Objects and Classes

Introduction to Java

Alan Hohn
Alan.M.Hohn@lmco.com

25 July 2013

Contents

Course Contents

- Getting started writing Java programs (last time)
- Java programming language basics (today is 1 of 4 sessions)
- Packaging Java programs (1 session)
- Core library features (6 sessions)
- Java user interfaces (2 sessions)

Java Portability

- A key motivation for Java is portability (ability to run on different platforms without recompiling)
- This is accomplished by compiling for a "virtual machine" with its own instruction set
  - Known, appropriately, as the Java Virtual Machine (JVM)
  - JVM instructions are called "bytecode"
  - Looks like assembly / machine language, but with added features like virtual function calls
- The JVM provides a way to run bytecode on a specific operating system / machine instruction set
- Generally, the same bytecode can be run unmodified on any JVM

Our Basic Java Example

```
package org.anvard.introtojava;

// Classes in other packages that we need
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class HelloName {

  public static void main(String[] args)
      throws Exception {
    System.out.print("What is your name? ");
    // 1 statement, 3 objects
    String name = new BufferedReader(
        new InputStreamReader(System.in)).readLine();
    if (name.length() > 10) {
      System.out.println("You have a long name.");
    }
    // Smart concatenation, but no operator overloading
    System.out.println("Hello, " + name);
  }
}
```

Java Memory Management

- That program raises questions about memory management
  - We used the `new` keyword to make some objects
  - We didn't "free" those objects, set them to `null`, or otherwise worry about them
- In Java, new objects are allocated on the "heap"
  - The JVM manages the heap
  - When the heap (or part of it) gets full, the JVM does "garbage collection"
  - This means identifying objects that are no longer referenced from live code and freeing the memory
- Object allocation happens all the time in Java
  - Most objects are short-lived
  - For example, the `readLine` method and the + operator both instantiated new string objects
  - Modern JVMs are optimized for lots of short-lived objects, so this is surprisingly performant

This Time

- Java Objects and Classes
- Brief overview of Object-Oriented Programming
- Distinction between primitive types and objects
- Distinction between static and instance variables

OOP for Programmers

- Every "new" idea in Object-Oriented Programming (OOP) is not really new
- The basic design purpose behind OOP is encapsulation
  - Information should be held in the smallest possible scope
  - Simplifies maintenance, analysis of behavior, and reuse
- OOP takes the `record` or `struct` and adds behavior to it
  - Keep the code that operates on a piece of data together with the data
  - Control the scope of both code and data so other code can only access through a defined interface
- OOP lets us play other tricks that we will see later, but the basic idea is still improved encapsulation, because the data and the code that operates on it "live" inside the same construct

Objects and Classes in Java

- (Almost) everything in Java is an object
- All the code we write will be in a "class"
  - A class defines what an object "looks like": its unique name, its behavior, what data it stores
  - There can be many objects that have the same class; each object is known as an "instance" of the class
  - All share the same behavior, but have (mostly) independent data
- Classes have "fields" that hold data, and "methods" that specify behavior

Simple Class

```
package org.anvard.introtojava.designpatterns;

import java.io.Serializable;

@SuppressWarnings("serial") // Annotation
public class Person implements Serializable {

  private Integer id; // Field
  private String theFirstName;
  private String lastName;

  public Person() {} // Constructor

  public Person(Integer id, String aFirstName,
      String lastName) {
    this.id = id;
    this.theFirstName = aFirstName;
    this.lastName = lastName;
  }
```

Simple Class (continued)

```
    public Integer getId() { // Method
      return id;
    }

    public void setId(Integer id) {
      this.id = id;
    }

    public String getFirstName() {
      return theFirstName;
    }

    public void setFirstName(String aFirstName) {
      this.theFirstName = aFirstName;
    }

    ...

}
```

Where Does a Class Reside?

