Java – Introduction (Part I)

Kurt Schmidt

Dept. of Computer Science, Drexel University

December 26, 2016

These notes are intended for students familiar with C++

Originally from Bruce Char & Vera Zaychik

Intro

Java is Object-Oriented

- Not *purely*, like Ruby, Smalltalk, Eiffel
  - More strictly than C++
  - But not everything is an object (the source of several of its awkward features)
- Everything belongs to a class
  - No global variables, nor functions
- Very portable
  - Programs run in a virtual machine (the *JVM*)

Similarities to C++

- Built-in (primitive) types:

  `boolean byte char int long float double`
- Same literals (where they exist)
  - Primitives are *not* instances of a class
- Branches

  `if if-else switch ?:`
- Loops

  `for while do-while break continue`
- Comments
  - `/* ...*/` – block comment
  - `// ...` – line comment

Similarities to C++

- `{}` – Defining and nesting scopes
- `;` separates statements
  - No longer used to end class definitions
- Exceptions
  - `throws` *errorlist*
  - `throw` *error*
  - `try{}`
  - `catch{}`
  - New one – `finally{}`

C++ Features Java Lacks

- No user-overloading of operators
  - No I/O operators, ≪ ≫
  - + − etc. don't work with `Integer`, `Float`
- No multiple inheritance[1]
  - (This is not nearly so bad as it seems)
- No default arguments (*why?!*)
- No explicit pointer notation

[1] We *do* have *mixins*; beyond the scope of these notes

Some Cool Things Java Has

- Interfaces
  - Rather than multiple inheritance
- Iterating `for` loop[1]

  `for( declaration : iterable )`
- Java won't accept `int`s, nor `elephant`s, where a `boolean` is expected
- All methods are virtual
- Inner classes (not this term)
- `/** ...*/` – Javadoc comment
- `finally`

[1] Which C++ now has

Other Differences

- No global anything
  - Everything is inside a class
- Single inheritance tree
  - `Object` is common ancestor of all classes
- Variables of type Object are references
  - Reference semantics apply, for assignment, passing into / out of functions
- Generics, rather than templates
  - Type-safe(r) containers
  - Can not hold primitive types
    - All primitive types have wrapper classes
    - Boxing and unboxing

Hello, World!

hello.java

```
/*
 * openjdk version "1.8.0_111"
 */

import java.io.* ;

public class hello
{
   public static String message = "Good morning" ;

   public static void main( String [] argv )
   {
      String user = System.getenv( "USER" ) ;

      System.out.println( message + ", " + user ) ;
   }
}
```

Java Classes

- Each class goes into a separate file[1]
  - Class `Foo` would be in a file named `Foo.java`
  - To compile:

```
$ javac hello.java
```

- Each class can have a static `main` method
  - To run, tell the jvm which class to start in (who's `main`):

```
$ java hello
```

[1] This isn't quite accurate, but, a fine place to start

The Java Classpath

- A list of directories (and jarfiles) where the JVM looks for (non-system) classes
  - The jarfile where a class is located
  - The directory where a class in unnamed package is located
  - The directory containing the root package
- Its use is much like the `PATH` environment variable
- By default, contains only the current directory
  - Classes without package specifiers in the same directory are in the same package

The Java Classpath

The classpath may be specified:

- With the `-classpath` (`-cp`) option, when invoking the JDK:

```
$ java -cp '/usr/share/java/junit4.jar:.' FooTest
```

- You can set (and export) `CLASSPATH` (less preferred)
- Inside the Ant file (coming soon)

Reference vs. Copy

Semantics – Objects vs. Primitives

- Source of many of Java's woes
- Variables of any primitive type use copy semantics, as in C
- Assignment and equality work as expected:
  - Assign the value stored in `b` to `a`

    ```
    a = b
    ```

  - Compare values stored in 2 different variables

    ```
    a == b
    ```

References

- If a variable is of type `Object` (an instance of some class that inherits from `Object`), it is a *reference*
  - Stores the object's location in memory
  - A pointer, without the notation
- Assignment and equality have different meaning:
  - Assign two references to the same object

    ```
    a = b
    ```

  - Check if both references refer to the same object

    ```
    a == b
    ```

References (cont.)

