Data Structures in Java

Java Review

9/14/2015

Daniel Bauer
and Larry Stead

Java Review - Contents (1)

- Writing and Running Java Programs.
- Structure of a Program:
  - Packages, Files, Classes.
  - Static/Non-Static Methods, visibility modifiers, method calls.
  - Primitive Types and Wrappers.

Java Review - Contents (2)

- Object Oriented Programming
  - Classes and Instance Objects.
  - Instance variables, Methods.
  - Inheritance, Polymorphism.
  - Interfaces and Abstract Classes.
  - Nested classes:
    - Static nested classes.
    - Inner classes.

Compiling and Running Java Code

MyClass.java *Source File(s)*

```
$javac MyClass.java
```

or

```
$javac *.java
```

MyClass.class *"ByteCode" Class Files*

```
$java MyClass
```

MyClass must contain a *public static main* method.

Java Program

- A Java program consists of one or more class definitions (not to be confused with ".class" files). These classes can refer to other classes
    - Java comes with a huge library of classes
    - 3rd party classes can also be used
- Exactly one class must have a "main" method. This is where program execution will begin

Building more Complex Projects

- Class files are sometimes collected into “.jar” files(an archive format like ‘tar’)
- For non-trivial programs, running these tools by hand becomes too complex, and some kind of “system building” tool is used, like ‘make’, ‘ant’, ‘maven’, or ‘gradle’. These can be quite tedious to setup.
- 'sbt' for Scala.
- IDEs will also build your code for you.

Java Comments

- Three types of comments in Java
  - `//` - rest of line is ignored
  - `/*  ...  */` - inside is ignored
    - easy way to comment out a block of code
    - does NOT nest
  - `/**    ..  */` - treated as comments by compiler, but used by Javadoc documentation builder

Classes

- A class defines a blueprint/design for objects.
- Once a class is defined, any number of objects may be “instantiated” or “built” by using the “`new`” operator, much like a car factory can crank out an arbitrary number cars based on a design.
- Each class instance can hold data values unique to it (instance variables).
- Classes provide methods that operate on the instances.
- Class instances may also refer to data shared among all instances (class variables).

Methods

- Generalization of a function(sqrt vs random)
- A *class method (aka static method)* can access local and class (static) variables.
- A *instance method* can access local, class, and instance variables
- This lets an instance of a class bundle together state and actions
- A method is a series of statements.
- Statements are built out of Expressions
- Expressions are built from literals, variables, and operators

Access Control

- `package pkg; class Foo{}` - accessible within pkg package
- `package pkg; public class Foo{}` - accessible anywhere
- `public` member - accessible anywhere class is accessible
- `protected` member - accessible within the package and in subclasses
- `private` member - only accessible within the class
- will mainly use ‘`public`’ or ‘`private`’

Rect class

```
public class Rect{
   private int x,y,w,h;

   public Rect(int x, int y, int w, int h){
     this.x = x; this.y = y;
     this.w = w; this.h = h;
   }

   public int area(){
     return w * h;
   }

   public void move(int x, int y){
     this. x = x; this.y = y;
   }
}
```

‘Mutable’ vs ‘Immutable’

- Rect is ‘mutable’
  - the move() method can change the ‘saved state’ of a Rect instance
- an ‘immutable’ Rect would be

```
public class Rect{
     final x,y, w, h;
…
```

- When is ‘immutable’ desirable?

Java Types

- A type is a set of things, like 32 bit integers, 64 bit integers, unicode characters…
- There are two “types of type” in Java
    - Primitive
    - Reference
- Debatable - newer languages don’t do this
    - Why does Java?

Primitives vs References

- Primitives
  - 7 predefined by Java, not extendable
  - Also known as an “immediate” or “unboxed”
- References
  - A reference “refers” to an Object
  - An “array” defines a reference type
  - A “class” defines a reference type
  - New classes and arrays can be defined by the developer

Primitives and their Class "Wrappers"

- Wrapper classes have useful methods for the data type
- Can have arrays of primitives

Primitive	**Wrapper**
boolean	Boolean
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing and Unboxing

```
// automatically convert from ‘int’ to
‘Integer’
int x = 34
Integer y = x

// automatically convert from
‘Integer’ to ‘int’
Integer x = 45
int y = x

// be careful - this will generate a
lot of garbage

for(Integer j = 0; j<1000000; j++)
 …
```

Arrays

- Some differences from C arrays
  - int `a[10]` won’t work, must use “new” to get storage
  - int `a[] = new int[10]`
  - 2d arrays
    - int `a[][] = new int[5][6]` is not a linear array in memory - it is an “array of arrays”
    - a 2d array can be “rectangular” or “ragged”
- “`int[] a`”, and “`int a[]`” are both valid declarations