- When a class is *loaded*, its `.class` file is read into memory in the permanent generation (PermGen)
- When an object is *instantiated* (e.g. `Person p = new Person()`), the JVM allocates space on the heap for its *instance data* (fields)
- The class stays in memory for the life of the JVM
- The instance stays on the heap until it is garbage collected

Object References

- An object can refer to another object, but it *does not* keep its own copy of that object's data
  - All objects are instantiated on the heap
  - A field just contains a *reference* to the other object
  - The `this` keyword is a reference to the "current" instance
- Object references are used for parameters, too
  - Java is pass-by-value, but what is passed in the case of an object parameter is always a reference to the object (similar to a pointer)
  - Only the object reference lives on the stack; the actual object is on the heap
  - Objects modified inside methods retain their modifications when the method returns[1]

[1] Except for primitive wrapper classes, see below

Primitives

```
private Integer id;
```

- This is an object reference that is an instance of `java.lang.Integer`, a class that is part of the JVM
- This means it is allocated on the heap and must eventually be garbage collected
- It also means it is allowed to be `null`, which can be useful in some cases

```
private int id2;
```

- This is a simple integer called a primitive
- It is more efficient in storage, but it cannot be null and does not have fields or methods
- The types of primitives are `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`
- All primitives in Java are signed

Primitives Example

```
package org.anvard.introtojava;

public class PrimitivesExample {

  private int i; // Default initial value 0

  private Integer objectI; // Default null

  public void setObjectI (Integer newValue) {
    objectI = newValue;
  }

  public void setI (int newValue) {
    i = newValue;
    newValue += 50;
  }
```

Primitives Example

```
    public static void main(String[] args) {
      PrimitivesExample ex = new PrimitivesExample();
      System.out.println("i: " + ex.i);
      System.out.println("objectI: " + ex.objectI);
      ex.objectI = new Integer(5);
      System.out.println("objectI: " +
          ex.objectI.toString()); // Method call
      ex.setObjectI(10);
      System.out.println("objectI: " +
          ex.objectI.toString());
      ex.setI(new Integer(20));
      System.out.println("i: " + ex.i);
      int val = 30;
      ex.setI(val);
      System.out.println("i: " + ex.i);
      System.out.println("val: " + val);
    }
}
```

Primitives Example

```
i: 0
objectI: null
objectI: 5
objectI: 10
i: 20
i: 30
val: 30          <-- Note no change in 'val'
```

Strings

- Strings are instances of `java.lang.String`
- The actual characters of the string typically exist on the heap
- Strings are immutable
- Modifying a string typically results in memory allocation for a new string
  - This has important implications for performance, discussed later
  - One exception is *string literals*
  - These are allocated in a single pool and reused

```
String a = "abc"; // a points to a string in the pool
String b = a; // Reuse: b points to the same string in the...
     pool
String c = a + b; // Brand new string "abcabc" allocated ...
    on the heap
```

Primitive Wrapper Classes

- Each primitive has a corresponding "wrapper" class (e.g. int and java.lang.Integer)
- These are objects: they exist on the heap, they can be null, they have methods
- However, they are also immutable
  - Changing the 'value' results in changing the instance to point to a new value
  - Might be a reused value from a pool or a new one
- Java SE 5 and later automatically converts between primitives and the corresponding wrapper class (a.k.a. autoboxing)

```
Integer a = new Integer(5); // Always a new instance
a = 6; // Auto-boxing, illegal before Java SE 5
a = Integer.valueOf(6) // This is what really happens
```

Static and Instance Variables

- The `static` keyword marks a field or method as belonging to the class as a whole, not one particular instance
- Only one copy of a static field exists for the whole class
  - The static field exists in PermGen, but it might be an object reference to an object on the heap
  - While the instance points to a live object, that object is never garbage collected (because the class is in PermGen and never goes away)
- Static fields and methods can be accessed *and should be accessed* without referring to an object instance
- Static fields and methods must be handled carefully in multi-threaded environments

Static Example

```
package org.anvard.introtojava;

public class StaticExample {

  private static int a;
  private int b;

  public static void main(String[] args) {
    StaticExample y = new StaticExample();
    StaticExample z = new StaticExample();
    y.a = 1; // Bad form
    y.b = 2;
    z.a = 3; // Bad form
    z.b = 4;
    System.out.print(
      String.format("y.a: %d y.b: %d, z.a: %d, z.b: %d\n",
      y.a, y.b, z.a, z.b));
    StaticExample.a = 5; // Better form
  }
}
// Output: y.a: 3 y.b: 2, z.a: 3, z.b: 4
```

Next Time

- Java Control Flow
- Operators
- Exception Handling

Credit in LMPeople

Last Time: LMPeople Course Code: 071409ILT01
This Time: LMPeople Course Code: 071409ILT03