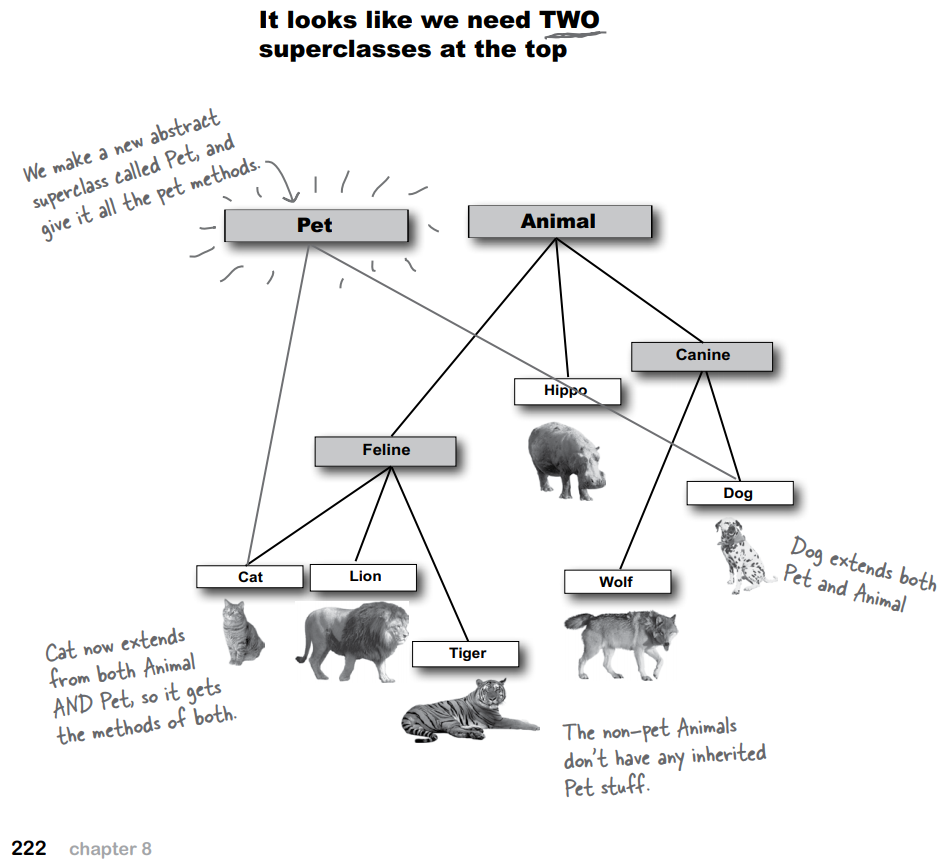
**Question:** ArrayList<Dog> gives back Dog references that don’t need to be cast, yet the ArrayList class uses Object in its methods, not Dog (or DotCom or anything else). What’s the special trick going on when you say ArrayList<Dog>?

**Answer:** In fact it is a special trick that ArrayList<Dog> gives back Dogs without you having to do any cast, since it looks like ArrayList methods don’t know anything about Dogs, or any type besides Object.

