

9 constructors and garbage collection

Life and Death of an Object



Objects are born and objects die. You're in charge of an object's lifecycle. You decide when and how to **construct** it. You decide when to **destroy** it. Except you don't actually *destroy* the object yourself, you simply *abandon* it. But once it's abandoned, the heartless **Garbage Collector (gc)** can vaporize it, reclaiming the memory that object was using. If you're gonna write Java, you're gonna create objects. Sooner or later, you're gonna have to let some of them go, or risk running out of RAM. In this chapter we look at how objects are created, where they live while they're alive, and how to keep or abandon them efficiently. That means we'll talk about the heap, the stack, scope, constructors, super constructors, null references, and more. Warning: this chapter contains material about object death that some may find disturbing. Best not to get too attached.

The Stack and the Heap: where things live

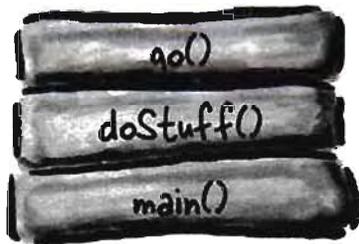
Before we can understand what really happens when you create an object, we have to step back a bit. We need to learn more about where everything lives (and for how long) in Java. That means we need to learn more about the Stack and the Heap. In Java, we (programmers) care about two areas of memory—the one where objects live (the heap), and the one where method invocations and local variables live (the stack). When a JVM starts up, it gets a chunk of memory from the underlying OS, and uses it to run your Java program. How *much* memory, and whether or not you can tweak it, is dependent on which version of the JVM (and on which platform) you're

running. But usually you *won't* have anything to say about it. And with good programming, you probably won't care (more on that a little later).

We know that all *objects* live on the garbage-collectible heap, but we haven't yet looked at where *variables* live. And where a variable lives depends on what *kind* of variable it is. And by "kind", we don't mean *type* (i.e. primitive or object reference). The two *kinds* of variables whose lives we care about now are *instance variables* and *local variables*. Local variables are also known as *stack variables*, which is a big clue for where they live.

The Stack

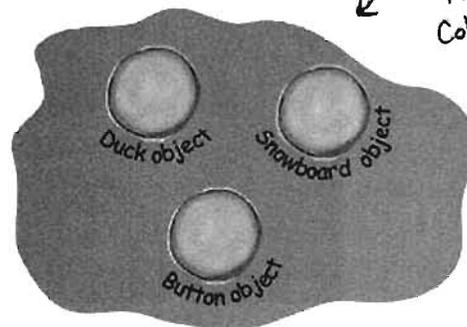
Where method invocations and local variables live



The Heap

Where **ALL** objects live

also known as "The Garbage-Collectible Heap"



Instance Variables

Instance variables are declared inside a *class* but *not* inside a *method*. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {  
    int size;  
}
```

← Every Duck has a "size" instance variable.

Local Variables

Local variables are declared inside a *method*, including *method parameters*. They're temporary, and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

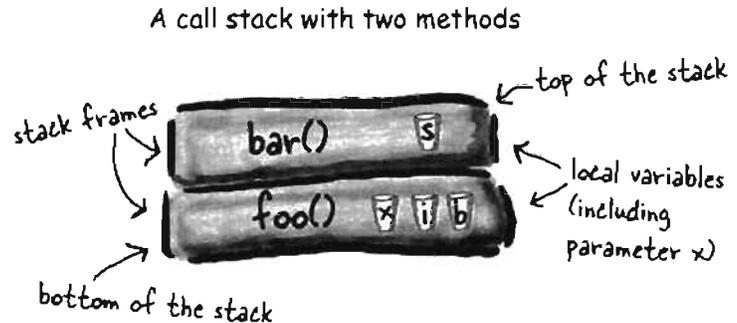
```
public void foo(int x) {  
    int i = x + 3;  
    boolean b = true;  
}
```

The parameter *x* and the variables *i* and *b* are all 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.

The method at the *top* of the stack is always the currently-running method for that stack (for now, assume there's only one stack, but in chapter 14 we'll add more.) 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()*.



The method on the top of the stack is always the currently-executing method.

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

public void crazy() {
    char c = 'a';
}
```

A stack scenario

The code on the left is a snippet (we don't care what the rest of the class looks like) with three methods. The first method (*doStuff()*) calls the second method (*go()*), and the second method calls the third (*crazy()*). Each method declares one local variable within the body of the method, and method *go()* also declares a parameter variable (which means *go()* has two local variables).

1 Code from another class calls *doStuff()*, and *doStuff()* goes into a stack frame at the top of the stack. The boolean variable named 'b' goes on the *doStuff()* stack frame.



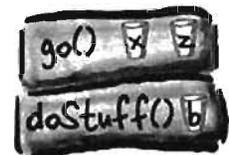
2 *doStuff()* calls *go()*, *go()* is pushed on top of the stack. Variables 'x' and 'z' are in the *go()* stack frame.



3 *go()* calls *crazy()*, *crazy()* is now on the top of the stack, with variable 'c' in the frame.



4 *crazy()* completes, and its stack frame is popped off the stack. Execution goes back to the *go()* method, and picks up at the line following the call to *crazy()*.

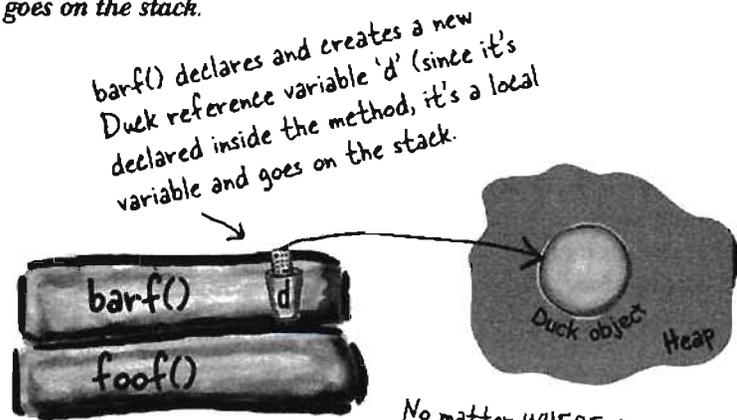


What about local variables that are objects?

Remember, a non-primitive variable holds a *reference* to an object, not the object itself. You already know where objects live—on the heap. It doesn't matter where they're declared or created. *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.

