

Crash Course in Java

Based on notes from D. Hollinger

Based in part on notes from J.J. Johns

also: Java in a Nutshell

Java Network Programming and
Distributed Computing

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?

- Network Programming in Java is very different than in C/C++
 - much more language support
 - error handling
 - no pointers! (garbage collection)
 - threads are part of the language.
 - some support for common application level protocols (HTTP).
 - dynamic class loading and secure sandbox execution for remote code.
 - source code and bytecode-level portability.

Java notes for C++ programmers

- Everything is an object.
 - Every object inherits from `java.lang.Object`
- No code outside of class definition!
 - No global variables.
- Single inheritance
 - an additional kind of inheritance: interfaces
- All classes are defined in `.java` files
 - one top level public class per file

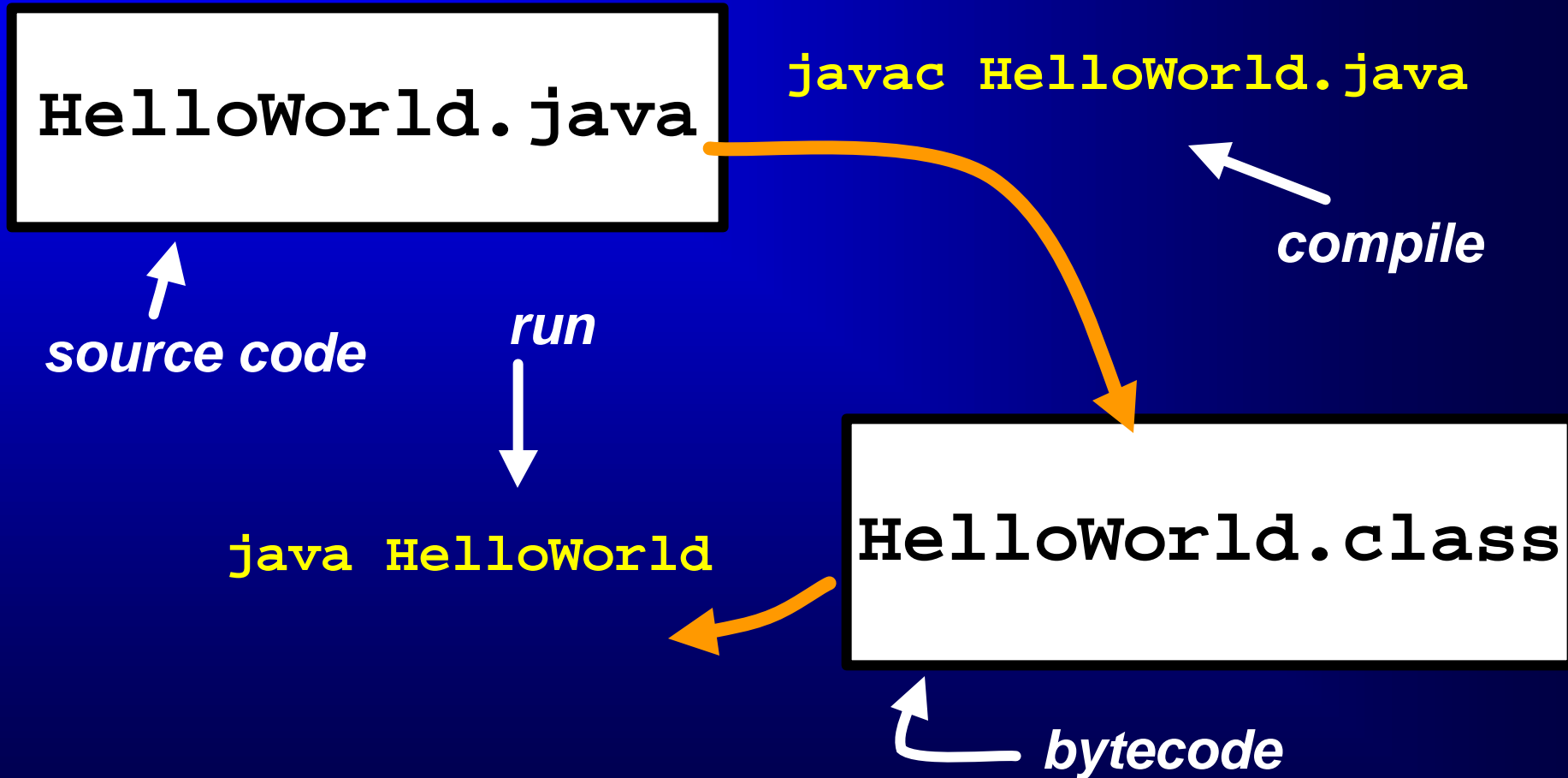
More for C++ folks

- Syntax is similar (control structures are very similar).
- Primitive data types similar
 - bool is not an int.
- To print to stdout:
 - `System.out.println();`

First Program

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

Compiling and Running



Java bytecode and interpreter

- bytecode is an intermediate representation of the program (class).
- The Java interpreter starts up a new “Virtual Machine”.
- The VM starts executing the users class by running it's `main()` method.

PATH and CLASSPATH

- The *java_home/bin* directory is in your \$PATH
- If you are using any classes outside the java or javax package, their locations are included in your \$CLASSPATH

The Language

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

Java 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

not an int!

Other Data Types

- *Reference types (composite)*
 - classes
 - arrays
- strings are supported by a built-in class named `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

- More of what you expect:
conditional: if, if else, switch
loop: while, for, do
break and continue (but a little different than with C/C++).

Exceptions

- Terminology:
 - *throw an exception*: signal that some condition (possibly an error) has occurred.
 - *catch an exception*: deal with the error (or whatever).
- 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.

Concurrent Programming

- Java is multithreaded!
 - threads are easy to use.
- 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

- Java is multithreaded!
 - threads are easy to use.
- Instead of mutex, use synchronized:

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

synchronized as a modifier

- You can also declare a method as synchronized:

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

Classes and Objects

- “All Java statements appear within methods, and all methods are defined within classes”.
- Java classes are very similar to C++ classes (same concepts).
- Instead of a “standard library”, Java provides a lot of Class 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.

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);
```

there are other ways to create objects...

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 = "I love " + "RPI";
```

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[][] mvals = 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<value.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 we can't mess with).
- 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: 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, the method accesses the same 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
 - 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` objects.

Sample Code

- `Sum.java`: reads command line args, converts to integer, sums them up and prints the result.
- `Sum1.java`: Same idea, this one creates a `Sum1` object, the constructor then does the work (instead of `main`).

More Samples

- Multiple Public classes:
 - need a file for each class.
 - Tell the compiler to compile the class with main().
 - automatically finds and compiles needed classes.
- Circle.java, CircleTest.java, Point.java, Threads.java, ExceptionTest.java