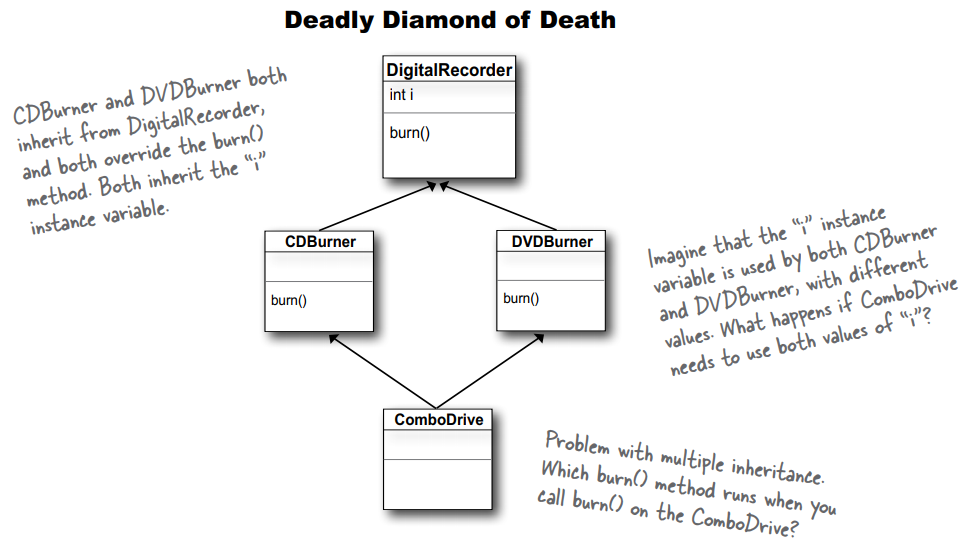
The short answer is that the compiler puts in the cast for you! When you say ArrayList<Dog>, there is no special class that has methods to take and return Dog objects, but instead the <Dog> is a signal to the compiler that you want the compiler to let you put ONLY Dog objects in and to stop you if you try to add any other type to the list. And since the compiler stops you from adding anything but Dogs to the ArrayList, the compiler also knows that it’s safe to cast anything that comes out of that ArrayList to a Dog reference. In other words, using ArrayList<Dog> saves you from having to cast the Dog you get back. But it’s much more important than that... because remember, a cast can fail at runtime, and wouldn’t you rather have your errors happen at compile time rather than, say, when your customer is using it for something critical?

But there’s a lot more to this story, and we’ll get into all the details in the Collections chapter.

### Multiple inheritance is not allowed in Java



Multiple inheritance has a problem known as The Deadly Diamond of Death, so multiple inheritance is not possible to do in Java. That means you can extend only one class (i.e. you can have only one immediate superclass).

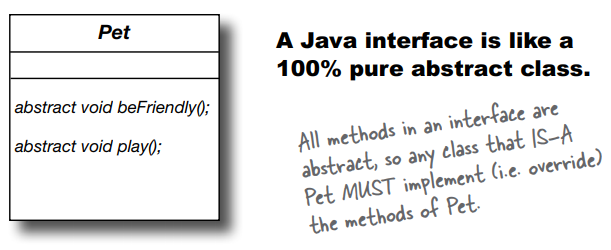


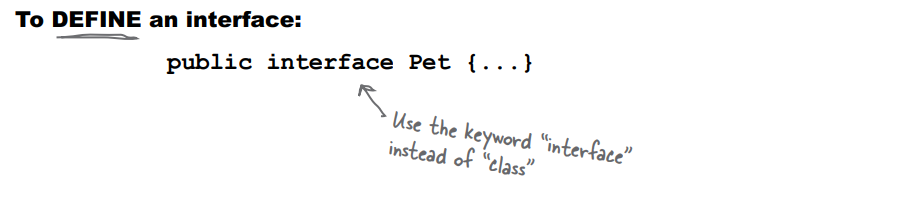
### interfaces

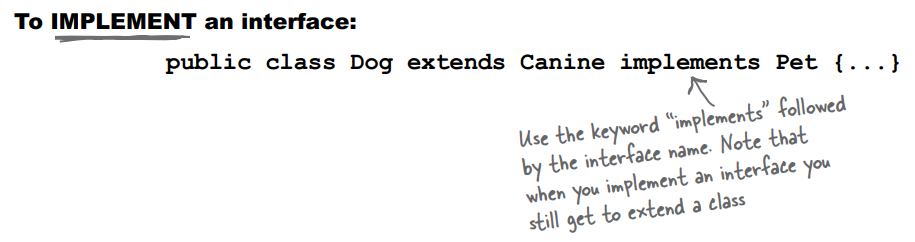
A Java interface solves your multiple inheritance problem by giving you much of the polymorphic benefits of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

An interface is like a 100% pure abstract class. It defines only abstract methods.

The way in which interfaces side-step the DDD is surprisingly simple: **make all the methods abstract**! That way, the subclass must implement the methods (remember, abstract methods must be implemented by the first concrete subclass), so at runtime the JVM isn’t confused about which of the two inherited versions it’s supposed to call.

