Section 11

Variables, Memory & Garbage Collection

Learning Outcomes

- After this section you should be able to:
  - Distinguish between variable types.
  - Outline the difference between Stack and Heap in relation to the JVM.
  - Describe the process of constructor chaining.
  - Distinguish between scope and lifespan in relation to variables.
  - Discuss how variables and objects become available for Garbage Collection.

Reading and studying recommended text is essential to improve your understanding of the above.

Variables

- A variable is a named location in a computer's memory that stores a value
- numbers and other data can be *remembered* in the computer's memory and can access that data through program elements called variables
- A variable has *three* properties:

  A memory location to store the value,

  The type of data stored in the memory location, and

  The name used to refer to the memory location

Java Variable Types

- Instance Variables (Non-Static Fields)
  - objects store their individual states in "non-static fields" (no static keyword). As their values are unique to each instance of a class, non-static fields are also known as instance variables
- Class variables (Static Fields)
  - any field declared with the static modifier and no matter how many times the class is instantiated there is only one copy of this variable
- Local Variables
  - local variables store temporary state inside a method and are only visible to the methods in which they are declared
- Parameters
  - parameters are variables that provide additional information to a method, parameters are always classified as "variables" not "fields"

Memory

- Heap (aka. Garbage Collectible Heap)
  - Area of memory where objects reside.
- Stack
  - Area of memory method calls and local variables reside.
- When JVM commences it is allocated memory by the OS.
- Where variables live depends on the type (instance or local).
  - Local variables live on the stack.
  - Instance variables live on the heap.

Instance Variables

- Instance variables
  - are declared inside a class but NOT inside a method.
  - represent the *fields* that each individual object has.
  - can contain different values for each individual object.
  - reside inside the object they belong to.

```
public class Animal {

    // instance variable, every Animal has a “name” instance variable
    private String name;

}
```

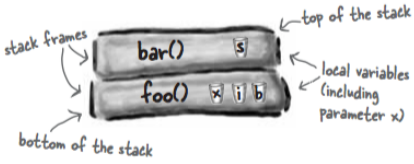
Local Variables

- Local variables
  - are declared inside a method, including method parameters
  - are temporary, live only as long as the method is on the stack (not reached closing curly bracket).

```
public void setName(String animalName){

        // animalName is a local varaible
        name = animalName;

}


 public int foo(int x) {

     // x, b and i are local variables
     int b = 2;
     int i = x + 1;
     return i;

 }
```

Methods are Stacked

- When you call a method, the method lands on top of a call stack.
  - What's pushed onto the stack is the stack frame, which holds the state of the method including which line of code is being executing and values for all of the local variables.
  - The method on top of the stack is always the currently-running method for the stack.
  - A method stays on the the stack until the method hits the closing curly brace (signifying the method has completed).
  - 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.

Stack Scenario

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

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

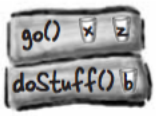
```
public void crazy(){
    char c = 'a';
}
```

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()**.



Local variables which are objects

- A non-primitive variable holds a reference to an object, not the object itself.
- Objects live on the heap, no matter where they are declared or created.
- If a local variable is a reference to an object, only the variable (the reference) goes on the stack.
- The object itself still goes on the heap.
- Note: Knowing the fundamentals of Stack and Heap is crucial in understanding scope, object creation and memory management

What about Instance Variables?

- When you create an instance of an object, the JVM allocates space on the heap for the object.
- The space required is large enough to house all of the object's instance variables.
- Instance variables reside *on the heap*, inside the object they belong to.
- Instance variables can be primitive types of reference types.
  - When a primitive type is declared, adequate space is allocated.
  - With a reference type is declared, only space for the reference is allocated. Space for the object is only allocated when the reference variable is assigned a new object.

Creating an object

- Three steps
  - Declare a reference variable.
  - Create an object.
  - Assign the object to the reference.

```
Dog a = new Dog();
```

- Not calling a Dog method, we are calling the Dog constructor.
- A Constructor:
  - contains code that runs when you instantiate an object.
  - has the same name as the class and no return type.
  - is often used to initialise the state of an object.

```
public Dog(){
    // code
}
```

- Should have a no argument constructor.
- Can be overloaded (different parameter list).

Constructor Chaining

- All the constructors in an objects inheritance tree must run when you make a new object.
- For an object to be fully-formed, all the superclass parts of itself must be fully-formed.
- Hence the super constructor must run.
- When a constructor is run it immediately calls it's superclass constructor, all the way up the chain until you get to the class Object constructor.

```
// constructor
public Beagle(){
    super();
}
```

Constructor

- Summary
  - A new Beagle object also IS-A Dog and IS-A Animal and IS-A Object.
  - If you want to make a Beagle you must also make the Dog and the Animal and Object part of the Beagle.
- Invoking one overloaded constructor from another.
  - 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 not both.

Constructor & this()

```
class Person {

    // Data Members
    private String  name;     // The name of this person
    private int     age;      // The age of this person
    private char    gender;   // The gender of this person

    // Default constructor
    public Person() {
        this("Not Given", 0, 'U');
    }

    // Constructs a new Person with passed name, age, and gender.
    public Person(String name, int age, char gender) {
        this.age    = age;
        this.name   = name;
        this.gender = gender;
    }

    …

}
```

Lifespan

- If a reference referring to an object is considered alive, the object is still alive on the heap.
- If an object's life depends on the the reference variables life, how long does a variable live for.
  - A local variable lives only within the method that declared the variable.
  - A local variable is in scope within it's own method.
  - An Instance variable lives as long as the object does. If the object is still alive so too are its instance variables.
  - Variable *animalName* is in scope only within the *setName()* method. But instance variable name is scoped for the life of the object as opposed to the life of the method.

Life & Scope

- 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 it 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 class Animal {

    private String name;

    public void setName(String animalName){
        name = animalName;
    }
    …
}
```

Garbage Collection

- Rules
  - A reference variable can only be used when it's in scope.
  - An object is still 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 alive on the heap.
- However
  - What happens when the Stack frame holding the reference gets popped off the stack at the end of the method?
  - The reference variable dies with the Stack frame and the abandoned object is now available for garbage collection.

Object-killers

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'.

Summary

- In this lecture we looked at:
  - Variable types.
  - Instance Variables VS local variables.
  - Stack & Heap.
  - Constructors & constructor overloading.
  - Constructor chaining.
  - Life & Scope.
  - Garbage Collection.