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck(24);  
    }  
}
```



there are no Dumb Questions

Q: One more time, WHY are we learning the whole stack/heap thing? How does this help me? Do I really need to learn about it?

A: Knowing the fundamentals of the Java Stack and Heap is crucial if you want to understand variable scope, object creation issues, memory management, threads, and exception handling. We cover threads and exception handling in later chapters but the others you'll learn in this one. You do not need to know anything about *how* the Stack and Heap are implemented in any particular JVM and/or platform. Everything you need to know about the Stack and Heap is on this page and the previous one. If you nail these pages, all the other topics that depend on your knowing this stuff will go much, much, much easier. Once again, some day you will SO thank us for shoving Stacks and Heaps down your throat.

BULLET POINTS

- ▶ Java has two areas of memory we care about: the Stack and the Heap.
- ▶ Instance variables are variables declared inside a class but outside any method.
- ▶ Local variables are variables declared inside a method or method parameter.
- ▶ 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.

If local variables live on the stack, where do instance variables live?

When you say `new CellPhone()`, Java has to make space on the Heap for that `CellPhone`. But how *much* space? Enough for the object, which means enough to house all of the object's instance variables. That's right, instance variables live on the Heap, inside the object they belong to.

Remember that the *values* of an object's instance variables live inside the object. If the instance variables are all primitives, Java makes space for the instance variables based on the primitive type. An `int` needs 32 bits, a `long` 64 bits, etc. Java doesn't care about the value inside primitive variables; the bit-size of an `int` variable is the same (32 bits) whether the value of the `int` is 32,000,000 or 32.

But what if the instance variables are *objects*? What if `CellPhone` HAS-A `Antenna`? In other words, `CellPhone` has a reference variable of type `Antenna`.

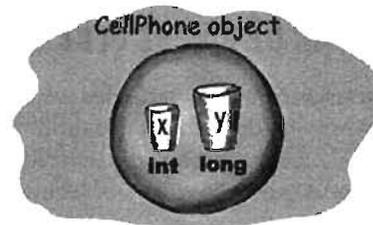
When the new object has instance variables that are object references rather than primitives, the real question is: does the object need space for all of the objects it holds references to? The answer is, *not exactly*. No matter what, Java has to make space for the instance variable *values*. But remember that a reference variable value is not the whole *object*, but merely a *remote control* to the object. So 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*.

Well then when does the `Antenna` *object* get space on the Heap? First we have to find out *when* the `Antenna` object itself is created. That depends on the instance variable declaration. 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.

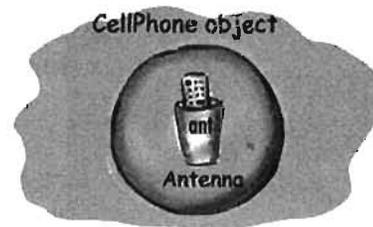
```
private Antenna ant;
```

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();
```

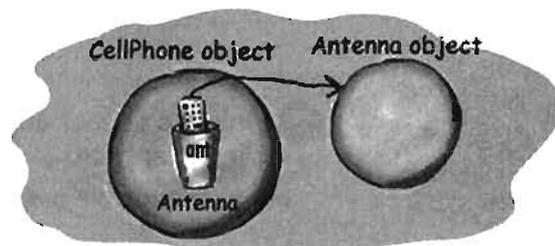


Object with two primitive instance variables. Space for the variables lives in the object.



Object with one non-primitive instance variable—a reference to an `Antenna` object, but no actual `Antenna` object. This is what you get if you declare the variable but don't initialize it with an actual `Antenna` object.

```
public class CellPhone {
    private Antenna ant;
}
```



Object with one non-primitive instance variable, and the `Antenna` variable is assigned a new `Antenna` object.

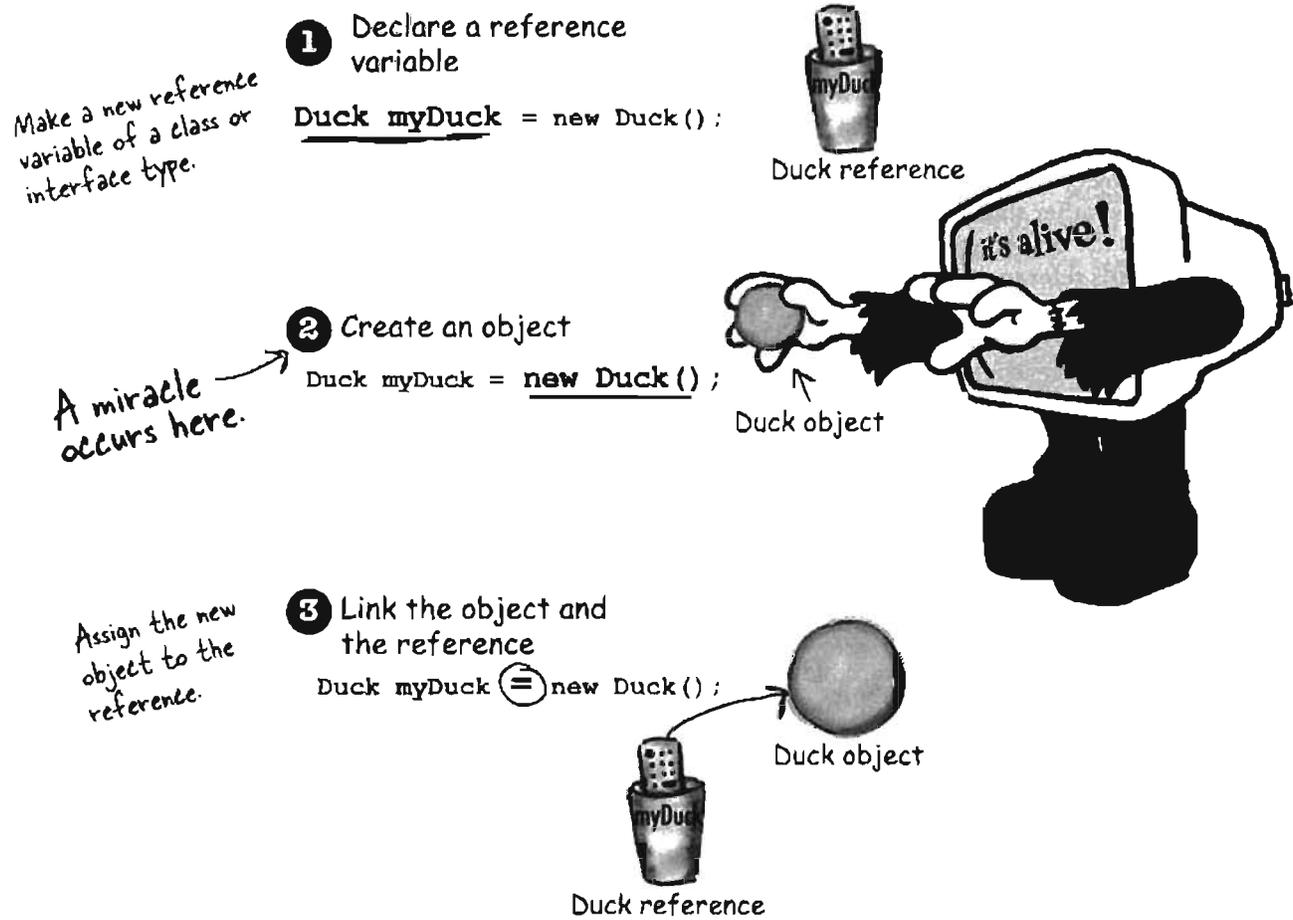
```
public class CellPhone {
    private Antenna ant = new Antenna();
}
```

The miracle of object creation

Now that you know where variables and objects live, we can dive into the mysterious world of object creation. Remember the three steps of object declaration and assignment: declare a reference variable, create an object, and assign the object to the reference.

But until now, step two—where a miracle occurs and the new object is “born”—has remained a Big Mystery. Prepare to learn the facts of object life. *Hope you're not squeamish.*

Review the 3 steps of object declaration, creation and assignment:



Are we calling a method named `Duck()`? Because it sure *looks* like it.

```
Duck myDuck = new Duck();
```

It looks like we're calling a method named `Duck()`, because of the parentheses.

No.

We're calling the *Duck constructor*.

A constructor *does* look and feel a lot like a method, but it's not a method. It's got the code that runs when you say **new**. In other words, *the code that runs when you instantiate an object*.

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.)

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.

But where is the constructor?

If we didn't write it, who did?

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:

```
public Duck() {
}

```

Notice something missing? How is this different from a method?