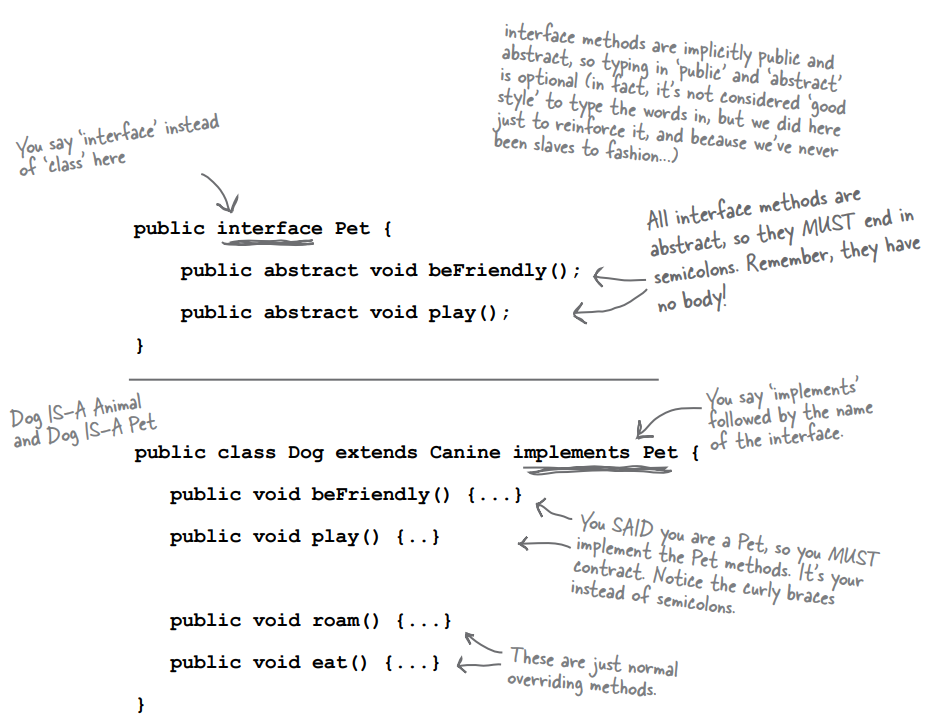




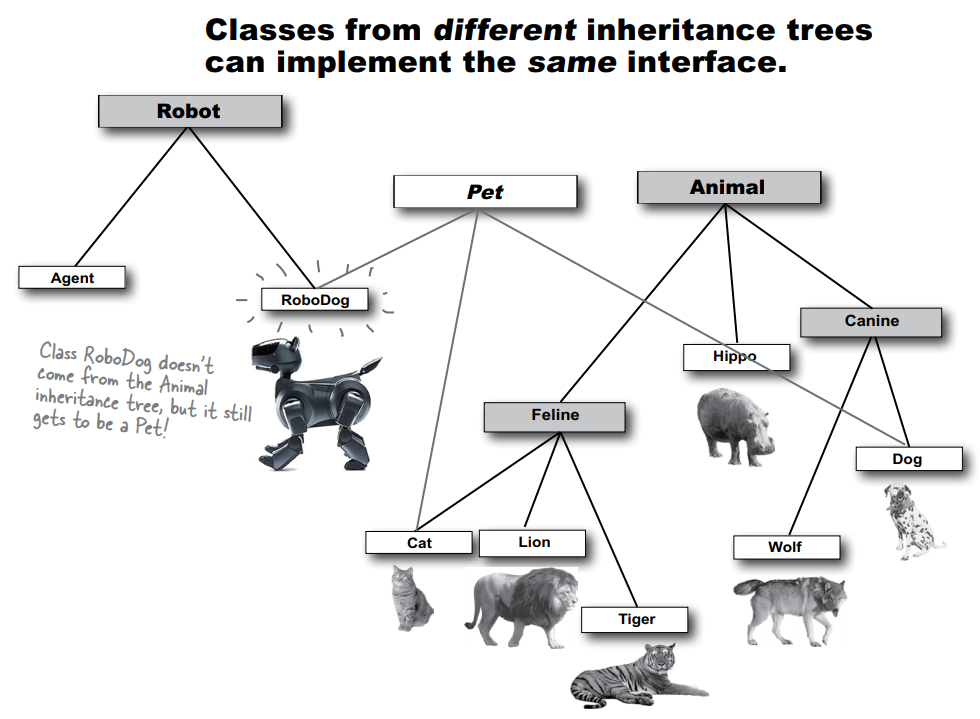


A class that implements an interface must **implement all the methods of the interface**, since all interface methods are implicitly public and abstract.

