Head First Java: Chapter 9 Notes

August 30, 2019

Memory Management in Java, the Stack, and the Heap

The stack and the heap are concepts that aid in understanding on how Java allocate memory to various entities in written programs.

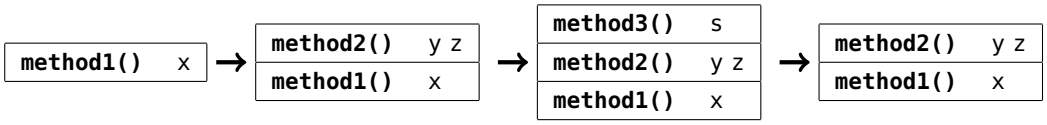
- Stack — The stack can be visualized as a literal stack of frames. Each frame is an invoked method within a class. Methods and variables can be thought of "living" on a stack. Consider:

```
public void method1() {
    String x = ''Doing stuff '';
    System.out.println(x);
    method2(x);
}

public void method2(String y) {
    String z = y + ''again '';
    System.out.println(z);
    method3();
}

public void method3  {
    String s = ''Nope I am done'';
    System.out.println(s);
}
```

Above is a series of methods that calls each other. When compiled and ran, the stack will initially look like this:



Based on the code, `method1()` was invoked first; it is the first frame in the stack. Within this stack, the *local variable* `x` is created by `method1` and "lives" in its frame on the stack. Because the `method1()` frame is at the top of the stack at this moment, `method1()` and its associated local variables (`x` in this case) is said to be *in scope* and therefore accessible. Next, `method1()` calls `method2()` within its own code. Another frame containing `method2()` and its local variables (`y` and `z`) is not added to the top of the stack. Because only the frame at the *top* of the stack is in scope, only the `method2()` frame and everything in it is in scope

and thus accessible from the rest of the program. While not in scope, `method1()` and its variables cannot be accessed, but it is still "alive" (*i.e.* it still exist in memory.) Within `method2()`, `method(3)` was called, and thus it is pushed to the top of the stack along with its local variable `s`. At this point, both `method2()` and `method1()` are no longer in scope and their variables are unavailable. Because `method3()` does not call any other methods itself, it completes its task (in this case, printing "Nope I am done" to the screen) and its frame along with its variables are "popped" off the stack. Once off the stack, all variables within that frame is "dead" and is permanently inaccessible. This stack continues to run with the `method2()` frame in scope now, and so forth. All local variables, regardless if they are primitive or object references, exist in the stack and behave the same way. In the case of object reference variables, the actual object being references do not exist in the stack, however, they exist in the heap.

- Heap — Java objects (not object references) exist in the memory heap. The heap is unlike the stack as they are not ordered and scopes has no meaning here. Recall object creation and object reference declaration in Java is a three step process:

  1. `Object o = new Object` — declaration of a new `Object` class object reference variable in the stack or the heap.
  2. `Object o = new Object` — instantiation of a new `Object` class object in the heap.
  3. `Object o = new Object` — connecting the object reference variable `o` either on the stack or in the heap to the actual object in the heap.

  While this is only one line of code in Java, three different things are occurring. If the object reference variable is a local variable of a method, then it exists in the frame of its creating method on the stack. However, if the object reference variable is an instance variable, then it exist in the heap. In either case, whatever object the object reference variable is referencing exists in the heap. Note that often object reference instance variable can be declared without assigning and instantiating an object (not possible with local variables), in this case the object reference variable still remains in the heap awaiting assignment.

  An important note can be observed here: object reference variables can be declared without assigning them to an object (existing or newly instantiated), but objects **CANNOT** be instantiated without assigning them to an object reference variable. The implication

here is that if the object reference variable assigned to an object ceased to exist in memory, the assigned object essentially also ease to exist (in reality, the object remains in the heap, but it is no longer accessible and is marked for *garbage collection*.)

Object Creation and Constructors

Objects are instantiated in Java by constructors. Explicitly writing constructor code within a class is optional as if they are unwritten the compiler will write one itself. When the `new` keyword is used during object reference variable declaration and object assignment/instantiation, the constructor is called (whether it is explicitly written code or implicitly written by the compiler.) Consider the class `Duck`:

```
public class Duck {
    public Duck() {             // Constructor code begins here
        System.out.println(''Quack'');
    }
}
```

The above code differs from the generic constructor code that the compiling would write if unwritten only with the `System.out.println("Quack")` line. By writing this `public Duck(){}` constructor code in the class, the compiling will no longer write the generic constructor into the bytecode but will run the explicitly written constructor instead. In this case, every time a `Duck` object is instantiated, the constructor will also print `"Quack"` to the screen.

