CS231: Lecture 4 (More Java)

- Recall Primitive data types: byte, short, int, long, float, double, Boolean, char. If your variable is not one of the 8 types we discussed before, then it is non primitive type (a variable whose type is an object). We call such variable object reference variable.

- **Java's memory model**
  - All variables are allocated space in memory to store something.
  - Java divides the computer's memory into two parts: a stack and a heap.
  - Stack is the section of memory where we will store temporary information such as method invocations, local variables;
  - Objects are stored in Heap memory.

  ![Stack and Heap diagram](image)

  - **Primitives vs objects**
    1. All primitive variables store their value in that space.
    2. All reference variables (object reference variable) store a pointer in that space to the object they refer to.

  - Primitive variable represents the actual value of the variable whereas object reference variable represents a way to get to the object.

- Example
Local variables vs fields

- There are two kinds of variables: instance variables/fields and local variables/parameters
- Every method has a stack frame with space for all its local variables (while it is being executed)
- Fields reside in the objects on the heap.

Local/Instance/Class Variable: [Check Table]

- Instance variables – declared inside a class. Must be initialized before use (else compiler error)
- Local variables – declared within a method

Parameters are just local variables in Java except that they are initialized by the caller instead of in the method body.

Local variables’ lifetimes consist of the time the method that contains them starts until the method returns.

Field's lifetimes consist of the time from the creation of the object until the garbage collection of the object.

- Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects
- [Garbage Collection](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html)

Local variable's scope is from the point of declaration to the end of the enclosing scope.

Field’s scope depends on whether they are public or private. The scope of private fields is the whole class body in which the fields are declared. The scope of public fields is everywhere.
<table>
<thead>
<tr>
<th></th>
<th>local var/params</th>
<th>instance var/field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>stack frame</td>
<td>heap</td>
</tr>
<tr>
<td>Lifetime</td>
<td>during method execution</td>
<td>object creation to garbage</td>
</tr>
<tr>
<td>Scope</td>
<td>remaining body of method</td>
<td>class if private, all if public</td>
</tr>
</tbody>
</table>

- Recall that the `new` command allocates memory for a new object, including space for all the fields of the object, clears out that memory, calls the constructor to initialize the fields, and then returns a pointer to the memory.
- You can have more than one constructor, as long as they differ on the number or types of the parameters.
- Method calls' parameters are **passed by value**
  - Primitive values are copied into the parameters
  - Objects are not copied. Instead the pointers to the objects are copied into the parameters.

- Example
  ```java
class Main
{
  public static void main(String[] args)
  {
    Foo f = new Foo("f");
    changeReference(f); // It won't change the reference!
    modifyReference(f); // It will modify the object that the reference variable "f" refers to!
  }
  public static void changeReference(Foo a)
  {
    Foo b = new Foo("b");
    a = b;
  }
  public static void modifyReference(Foo c)
  {
    c.setAttribute("c");
  }
}
```

1. Declaring a reference named `f` of type `Foo` and assign it to a new object of type `Foo` with an attribute "f".
   ```java
   Foo f = new Foo("f");
   ```
From the method side, a reference of type Foo with a name a is declared and it’s initially assigned to null.

2. `public static void changeReference(Foo a)`

3. As you call the method changeReference, the reference a will be assigned to the object which is passed as an argument.
   `changeReference(f);`

4. Declaring a reference named b of type Foo and assign it to a new object of type Foo with an attribute "b".
   `Foo b = new Foo("b");`

5. a = b is re-assigning the reference a NOT f to the object whose its attribute is "b".

6. As you call `modifyReference(Foo c)` method, a reference c is created and assigned to the object with attribute "f".
7. `c.setAttribute("c");` will change the attribute of the object that reference `c` points to it, and it's same object that reference `f` points to it.

(http://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value)

- Methods that return a value return the actual value if primitive and return a pointer to the object if it is not primitive.

- Recall that fields are initialized to default values when an object is constructed. What if one field was of type String? Can you guess the default value?

- Objects default to null; ints, longs and shorts to 0; Strings to null; booleans to false (Recall - Default Values: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value (for fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>char</td>
<td>\u0000'</td>
</tr>
<tr>
<td>String (or any object)</td>
<td>null</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
</tbody>
</table>