### Making and Implementing the Pet interface



### Classes from different inheritance trees can implement the same interface.



### when you use an interface as a polymorphic type

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. Do you want an object to be able to save its state to a file? Implement the Serializable interface. Do you need objects to run their methods in a separate thread of execution? Implement Runnable. You get the idea. You’ll learn more about Serializable and Runnable in later chapters, but for now, remember that classes from any place in the inheritance tree might need to implement those interfaces. Nearly any class might want to be saveable or runnable.

### a class can implement multiple interfaces

A Dog object IS-A Canine, and IS-A Animal, and IS-A Object, all through inheritance.

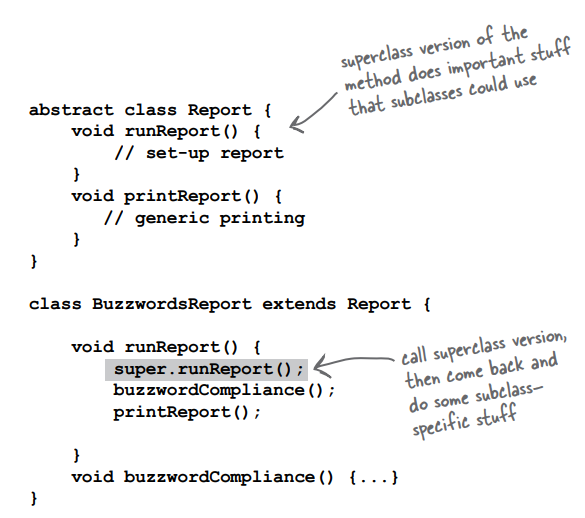
A Dog IS-A Pet through interface implementation, and the Dog might implement other interfaces as well. You could say:

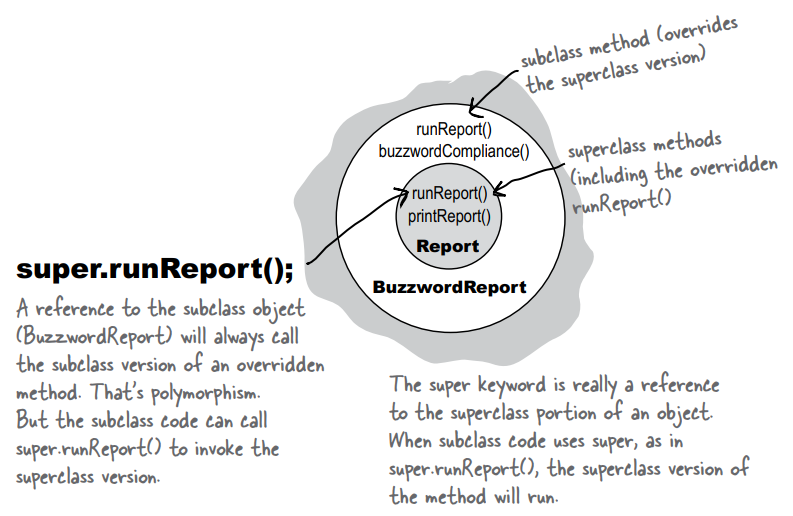


### How do you know whether to make a class, a subclass, an abstract class, or an interface?

* 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 behaviours.
* 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.