Written explicit constructors into a class is useful especially when external information is needed (either from the user or from elsewhere in the program) for the instantiation of the object. Consider the following modified `Duck` class:

```
public class Duck {
    private int size;       // Instance variable
    public Duck() {         // Constructor
        System.out.println(size);
    }
    public void setSize(int newSize) { // Setter method
        size = newSize;
    }
}
```

The danger here is that a new `Duck` object can be instantiated by `Duck aDuck = new Duck();` without any problems, but the user or the rest of the program will have to immediately run the `aDuck.setSize()` method, otherwise the `Duck` object will not have a size, which can cause unexpected behavior for the rest of the program. One way to avoid this is to assign a default value to the instance variable `size`

(*e.g.* `private int size = 42;`), but due to encapsulation this is not enough if flexibility is needed for users to set there own `size` value. In this case, multiple constructors should be coded by *overloading*:

```
public class Duck {
    private int size;
    public Duck() {
        this(42);
    }
    public Duck(int userSize) {
        size = userSize;
        System.out.println(size);
    }
}
```

When a new `Duck` object is instantiated, one or the other constructor above will be called depending if an appropriate argument is passed or not. In this case, if an `int` argument was passed, a new `Duck` object is instantiated with the user supplied size, which is also printed on the screen. If no argument was passed, then the constructor with the `this()` method is called. The `this()` helps with consistency if multiple constructor is needed; it will called the most appropriate constructor for the current object depending on the argument passed to `this()`. In this case, `this(42);` passed a valid `int` argument for the `public Duck(int userSize) {}` constructor, therefore it acts like a default if the user or the rest of the program instantiated a new `Duck` object without supplying an argument, and assigning `42` to `size`. If `this()` is to be used in a constructor, it **must** be the first line of the code.

In summary:

1. A constructor is the code that runs when a new object is instantiated (*e.g.* the `new` keyword was used).
2. A constructor must have the same name as the class it is written in, but have no return type.
3. A constructor is implicitly written by the compiler if not explicitly written in the class.
4. Constructors can be *overloaded*, meaning multiple constructors can be written for the same class. However, the overloaded constructors must have different argument lists (argument orders, types, etc.)

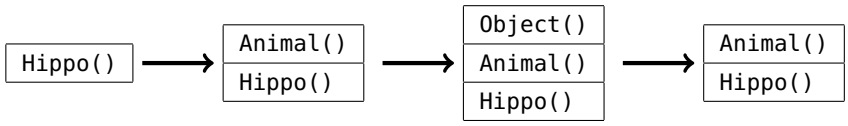
Constructors and Inheritance

Constructors exist in all classes, whether or not they are a superclass, a subclass, or both. Consider the following:

```
public class Animal {
    public Animal() {
        System.out.println(‘‘Making an animal’’)
    }
}

public class Hippo extends Animal {
    public Hippo() {
        System.out.println(‘‘Making a Hippo’’)
    }
}
```

Constructors behaves like methods, and thus they exist on the stack. When a `Hippo` object is instantiated (by `Hippo h = new Hippo();` for example), the stack evolves:



The `Hippo()` constructor is first called, and remain on the top of the stack. However, `Hippo()` inherits from `Animal`, its superclass, and therefore before any code in `Hippo()` is ran, the constructor `Animal()` is pushed to the top of the stack, putting `Hippo()` and any of its code (and if exist, variables) out of scope and accessibility. Recall that all objects are ultimately subclasses of class `Object`, so its constructor `Object()` is pushed to the top of the stack. Any code in `Animal()` is not out of scope as well until `Object()` completes and is popped off the stack. Hence, the screen output when instantiating a `Hippo` object will first be “`Making an Animal`” then “`Making a Hippo`”.

When a subclass object is instantiated, its superclass constructor is always called first, whether or not it is explicitly written out. The listing below is essentially the same from the one above:

```
public class Animal {
    public Animal() {
        super();
        System.out.println(‘‘Making an animal’’)
    }
}

public class Hippo extends Animal {
    public Hippo() {
        super();
        System.out.println(‘‘Making a Hippo’’)
    }
}
```

The `super()` above is a call of the immediate superclass constructor. Therefore, the `super()` in `Hippo()` calls the `Animal()` constructor, and the `super()` in `Animal()` calls the `Object()` constructor. If `super()` is not explicitly stated in a constructor, the compiler will write it implicitly. If written implicitly by the compiler, `super()` is always called without any arguments.

