LESSON 1:
48: XML introduction
- layout_gravity for center of layout
- gravity for center of widget

49: Random Class

51: Inner & Nested class

53: Nested Interface & Anonymous Class & Delegation
- Nested Interface
- Static inner class = declared on class level to use multiple times
- Anonymous class = only used once & defined when function is called

57: How to use basic android from AppCompatActivity
- 63: 3 methods for onClickListener

LESSON 2:
79: How to use BigDecimal class

84: Exceptions and throwing

85: Static Factory Method (Tea & teh example)
- A static method that returns instance of the class

86: Toasts

87: Logcat

88: Explicit Intents
- How to put extras and get extras

90: Activity Lifecycle

92: SharedPreferences & onPause

93: Options menu

94: Builder Design Pattern
- Basically set, set, then build, will return the parent class with objects set
- Not sure why this is even used

96: (DELETED) URI (Universal Resource Indicators)

97: Implicit Intents
- How to use maps etc
- Don't forget to change the manifest

99: Unit Testing
- Use @Test and assertEquals

LESSON 3:
120: Abstract Classes

122: Template Method Design Pattern
- One abstract class to enforce methods to be implemented in other classes
- e.g. Drink = abstract class, all other drinks have to follow it

124: Generic Classes & Interfaces
- ArrayList is a generic CLASS
- Comparable<T> is a generic INTERFACE

126: Executor + Handlers
- Can only access final/immutables

129: Generic Class
- Container<T>, CHECK CODE
- Container MUST be final to be used in executor

130: Looper & Handler
- Get a handler based on the looper

LESSON 4:
144: Recap on 2 & 3

149: startActivityForResult(intent, REQUEST_CODE)

setResult(Activity.RESULT_OK, returnIntent) --> May be CANCELED if user backed out
finish()

onActivityResult

155: Strategy Design Pattern
- Setting interfaces that have been implemented as the inputs
- public void setInterface(Interface interface) { this.interface = interface; }

Overview:
- Delegation = parts of behaviour is handed over
  - flying behaviour delegated to FlyBehaviour
  - FlyBehaviour is an interface that forces some method

- MallardDuck = subclassing abstract class Duck
  - We use delegation and setFlyBehaviour,etc

158: Adapter Design Pattern
- Client to loop through all objects and make them do something
- Used for recycler view
- Make classes compatible so that the client loop still works

Overview:
- MallardDuck implements INTERFACE Duck & overrides
- Adapter design pattern uses Ducks as the object type, so that many
  other types of duck can be used for the same loop

164: RecyclerView
- Use .notifyDataSetChanged() to update the view

- Strategy Design Pattern
  - Retrieving data by RecyclerView.Adapter
  - Managing layout by LinearLayoutManager
    - recyclerView is like MallardDuck and adapter/layoutManager is like FlyBehaviour

- Adapter Design Pattern
  - RecyclerView.Adapter<VH> is like Duck, but is an ABSTRACT class that forces the use of
    - onCreateViewHolder
      - Called on run-time each time data is added & returns an updated <VH>
    - onBindViewHolder
      - Get data & attach them to charaViewHolder accordingly
    - getItemCount
      - Simply return number of items to be seen

    - Generally, <VH> is usually defined as a static inner class
      <VH> is an abstract class with no abstract methods

170: Make RecyclerView items responds to clicks
- Option 2 is used in the code
- Option 1 is commented out

171: itemTouchHelper
- Meant for swiping

176: Relationship for classes

Alignment

Note the difference between the two:

- To align a widget within a layout, use the **`layout_gravity`** attribute (child to parent)
- To align the contents of a widget within itself, use the **`gravity`** attribute (parent to child)

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="AA"/>
```

The widget aligns itself to the centre of the layout.

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:text="AA"/>
```

The contents of the widget aligns itself in the centre of the widget.

To understand the difference:

In which scenario would the **`gravity`** attribute have no effect?

- when width is wrap_content

In which scenario would the **`layout_gravity`** attribute have no effect?

- when width is match_parent

Random Class

In many applications it is useful to generate random numbers.

In Java, you do it by getting an instance of the **`Random`** class.

In this class there are three useful methods

- **`nextInt()`** gives you an integer between 0 and 2^{32} (exclusive)
- **`nextInt(n)`** gives you an integer between 0 and n (exclusive)
- **`nextDouble()`** gives you a double between 0.0 and 1.0

```
Random r = new Random();
r.nextInt();
r.nextInt(100);
r.nextDouble();
```

Random number generators usually need to be initialized with a seed.