Where's the return type? If this were a method, you'd need a return type between "public" and "Duck()".

```
public Duck() {
    // constructor code goes here
}

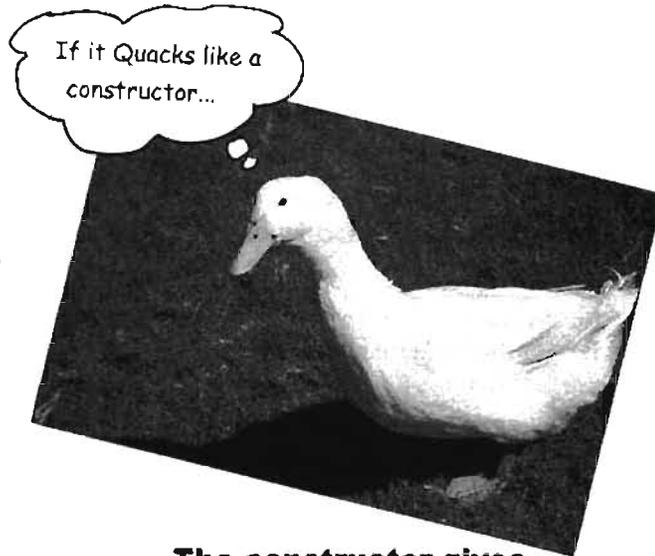
```

Its name is the same as the class name. That's mandatory.

constructing a new Duck

Construct a Duck

The key feature of a constructor is that it runs *before* the object can be assigned to a reference. That means you get a chance to step in and do things to get the object ready for use. In other words, before anyone can use the remote control for an object, the object has a chance to help construct itself. In our Duck constructor, we're not doing anything useful, but it still demonstrates the sequence of events.



```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

← Constructor code.

The constructor gives you a chance to step into the middle of new.

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
    }  
}
```

← This calls the Duck constructor.



Sharpen your pencil

A constructor lets you jump into the middle of the object creation step—into the middle of *new*. Can you imagine conditions where that would be useful? Which of these might be useful in a Car class constructor, if the Car is part of a Racing Game? Check off the ones that you came up with a scenario for.

- Increment a counter to track how many objects of this class type have been made.
- Assign runtime-specific state (data about what's happening NOW).
- Assign values to the object's important instance variables.
- Get and save a reference to the object that's *creating* the new object.
- Add the object to an ArrayList.
- Create HAS-A objects.
- _____ (your idea here)

Initializing the state of a new Duck

Most people use constructors to initialize the state of an object. In other words, to make and assign values to the object's instance variables.

```
public Duck() {
    size = 34;
}
```

That's all well and good when the Duck class *developer* knows how big the Duck object should be. But what if we want the programmer who is *using* Duck to decide how big a particular Duck should be?

Imagine the Duck has a size instance variable, and you want the programmer using your Duck class to set the size of the new Duck. How could you do it?

Well, you could add a setSize() setter method to the class. But that leaves the Duck temporarily without a size*, and forces the Duck user to write *two* statements—one to create the Duck, and one to call the setSize() method. The code below uses a setter method to set the initial size of the new Duck.

```
public class Duck {
    int size; ← instance variable

    public Duck() {
        System.out.println("Quack"); ← constructor
    }

    public void setSize(int newSize) { ← setter method
        size = newSize;
    }
}
```

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck();
        d.setSize(42);
    }
}
```

There's a bad thing here. The Duck is alive at this point in the code, but without a size! * And then you're relying on the Duck-user to KNOW that Duck creation is a two-part process: one to call the constructor and one to call the setter.

*Instance variables do have a default value. 0 or 0.0 for numeric primitives, false for booleans, and null for references.

there are no Dumb Questions

Q: Why do you need to write a constructor if the compiler writes one for you?

A: If you need code to help initialize your object and get it ready for use, you'll have to write your own constructor. You might, for example, be dependent on input from the user before you can finish making the object ready. There's another reason you might have to write a constructor, even if you don't need any constructor code yourself. It has to do with your superclass constructor, and we'll talk about that in a few minutes.

Q: How can you tell a constructor from a method? Can you also have a method that's the same name as the class?

A: Java lets you declare a method with the same name as your class. That doesn't make it a constructor, though. The thing that separates a method from a constructor is the return type. Methods *must* have a return type, but constructors *cannot* have a return type.

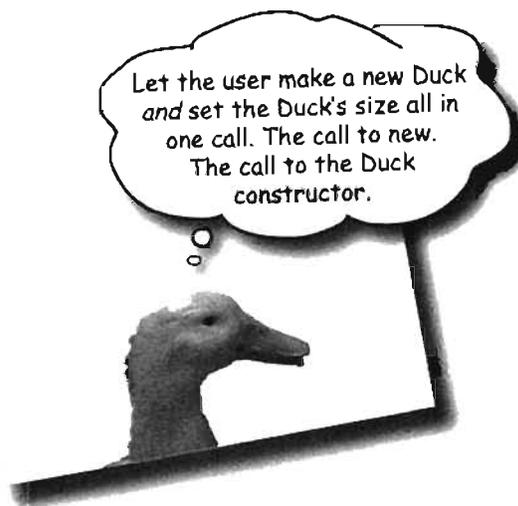
Q: Are constructors inherited? If you don't provide a constructor but your superclass does, do you get the superclass constructor instead of the default?

A: Nope. Constructors are not inherited. We'll look at that in just a few pages.

Using the constructor to initialize important Duck state*

If an object shouldn't be used until one or more parts of its state (instance variables) have been initialized, don't let anyone get ahold of a Duck object until you're finished initializing! It's usually way too risky to let someone make—and get a reference to—a new Duck object that isn't quite ready for use until that someone turns around and calls the `setSize()` method. How will the Duck-user even *know* that he's required to call the setter method after making the new Duck?

The best place to put initialization code is in the constructor. And all you need to do is make a constructor with arguments.



```
public class Duck {
    int size;

    public Duck(int duckSize) {
        System.out.println("Quack");

        size = duckSize;

        System.out.println("size is " + size);
    }
}
```

↙ Add an int parameter to the Duck constructor.

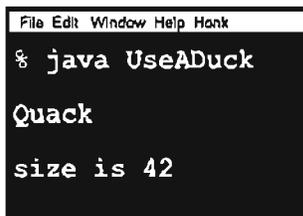
↙ Use the argument value to set the size instance variable.

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck (42);
    }
}
```

↗ This time there's only one statement. We make the new Duck and set its size in one statement

↙ Pass a value to the constructor.



*Not to imply that not all Duck state is not unimportant.

Make it easy to make a Duck

Be sure you have a no-arg constructor

What happens if the Duck constructor takes an argument? Think about it. On the previous page, there's only *one* Duck constructor—and it takes an `int` argument for the *size* of the Duck. That might not be a big problem, but it does make it harder for a programmer to create a new Duck object, especially if the programmer doesn't *know* what the size of a Duck should be. Wouldn't it be helpful to have a default size for a Duck, so that if the user doesn't know an appropriate size, he can still make a Duck that works?

Imagine that you want Duck users to have *TWO* options for making a Duck—one where they supply the Duck size (as the constructor argument) and one where they don't specify a size and thus get your default Duck size.

You can't do this cleanly with just a single constructor. Remember, if a method (or constructor—same rules) has a parameter, you *must* pass an appropriate argument when you invoke that method or constructor. You can't just say, "If someone doesn't pass anything to the constructor, then use the default size", because they won't even be able to compile without sending an `int` argument to the constructor call. You *could* do something clunkily like this:

```
public class Duck {
    int size;

    public Duck(int newSize) {
        if (newSize == 0) {
            size = 27;
        } else {
            size = newSize;
        }
    }
}
```

If the parameter value is zero, give the new Duck a default size, otherwise use the parameter value for the size. NOT a very good solution.