- Use `clone` method to create a copy of an object
  - Only if object implements `Cloneable`
  - `clone` is broken-ish
  - Copy constructor doesn't work with generics
- Use `equals` method to see if two distinct objects are equivalent
  - Inherited from `Object`
  - Works fine
  - Should be overloaded if object, in turn, contains references to other objects
- Deep vs. shallow copy, comparison

Strings

`String` Class

- The only built-in non-primitive (I think)
  - (Okay, maybe `Object`)
  - Compiler knows about it
  - Compiler wraps string literals in a `String` object
- `String`s are *immutable*
  - There are other classes for efficient string processing
- Concatenation
  - + works with `String`s
  - Primitives are *coerced* into `String`s

Arrays

Primitive Arrays

- Much like C arrays
  - Allocated from the heap:

    ```
    int [] ia = new int[ 20 ] ;
    int [] ja = { 12, 8, 392 } ;
    ```
  - Must be resized manually
  - Use reference semantics
- A single, final attribute, `length`
- Indexed using `[]`

Array Example


```
import java.io.*;
public class example2 {
  public static void main( String [] argv ) {
     int sum = 0;
     int [] temps = { 65, 87, 72, 75 };
     for( int i=0; i<temps.length; ++i )
        sum += temps[i];
     System.out.print( "# of samples: " + temps.length );
     System.out.println( ", avg: " + sum/temps.length );
  } // main
} // class example2
```

Resizing an Array


```java
public class arrEg {
  public static int[] resize( int [] a, int newsize ) {
    int [] rv = new int[newsize] ;

    for( int i=0; i<a.length; ++i )
      rv[i] = a[i] ;
    return rv ;
  }

  public static void main( String[] args ) throws Exception {
    int [] a = { 74, 011, 23, 0xff };
      // Want to add more items. Get bigger array
    int [] t = resize( a, 2*a.length ) ;
    a = t; // the old array is now marked for deletion
    t = null;
    a[4] = 47 ;
    for( int i : a )
      System.out.print( i + ", " ) ;
  } // main
} // class arrEg
```

Iterating `for` Loop

- Works with primitive arrays, and any generic, `Iterable` container

```
int [] temps = { 65, 87, 72, 75 } ;
for( int i : temps )
   sum += i ;
```

- An example of boxing and unboxing

```
import java.util.ArrayList ;

public class al {
  static public void main( String [] args ) {
    ArrayList<Integer> v = new ArrayList<Integer>( ) ;
    v.add( 72 ) ;
    ...
    for( int i : v )
      ... ;
  } // main
} // class al
```

Classes

Java Classes

- Each class needs an access modifier
  - Default is package
- Each member takes an access modifier
  - Default is package[1]

Default Packages

Without an implicit package specificier, all classes in the same directory are in the same package.

- All methods are virtual
- Static attributes can be initialised at declaration

[1] Please, no explicit `package` declarations this term

`final` Modifier

- Attributes:
  - Must be initialised
  - If a primitive type, value can not be changed
  - If a reference to an `Object`, the *reference* may not be changed
    - But the object referenced may still be modified
- `final` methods may not be overriden
- `final` classes may not be extended

Static Attributes

- Also called a *class attribute*
- A single variable, shared by all instances of the class
  - Don't need an instance to access
- Consider the interest rate on `SavingsAcct` class
  - Each instance would have its own account number, etc.
  - Each would *share* today's interest rate:

```
SavingsAcct.rate
```

- `static public` is how we implement system-wide globals in Java
  - Consider `Math.Pi`

Static Methods

- Can not access instance data
- Don't need an instance to access
  - Recall our *SavingsAcct* class

```
SavingsAcct.getRate() ;
```

- `static public` is how we implement "global" library functions
  - Consider `Math.sin()`, `Math.log()`, `Math.floor()`, etc.

