Concurrent Object-Oriented Programming

Java (VRH 7.7,8.6)

Carlos Varela
RPI

October 29th, 2007

Partly adapted with permission from:
D. Hollinger, J.J. Johns, RPI

Overview

- Crash Course in Java
- Types and Classes
- Method Overloading; Multimethods
- Concurrency support

What is Java?

- A programming language.
  - As defined by Gosling, Joy, and Steele in the Java Language Specification
- A platform
  - A virtual machine (JVM) definition.
  - Runtime environments in diverse hardware.
- A class library
  - Standard APIs for GUI, data storage, processing, I/O, and networking.

Why Java?

- Java has substantial differences with C++
  - error handling (compiler support for exception handling checks)
  - no pointers (garbage collection)
  - threads are part of the language
  - dynamic class loading and secure sandbox execution for remote code
  - source code and bytecode-level portability

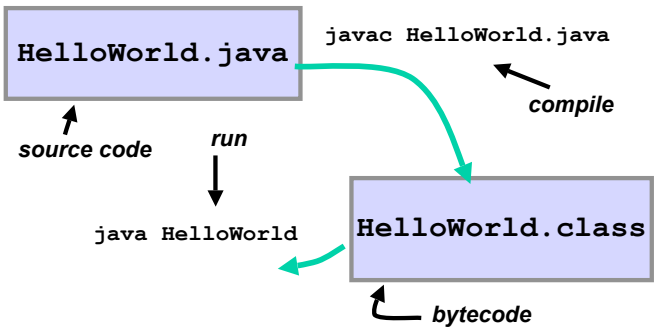
Java notes for C++ programmers

- (Almost) everything is an object.
  - Every object inherits from java.lang.Object
  - Primitive data types are similar: boolean is not an int.
- No code outside of class definitions
  - No global variables
- Single class inheritance
  - an additional kind of inheritance: multiple interface inheritance
- All classes are defined in `.java` files
  - one top level public class per file

First Program

```
public class HelloWorld {
  public static void main(String args[]) {
   System.out.println("Hello World");
  }
}
```

Compiling and Running



Java bytecode and interpreter

- Java bytecode is an intermediate representation of the program (stored in .class file)
- The Java interpreter starts up a new "Virtual Machine".
- The VM starts executing the user's class by running its `main()` method.

PATH and CLASSPATH

- PATH and CLASSPATH are environment variables that tell your operating system where to find programs.
- The *java_home/bin* directory should be in your $PATH
- If you are using any classes outside the java or javax packages, their locations must be included in your $CLASSPATH

The Language

- Data types
- Operators
- Control Structures
- Classes and Objects
- Packages

Java Primitive Data Types

- Primitive Data Types:
  - `boolean` **true** or **false**
  - `char` unicode (16 bits)
  - `byte` signed 8 bit integer
  - `short` signed 16 bit integer
  - `int` signed 32 bit integer
  - `long` signed 64 bit integer
  - `float`,`double` IEEE 754 floating point

Other Data Types

- *Reference types (composite)*
  - objects
  - arrays
- strings are supported by a built-in class named `String (java.lang.String)`
- string literals are supported by the language (as a special case).

Type Conversions

- Conversion between integer types and floating point types.
  - this includes **`char`**
- No automatic conversion from or to the type **`boolean`**.
- You can force conversions with a cast – same syntax as C/C++.

```
int i = (int) 1.345;
```


Operators

- Assignment: =, +=, -=, *=, …
- Numeric: +, -, *, /, %, ++, --, …
- Relational: ==. !=, <, >, <=, >=, …
- Boolean: &&, ||, !
- Bitwise: &, |, ^, ~, <<, >>, …

Just like C/C++!

Control Structures

Conditional statements:

if, if else, switch

Loop statements:

while, for, do

Exceptions

- Terminology:
  - *throw an exception*: signal that some condition (possibly an error) has occurred.
  - *catch an exception*: deal with the error.
- In Java, exception handling is necessary (forced by the compiler)!

Try/Catch/Finally

```
try {
  // code that can throw an exception
} catch (ExceptionType1 e1) {
  // code to handle the exception
} catch (ExceptionType2 e2) {
 // code to handle the exception
} catch (Exception e) {
 // code to handle other exceptions
} finally {
 // code to run after try or any catch
}
```


Exception Handling

- Exceptions take care of handling errors
  - instead of returning an error, some method calls will throw an exception.
- Can be dealt with at any point in the method invocation stack.
- Forces the programmer to be aware of what errors can occur and to deal with them.