### Invoking the superclass version of a method

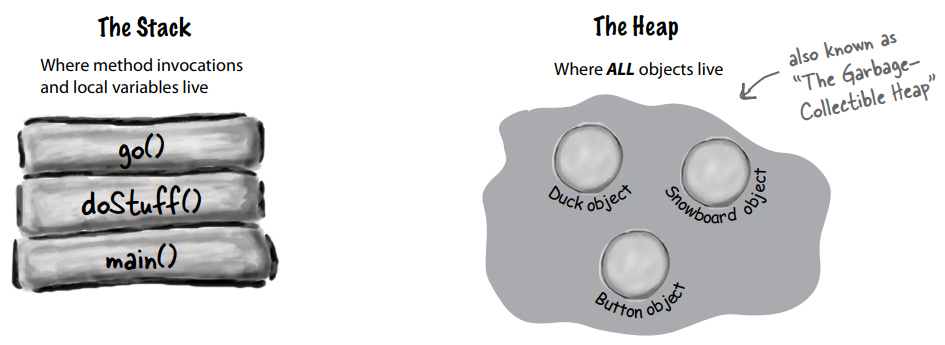




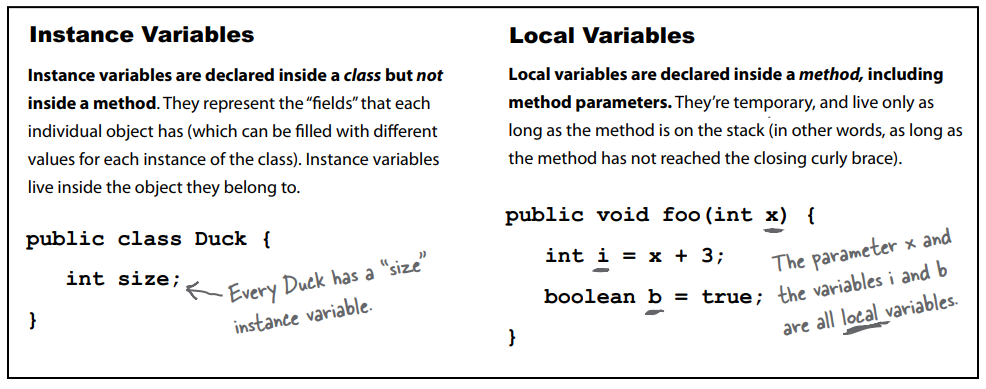
## Chapter 9 Life and Death of an Object

### the stack and the heap

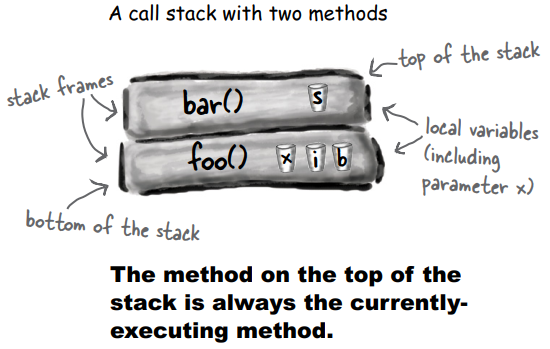
Java has two areas of memory we care about: the Stack and the Heap.