But that means the programmer making a new Duck object has to *know* that passing a "0" is the protocol for getting the default Duck size. Pretty ugly. What if the other programmer doesn't know that? Or what if he really *does* want a zero-size Duck? (Assuming a zero-sized Duck is allowed. If you don't want zero-sized Duck objects, put validation code in the constructor to prevent it.) The point is, it might not always be possible to distinguish between a genuine "I want zero for the size" constructor argument and a "I'm sending zero so you'll give me the default size, whatever that is" constructor argument.

You really want *TWO* ways to make a new Duck:

```
public class Duck2 {
    int size;

    public Duck2() {
        // supply default size
        size = 27;
    }

    public Duck2(int duckSize) {
        // use duckSize parameter
        size = duckSize;
    }
}
```

To make a Duck when you know the size:

```
Duck2 d = new Duck2(15);
```

To make a Duck when you do *not* know the size:

```
Duck2 d2 = new Duck2();
```

So this two-opts-to-make-a-Duck idea needs two constructors. One that takes an `int` and one that doesn't. **If you have more than one constructor in a class, it means you have *overloaded* constructors.**

Doesn't the compiler always make a no-arg constructor for you? No!

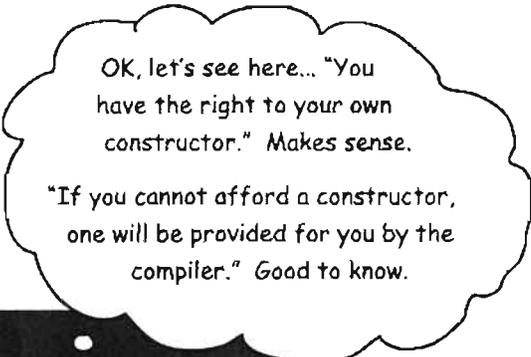
You might think that if you write *only* a constructor with arguments, the compiler will see that you don't have a no-arg constructor, and stick one in for you. But that's not how it works. The compiler gets involved with constructor-making *only if you don't say anything at all about constructors.*

If you write a constructor that takes arguments, and you *still* want a no-arg constructor, you'll have to build the no-arg constructor yourself!

As soon as you provide a constructor, ANY kind of constructor, the compiler backs off and says, "OK Buddy, looks like you're in charge of constructors now."

If you have more than one constructor in a class, the constructors MUST have different argument lists.

The argument list includes the order and types of the arguments. As long as they're different, you can have more than one constructor. You can do this with methods as well, but we'll get to that in another chapter.



Overloaded constructors means you have more than one constructor in your class.

To compile, each constructor must have a *different* argument list!

The class below is legal because all four constructors have different argument lists. If you had two constructors that took only an int, for example, the class wouldn't compile. What you name the parameter variable doesn't count. It's the variable *type* (int, Dog, etc.) and *order* that matters. You *can* have two constructors that have identical types, *as long as the order is different*. A constructor that takes a String followed by an int, is *not* the same as one that takes an int followed by a String.

Four different constructors means four different ways to make a new mushroom.



```
public class Mushroom {
```

```
    public Mushroom(int size) { }
```

```
    public Mushroom() { }
```

```
    public Mushroom(boolean isMagic) { }
```

```
    public Mushroom(boolean isMagic, int size) { }
```

```
    public Mushroom(int size, boolean isMagic) { }
```

```
}
```

these two have the same args, but in different order, so it's OK }

when you know the size, but you don't know if it's magic

when you don't know anything

when you know if it's magic or not, but don't know the size

when you know whether or not it's magic, AND you know the size as well

BULLET POINTS

- ▶ 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.
- ▶ You can use a constructor to initialize the state (i.e. the instance variables) of the object being constructed.
- ▶ If you don't put a constructor in your class, the compiler will put in a default constructor.
- ▶ The default constructor is always a no-arg constructor.
- ▶ If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.
- ▶ If you want a no-arg constructor, and you've already put in a constructor with arguments, you'll have to build the no-arg constructor yourself.
- ▶ Always provide a no-arg constructor if you can, to make it easy for programmers to make a working object. Supply default values.
- ▶ Overloaded constructors means you have more than one constructor in your class.
- ▶ Overloaded constructors must have different argument lists.
- ▶ You cannot have two constructors with the same argument lists. An argument list includes the order and/or type of arguments.
- ▶ Instance variables are assigned a default value, even when you don't explicitly assign one. The default values are 0/0.0/false for primitives, and null for references.

overloaded constructors



Match the new Duck() call with the constructor that runs when that Duck is instantiated. We did the easy one to get you started.

```
public class TestDuck {  
    public static void main(String[] args){  
        int weight = 8;  
        float density = 2.3F;  
        String name = "Donald";  
        long[] feathers = {1,2,3,4,5,6};  
        boolean canFly = true;  
        int airspeed = 22;  
  
        Duck[] d = new Duck[7];  
        d[0] = new Duck();  
        d[1] = new Duck(density, weight);  
        d[2] = new Duck(name, feathers);  
        d[3] = new Duck(canFly);  
        d[4] = new Duck(3.3F, airspeed);  
        d[5] = new Duck(false);  
        d[6] = new Duck(airspeed, density);  
    }  
}
```

```
class Duck {  
    int pounds = 6;  
    float floatability = 2.1F;  
    String name = "Generic";  
    long[] feathers = {1,2,3,4,5,6,7};  
    boolean canFly = true;  
    int maxSpeed = 25;  
  
    public Duck() {  
        System.out.println("type 1 duck");  
    }  
  
    public Duck(boolean fly) {  
        canFly = fly;  
        System.out.println("type 2 duck");  
    }  
  
    public Duck(String n, long[] f) {  
        name = n;  
        feathers = f;  
        System.out.println("type 3 duck");  
    }  
  
    public Duck(int w, float f) {  
        pounds = w;  
        floatability = f;  
        System.out.println("type 4 duck");  
    }  
  
    public Duck(float density, int max) {  
        floatability = density;  
        maxSpeed = max;  
        System.out.println("type 5 duck");  
    }  
}
```

Q: Earlier you said that it's good to have a no-argument constructor so that if people call the no-arg constructor, we can supply default values for the "missing" arguments. But aren't there times when it's impossible to come up with defaults? Are there times when you should not have a no-arg constructor in your class?

A: You're right. There are times when a no-arg constructor doesn't make sense. You'll see this in the Java API—some classes don't have a no-arg constructor. The Color class, for example, represents a... color. Color objects are used to, for example, set or change the color of a screen font or GUI button. When you make a Color instance, that instance is of a particular color (you know, Death-by-Chocolate Brown, Blue-Screen-of-Death Blue, Scandalous Red, etc.). If you make a Color object, you must specify the color in some way.

```
Color c = new Color(3,45,200);
```

(We're using three ints for RGB values here. We'll get into using Color later, in the Swing chapters.) Otherwise, what would you get? The Java API programmers *could* have decided that if you call a no-arg Color constructor you'll get a lovely shade of mauve. But good taste prevailed. If you try to make a Color without supplying an argument:

```
Color c = new Color();
```

The compiler freaks out because it can't find a matching no-arg constructor in the Color class.

```
File Edit Window Help StopBeingStupid  
cannot resolve symbol  
:constructor Color()  
location: class  
java.awt.Color  
Color c = new Color();  
^  
1 error
```

Nanoreview: four things to remember about constructors

- A constructor is the code that runs when somebody says `new` on a class type

```
Duck d = new Duck() ;
```

- A constructor must have the same name as the class, and no return type

```
public Duck(int size) { }
```

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

```
public Duck() { }
```

- You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```

Doing all the Brain Barbells has been shown to produce a 42% increase in neuron size. And you know what they say, "Big neurons..."



What about superclasses?