Classes and Objects

- All Java statements appear within methods, and all methods are defined within classes.
- Instead of a "standard library", Java provides a set of packages with classes supported in all Java implementations.

Defining a Class

- One top level public class per .java file.
  - typically end up with many .java files for a single program.
  - One (at least) has a static public main() method.
- Class name must match the file name!
  - compiler/interpreter use class names to figure out what file name is.
- Package hierarchy should match directory structure.

Sample Class

(from Java in a Nutshell)

```
public class Point {
  public double x,y;
  public Point(double x, double y) {
      this.x = x; this.y=y;
  }
  public double distanceFromOrigin(){
      return Math.sqrt(x*x+y*y);
  }
}
```


Objects and **new**

You can declare a variable that can hold an object:

```
Point p;
```

but this doesn't create the object!

You have to use new:

```
Point p = new Point(3.1,2.4);
```


Using objects

- Just like C++:
  - `object.method()`
  - `object.field`
- BUT, never like this (no pointers!)
  - `object->method()`
  - `object->field`

Strings are special

- You can initialize Strings like this:

```
String blah = "I am a literal ";
```

- Or this (+ String operator):

```
String foo = "RPI" + " is great";
```

Arrays

- Arrays are supported as a second kind of reference type (objects are the other reference type).
- Although the way the language supports arrays is different than with C++, much of the syntax is compatible.
  - however, creating an array requires **new**

Array Examples

```
int x[] = new int[1000];

byte[] buff = new byte[256];

float[][] vals = new float[10][10];
```


Notes on Arrays

- index starts at 0.
- arrays can't shrink or grow.
  - e.g., use Vector instead.
- each element is initialized.
- array bounds checking (no overflow!)
  - ArrayIndexOutOfBoundsException
- Arrays have a *.length*

Array Example Code

```
int[] values;

int total=0;

for (int i=0;i<values.length;i++) {
  total += values[i];
}
```


Array Literals

- You can use array literals like C/C++:

```
int[] foo = {1,2,3,4,5};

String[] names = {"Joe", "Sam"};
```


Reference Types

- Objects and Arrays are *reference types*
- Primitive types are stored as values.
- Reference type variables are stored as references (pointers that are not first-class).
- There are significant differences!

Primitive vs. Reference Types

```
int x=3;
int y=x;
```

There are two copies of the value 3 in memory

```
Point p = new Point(2.3,4.2);
Point t = p;
```

There is only one Point object in memory!

```
Point p = new Point(2.3,4.2);
Point t = new Point(2.3,4.2);
```


Passing arguments to methods

- Primitive types are passed by value: the method gets a copy of the value. Changes won't show up in the caller.
- Reference types: the method gets a copy of the reference, so the method accesses the same object
  - However, the object reference is passed by value. Changing the reference does not change the outside object!

Example

```
int sum(int x, int y) {
  x=x+y;
  return x;
}

void increment(int[] a) {
  for (int i=0;i<a.length;i++) {
      a[i]++;
  }
}
```


Comparing Reference Types

- Comparison using == means:
  - "are the references the same?"
  - (do they refer to the same object?)
- Sometimes you just want to know if two objects/arrays are identical copies.
  - use the **`.equals()`** method
    - you need to write this for your own classes!

Packages

- You can organize a bunch of classes and interfaces into a *package*.
  - defines a namespace that contains all the classes.
- You need to use some java packages in your programs, e.g.
  - java.lang java.io, java.util

Importing classes and packages

- Instead of **`#include`**, you use **`import`**
- You don't have to import anything, but then you need to know the complete name (not just the class, the package).
  - if you **`import java.io.File`** you can use **`File`** objects.
  - If not – you need to use **`java.io.File`** inside the program.
- You need not import **`java.lang`** (imported by default).

Compiling

- Multiple Public classes:
  - need a file for each class.
  - Telling the compiler to compile the class with main().
    - automatically finds and compiles needed classes.

