Constructor Initialization

Öznur ALKAN

The Process of Creating an Object

- Consider a class called **Dog**:
- The first time an object of type **Dog** is created *or* the first time a **static** method or **static** field of class **Dog** is accessed, the Java interpreter must locate **Dog.class**, which it does by searching through the classpath.
- As **Dog.class** is loaded, all of its **static** initializers are run.
- Thus, **static** initialization takes place only once, as the **Class** object is loaded for the first time.
- Super class constructors are called in a chained manner.
- When you create a **new Dog( )**, the construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.
- This storage is wiped to zero, automatically setting all the primitives in that **Dog** object to their default values (zero for numbers and the equivalent for **boolean** and **char**) and the references to **null**.
- Any initializations that occur at the point of field definition are executed.
- Constructors are executed.

Order of Initialization

1. Static variables initialized (in textual order and if not previously initialized)
2. Super() call in the constructor (explicit or implicit)
3. instance variables initialized (in textual order)
4. Remaining body of the constructor after super() call is executed

Constructor Initialization

- constructor can be used to perform initialization
- this gives greater flexibility because you can call methods and perform actions at run time to determine the initial values.
- if you say:

```
class Counter {
        int i;
        Counter() { i = 7; } ,
        // . . .
}
```

- then **i** will first be initialized to 0, then to 7.
- This is true with all the primitive types and with object references, including those that are given explicit initialization at the point of definition.
- For this reason, the compiler doesn't try to force you to initialize elements in the constructor at any particular place, or before they are used
- initialization is already guaranteed.

Order of Initialization

- Within a class, the order of initialization is determined by the order that the variables are defined within the class.
- The variable definitions may be scattered throughout and in between method definitions
- but the variables are initialized before any methods can be called—even the constructor.

Order of Initialization

```
class Tag {
    Tag(int marker) {
        System.out.println("Tag(" + marker + ")");
    }
}

class Card {
    Tag t1 = new Tag(1); // Before constructor
    Card() {
        // Indicate we're in the constructor:
        System.out.println("Card()");
        t3 = new Tag(33); // Reinitialize t3
    }
    Tag t2 = new Tag(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Tag t3 = new Tag(3); // At end
}
```

```
public class OrderOfInitialization {
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Shows that construction is done
    }
}
```

OUTPUT:

Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()

Static data initialization

- When the data is **static**, the same thing happens;
- if it's a primitive and you don't initialize it, it gets the standard primitive initial values.
- If it's a reference to an object, it's **null** unless you create a new object and attach your reference to it.
- If you want to place initialization at the point of definition, it looks the same as for non-**static**s.
- There's only a single piece of storage for a **static**, regardless of how many objects are created.
- But the question arises of when the **static** storage gets initialized.
- An example makes this question clear:

```
class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    b2.f(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}
```

```
class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    b4.f(2);
  }
  void f3(int marker) {
    System.out.println("f3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}
```

```
public class StaticInitialization {

public static void main(String[] args) {
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        t2.f2(1);
        t3.f3(1);
 }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
}
```

OUTPUT:

```
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

Static data initialization

- **Bowl** allows you to view the creation of a class
- **Table** and **Cupboard** create **static** members of **Bowl** scattered through their class definitions.
- Note that **Cupboard** creates a non-**static Bowl b3** prior to the **static** definitions.
- From the output, you can see that the **static** initialization occurs only if it's necessary.
- If you don't create a **Table** object and you never refer to**Table.b1** or **Table.b2**, the **static Bowl b1** and **b2** will never be created.
- They are initialized only when the *first* **Table** object is created (or the first**static** access occurs).
- After that, the **static** objects are not reinitialized.
- The order of initialization is **static**s first, if they haven't already been initialized by a previous object creation, and then the non-**static** objects.

FREE TOPICS

== and equals()

- == is an operator.
- equals is a method defined in the Object class
- **Default implementation** of equals() class provided by java.lang.Object **compares memory location and only** return true if two reference variable are pointing to same memory location i.e. essentially they are same object.
- == checks:
  - For reference types→ if two objects have the same address in the memory
  - For primitive types→checks if they have the same value.

For Strings

- equals() is overridden.
- equals( )method compares the characters inside a String object.
- == operator compares two object references to see whether they refer to the same instance.
- // equals() vs ==
- class EqualsNotEqualTo {

```
    public static void main(String args[]) {
      String s1 = "Hello";
      String s2 = new String(s1);
      System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
      System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}

//Hello equals Hello -> true
  Hello == Hello -> false
```

- In Java, when the **“==”** operator is used to compare 2 objects, it checks to see if the objects refer to the same place in memory.
- String obj1 = new String("xyz");
  String obj2 = new String("xyz");
  if(obj1 == obj2)
  System.out.println("obj1==obj2 is TRUE");
  else
  System.out.println("obj1==obj2 is FALSE");
- Even though the strings have the same exact characters (“xyz”), The code above will actually output: obj1==obj2 is FALSE
- Java String class actually overrides the default **equals()** implementation in the Object class – and it overrides the method so that it checks only the values of the strings, not their locations in memory.

- This means that if you call the equals() method to compare 2 String objects, then as long as the actual sequence of characters is equal, both objects are considered equal.
- String obj1 = new String("xyz");

```
String obj2 = new String("xyz");
if(obj1.equals(obj2))
    System.out.printlln("obj1==obj2 is TRUE");
else System.out.println("obj1==obj2 is FALSE");
```

This code will output the following:

- obj1==obj2 is TRUE