Access Modifiers

`private` No access outside class

default package – Only to classes in package

`protected` Access given to classes in package, subclasses

`public` All have access

- A class may be either *public*, or default
- Class members may have any modifier
- Note, member modifiers can *not* grant accesses not granted by class

`main` Method

- Every class may have a `main`

  ```
  public static void main( String [] args )
  ```
- Entry point, potentially
  - Class to start in must be identified to JRE
  - No instances yet, so, must be `static`
- No return value from `main`
- Use static methods for a traditional C-like program

Command-line Arguments

- Single parameter to `main`
- Java array of `String`s
- Argument 0 is *not* the name of the program, class, etc.

```
public static void main( String [] args )
{
   for( int i=0; i<args.length; ++i )
      System.out.printf( "%d %s\n", i, args[i] ) ;
}
```

Constructors

- Similar to C++
  - No return type
- If none is provided, default c'tor is used
- If *c'tor* is provided, default is no longer implicitly available
- There is no destructor
  - See `close()` and `finalize()`
    - `finalize()` is unreliable
    - Should not be used as a destructor
    - Output streams should be closed explicitly
    - See the try-resource syntax, since Java 1.7

super

- Use it to call one of the parent class' c'tor, to initialise the parent sub-object

  ```
  super( name ) ;
  ```

  - Place it as first line in child's c'tor
  - If absent, parent's default c'tor is called
- Use it to call parent's version of overridden method

```
public class Professor extends Person {
  public String toString()
  { return "Prof. " + super.toString() ; }
  ...
}
```

Importing Classes from Packages

- You can import individual classes:

```
import java.io.ObjectInputStream ;
```

- Import all classes in a particular package
  - E.g., all classes in the `java.net` package:

```
import java.net.* ;
```

I/O

Input / Output

- Messier than usual
  - Many classes
  - Note the difference between reading ASCII input vs. raw data
  - Choose the right one for the job
- Always call `close` explicitly
  - Especially output streams
  - No guarantee that `finalize` will be called
  - Since Java 1.7, see the try-with-resources syntax

Text Input – `Scanner`

Very handy class for formatted ASCII input

- Can be wrapped around:
  - A `String`

```
Scanner src = new Scanner( "Yo, parse *this*.\nAnd this." ) ;
```

  - A `File` object

```
Scanner src = new Scanner( new FileReader( "data.csv" )) ;
```

  - `InputStream`

```
Scanner src = new Scanner( System.in ) ;
```

- `Scanner` will open a file for you

```
Scanner src = new Scanner( new Path( "../Files/input.src" )) ;
```

Scanner Examples

- Can read by lines:

```
while( src.hasNextLine() )
   l = src.nextLine() ;
```

- By words:

```
String s ;
while( src.hasNext() )
   s = src.next() ;
```

- Or, by tokens, over the primitive types

```
int i ;
while( src.hasNextInt() )
   i = src.nextInt() ;
```

Use Scanner to Parse a String

- By default, token delimiter is white space
  - Can be changed
  - Can use a regular expression (`Pattern`) to describe the delimeter

```
String s = "Parse-*-this-*-up" ;
Scanner src = new Scanner( s ).useDelimiter( "-*-" ) ;
ArrayList<String> fields = new ArrayList<String>() ;

while( src.hasNext() )
   fields.add( src.getNext() ) ;
```

Writing Text – `PrintWriter`

- Use `PrintWriter` to write text
- Can be wrapped around a `File` or an `OutputStream`, `Writer`, or just a filename
- `print` , `println` , `printf` provide familiar behavior
- There are a handful of `write` methods, for writing characters and strings
- Nothing for writing a `byte`

Reading Raw Data

Use `BufferedInputStream` or `FileInputStream` to read Bytes. `ByteArrayInputStream` also looks promising.

```
import java.io.BufferedInputStream ;
import java.io.IOException ;
public class readBytes {
  public static int wordLen = 8 ; // # of bytes to be read each time
  public static void main( String [] argv ) throws IOException
  {
    BufferedInputStream is = new BufferedInputStream( System.in ) ;
    byte [] buff = new byte[wordLen] ;
    int r ;
    while( (r=is.read(buff, 0, wordLen)) != -1 )
    {
      for( int i=0; i<r; ++i )
        System.out.printf( "%x ", buff[i] ) ;
      System.out.print( "\n" ) ;
    } // while
    is.close() ;
  } // main
} // class readBytes
```

Use `PrintStream` to Write Bytes (Binary Data)

- `System.out` and `System.err` are instances of `PrintStream`
- `printf` , `print` , `println` overloaded for all primitives
- Can be wrapped around a `File` or an `OutputStream`, or just a filename
- Uses `write` to write bytes

```
PrintStream of ;
if( argv.length == 0 )
   f = System.in ;
else
   f = new PrintStream( argv[0] ) ;
...
f.close() ;
```

