Introduction To Weeks 8 - 11

About Myself

Me: Dr Norman Lee

Office: 1.402.30

Tel: 6499 8215

Email: norman_lee@sutd.edu.sg

Queries on this course, please put on the respective channels in MS Teams

Getting Ready For Android Programming

“Official Material” - Android Developer Fundamentals Version 2

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

This set of teaching materials was mostly prepared by Dr Norman Lee.

- In lesson 3 below, we deviate from the above reference. Instead of using AsyncTask (which becomes deprecated), we learn how to program a concurrent Android app using the java.util.concurrent package.

Correspondence

This table shows the connection between our lessons and the materials in the android developer fundamentals

Our course	**Android Developer Fundamentals Concepts** **All from Version 2 unless stated**
Lesson 0 **SELF-STUDY From Week 6**	Lesson 1.1
Lesson 1 **SELF-STUDY From Week 6**	Lesson 1.2, Lesson 1.3
Lesson 2	Lesson 2, Lesson 9.1, Lesson 3.2, Lesson 4.1 - 4.2
Lesson 3	~~Lesson 7.1 - 7.2~~ This Blog Post
Lesson 4	Lesson 4.5, Lesson 9.0
Lesson 5	Nil

Android Device

Getting an **Android device** is highly recommended, Android version 5.0 or higher is recommended.
(To check, Open the Settings app → About Phone → Android Version)

If not, you can use the **emulator** in Android studio. However, be warned that students' previous experience with the emulator is that it can take a long time to load and consume a lot of system resources.

Another option for an emulator is **Genymotion**. However, your experience may vary with this emulator.

Other Resources

One thing to note

These resources can be out of date for the following reasons

- New libraries get introduced
- Code in ex gets deprecated

For this reason, books and online resources get out of date very quickly.

Still they can be useful.

Here are the resources

Please use at your discretion

- Codepath guides to android https://guides.codepath.com/android
- Online course on Udacity https://www.udacity.com/course/new-android-fundamentals--ud851
- Stanford CS193A http://web.stanford.edu/class/cs193a/
- Build your first android App https://developer.android.com/codelabs/build-your-first-android-app#0

My Android Experience

- **Frustrating** - code in tutorials is found to be deprecated
- **Moves fast** - new ways of doing things are released very quickly
- **Lots of applications** - commercial 43” screens run android!

How we’ll learn

- I will first highlight and reinforce the Java that you need to know
- Relevant android features will also be introduced to you
- Then we will code an app to apply those features

Java Revision

Some or most of the Java you need to know

equals vs == (1)

Recall that

- `==` checks for whether two **primitive type variables have the same value**
- `==` checks for whether two reference variables have the same **object reference**
- **`.equals()`** is to check for whether two objects have the **same contents**

If you are writing your own classes, a good practice is to override **`equals()`** if you wish to provide the ability to check for the same contents.

```
BigDecimal b1 = new BigDecimal("1.23");
BigDecimal b2 = new BigDecimal("1.23");
BigDecimal b3 = b1;

System.out.println(b1 == b2);
System.out.println(b1.equals(b2));
System.out.println(b1 == b3);
```

What is printed on the screen?

equals vs == (2)

Be aware that Strings are **interned**. In this process,

if you assign the same String literal to two different variables, the compiler will

- store **one** instance of the string in the String pool
- make both variables have the same reference to the same string

Interning will not happen if the **`new`** keyword is used. But, you should avoid using the new keyword with Strings.

Thus, it is a bad habit to check if two strings are the same (i.e. same contents) using `==` and you should always use **`equals()`**.

```
String s1 = "pikachu";
String s2 = "pikachu";
String s3 = new String("pikachu");

System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s3));
```

What is printed on the screen?

Scope of Variables

Recall that variables have scope. For example, variables declared in a method are local to the method. It is also possible to have variables declared within a pair of braces, which have local scope within that pair of braces.

```
for(int i = 0; i <= 10; i++){
   String out = "Green Bottle " + i;
   System.out.println(out);
}
String out = "No more green bottles";
System.out.println(out);
```

What is printed on the screen?

A. Green Bottle 9
B. Green Bottle 10
C. No more green bottles
D. There will be compilation error as `out` cannot be declared in two places.

Reference vs Primitive Types

For each of the **primitive types** (int, double etc), Java provides a **wrapper class** (Integer, Double etc) whose objects store exactly one value of the associated primitive type.

These wrapper classes can then be used in **`Collections`** objects e.g. **`ArrayList<>`**, as these are not designed to accommodate primitive types.

It is possible to assign a primitive type to its wrapper object and vice versa.

