**B”H**

**Chapter 9**

**Constructors and Garbage Collection**

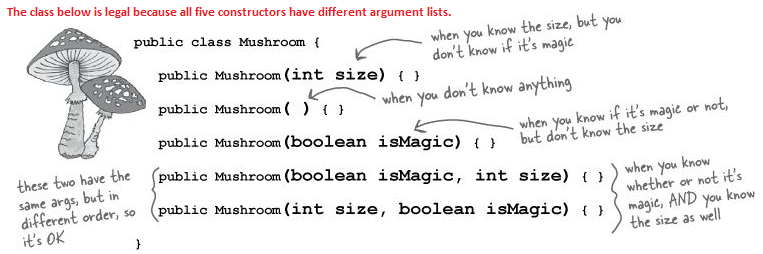
* There are two areas of memory
  + The heap – where objects and instance variables live
  + The stack - where method invocations and local variables (also known as stack variables) live
* Instance variables are declared inside a class but not inside a method.
* Local variables are declared inside a method, including method parameters.
* Methods are stacked
* The method at the top of the stack is always the currently-running method for that stack (see Chap 14 to use multiple stacks)



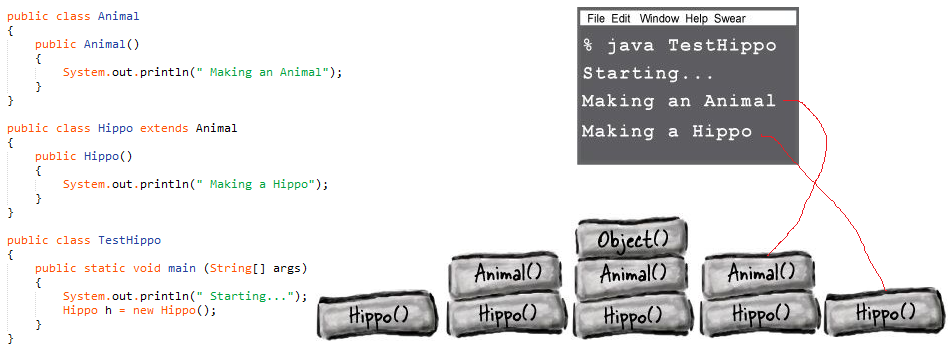
* If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack. The object itself still goes in the heap.
* Knowing the fundamentals of the Java Stack and Heap is crucial if you want to understand variable scope, object creation issues, memory management, threads, and exception handling.

**Constructors**

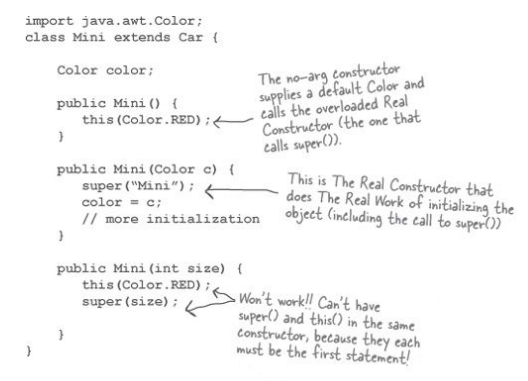
* A constructor does look and feel a lot like a method, but it’s not a method. It’s got the code that runs when you say new. In other words, the code that runs when you instantiate an object.
* The only way to invoke a constructor is with the keyword new followed by the class name. (You can call a constructor from within another constructor, with restrictions, but we’ll get into all that later in the chapter.)
* If you don’t write a constructor for your class the compiler writes one for you
* Constructors must have the same name as the class
* 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 (like initialize the state of the object) 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.
* 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.
* In general, if possible, you should provide a no-arg constructor
* 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 (Java won’t create a default one)
* Overloaded constructors means you have more than one constructor in your class. To compile, each constructor must have a different argument list.



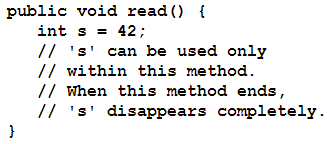
* Constructors can be public, protected, private, or default
* All the constructors in an object’s inheritance tree must run when you make a new object. This all happens in a process called Constructor Chaining.
* Every class has a constructor
* Even abstract classes have constructors. Although you can 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.

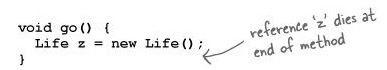
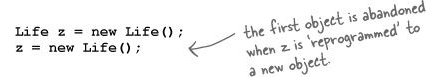
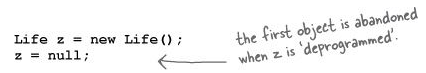


* super() calls the super constructor.
* If you do not put in the call to super() the compiler will put a call to super() in each of your overloaded constructors. Unless the constructor calls another overloaded constructor (you’ll see that soon).
  + The compiler-inserted call to super() is always a no-arg call.
  + You can add one with arguements
* The superclass parts of an object have to be fully-formed (completely built) before the subclass parts can be constructed.
* You can invoke one overloaded constructor from another (to avoid code duplication etc.) like this:
  + this() or this(aString) or this(27,x) etc.
* 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!
* Every constructor can have a call to super() or this(), but never both.
* When used, the call to super() or this() must be the first statement in the constructor



* 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 the object will die.
* A **local** variable lives only within the method that declared the variable.



* The ‘s’ variable is in scope only within its own method. No other code in the class (or any other class) can see ‘s’.
* An **instance variable** lives as long as the object does. If the object is still alive, so are its instance variables.
* An **object** becomes eligible for GC when its last live reference disappears.
* Three ways to get rid of an object’s reference:
  1. 
  2. 
  3. 
* When you set a reference to null, you’re deprogramming the remote control. If you use the dot operator on a null reference, you’ll get a NullPointerException at runtime.