When you make a Dog, should the Canine constructor run too?

If the superclass is abstract, should it even *have* a constructor?

We'll look at this on the next few pages, so stop now and think about the implications of constructors and superclasses.

^{there are no}
Dumb Questions

Q: Do constructors have to be public?

A: No. Constructors can be `public`, `private`, or `default` (which means no access modifier at all). We'll look more at `default` access in chapter 16 and appendix B.

Q: How could a `private` constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

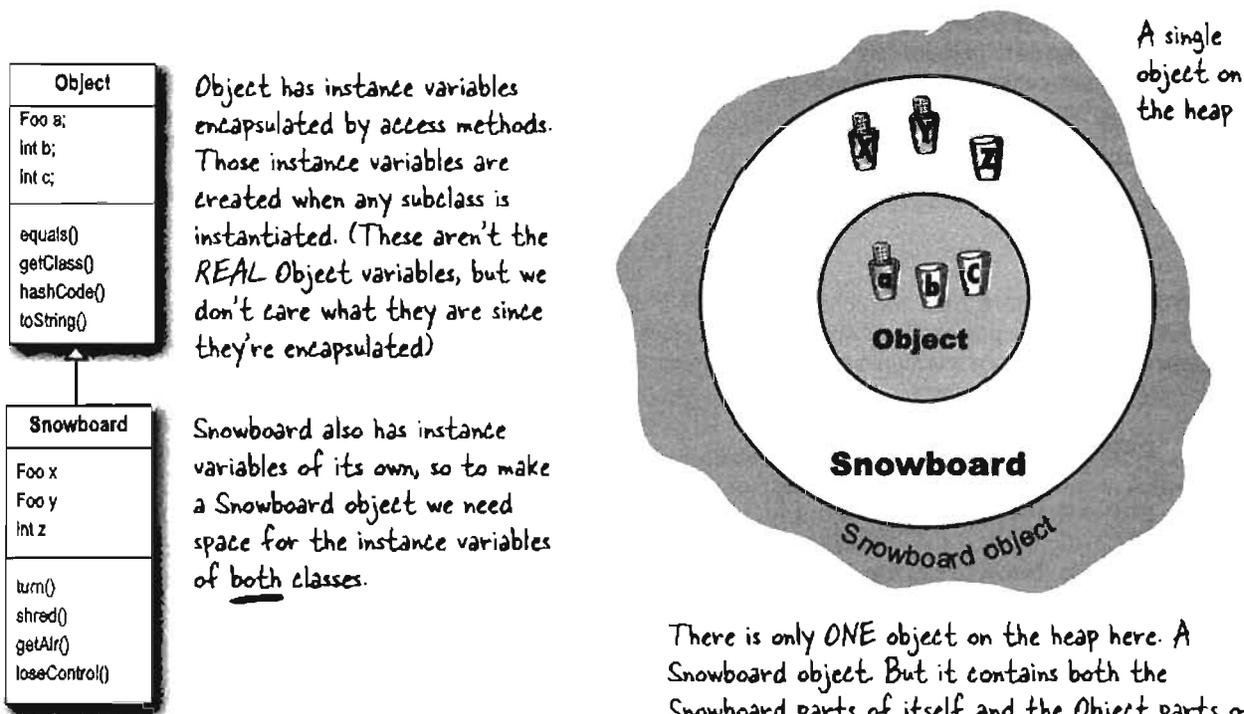
A: But that's not exactly right. Marking something `private` doesn't mean *nobody* can access it, it just means that *nobody outside the class* can access it. Bet you're thinking "Catch 22": Only code from the *same* class as the class-with-private-constructor can make a new object from that class, but without first making an object, how do you ever get to run code from that class in the first place? How do you ever get to anything in that class? *Patience grasshopper*. We'll get there in the next chapter.

space for an object's superclass parts

Wait a minute... we never DID talk about superclasses and inheritance and how that all fits in with constructors.

Here's where it gets fun. Remember from the last chapter, the part where we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its *own* declared instance variables, but also *everything from its superclasses* (which, at a minimum, means class Object, since *every* class extends Object).

So when an object is created (because somebody said `new`; there is *no other way* to create an object other than someone, somewhere saying `new` on the class type), the object gets space for *all* the instance variables, from all the way up the inheritance tree. Think about it for a moment... a superclass might have setter methods encapsulating a private variable. But that variable has to live *somewhere*. When an object is created, it's almost as though *multiple* objects materialize—the object being new'd and one object per each superclass. Conceptually, though, it's much better to think of it like the picture below, where the object being created has *layers* of itself representing each superclass.



There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard parts of itself and the Object parts of itself. All instance variables from both classes have to be here.

The role of superclass constructors in an object's life.

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

Let that sink in.

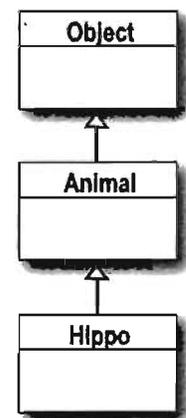
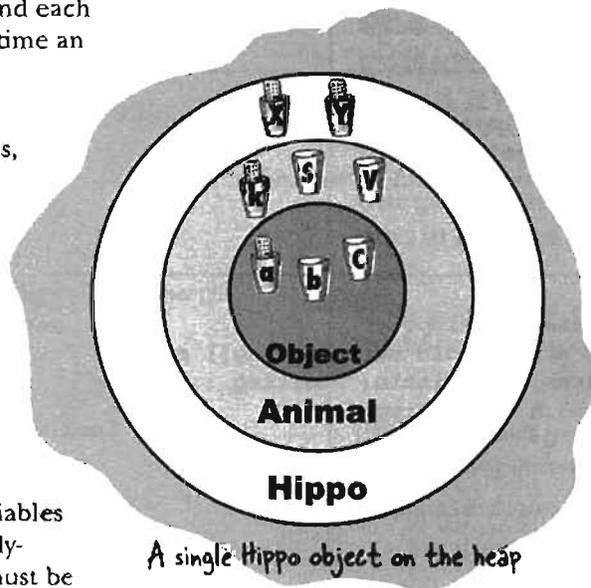
That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

Saying **new** is a Big Deal. It starts the whole constructor chain reaction. And yes, even abstract classes have constructors. Although you can never say **new** on an abstract class, an abstract class is still a superclass, so its constructor runs when someone makes an instance of a concrete subclass.

The super constructors run to build out the superclass parts of the object. Remember, a subclass might inherit methods that depend on superclass state (in other words, the value of instance variables in the superclass). For an object to be fully-formed, all the superclass parts of itself must be fully-formed, and that's why the super constructor *must* run. All instance variables from every class in the inheritance tree have to be declared and initialized. Even if **Animal** has instance variables that **Hippo** doesn't inherit (if the variables are private, for example), the **Hippo** still depends on the **Animal** methods that *use* those variables.

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

On the next few pages, you'll learn how superclass constructors are called, and how you can call them yourself. You'll also learn what to do if your superclass constructor has arguments!



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.

Making a Hippo means making the Animal and Object parts too...

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}

public class Hippo extends Animal {
    public Hippo() {
        System.out.println("Making a Hippo");
    }
}

public class TestHippo {
    public static void main (String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

Sharpen your pencil

What's the real output? Given the code on the left, what prints out when you run TestHippo? A or B? (the answer is at the bottom of the page)

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

1 Code from another class says `new Hippo()` and the `Hippo()` constructor goes into a stack frame at the top of the stack.



2 `Hippo()` invokes the superclass constructor which pushes the `Animal()` constructor onto the top of the stack.



3 `Animal()` invokes the superclass constructor which pushes the `Object()` constructor onto the top of the stack, since `Object` is the superclass of `Animal`.