### instance variables VS local variables

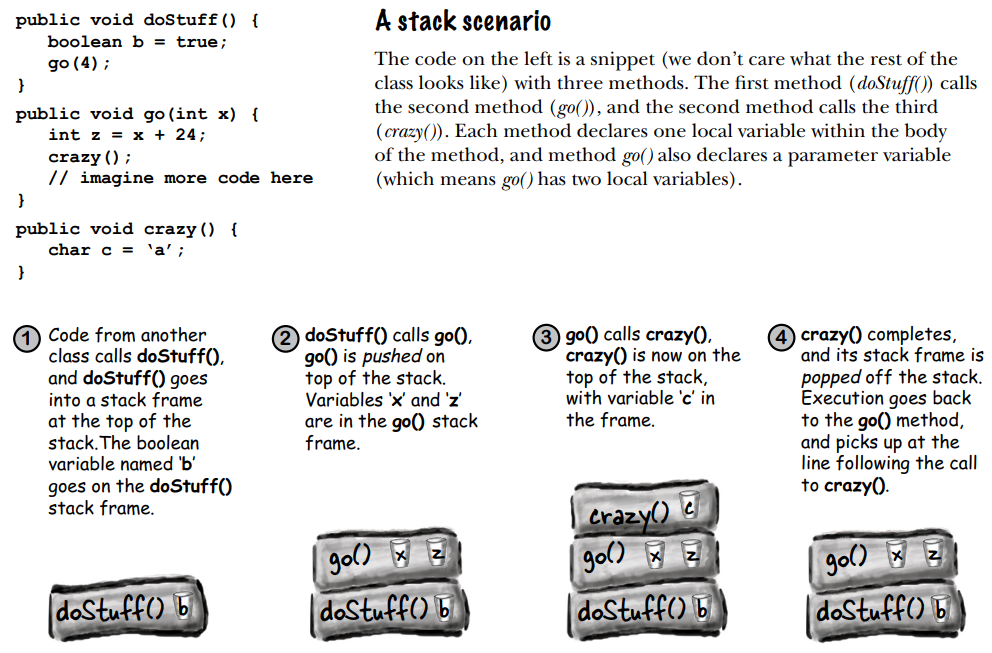


### Methods are stacked



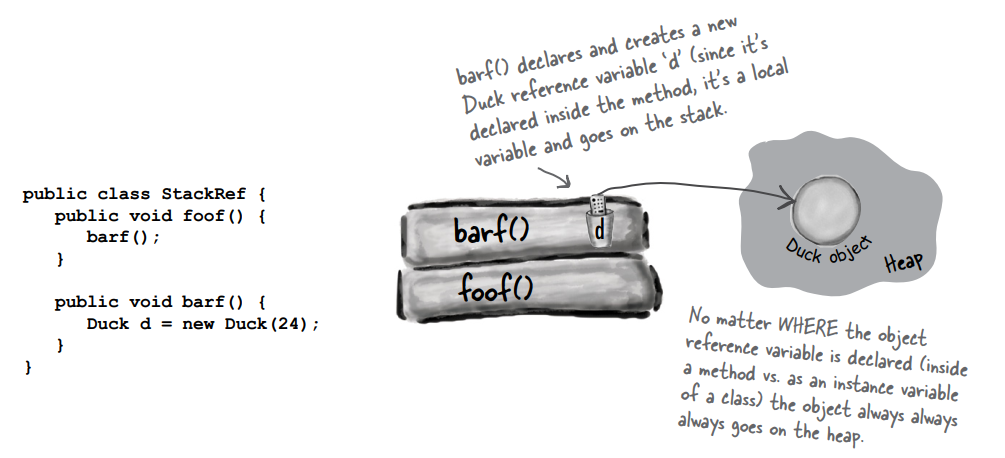
When you call a method, the method lands on the top of a call stack. That new thing that’s actually pushed onto the stack is the **stack frame**, and it holds the state of the method including which line of code is executing, and the values of all local variables.

A method stays on the stack until the method hits its closing curly brace (which means the method’s done). If method foo() calls method bar(), method bar() is stacked on top of method foo().



### object references on the stack

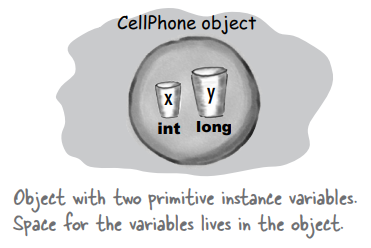
If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack. The object itself still goes in the heap.



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

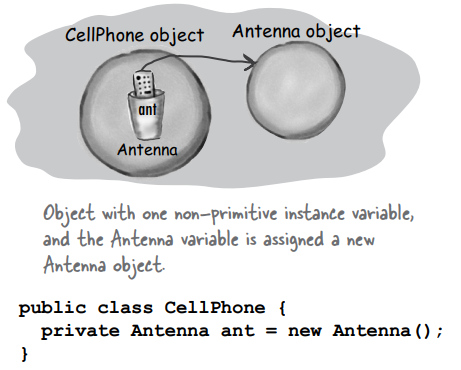
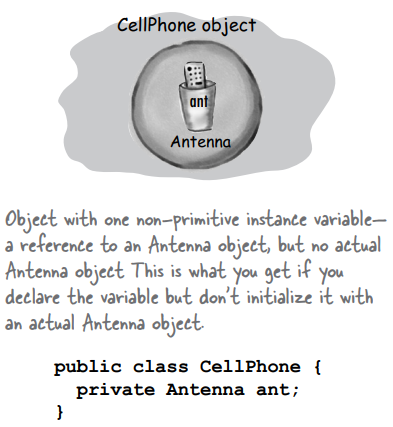
All objects live in the heap, regardless of whether the reference is a local or instance variable.