Arrays

- Can have arrays of Primitives and Object References
- Many useful array methods can be found on `java.util.Arrays`
  - If you print an array, you get a useless hex address
  - `Arrays.toString(a)` will generate a string representation of the array contents. Useful for debugging

Java Variables have a Type

- Some languages, like Python, have untyped variables
    - `x = 1`
    - `x = “string”`
- In Java
    - `int x = 1`
    - `x = “string”` - compiler error!!
- Tradeoff - typed variables are more verbose, but they enable more checking by the compiler. There is a large class of errors that can be made in Python that are impossible in Java
- Typed variables also make code more human readable

Java Memory Abstraction

- You can not generate a reference to an arbitrary location in memory
- Java variables either point to primitives or are references to objects.
- Unlike C/C++, Java does NOT support “pointer arithmetic”- no segfaults!

Java Automatic Memory Management Model

- In languages like C/C++ the programmer needs to reserve space for data (*malloc*) and explicitly *free* it when no longer needed.
- Java automatically allocates memory and automatically frees space occupied by objects no longer in use.
    - Pro: Much simpler, Con: Be careful. Still can have memory leaks.
- Things get stored in two places
    - Call Stack frame - “short term” (life of the stack frame) storage for primitives, memory freed by popping the stack.
        - Factorial example, recursion
    - Heap - “long term”(completely independent of stack frame where it was created) storage of reference types, memory freed by Garbage Collector

“null”

- Can’t get a “bad reference” in Java
- But, you can get a null

```
Foo f = new Foo();
// an instance method defined on Foo
f.fooMethod();
f = null;
// Will get a NullPointerException, which can be caught
// no SEGFAULT!!!
f.fooMethod();
```

Operators - Arithmetic

- +, -, *, /, %, ++, --
- division of two integers drops the remainder
  - % will yield remainder
- Oddly, there is no exponentiation operator
  - If you want to cube an integer, write j*j*j
  - For doubles, can use Math.pow
- + also does string concatenation
  - “abc” + “123” => “abc123”

Operators - Comparison

- ==, !=, <, <=, >, >=
- ==, !=
  - for primitives, == is true if the values are the same
  - for objects, == is true if the references are to the SAME object. for example new Rect(1, 2, 3, 4) == new Rect(1, 2, 3, 4) is false

Operators - Boolean

- AND &, OR |
    - evals both args
- Conditional, AND &&, OR ||
    - will stop eval at first opportunity
- Negation, !
- XOR - ^

Operators - Bit

- Bitwise Complement - ~
- Bitwise AND &, OR |, XOR ^
- Left Shift <<
- Right Shift, Signed >>, Unsigned >>>

Object Oriented Programing

Object Based Programming(OOP)

- Quick history
  - Sketchpad(1963) - astonishing program - invented objects, GUIs, CAD, and constraint systems
  - Simula(1967) - many of the features we see today
  - Smalltalk(1980) - very influential, full environment
  - Flavors(1982)/CLOS
  - C++(1985)
  - Java(1995)

Classes

- Usually a blueprint for instantiating objects, but not all classes are instantiated (also used in Java to group static methods.)
- Consists of some number of “members” - variables and methods(generalization of a function).
- Members can be defined on the class(static) or on the instances.
- Members are accessed with “.” operator
- Access to members may be restricted.

Encapsulation & Modularity

- a class ties together a bundle of related functionality
  - data
  - methods
  - access control
  - external interface
  - documentation

Code Reuse

- Ideally code is never written more than once, or duplicated
- Causes tremendous grief in large systems
- Want to promote maximal code reuse
- Sometimes an existing piece of functionality is close to what you want, but needs to be “tweaked”
- Example - I like rectangles, but I want a square

Special Case Classes

- java.lang.Math
  - Collection of functions and constants
  - All members static
  - Never instantiated
- Main class of a program (often only contains a `public static main` method.)

Special Case Class: All Instance Variables

- class Student {String uni; String first; String last;}
  - Student s = new Student();
  - s.first = “larry”;
  - System.out.println(s.last); // prints out null

General Class

- Package declaration, followed by arbitrary mix of class and instances members

```
package edu.columbia.cs.lstead;

class Vector2D {
  static int final dimensions = 2;
  double v[2] = new double[2];

  double mag() {
   double sum = 0;
   for(int j = 0; j<dimensions; j++)
     sum += v[j]*v[j];
   return Math.sqrt(sum);
  }
}
```

Method Overloading(works for Constructors too)

```
// same method name can be defined
multiple times with different arguments
// return type must be the same

class Foo {
  Foo() { ... }
  Foo(int z) { ... }
  Foo(int x, double y) { ... }

  int bar(int a, int b) { ... }
  int bar(int a) { ... }
  int bar() { ... }
 }
```

Constructors

- Can perform initializations at instantiation time
- Constructor have no return type(not even void)

```
Vector2D(double x, double y) {
  v[0] = x;
  v[1] = y;
}
```

Default Constructor

- `class Vector2D {...}` - gets default constructor
- `class Vector2D {Vector2D() { ... } }` - replaces default constructor
- `class Vector2D {Vector2D(double xy) { ... } }` - removes default constructor!