If you need the sequence of random numbers to be the same, you use the same seed.

If not, one way to get a changing seed is to use the Date object.

```
Date d = new Date();
Random r = new Random(d.getTime());
```

Nested Classes

A class definition can contain class definitions. We call these classes **nested classes**.

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

Inner Class

A nested class that is not declared static is called an **Inner Class**.

- To instantiate an inner class, you need an instance of the outer class, which is usually called the **enclosing class**.
- The inner class can access all methods and variables of the enclosing outer class.

```
public class OuterClass {

   int a;
   OuterClass(){  a = 10;  }
   void outerPrintA(){  System.out.println(a);  }

   class InnerClass{
       int c;

       InnerClass(){ c = 100; }

       void innerPrintA(){  System.out.println(a);  }

       OuterClass giveBackOuter(){  return OuterClass.this; }
   }
}
```

Activity. For **`OuterClass`**, complete the main function below to illustrate the following properties.

```
public class TestOuterClass {
   public static void main(String[] args){
       //Instantiate OuterClass
       OuterClass outerClass = new OuterClass();

       //Instantiate the InnerClass
       OuterClass.InnerClass innerClass = outerClass.new
InnerClass();
       //Show that InnerClass can access variables in OuterClass

       //Show that InnerClass stores a reference to OuterClass
   }
}
```

Alternative way to instantiate:
OuterClass.InnerClass innerClass = new OuterClass().new InnerClass();
innerClass.outerPrintA();
outerClass = innerclass.giveBackOuter();

Type text here

Static Nested Classes

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.

Activity.

- Modify **`OuterClass.java`** by declaring **`InnerClass`** as static and adjusting other parts of the class accordingly e.g. which other variables must be static? Which methods do not work anymore?
- Write code to show that you can instantiate OuterClass and InnerClass separately.

```
public class OuterClass {
    int a;
    static int b;

    OuterClass() { a = 10; b = 1;}

    void outerPrintA() { System.out.println(a); }

    static void outerPrintB() { System.out.println(b); }

    static class InnerClass {
        static int c;

        InnerClass() { c = 100; b = 100;}

        void innerPrintB() { System.out.println(b); }

        void innerAccessOuter() { OuterClass.outerPrintB(); }
    }
}
```

InnerClass can no longer access OuterClass attribute a, neither can it access the outerPrintA method.

It can however access b and outerPrintB. It will also no longer give back the reference to the outerClass

To test, use new OuterClass.InnerClass();

Nested Interface & Anonymous Classes

Recall that interfaces make your code reusable. We may nest interfaces as well. Recall also that Interfaces are inherently static. In the following code, any object that implements **`Foo.Bar`** interface can be passed to **thirsty()**.

```
public class Foo {

   interface Bar{
       void drink();
   }

   Foo(){
   }

   void thirsty(Bar bar){
       bar.drink();
   }

}
```

nested interface

constructor

method

The inner class **`C`** implements **`Foo.Bar`** and an instance is passed to **`thirsty()`**. The inner class is declared **`static`** because it is invoked from **`main()`**.

```
public class TestFoo {

   public static void main(String[] argv){
       Foo f = new Foo();
       f.thirsty( new C() );
   }

   static class C implements Foo.Bar {
       @Override
       public void drink() {
           System.out.println("gulp");

       }
   }
}
```

static inner class

Anonymous Class

Often, if the Inner Class is used only once, an alternative is an **Anonymous Inner Class**, to avoid declaring too many classes. You may not want to have too many inner classes that are practically used only once.

The following code shows how the **`TestFoo`** example above can be implemented using an anonymous inner class.

```
public class TestFoo1 {

   public static void main(String[] argv){

       Foo f = new Foo();
       f.thirsty( new Foo.Bar(){
           @Override
           public void drink() {
               System.out.println("Gulp");
           }
       });
   }
}
```

The **`new`** keyword is used to instantiate an object that implements **`Foo.Bar`**. Because **`Foo.Bar`** is an interface, the implementation is then specified immediately.

As you can see, we have an **anonymous class** because

- We do not name the class that implements the interface
- We do not assign a variable name to the class that implements the interface

We see nested interfaces, nested static classes and anonymous classes in Android programming frequently.

e.g. in Executors, new Runnable is an anonymous class

Delegation

We go back to the **`Foo`** class and notice that what happens when **`thirsty()`** is executed depends on objects implementing **`Foo.Bar`** that are passed to it.