`BufferedWriter`, `BufferedOutputStream`, `FileWriter`, Other Thoughts

- The `Print`*xxx* member functions do not throw exceptions
  - See `checkError`, `clearError`
- The `Buffered`*xxx* *do* throw exceptions
  - But, they lack the handy `print`, `printf` functions
  - I truly don't know why, nor which is preferrable
- Understand the difference between *text* and *binary data*
- Do *not* infer from the Java library that streams should be associated with binary data, while files should be text
  - You can send a text file to another computer
  - A disk file, say, `mp3`, is raw data
  - A file *is* a stream

Containers

Standard Library Containers

- Some standard, useful containers (today)
  - Note, most containers have multi-threaded analogs in the library
- These are *generic* containers
- Can only hold `Objects`, and descendants (no primitives)
- Many (but not all) implement the `Collection` interface
  - The others probably implement `Iterable`

Generics

- Like C++ templates (sorta)
- Allow containers to hold *particular* `Objects`
  - Makes code more type-safe
- Primitives are automatically *boxed* into appropriate objects (`Integer`, `Double`, etc.) when inserted
  - And unboxed when returned

Multi-Threaded Programs

Many of the following containers have thread-safe counterparts, which would run slower. Beyond the scope of this discussion.

The `Iterable<T>` Interface

- Any implementing class has these methods:
  - `iterator` – Returns an `Iterator` over the elements
  - `spliterator` – Returns a `Spliterator` over the elements[1]
  - `forEach( Consumer<? super T> action)` – Applies `action` to each element of the Iterable[2]

[1] An *early-binding, fail-fast* iterator

[2] See lambda forms

The `Collection<T>` Interface

- Inherited from the Iterable Interface
- Also provides the following (among others):
  - `add`, `addAll`
  - `contains`, `isEmpty`, `size`, `clear`
  - `remove`, `removeIf`, `retainAll`

The `Collections` Algorithms

- Many handy algorithms that work on objects that implement `Collection<T>`
  - Maybe[1]
  - Others need `T` to implement `Comparable`
- These containers apparently have some notion of indexing
  - (This doesn't imply constant-time access)
- Here are a few handy ones:
  - `replaceAll`, `swap`, `fill`, `frequency`, `reverse`, `shuffle`
  - `max`, `min`, `binarySearch`, `sort`

[1] Not all algorithms apply to all containers, even if they're a `Container`

ArrayList<T> – a Vector

- The preferred vector, these days
- Inherits from AbstractList
- Implemented interfaces include List<E>, Collection<E>, Iterable<E>
- Some useful methods:
  - add(T elem[, int index])
  - clear()
  - contains(Object elem)
  - get(int index)
  - set(int index, T elem)
  - size()
  - iterator()

LinkedList<T>

A doubly-linked list implementation of the `List` interface

- No constant-time access of elements
- Can modify anywhere in constant time
- Interfaces include `Collection<E>`, `Iterable<E>`, `Deque<E>`, `List<E>`, `Queue<E>`
  - Note, there is no stack interface
  - `Stack<E>` is a class, built on `Vector<E>`
  - Java gives me grey hair

`ArrayDeque<T>`

- Deque – doubly-ended queue
  - Pronounced "deck"
- Supports constant-time insert and delete at front, also
- Much like a vector
  - Indexed (constant-time access)
  - Modification of middle still linear operation
- Interfaces include `Collection<E>`, `Deque<E>` and `Queue<E>`
- Probably best choice for a queue
- Can easily be a stack

The `Set<T>` Interface

- Implemented by:
  - `HashSet<E>`, `LinkedHashSet<E>`, `TreeSet<E>`
- Behaviors include:
  - `contains`, `add`, `remove`

The Map<K,V> Interface

- Implemented by several classes, including:
  - `HashMap<E>`, `LinkedHashMap<E>`, `TreeMap<E>`
- `HashTable` is a child of `Dictionary`, and has been overtaken by the `AbstractMap` classes
- Behaviors include:
  - `clear`, `hasKey`, `hasValue`, `get`, `remove`