Java Primer

- Review of Java basics
- Primitive vs Reference Types
- Classes and Objects
- Class Hierarchies and Interfaces
- Exceptions
- Generics

Reading: Lambert and Osborne, Appendix A & Sections 1.2 and 2.1–2.7

1. Review of Java Basics

1.1 Primitive Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>default value</th>
<th>size</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
<td>8 bits</td>
<td>−128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
<td>16 bits</td>
<td>−2^15 to 2^15 − 1</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
<td>32 bits</td>
<td>−2^31 to 2^31 − 1</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
<td>64 bits</td>
<td>−2^63 to 2^63 − 1</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
<td>32 bits</td>
<td>?</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
<td>64 bits</td>
<td>?</td>
</tr>
<tr>
<td>char</td>
<td>'u0000'</td>
<td>16 bits</td>
<td>0 − 2^16</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
<td>?</td>
<td>{true, false}</td>
</tr>
</tbody>
</table>

1.2 Local Variables

**Scope**: block in which defined

```java
for (int i=0; i<4; i++) {
    // do something with i
}  
System.out.println(i);
```

Result?

1.3 Expressions

Built from variables, values, and **operators**.

- **arithmetic**: +, −, *, /, %, ...
- **logical**: &&, ||, !, ...
- **relational**: =, !=, <, >, <=, >=, ==, !=, equals
- **instanceOf**
- **ternary**: ? (e.g. x > 0 ? x : −x)
1.4 Control Statements

if and if-else

\[
\begin{align*}
\text{if } & (<\text{boolean expression}>) \\
& <\text{statement}> \\
\text{if } & (<\text{boolean expression}>) \\
& <\text{statement}> \\
\text{else} & \\
& <\text{statement}>
\end{align*}
\]

where \(<\text{statement}>\) is a single or compound statement.

Example

\[
\begin{align*}
\text{for } & (\text{int } i=0; i<4; i++) \ \text{System.out.println}(i); \\
0 & \\
1 & \\
2 & \\
3 & \\
\text{for } & (\text{String } s=\text{"\text{"aaaa\text{"); s=s+\text{"a\text{") \\
& \ \text{System.out.println}(s.length());}
\end{align*}
\]

In Java 5 we also have an enhanced for loop:

\[
\begin{align*}
\text{int} & [\] \ \text{array} = \{0,2,4\}; \\
\text{for } & (\text{int } i : \ \text{array}) \\
& \ \text{System.out.println}(\text{"i is: "} + i);
\end{align*}
\]

while, do-while, and for

\[
\begin{align*}
\text{while } & (<\text{boolean expression}>) \\
& <\text{statement}> \\
\text{do} & \\
& <\text{statement}> \\
\text{while } & (<\text{boolean expression}>) \\
\text{for } & (<\text{initialiser list}; <\text{termination list}; <\text{update list}> \\
& <\text{statement}>
\end{align*}
\]

Arrays

Declaration

\[
\begin{align*}
<\text{type}> & [...] <\text{name}>; \\
<\text{type}> & [...] <\text{name}>
\end{align*}
\]

Instantiation

\[
\begin{align*}
<\text{name}> & = \text{new } <\text{type}>[<\text{int-exp}>]; \\
<\text{name}> & = \text{new } <\text{type}>[<\text{int-exp}>]...[<\text{int-exp}>];
\end{align*}
\]

Example

\[
\begin{align*}
\text{int} & [] \ \text{matrixArray}; \\
\text{matrixArray} & = \text{new int}[\text{rows}][\text{columns}]; \\
\text{int} & [] \ \text{array} = \{0,2,4\};
\end{align*}
\]
1.5 Methods

Methods have the form (ignoring access modifiers for the moment)

<return type> <name> (<parameter list>) {
    <local data declarations and statements>
}

Example

void set (int i, int j, int value) {
    matrixArray[i][j]=value;
}

int get (int i, int j) {return matrixArray[i][j];}

Parameters are passed by value:

// a method...
void increment (int i) {i++;}

// some code that calls it...
i=7;
increment(i);
System.out.println(i);

Result?