Arguments can be passed to `super()` if it is appropriate. If we modify the `Animal` and `Hippo` class:

```
public abstract class Animal {
    private String name;
    public String getName() {
        return name();
    }
    public Animal(String userName) {
        name = userName;
    }
}

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

The `Animal` class is now flagged `abstract` and therefore cannot be instantiated. It also contains an encapsulated `String` variable `name` that is not assigned value, a getter method that returns `name`, and a constructor that requires a `String` argument `userName` that it assigns to `name`. `Hippo` is a subclass of `Animal`, its constructor is explicitly written to require `String` argument `hippoName`. Within the code body of `Hippo()`, `super()` is explicitly called and passing the argument `hippoName`. Because `Animal` is the immediate superclass of `Hippo`, when `super(hippoName)` was called, the `Animal()` constructor was called and `hippoName` is passed up the inheritance tree. When a `Hippo` object is instantiated with, for example, the `String` argument "Buffy", all constructors within the direct inheritance tree are called in the order of `Object()`, `Animal()`, and `Hippo`. An `Object` is instantiated first, then the `Animal` is instantiated and wrapped around the `Object` object, within it the instance variable `name`, to which "Buffy" is assigned its value, and finally the `Hippo` is instantiated wrapping around the `Animal`.

If the superclass has overloaded constructors (multiple constructors), when calling `super()` in its subclass' constructor the argument type and order will determine which superclass constructor is called. Consider:

```
public class Boo {
```

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

The class `Boo` is overloaded with three constructors, each requiring different argument(s) type and order. If a subclass exist, it must include an explicit call to its superclass constructor, because a no-argument constructor for `Boo` was not provided. Calling `super()` in a subclass constructor will not compile. There are many possibilities here, for example, if `super("boo")` was used, then the `public Boo(String s) {}` constructor is called. If `super(42)` was used, then the `public Boo(int i) {}` is called, and so forth.

Note that if `super()` is to be used in a constructor, it **must** be the first line of code. Therefore, `this()` and `super()` cannot be used in the same constructor.

Garbage Collection

As mentioned previously, when objects in the heap "dies" or become permanently inaccessible in the memory, the JVM will automatically marked it as eligible for *garbage collection*. This means that when JVM determines that more memory is needed, it will overwrite memories eligible for garbage collection that is no longer in need. Everything in Java occupies memory. Things that exist on the stack (methods, local variables, constructors, etc.) are marked for garbage collection once it is popped off the stack (note that being out of scope does not mean being marked for garbage collection, the inaccessibility is temporary in this case.) Instance variables exist in objects in the heap, and they are "alive" as long as the object they exist in are still alive (*i.e.* NOT marked for garbage collection.)

Another way to look at this is that anything on the stack remains alive and accessible (if not now, then later), but once popped off the stack (the frame it resides in completes), it is marked for garbage collection and is gone forever. Things that are in the heap remains alive as long as a reference to it exist. Generally speaking, there are two main categories of things in Java that exists in the heap: Object reference variables, and objects themselves. Objects remain alive as long as there is some reference to it from the stack. Recall from previously that an object cannot be instantiated without assigning to a reference variable (`Object o = New Object()`). If that link is severed, then the object itself is marked for garbage collection and is gone forever. There are a number of ways this can occur:

1. The frame where the object reference variable resides in is popped off the stack. This occurs when a method (or a constructor etc.)

completes its task. The object reference variable is marked for garbage collection and is gone forever, therefore the same is true to the object itself.

2. The object reference variable is assigned to another object. Since an object reference variable can only be assigned to a single object at anytime, the original object no longer has a reference on the stack, and is marked for garbage collection.

3. The object reference variable is set to `null`. This is the same case as above. Essentially Java tells the object reference variable to sever its link to the object. As above, the object itself is then marked for garbage collection.

Object reference variables themselves is a more complicated case as they can both exist on the stack as local variables or within an object in the heap as instance variables (this is true to primitive variables as well). Object reference variables that exist on the stack obeys the same rules, once popped off the stack they are marked for garbage collection. In the heap, object reference variables do not exist on their own but are inside actual objects. In this case, these object reference variables in the heap exist as long as the objects they reside remains "alive". If those objects are marked for garbage collection, so do all instance variables that reside within.