### Where do instance variables live?



If the instance variables are all primitives, Java makes space for the instance variables based on the primitive type.

If CellPhone has an instance variable declared as the non-primitive type Antenna, Java makes space within the CellPhone object only for the Antenna’s remote control (i.e. reference variable) but not the Antenna object.



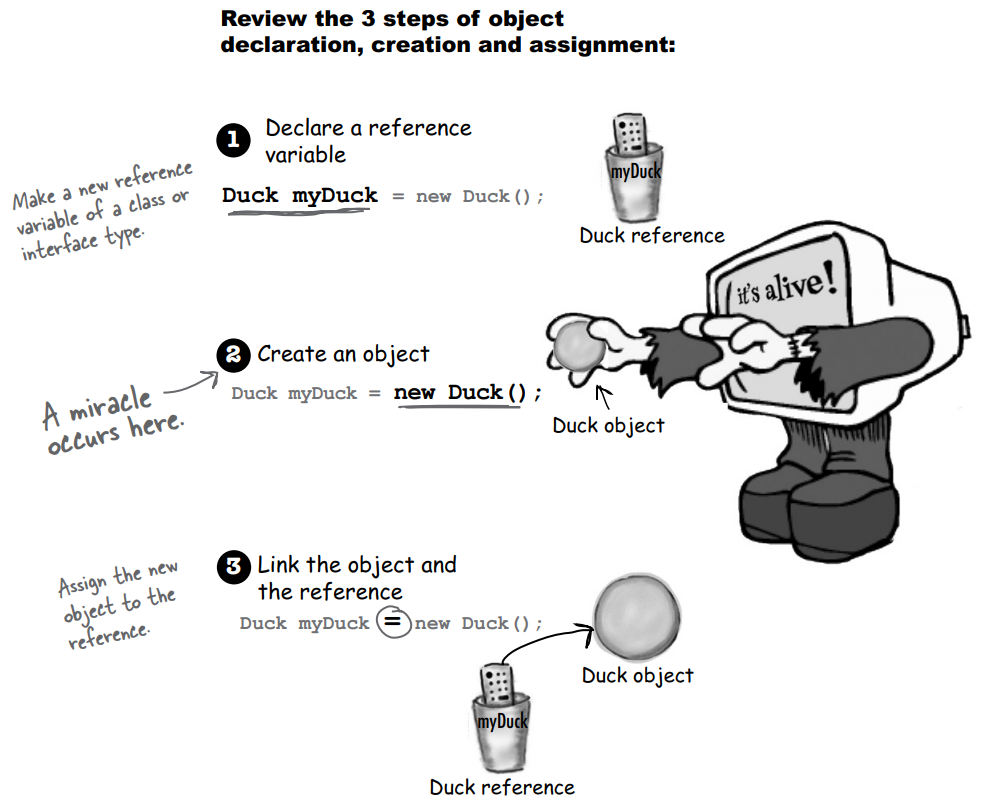
If the instance variable is declared but no object is assigned to it, then only the space for the reference variable (the remote control) is created.

**private Antenna ant;**

No actual Antenna object is made on the heap unless or until the reference variable is assigned a new Antenna object.

**private Antenna ant = new Antenna();**

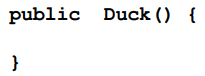
### The miracle of object creation



A constructor has the code that runs when you instantiate an object. In other words, the code that runs when you say new on a class type. Every class you create has a constructor, even if you don’t write it yourself.

The only way to invoke a constructor is with the keyword new followed by the class name. The JVM finds that class and invokes the constructor in that class. (OK, technically this isn’t the only way to invoke a constructor. But it’s the only way to do it from outside a constructor. You can call a constructor from within another constructor, with restrictions, but we’ll get into all that later in the chapter.)

You can write a constructor for your class (we’re about to do that), but if you don’t, the compiler writes one for you! Here’s what the compiler’s default constructor looks like:



How is this different from a method?