2. Primitive Types vs Reference Types

Primitive types

• fixed size
• size doesn’t change with reassignment
⇒ store value alongside variable name

Reference types (eg. Arrays, Strings, Objects)

• size may not be known in advance
• size may change with reassignment
⇒ store address alongside variable name

integer i = 15;

Object b = new Object();

The variable holds a pointer or reference to the object’s data
⇒ reference types
```java
int[] a = {0, 1, 2, 3};
int[] b = a;
b[0]++;
System.out.println(a[0]);

Result?

// a method...
void incrementAll (int[] a) {
    for (int i=0; i<a.length; i++) a[i]++;
}

// some code that calls it...
int[] b = {0, 1, 2, 3};
incrementAll(b);
System.out.println(b[0]);

Result?

class Box {
    // instance variables
    double width, length, height;

    // constructor method
    Box (double w, double l, double h) {
        width = w;
        length = l;
        height = h;
    }

    // additional method
    double volume () {return width * length * height;}
}

3. Classes and Objects

3.1 What are they?

Aside from a few built-in types (arrays, strings, etc) all reference types are defined by a class.

A class is a chunk of software that defines a type, its attributes or instance variables (also known as member variables), and its methods...

3.2 Constructors

The runtime engine creates an object or instance of the class each time the new keyword is executed:

Box squareBox, rectangularBox;
...
squareBox = new Box(20, 20, 20);
rectangularBox = new Box(20, 30, 10);
### 3.3 Different Kinds of Methods

- **constructor** — tells the runtime engine how to initialise the object
- **accessor** — returns information about an object’s state without modifying the object
- **mutator** — changes the object’s state

### 3.4 Packages

A collection of related classes. E.g. `java.io`

In Java:
- must be in same directory
- directory name matches package name

Specifying your own package

```java
package myMaths;

class Matrix {
    ...
}
```

If you don’t specify a package Java will make a default package from all classes in the directory.

Using someone else’s package

```java
package myMaths;
import java.io.*;

class Matrix {
    ...
}
```

Note that `java.lang.*` is automatically imported.

### 3.5 Access Modifiers

Specify access to classes, variables, and methods.

- **public** — accessible by all
- **private** — access restricted to within class
- **(none)** — access restricted to within package
- **protected** — access to package and subclasses
3.6 The `static` Keyword

Used for methods and variables in classes that don’t create objects, or for variables shared by all instances of a class.

Example:

```java
public class MatrixTest {
    static Matrix m;
    public static void main (String[] args) {
        m = new Matrix(2,2);
        m.set(0,0,1);
        ...
    }
}
```

called class variables and class methods.

Also used for "constants".

Example:

```java
public class Matrix {
    static final int MAX_SIZE=100;
    private int[][] matrixArray;
    ...
}
```

Keyword `final` means the value cannot be changed at runtime.

4. Class Hierarchies

Classes can be built from, or extend other classes.

Example:

```java
public class Shape {
    private double xPos, yPos;
    public void moveTo (double xLoc, double yLoc) {
        xPos = xLoc;
        yPos = yLoc;
    }
    ...
}
```

(public more detail: see Lambert and Osborne, Section 2.5)
public class Circle extends Shape {
    private double radius;

    public double area () {
        return Math.PI * radius * radius;
    }
}

While we will not be using hierarchies extensively in this unit, we will be using some very important features of them...

1. Any superclass reference (variable) can hold and access a subclass object.

   Example:
   ```java
   public class ShapeTest {
       public static void main (String[] args) {
           Shape sh; // declare reference of type Shape
           sh = new Circle(); // hold a Circle object in sh
           sh.moveTo(2.0,3.0); // access a Shape method
           double a=sh.area(); // access a Circle method
           ...
       }
   }
   ```

2. All Java classes are (automatically) subclasses of Object

   Example:
   ```java
   Object holdsAnything;
   holdsAnything = new Circle();
   holdsAnything = new Rectangle();
   holdsAnything = new Shape();
   ```

   Example:
   ```java
   Object[] arrayOfAnythings = new Object[10];
   arrayOfAnythings[0] = new Circle();
   arrayOfAnythings[1] = new Rectangle();
   arrayOfAnythings[2] = new Shape();
   ```

Java provides wrappers for all primitives to allow them to be treated as Objects:

⇒ Character, Boolean, Integer, Float, ...

See the Java API for details.

Note: A new feature in Java 1.5 is autoboxing — automatic wrapping and unwrapping of primitives.

⇒ Compile time feature — doesn’t change what is “really” happening.
4.1 Casting

While a superclass variable can be assigned a subclass object, a subclass variable cannot be assigned an object held in a superclass, even if that object is a subclass object.

Example:

```java
Object o1 = new Object();    // OK
Object o2 = new Character('a'); // OK
Character c1 = new Character('a'); // OK
Character c2 = new Object();    // Error

o1 = c1;                      // OK
c1 = o1;                      // Error
```

In the last statement, even though `o1` is now “holding” something that was created as a `Character`, its reference (its class) is `Object`.

To get the “`Character`” back, we have to cast it back down the hierarchy:

```java
o1 = c1;                      // OK
c1 = (Character) o1;         // OK - casted back to `Character`
```

4.2 Object Oriented Programming in Java

Some object oriented features worth remembering are:

- **Abstraction**: the ability to treat different types of objects as a common type.
- **Polymorphism**: how one method can change its behavior in different classes.
- **Inheritance**: reusing methods and variables from super classes.
- **Encapsulation**: information hiding, and containing other classes.

To demonstrate these properties, let’s reconsider the Shape example. This time, we first define a class `Point`...

```java
public class Point {

    private double xPos, yPos;

    public Point(double x, double y){
        xPos = x;
        yPos = y;
    }

    public void moveTo (double xLoc, double yLoc) {
        xPos = xLoc;
        yPos = yLoc;
    }
}
```
...and use Point to define Shape. This is **encapsulation**.

```java
public abstract class Shape {
    private Point p;
    
    public Shape(double x, double y) {
        p = new Point(x, y);
    }
    
    public void moveTo(double xLoc, double yLoc) {
        p.moveTo(xLoc, yLoc)
    }
    
    public double area(); // an abstract method - more later
}
```

Circle inherits from Shape...

```java
public class Circle extends Shape {
    private double radius;
    
    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }
    
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

as does Rectangle. This demonstrates **inheritance**.

```java
public class Rectangle extends Shape {
    private double height;
    private double width;
    
    public Rectangle(double x, double y, double height, double width) {
        super(x, y);
        this.height = height;
        this.width = width;
    }
    
    public double area() {
        return width * height;
    }
}
```

We can now treat Rectangles and Circles as the more general type Shape:

**Example:**

```java
public class ShapeTest {
    
    public static void main (String[] args) {
        Shape[] sA = new Shape[2];
        sA[0] = new Circle(1,1,1);
        sA[1] = new Rectangle(1,1,1,1); // This is abstraction
        for(int i = 0; i < 2; i++)
            sA[i].moveTo(0,0);
        int totArea = 0;
        for(int i = 0; i < 2; i++)
            totArea += sA[i].area(); // This is polymorphism
    }
}
```
Note:

- We did not need to specify how a Shape’s area is calculated. This means that we are never able to construct just a Shape.
- We chose to encapsulate a Point, rather than inherit from it (a shape is not a point). Inheritance should be used sparingly. Always consider composition first.
- Once Circles and Rectangles can be treated as shapes we can have an array that contains both.
- The method area() was different for both shapes, but we did not need to cast. The Java virtual machine will determine which method is called.
- A good example of inheritance in the API is java.awt (e.g., a Window is a Container which is a Component which is an Object). However, in general inheritance hierarchies should be fairly shallow.

5. Interfaces and Abstract Classes

An interface:

- looks much like a class, but uses the keyword interface
- contains a list of method headers — name, list of parameters, return type (and exceptions)
- no method contents (they are called abstract, but abstract classes may have some methods specified)
- can only have constant variables declared
- no public/private necessary — they are implicitly public
- can implement multiple interfaces

Effectively, interfaces present all the OO advantages above, except inheritance.