“this” and “super”

- Inside a instance method, ‘`this`’ refers to the object itself
- Inside a constructor, can “call” other constructors

  -
    ```
    Foo(int n) { this(n, 1)}
    Foo(int n, int m) {...}
    ```
- Inside a constructor, `super(...)` will call constructor of superclass. If used, `super(...)` must be the first statement in the constructor
- With a chain of “`super()`”, highest superclass is executed first

Inheritance

- If an existing class “almost” does what I want, I can “reuse” that class by using inheritance

```
class JustWhatIWant extends AlmostWhatIWant {
  /** new instance variables */
  String coolNewName;
  /** new method I want */
  void awesomeNewFunctionality(...) {...}
  /** completely replace superclass method */

  @Override
  void redoExistingMethod(...) {…}

  /** run superclass method, then do new stuff
  @Override
  void modifyExistingMethod(...) {
    super(...); …
  }
```

Abstract Classes and Methods

- An abstract class, or a class with abstract methods, cannot be instantiated
- Purpose is to partially constrain/guide any implementation class

```
abstract class Foo  {
  int cnt;
  // method must be implemented by subclass
  abstract void beGreat();
  // method can be used by subclass
  void incr() { cnt++; }
}
```

Polymorphism

- A subclass can “pass” for any of its superclasses

```
class Shape{abstract void draw(); …};
class Rect extends Shape{void draw(){…}; …};
class Circle extends Shape{void draw(){…}; …};

Shape s = new Rect();
s.draw();
s = new Circle();
s.draw();
```

java.lang.Object

- All classes without an “extends” declaration, “extend” Object
- important methods on Object
  - `equals(), hashCode()` - important for Collections
  - `toString()` - customize how object prints
  - `finalize()` - almost always a bad idea to use it

Object.toString() method

- Default way an object is printed is not very useful
- Showing some state very helpful in debugging

```
class Vector2D {

  String toString() {
    return “V2(“ + v[0] + “, “ + v[1]
    + “)”;
  }

}
```

java.lang.String

- Only object that has literal and operator
    - `String s = “foo” + “bar”`
- String is a “final” class - can not be extended
    - Strings are used extensively in Java, so efficiency is critical
- String is a ***immutable*** class - methods like `substring()`, `toUpperCase()` return a NEW string
- Many handy constructors and methods

Redefining equals() and hashCode()

- Often useful to redefine the equals() to implement a more general concept of equality. for example, might consider two objects equal if certain fields have the same value

  - sometimes it is useful to not compare all fields. fields not compared can be different, but objects are still “equals”

- if equals() is redefined, hashCode() must be redefined as well to be “consistent with equals”. if two objects are “equals”, they must have the same hashCode

Single vs Multiple Inheritance

- Multiple - C++, Python, CLOS
  - Drawbacks
    - considerably more complicated than Single
    - difficult to design correctly
    - efficiency issues
- Java - Single
  - Opted for simplicity

Interfaces

- Java side steps the multiple inheritance problem
- An interface specifies what an object must do, not what it is
- Integral part of the Collections framework
- Classes can “implement” any number of interfaces

Error System

- When a problem occurs, Java puts relevant info into a Throwable object, and “throws” it
- `java.lang.Throwable` - Top class
    - `java.lang.Error` - Bad stuff you can’t do anything about, JVM internal problems
    - `java.lang.Exception` - Things you can deal with
        - Attempt to open a file, but path is bad - could reprompt the user for path
        - Can’t make network connection - wait awhile and retry

Try Block

```
try {
  // code that might break
  …
} catch (ExceptionClass e) {
   System.out.println(e);
} finally {
   // always runs
}
```

Throws Declaration

- If checked exception is not handled in a try block, it must be thrown.
- Each method on the call stack will be search for an appropriate handler
- If no handler is found, program will use “default handler” - prints some info, and terminates program

```
void foo() throwsFileNotFoundException {
    ...
}
```

Exception Signaling

- You can define your own Exception class

```
class HorribleProblem extends Exception {
  // can save useful state on instance variables

  HorribleProblem(args…) {
    super(stringMessageToDisplay);
  }
}

...

// throw it
throw(new HorribleProblem(...));
```