While the **`Integer.valueOf()`** static method will convert an **`int`** (and a **`String`** as well) to an **`Integer`**, you can directly assign a primitive type. Java will convert the primitive type to the reference type and this is called **autoboxing**.

```
//Instead of writing
Integer i = Integer.valueOf(3);
// you can directly write
Integer k = 3; // autoboxing
```

The reverse is **unboxing**, where a variable of a reference type is assigned a primitive type.

```
Integer k = 3; // autoboxing
int x = k; //unboxing
```

Consider the following code. Does autoboxing or unboxing happen?

```
List<Integer> arrayList = new ArrayList<>();
arrayList.add(1);
```

Consider the following loops

```
// LOOP 1
int total = 0;
for( int i = 0; i < Integer.MAX_VALUE; i++){
   total = total + i;
}


// LOOP 2
Integer sum = 0;
for( int i = 0; i < Integer.MAX_VALUE; i++){
   sum = sum + i;
}
```

There are two loops in the example above. Notice that loop 2 uses **`sum`** declared with the **wrapper class `Integer`**, while loop 1 uses **`total`** declared as a primitive type **`int`**.
Which loop takes a shorter time to run?

A. Both Loops take the same time, because they take the same number of operations
B. Both Loops take the same time, because the compiler will optimise for speed
C. Both A & B
D. Loop 1 is faster
E. Loop 2 is faster

ArrayList

```
List<Integer> a = new ArrayList<>();
a.add(1);
a.add(2);
a.add(1,3);
a.add(5);
System.out.println(a.toString());
```

What is printed on the screen?

Access modifiers and the final keyword

There are **four access modifiers**. This table summarises them.

If a variable / method is	**It can be accessed …**			
	… within the same class	**… within the same package by other classes / methods**	**… in other packages but in subclasses**	**… in other packages by other classes / methods**
`private`	**Yes**	**No**	**No**	**No**
package-private (no modifier)[1]	**Yes**	**Yes**	**No**	**No**
`protected`	**Yes**	**Yes**	**Yes**	**No**
`public`	**Yes**	**Yes**	**Yes**	**Yes**

[1] It is more accurate to say ”package-private”. Some people say “default”, however this may give the impression that the keyword **`default`** (introduced in later versions of Java) is used.

Private vs Public

```
class Point2D{
    private double x;
    private double y;

    Point2D(){
        //code not shown
    }

    Point2D(double x, double y){
        this.x = x;
        this.y = y; }

    public double getX() { return x; }
    public double getY() { return y; }
}

class Point3D extends Point2D{

    private double z;

    Point3D( double x, double y, double z ){
    }
}
```

Complete the constructor for Point3D.

The process of **Constructor Chaining** happens when a child class constructor is called. If there is no explicit call to a superclass constructor, the compiler automatically places **`super()`** within it. Thus for classes that are meant to be extended, you ought to place a no-arg constructor to provide default behaviour at instantiation.

Access modifiers

```
package p1;

public class A {

   private int a;

   public A( int a){
      this.a = a;
   }

   // continued next column
```

```
   protected int getA() {
        return a;
   }

   int get2A(){
        return 2*a;
   }

   public int get3A(){
         return 3*a;
   }
}
```

Consider the class `A` in package `p1`.

When in package `p2`, which of the methods of `A` can be executed?

```
package p2;
import p1.A;

public class Test {

   public static void main(String[] args) {
       A a = new A(2);
       System.out.println( ? );
   }
}
```

A. `a.getA()`
B. `a.get2A()`
C. `a.get3A()`
D. None of the above
E. All of the above

Recall Polymorphism, Overriding vs Overloading, Generics

We see **overriding**, **overloading and generics** in android very often, so it is good to recap these concepts.
There are three kinds of Polymorphisms, namely **Subtype polymorphism** and **Ad Hoc Polymorphism** and **Parametric Polymorphisms**

Subtype Polymorphism allows variables of a subclass to be used in the context where a superclass is expected. Thus, in the example below, variable **g** is referencing a **Hound** object, but it can be declared as an instance of the **Dog** class. Putting an object of the subtype in the context of the supertype is called upcasting. Upcasting does not override behaviours in the (upcasted, subclass) object.

To override a method in a super-class, the **method signature** in the subclass must be the same. The **@Override** annotation allows the compiler to help you check if you have got this condition correct. The methods that are available depend on the *declared type*. If a method is overridden in the subclass, in **dynamic binding**, the Java VM decides which method to invoke, starting from the *actual type*.

```
abstract class Dog{
    public void bark(){ System.out.println("woof"); }
    public void drool(){ System.out.println("drool");}
}

class Hound extends Dog{
    public void sniff(){ System.out.println("sniff ");}
    @Override public void bark(){ System.out.println("growl");}
    public void drool(int time){ System.out.println("drool" + time);}
}
```

Given **Dog g = new Hound();**

- What will you see on the screen for **g.bark()** ?
- What will you see on the screen for **g.drool(1)** ?
- What will you see on the screen for **g.drool()** ?
- What will you see on the screen for **g.sniff()**?

Subtype Polymorphism

```
class A {

   void f(int x){System.out.println("Af");}
   void h(int x){System.out.println("Ah");}
}

class B extends A{

   void f(int x){System.out.println("Bf");}
   void g(int x){System.out.println("Bg");}
}
```

Given

`A x = new B();`

Which of the following can subsequently be executed?