Example:

```java
public interface Matrix {
    public void set (int i, int j, int value);
    public int get (int i, int j);
    public void transpose ();
}
```

Classes can implement an interface:

Implementation 1:

```java
public class MatrixReloaded implements Matrix {
    private int[][] matrixArray;
    public void transpose () {
        // do it one way
    }
    ...
}
```

Implementation 2:

```java
public class MatrixRevolutions implements Matrix {
    private int[][] somethingDifferent;
    public void transpose () {
        // do it yet another way
    }
}
```
Why use interfaces?

1. Can be used like a superclass:

Example:
```java
Matrix[] myMatrixHolder = new Matrix[10];
myMatrixHolder[0] = new MatrixReloaded(2,2);
myMatrixHolder[1] = new MatrixRevolutions(20,20);
...
myMatrixHolder[0] = myMatrixHolder[1];
```

2. Specifies the methods that any implementation must implement.

Example:
```java
Matrix[] myMatrixHolder = new Matrix[10];
myMatrixHolder[0] = new MatrixReloaded(2,2);
myMatrixHolder[1] = new MatrixRevolutions(20,20);
...
for (int i=0; i<10; i++)
    myMatrixHolder[i].transpose();
```

Note: this doesn’t mean the methods are implemented correctly.

This is an important software engineering facility

- follows on from Information Hiding in Topic 1
- allows independent development and maintenance of libraries and programs that use them
- will be used extensively in this unit to specify ADTs
- also used to add common functionality to all objects, e.g. `Serializable`, `Cloneable`

More examples — see the Java API

eg. the `Collection` interface, also `Runnable`, `Throwable`, `Iterable`, and `List`.

6. Exceptions

- special built-in classes
- used by Java to determine what to do when something goes wrong
- `thrown` by the Java virtual machine (JVM)
Example program

```java
int[] myArray = {0,1,2,3};
System.out.println("The last number is:");
System.out.println(myArray[4]);
```

Output

```
The last number is:
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
```
```
at Test.main(Test.java:31)
```
```
Process Test exited abnormally with code 1
```

We can throw exceptions ourselves.

```java
if (<condition>)
    throw new <exception type> (<message string>);
```

Example:

```java
double squareRoot (double x) {
    if (x < 0)
        throw new ArithmeticException("Can’t find square root of -ve number.");
    else {
        // calculate and return result
    }
}
```

Have a look for ArithmeticException in the Java API.

---

Two types of exceptions:

**checked** — most Java exceptions
  — must be caught by the method, or passed (thrown) to the calling method
**unchecked** — RuntimeException and its subclasses
  — don’t need to be handled by programmer (JVM will halt)

To catch an exception, we use the code:

```java
try {
    codeThatThrowsException();
} catch(Exception e) {
    codeThatDealsWithException(e);
}
```

For simplicity, we will primarily use unchecked exceptions in this unit.

---

Compiling and running Java

There are various ways to compile and run Java, but the command line is the most ubiquitous. The command:

```bash
> javac myClass.java
```

will create the file myClass.class in the current directory. The command:

```bash
> java myClass
```

will execute the main method of the class myClass.
Objects and Generic Data Structures

```java
/**
   * Block representation of a queue (of objects).
   */
public class QueueBlock {
    private Object[] items; // array of Objects
    private int first;
    private int last;

    public Object dequeue() throws Underflow { // returns an Object
        if (!isEmpty()) {
            Object a = items[first];
            first++;
            return a;
        }
        else throw new Underflow("dequeueing from empty queue");
    }
}
```

6.1 Wrappers

The above queue is able to hold any type of object — that is, an instance of any subclass of the class `Object`. (More accurately, it can hold any reference type.)

But there are some commonly used things that are not objects — the primitive types.

In order to use the queue with primitive types, they must be “wrapped” in an object.

Recall from Topic 4 that Java provides wrapper classes for all primitive types.