4 `Object()` completes, and its stack frame is *popped* off the stack. Execution goes back to the `Animal()` constructor, and picks up at the line following `Animal`'s call to its superclass constructor.



The first one. A. The `Hippo()` constructor is invoked first, but it's the `Animal` constructor that finishes first.

How do you invoke a superclass constructor?

You might think that somewhere in, say, a Duck constructor, if Duck extends Animal you'd call `Animal()`. But that's not how it works:

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        BAD! → Animal (); ← NO! This is not legal!
        size = newSize;
    }
}
```

The only way to call a super constructor is by calling `super()`. That's right—`super()` calls the *super constructor*.

What are the odds?

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        super (); ← you just say super()
        size = newSize;
    }
}
```

A call to `super()` in your constructor puts the superclass constructor on the top of the Stack. And what do you think that superclass constructor does? *Calls its superclass constructor*. And so it goes until the Object constructor is on the top of the Stack. Once `Object()` finishes, it's popped off the Stack and the next thing down the Stack (the subclass constructor that called `Object()`) is now on top. *That* constructor finishes and so it goes until the original constructor is on the top of the Stack, where *it* can now finish.

And how is it that we've gotten away without doing it?

You probably figured that out.

Our good friend the compiler puts in a call to `super()` if you don't.

So the compiler gets involved in constructor-making in two ways:

① If you *don't* provide a constructor

The compiler puts one in that looks like:

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

② If you *do* provide a constructor but you do *not* put in the call to `super()`

The compiler will put a call to `super()` in each of your overloaded constructors.* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to `super()` is always a no-arg call. If the superclass has overloaded constructors, only the no-arg one is called.

*Unless the constructor calls another overloaded constructor (you'll see that in a few pages).

Can the child exist before the parents?

If you think of a superclass as the parent to the subclass child, you can figure out which has to exist first. *The superclass parts of an object have to be fully-formed (completely built) before the subclass parts can be constructed.* Remember, the subclass object might depend on things it inherits from the superclass, so it's important that those inherited things be finished. No way around it. The superclass constructor must finish before its subclass constructor.

Look at the Stack series on page 248 again, and you can see that while the Hippo constructor is the *first* to be invoked (it's the first thing on the Stack), it's the *last* one to complete! Each subclass constructor immediately invokes its own superclass constructor, until the Object constructor is on the top of the Stack. Then Object's constructor completes and we bounce back down the Stack to Animal's constructor. Only after Animal's constructor completes do we finally come back down to finish the rest of the Hippo constructor. For that reason:

Eewwww... that is SO creepy. There's no way I could have been born before my parents. That's just wrong.



The call to super() must be the first statement in each constructor*

Possible constructors for class Boop

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

```
 public Boop(int i) {  
    super();  
    size = i;  
}
```

These are OK because the programmer explicitly coded the call to super(), as the first statement

```
 public Boop() {  
}
```

```
 public Boop(int i) {  
    size = i;  
}
```

These are OK because the compiler will put a call to super() in as the first statement

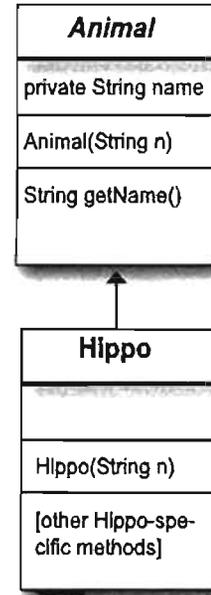
```
 public Boop(int i) {  
    size = i;  
    super();  
}
```

BAD!! This won't compile! You can't explicitly put the call to super() below anything else.

*There's an exception to this rule; you'll learn it on page 252.

Superclass constructors with arguments

What if the superclass constructor has arguments? Can you pass something in to the *super()* call? Of course. If you couldn't, you'd never be able to extend a class that didn't have a no-arg constructor. Imagine this scenario: all animals have a name. There's a *getName()* method in class *Animal* that returns the value of the *name* instance variable. The instance variable is marked private, but the subclass (in this case, *Hippo*) inherits the *getName()* method. So here's the problem: *Hippo* has a *getName()* method (through inheritance), but does not have the *name* instance variable. *Hippo* has to depend on the *Animal* part of himself to keep the *name* instance variable, and return it when someone calls *getName()* on a *Hippo* object. But... how does the *Animal* part get the name? The only reference *Hippo* has to the *Animal* part of himself is through *super()*, so that's the place where *Hippo* sends the *Hippo*'s name up to the *Animal* part of himself, so that the *Animal* part can store it in the private *name* instance variable.



```

public abstract class Animal {
    private String name;
}

public String getName() {
    return name;
}

public Animal(String theName) {
    name = theName;
}
    
```

Annotations for the code above:

- ← All animals (including subclasses) have a name
- ← A getter method that Hippo inherits
- ← The constructor that takes the name and assigns it the name instance variable

```

public class Hippo extends Animal {
    public Hippo(String name) {
        super(name);
    }
}
    
```

Annotations for the code above:

- ← Hippo constructor takes a name
- ← it sends the name up the stack to the Animal constructor

```

public class MakeHippo {
    public static void main(String[] args) {
        Hippo h = new Hippo("Buffy");
        System.out.println(h.getName());
    }
}
    
```

Annotations for the code above:

- ← Make a Hippo, passing the name "Buffy" to the Hippo constructor. Then call the Hippo's inherited getName()

The Animal part of me needs to know my name, so I take a name in my own Hippo constructor, then pass the name to super()



calling overloaded constructors

Invoking one overloaded constructor from another

What if you have overloaded constructors that, with the exception of handling different argument types, all do the same thing? You know that you don't want *duplicate* code sitting in each of the constructors (pain to maintain, etc.), so you'd like to put the bulk of the constructor code (including the call to `super()`) in only *one* of the overloaded constructors. You want whichever constructor is first invoked to call The Real Constructor and let The Real Constructor finish the job of construction. It's simple: just say `this()`. Or `this(aString)`. Or `this(27, x)`. In other words, just imagine that the keyword `this` is a reference to **the current object**

You can say `this()` only within a constructor, and it must be the first statement in the constructor!

But that's a problem, isn't it? Earlier we said that `super()` must be the first statement in the constructor. Well, that means you get a choice.

Every constructor can have a call to `super()` or `this()`, but never both!

```
class Mini extends Car {
```

```
    Color color;
```

```
    public Mini() {  
        this(Color.Red);  
    }
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls `super()`).

```
    public Mini(Color c) {  
        super("Mini");  
        color = c;  
        // more initialization  
    }
```

This is The Real Constructor that does The Real Work of initializing the object (including the call to `super()`)

```
    public Mini(int size) {  
        this(Color.Red);  
        super(size);  
    }  
}
```

Won't work!! Can't have `super()` and `this()` in the same constructor, because they each must be the first statement!

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!

```
File Edit Window Help Drive
```

```
javac Mini.java
```

```
Mini.java:16: call to super must  
be first statement in constructor
```

```
    super();  
    ^
```



Some of the constructors in the SonOfBoo class will not compile. See if you can recognize which constructors are not legal. Match the compiler errors with the SonOfBoo constructors that caused them, by drawing a line from the compiler error to the "bad" constructor.