`x.f(1); //statement (i)`

`x.g(1); //statement (ii)`

`x.h(1); //statement (iii)`

(a) (i) only

(b) (i) and (ii)

(c) (i) and (iii)

(d) (i), (ii) and (iii)

Overloading

Overloading allows a single method name to be shared across different implementations with different types of input parameters. Java run-time decides which particular implementation to be called based on the actual argument type. This is also known as **Ad Hoc Polymorphism**

```
public void log(Integer x) {
    System.out.println(x.toString());
}

public void log(String s) {
    System.out.println(s);
}
```

Generics

In Java **Parametric Polymorphism** exists in the form of Generics. Generics are type parameters, often used in augmenting some type constructors, e.g. ArrayList<>, Optional<>,

In the following the implementations of getFirst(), getSecond(), swap() are independent of what T and S are.

```
public class Pair<F,S> {

   private F first;
   private S second;

   Pair( F first, S second ){
       this.first = first;
       this.second = second;
   }

   public F getFirst() {
       return first;
   }

   public S getSecond() {
       return second;
   }

   public Pair<S,F> swap(){
       return new Pair<S,F>( this.second, this.first);
   }

}
```

Hence, in the following, what is the missing declaration for **`pair.swap()`**?

```
import java.math.BigDecimal;

public class TestPair {

   public static void main(String[] args) {

       Pair<Integer, BigDecimal> pair = new Pair( 1 , new BigDecimal("1.23"));
       ?? = pair.swap()
   }
}
```

Subtype Polymorphism vs Parametric Polymorphism vs Ad Hoc Polymorphism

- Parametric Polymorphism
    - by making the underlying type into a type parameter
    - one and only one piece of code shared by multiple instances of types
- Subtype Polymorphism
    - by making use of the subtyping and inheritance
    - the code for super class is unchanged
    - requires overriding methods
- Ad Hoc Polymorphism
    - By reusing the same method name for different implementation given different input arguments

Interfaces (1)

An interface is like a contract for the implementations of classes. It acts as a *supertype* for all classes that implement it.

```
interface I {
   void m(int x);
}

class K implements I{
   void m(int x){System.out.println("m");}
}
```

Which of the following statements is/are legal?

(i) `K x = new K();`

(ii) `K x = new I();`

(iii) `I x = new K();`

(iv) `I x = new I();`

(a) (i) only

(b) (i) and (ii)

(c) (i) and (iii)

(d) (i), (ii) and (iii)

Interfaces help in the maintenance of software.

Bearing in mind interface **`I`** and class **`K`** implements **`I`** (defined above)

Which method below is better?

```
void firstMethod(K k){ //do something;}
void secondMethod(I i){ //do something;}
```

A method that takes in an interface is more flexible.
It will be able to accept any object that implements that interface.
Suppose you create a new class implementing **I** that has a better implementation of **m**, you are able to pass it to **`secondMethod`** without having to change its signature.

Interfaces (2)

All method signatures in interfaces are automatically abstract, you do not need to specify the keyword.

```
interface Pokemon{

    void adjustCP(int value);
    void attack();
    void defend();
}

class Bulbasaur implements Pokemon{

    void adjustCP(){
        //code not shown
    }

    void attack(){
        //code not shown
    }

}
```

In the code above, which method(s) does class **`Bulbasaur`** still need to implement?

(a) **`defend()`**

(b) **`adjustCP(int)`**

(c) **`attack()`** and **`defend()`**

(d) **`defend()`** and **`adjustCP(int)`**

Similar to abstract class, interface leaves the implementation details to its sub-classes. In contrast to abstract class,

- All methods in interfaces are abstract
- A class can implement multiple interfaces

Exceptions (1)

```
public class TestExceptions1 {

   public static void main(String[] args){
       try{
           f(-1);
           System.out.print("R");
       }catch(Exception e){
           System.out.print("S") ;
       }
   }

   static void f(int x) throws Exception {
      if( x < 0) throw new Exception();
      System.out.print("P");
   }
}
```

In the code above, what is printed out?

(a) S

(b) PRS

(c) RS

(d) PR

Exceptions (2)

```
public class TestExceptions2 {

   public static void main(String[] args){
       try{
           f(-1);
           System.out.print("R");
       }catch(Exception e){
           System.out.print("S") ;
       }
   }

   static void f(int x) throws Exception {
       try{
           if( x < 0) throw new Exception();
           System.out.print("P");
       }catch( Exception e){
           System.out.print("Q");
       }
   }
}
```

In the code above, what is printed out?

(a) Q **(b)** S

(c) QR **(d)** QRS

Points to note

- When an **exception** is thrown, the Java runtime searches through the **call stack** to find the first method that will handle the exception.
- The `finally` block is always executed regardless of what happens in the `try` block.
- It is good programming practice to specify exactly the type of exception that is handled in each catch block, as you will have specific details of the exception that occurred. Hence code examples here are not good …