Autoboxing — Note for Java 1.5

Java 1.5 provides autoboxing and auto-unboxing — effectively, automatic wrapping and unwrapping done by the compiler.

```java
Integer i = 5;
int j = i;
```

However:

- Can lead to unintuitive behaviour. Eg:
  ```java
  Long w1 = 1000L;
  Long w2 = 1000L;
  if (w1 == w2) {
      // do something
  }
  ```
  may not work. Why?

- Can be slow. Eg. if a, b, c, d are Integers, then
  ```java
  d = a * b + c
  ```
  becomes
  ```java
  d.valueOf(a.intValue() * b.intValue() + c.intValue())
  ```

For more discussion see:

http://chaoticjava.com/posts/autoboxing-tips/
6.2 Casting

Recall that in Java we can assign "up" the hierarchy — a variable of some class (which we call its reference) can be assigned an object whose reference is a subclass.

However the converse is not true — a subclass variable cannot be assigned an object whose reference is a superclass, even if that object is a subclass object.

In order to assign back down the hierarchy, we must use casting.

This issue occurs more subtly when using ADTs. Recall our implementation of a queue...

```java
public class QueueBlock {
    private Object[] items; // array of Objects
    ...
    public Object dequeue() throws Underflow { // returns an Object
        if (!isEmpty()) {
            Object a = items[first];
            first++;
            return a;
        }
        else...
    }
}
```

Consider the calling program:

```java
QueueBlock q = new QueueBlock();
String s = "OK, I'm going in!";
q.enqueue(s); // put it in the queue
s = q.dequeue(); // get it back off ???
```

The last statement fails. Why?

The queue holds `Object`. Since `String` is a subclass of `Object`, the queue can hold a `String`, but its reference in the queue is `Object`. (Specifically, it is an element of an array of `Objects`.)

dequeue() then returns the "String" with reference `Object`.

The last statement therefore asks for something with reference `Object` (the superclass) to be assigned to a variable with reference `String` (the subclass), which is illegal.

We have to cast the `Object` back "down" the hierarchy:

```java
s = (String) q.dequeue(); // correct way to dequeue
```

6.3 Generics

Java 1.5 provides an alternative approach. Generics allow you to specify the type of a collection class:

```java
Stack<String> ss = new Stack<String>();
String s = "OK, I'm going in!";
ss.push(s);
s = ss.pop()
```

Like autoboxing, generics are handled by compiler rewrites — the compiler checks that the type is correct, and substitutes code to do the cast for you.
Writing Generic Classes

```java
/**
* A simple generic block stack for
* holding object of type E
**/
class Stack<E> {
    private Object[] block;
    private int size;
    public Stack(int size) {
        block = new Object[size];
    }
    public E pop() {
        return (E) block[--size];
    }
    public void push(E el) {
        block[size++] = el;
    }
}
```

Using Generic Classes

```java
public static void main(String[] args) {
    //create a Stack of Strings
    Stack<String> s = new Stack<String>(10);
    s.push("abc");
    System.out.println(s.pop());
    //create a stack of Integers
    Stack<Integer> t = new Stack<Integer>(1);
    t.push(7);
    System.out.println(t.pop());
}
```

How Generics Work

The program:

```java
Stack<String> ss = new Stack<String>(10);
String s = "OK, I’m going in!";
ss.push(s);
s = ss.pop();
```

is converted to:

```java
Stack<Object> ss = new Stack<Object>(10);
String s = "OK, I’m going in!";
ss.push(s);
s = (String) ss.pop();
```

at compile time. Generics allow the compiler to ensure that the casting is correct, rather than the runtime environment.

Some Tricks with Generics...

Note that `Stack<String>` is not a subclass of `Stack<Object>` (because you can't put an `Integer` on a stack of `Strings`).

Therefore, polymorphism won't allow you to define methods for all stacks of subclasses of `String`. e.g.

```java
public int printAll(Stack<Object>);
```

Java 5 allows wildcards to overcome this problem:

```java
public int printAll(Stack<?>);
```

or even

```java
public int printAll(Stack<? extends Object>);
```
Generics in Java are complex and are the subject of considerable debate. While you may not need to write them often, it is important you understand them as the Java Collection classes all use generics.

Some interesting articles:


http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consid_1.html