Access Control

- **Public** – everyone has access
- **Private** – no one outside this class has access
- **Protected** – subclasses have access
- **Default** – package-access

Final Modifier

- final class – cannot be subclassed
- final method – cannot be overriden
- final field – cannot have its value changed. Static final fields are compile time constants.
- final variable – cannot have its value changed

Static Modifier

- static method – a class method that can only be accessed through the class name, and does not have an implicit *this* reference.
- static field – A field that can only be accessed through the class name. There is only 1 field no matter how many instances of the class there are.

Classes vs Types

- Every object o has a class c.
- Is c the type of the object?
- Suppose d < c (d is a subclass of c) then an object o2 of class d can be used anywhere an object of class c is used (called *subclass polymorphism*).
- Therefore, an object o is of type c if and only if o's class d is either:
  1. = c, or
  2. < c

instanceof operator

- Dynamically checks for an object's type.

  o instanceof t
- tests whether the value of o has type t (whether the class of o is assignment compatible with reference type t).

Interfaces

- A Java interface lists a number of method signatures for methods that need to be implemented by any class that "implements" the interface.
- E.g.:

```
public interface Figure {
  public double getArea() {}
}
```


Interfaces

- A Java class that implements an interface must provide an implementation for all the methods in the interface.
- E.g.:

```
public class Point implements Figure {
  ...
  public double getArea() { return 0.0 }
}
```


Multiple Interface Inheritance

- A Java class may implement more than one interface
- E.g.:

```
public class Circle implements Figure, Fillable {
  ...
  public double getArea() {
    return Math.PI * radius * radius;
  }
  public void fill(Color c) {…}
}
```


Using Interfaces as Types

- The Java language allows the usage of interfaces as types for polymorphism. E.g., it knows that any object of a class that implements the Figure interface will have a getArea() method:

```
public double totalArea(Figure[] figures) {
    // sum everything up
       double total=0.0;
       for (int i=0;i<figures.length;i++) {
           total += figures[i].getArea();
       }
       return total;
  }
}
```


Method Overloading

- In a statically typed language, a method can be overloaded by taking arguments of different types.
- E.g.:

```
public int m(Circle c){ return 1;}
public int m(String s){ return 2;}
```

- The return type cannot be overloaded.
- The types can be related, e.g:

```
public int m(Object o){ return 1;}
public int m(String s){ return 2;}
```


Method Dispatching and Multimethods

- Which method gets dispatched can be decided at compile-time based on declared argument types information (Java), or at run-time with multi-methods (Smalltalk, SALSA).

```
public int m(Object o){ return 1;}
public int m(String s){ return 2;}
```

```
Object o = new Object();
String s = new String("hi");
Object os = new String("foo");
m(o);   // returns 1
m(s);   // returns 2
m(os);  // Static dispatch
        // returns 1; (Java)
        // Dynamic dispatch
        // returns 2. (SALSA)
```

Concurrent Programming in Java

- Java is multi-threaded.
- Two ways to create new threads:
  - Extend java.lang.Thread
    - Overwrite "run()" method.
  - Implement Runnable interface
    - Include a "run()" method in your class.
- Starting a thread
  - new MyThread().start();
  - new Thread(runnable).start();

The synchronized Statement

- To ensure only one thread can run a block of code, use synchronized:

```
synchronized ( object ) {
   // critical code here
}
```

- Every object contains an internal lock for synchronization.

synchronized as a modifier

- You can also declare a method as synchronized:

```
synchronized int blah(String x) {
 // blah blah blah
}
```

equivalent to:

```
int blah(String x) {
  synchronized (this) {
 // blah blah blah
  }
}
```


Concurrency and state are tough when used together

- Execution consists of multiple threads, all executing independently and all using shared memory
- Because of interleaving semantics, execution happens as if there was one global order of operations
- Assume two threads and each thread does k operations. Then the total number of possible interleavings is $\binom{2k}{k}$ This is exponential in k.
- One can program by reasoning on all possible interleavings, but this is extremely hard. What do we do?

Exercises

62. Do Java and C++ object abstractions completely encapsulate internal state? If so, how? If not, why?
63. Do Java and C++ enable static access to methods defined in classes arbitrarily high in the inheritance hierarchy? If so, how? If not, why?
64. Do Java and C++ allow multiple inheritance? If so, how? If not, why?
65. *Write, compile and execute a Java program in your laptop.