```
public class Boo {
    public Boo(int i) { }
    public Boo(String s) { }
    public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {

    public SonOfBoo() {
        super("boo");
    }

    public SonOfBoo(int i) {
        super("Fred");
    }

    public SonOfBoo(String s) {
        super(42);
    }

    public SonOfBoo(int i, String s) {
    }

    public SonOfBoo(String a, String b, String c) {
        super(a,b);
    }

    public SonOfBoo(int i, int j) {
        super("man", j);
    }

    public SonOfBoo(int i, int x, int y) {
        super(i, "star");
    }
}
```



```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String, java.la
ng.String)
```

```
File Edit Window Help Yadayadayaada
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int, java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

Now we know how an object is born, but how long does an object *live*?

An *object's* life depends entirely on the life of references referring to it. If the reference is considered "alive", the object is still alive on the Heap. If the reference dies (and we'll look at what that means in just a moment), the object will die.

So if an object's life depends on the reference variable's life, how long does a *variable* live?

That depends on whether the variable is a *local* variable or an *instance* variable. The code below shows the life of a local variable. In the example, the variable is a primitive, but variable lifetime is the same whether it's a primitive or reference variable.

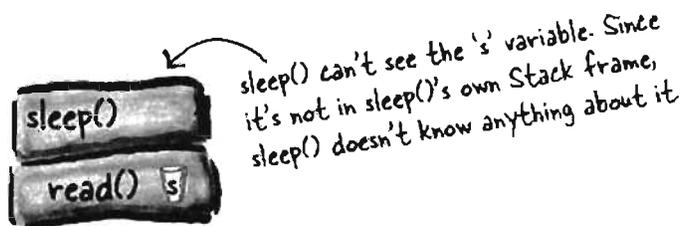
```
public class TestLifeOne {
```

```
    public void read() {
        int s = 42;
        sleep();
    }
```

's' is scoped to the read() method, so it can't be used anywhere else

```
    public void sleep() {
        s = 7;
    }
```

BAD!! Not legal to use 's' here!



The variable 's' is alive, but in scope only within the read() method. When sleep() completes and read() is on top of the Stack and running again, read() can still see 's'. When read() completes and is popped off the Stack, 's' is dead. Pushing up digital daisies.

1 A local variable lives only within the method that declared the variable.

```
public void read() {
    int s = 42;
    // 's' can be used only
    // within this method.
    // When this method ends,
    // 's' disappears completely.
}
```

Variable 's' can be used *only* within the read() method. In other words, *the variable is in scope only within its own method*. No other code in the class (or any other class) can see 's'.

2 An instance variable lives as long as the object does. If the object is still alive, so are its instance variables.

```
public class Life {
    int size;

    public void setSize(int s) {
        size = s;
        // 's' disappears at the
        // end of this method,
        // but 'size' can be used
        // anywhere in the class
    }
}
```

Variable 's' (this time a method parameter) is in scope only within the setSize() method. But instance variable size is scoped to the life of the *object* as opposed to the life of the *method*.

The difference between **life** and **scope** for local variables:

Life

A local variable is *alive* as long as its Stack frame is on the Stack. In other words, *until the method completes*.

Scope

A local variable is in *scope* only within the method in which the variable was declared. When its own method calls another, the variable is alive, but not in scope until its method resumes. *You can use a variable only when it is in scope*.

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // imagine more code here
}

public void crazy() {
    char c = 'a';
}
```



- 1 `doStuff()` goes on the Stack. Variable 'b' is alive and in scope.



- 2 `go()` plops on top of the Stack. 'x' and 'z' are alive and in scope, and 'b' is alive but *not* in scope.



- 3 `crazy()` is pushed onto the Stack, with 'c' now alive and in scope. The other three variables are alive but out of scope.



- 4 `crazy()` completes and is popped off the Stack, so 'c' is out of scope and *dead*. When `go()` resumes where it left off, 'x' and 'z' are both alive and back in scope. Variable 'b' is still alive but out of scope (until `go()` completes).

While a local variable is alive, its state persists. As long as method `doStuff()` is on the Stack, for example, the 'b' variable keeps its value. But the 'b' variable can be used only while `doStuff()`'s Stack frame is at the top. In other words, you can use a local variable *only* while that local variable's method is actually running (as opposed to waiting for higher Stack frames to complete).

What about reference variables?

The rules are the same for primitives and references. A reference variable can be used only when it's in scope, which means you can't use an object's remote control unless you've got a reference variable that's in scope. The *real* question is,

“How does *variable* life affect *object* life?”

An object is alive as long as there are live references to it. If a reference variable goes out of scope but is still alive, the object it *refers* to is still alive on the Heap. And then you have to ask... “What happens when the Stack frame holding the reference gets popped off the Stack at the end of the method?”

If that was the *only* live reference to the object, the object is now abandoned on the Heap. The reference variable disintegrated with the Stack frame, so the abandoned object is now, *officially*, toast. The trick is to know the point at which an object becomes *eligible for garbage collection*.

Once an object is eligible for garbage collection (GC), you don't have to worry about reclaiming the memory that object was using. If your program gets low on memory, GC will destroy some or all of the eligible objects, to keep you from running out of RAM. You can still run out of memory, but *not* before all eligible objects have been hauled off to the dump. Your job is to make sure that you abandon objects (i.e, make them eligible for GC) when you're done with them, so that the garbage collector has something to reclaim. If you hang on to objects, GC can't help you and you run the risk of your program dying a painful out-of-memory death.

An object's life has no value, no meaning, no point, unless somebody has a reference to it.

If you can't get to it, you can't ask it to do anything and it's just a big fat waste of bits.

But if an object is unreachable, the Garbage Collector will figure that out. Sooner or later, that object's goin' down.



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'.

Object-killer #1

Reference goes out of scope, permanently.



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

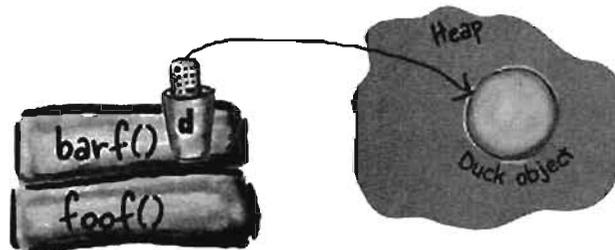
    public void barf() {
        Duck d = new Duck();
    }
}
```



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



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



The new Duck goes on the Heap, and as long as *barf()* is running, the 'd' reference is alive and in scope, so the Duck is considered alive.

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



Uh-oh. The 'd' variable went away when the *barf()* Stack frame was blown off the stack, so the Duck is abandoned. Garbage-collector bait.

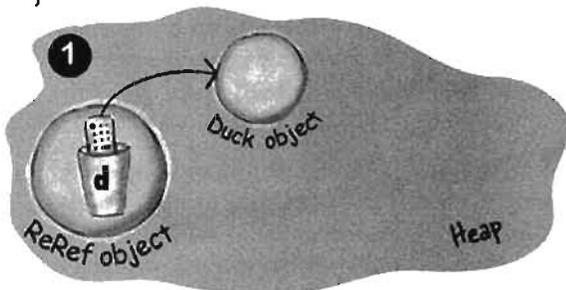
Object-killer #2

Assign the reference to another object

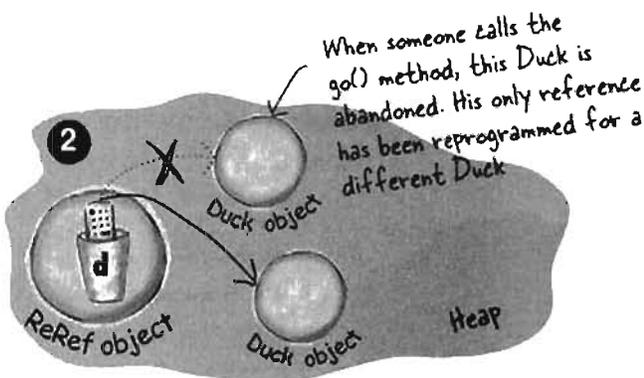


```
public class ReRef {  
    Duck d = new Duck();  
  
    public void go() {  
        d = new Duck();  
    }  
}
```

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



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



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



Object-killer #3

Explicitly set the reference to null



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

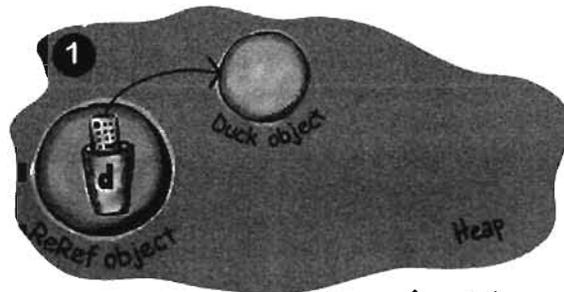
    public void go() {
        d = null;
    }
}
```

The meaning of null

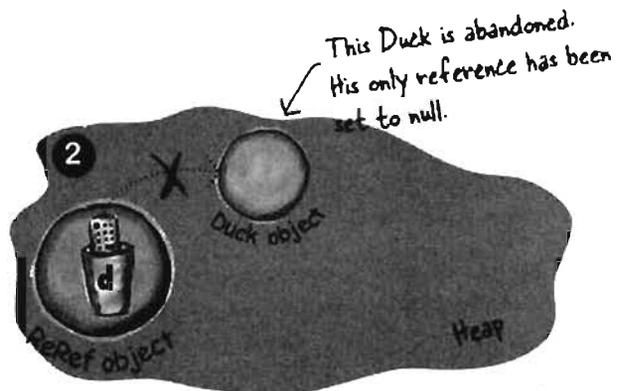
When you set a reference to `null`, you're deprogramming the remote control. In other words, you've got a remote control, but no TV at the other end. A null reference has bits representing 'null' (we don't know or care what those bits are, as long as the JVM knows).

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

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

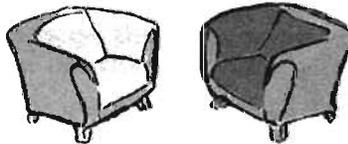


The new Duck goes on the Heap, referenced by 'd'. Since 'd' is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



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

Fireside Chats



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

Instance Variable

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

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

My apologies. I understand completely.

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

Local Variable

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

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

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

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

Instance Variable

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

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

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

They let us *drink*?

Local Variable

frame gets to run again. On the one hand, we get to be active again. On the other hand, the clock starts ticking again on our short lives. The more time our method spends running, the closer we get to the end of the method. We *all* know what happens then.

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

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

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

exercise: Be the Garbage Collector



BE the Garbage Collector

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

```
public class GC {
    public static GC doStuff() {
        GC newGC = new GC();
        doStuff2(newGC);
        return newGC;
    }

    public static void main(String [] args) {
        GC gc1;
        GC gc2 = new GC();
        GC gc3 = new GC();
        GC gc4 = gc3;
        gc1 = doStuff();

        A

        // call more methods
    }

    public static void doStuff2(GC copyGC) {
        GC localGC
    }
}
```

1 copyGC = null;

2 gc2 = null;

3 newGC = gc3;

4 gc1 = null;

5 newGC = null;

6 gc4 = null;

7 gc3 = gc2;

8 gc1 = gc4;

9 gc3 = null;



Popular Objects

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

Good Luck !

```

class Bees {
    Honey [] beeHA;
}

class Raccoon {
    Kit k;
    Honey rh;
}

class Kit {
    Honey kh;
}

class Bear {
    Honey hunny;
}

public class Honey {
    public static void main(String [] args) {
        Honey honeyPot = new Honey();
        Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
        Bees b1 = new Bees();
        b1.beeHA = ha;
        Bear [] ba = new Bear{5};
        for (int x=0; x < 5; x++) {
            ba[x] = new Bear();
            ba[x].hunny = honeyPot;
        }
        Kit k = new Kit();
        k.kh = honeyPot;
        Raccoon r = new Raccoon();

        r.rh = honeyPot;
        r.k = k;
        k = null;
    } // end of main
}

```

Here's a new Raccoon object!

Here's its reference variable 'r'.

puzzle: Five Minute Mystery



Five-Minute Mystery



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

"What ratios are you simulating?", Tom asked.

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

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

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

```
import java.util.*;
class V2Radiator {
    V2Radiator(ArrayList list) {
        for(int x=0; x<5; x++) {
            list.add(new SimUnit("V2Radiator"));
        }
    }
}

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

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

Five-Minute Mystery continued...

```

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

class SimUnit {
    String botType;
    SimUnit(String type) {
        botType = type;
    }
    int powerUse() {
        if ("Retention".equals(botType)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

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

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

object lifecycle

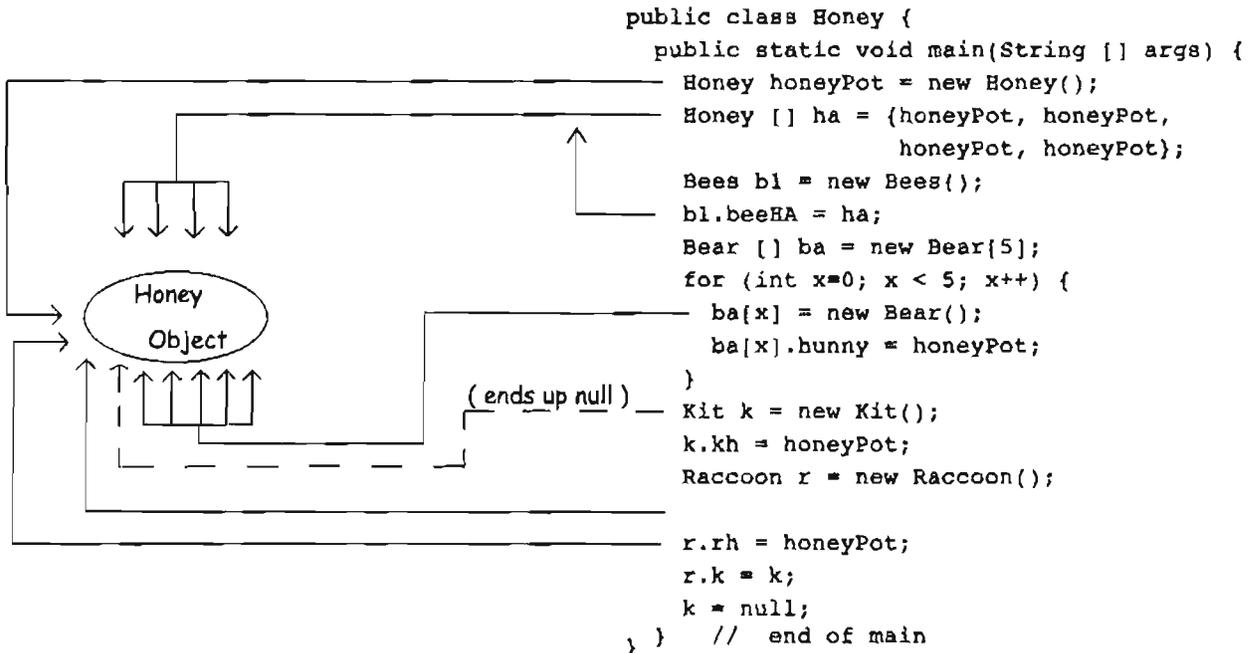


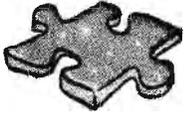
G.C.

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

Popular Objects

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





Five-Minute Mystery Solution

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

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