In other words, the behaviour of **`thirsty()`** is **delegated** to objects that implement **`Foo.Bar`**, (thanks to subtype polymorphism).

This illustrates two design principles:
Program to a supertype - because the input to **`thirsty()`** is an interface, it can accept any object that implements **`Foo.Bar`**.

Favour composition over inheritance - since **`thirsty()`** can accept any object that implements **`Foo.Bar`**, the objects of the **`Foo`** class become more flexible and its behaviour can change at runtime.

```
public class Foo {

   interface Bar{
       void drink();
   }

   Foo(){
   }

   void thirsty(Bar bar){
       bar.drink();
   }

}
```

The Android Programming You need to know

onCreate is called when the Activity is first launched

Within the **MainActivity** class, you would see this code

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
}
```

The **onCreate** method is called whenever your Activity is first launched e.g. when the user clicks on your app icon.

This method is part of the methods in the **Android activity life cycle**, which will be discussed in the next lesson.

You write code in **onCreate** to implement what you want the user to see when the activity is launched.

The R class contains resource IDs to the resources in the res folder.

When the app is compiled, an **R class** is generated that contains IDs to the resources in the **res** folder.

Since **activity_main.xml** is stored in the layout folder, its R class reference is **R.layout.activity_main**.

In onCreate, the layout is first inflated

R.layout.activity_main is passed to the **setContentView** method to **inflate the layout**. In this process, Android reads the XML code in the layout file and instantiates objects in the memory that represent each of the widgets on the Activity.

When a View object is clicked, what happens next is specified by calling setOnClickListener()

Typically, we want Button to be clicked, but it is also possible to have other widgets clicked, including TextView, LinearLayout etc.

The input to **`setOnClickListener`** is an object of a class that implements the **`View.OnClickListener`** **interface**. There is one method to implement, called **`onClick`**. You may implement this in several ways, here I list three ways:

Choice 1. As an inner class in `MainActivity`.

This method shows you clearly what you are doing. But it may cause your MainActivity.java to become bloated with inner classes that you use only once.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);
       button.setOnClickListener( new ClickMe());
   }

   //*** this is an inner class ***
   class ClickMe implements View.OnClickListener{

       @Override
       public void onClick(View v) {
               //code goes here
       }
   }
}
```

Choice 2 (Recommended). As an anonymous class that is defined in the input to setOnClickListener():

This is the recommended method because it is used very frequently, and in many other situations.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

       button.setOnClickListener(new View.OnClickListener() {
           @Override
           public void onClick(View v) {
                 //code goes here
           }
       });
   }
}
```

Choice 3. Define an instance method in MainActivity specifying what is to be done. Then specify it as an attribute in the widget.

Although it looks straightforward and easy, I don't recommend this choice, for the following reasons.

- Many code examples used in teaching android use anonymous classes instead
- This does not work in many other situations.

Define an instance method in MainActivity with any name you like and the following signature.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

   }
   //this method is what myButton1 will do
   public void whenClick(View view){
        //code here
   }
}
```

Then in the XML file, the button widget will have the **`onClick`** attribute.

```
<Button
   android:id="@+id/myButton1"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:onClick="whenClick"
   android:text="Click Me"/>
```

What you need to know (Android/Java)

Use the BigDecimal class for financial calculations

Types such as double and float are not suitable for financial calculations because often, accuracy is demanded, but you will encounter floating point errors with these types.

```
System.out.println(0.7 + 0.1); //do you get 0.8?
```

Why?

In computer every decimal value is represented in two parts,

1. The integer value **i**
2. The **c** represents the scale.
3. Every decimal value can be encoded as **i * b**$^{-c}$ where **b** is the *base*, can be 10, 2 etc.

E.g. if b is 10, 0.1 is represented as 1 * 10^{-1} and 0.25 is represented as 25 * 10^{-2} (or normalized as 2*10^{-1} + 5 * 10^{-2}. If b is 3, ⅓ is represented as 1 * 3^{-1}. Note that ⅓ in 10-base representation is an approximation. In most of the computer run-time, the base is 2 (binary), hence 0.5 is 1*2^{-1}, 0.25 is 1*2^{-2} and 0.75 is 3*2^{-2} or 1*2^{-1} + 1*2^{-2}. Unfortunately in a 2-base system, 0.1 has to be approximated, just like ⅓ has to be approximated in a 10-base system, but perfectly accurate in a 3-base system.

The **`BigDecimal`** class provides a natural representation of decimal by representing numbers in a **10-base** system with two parts

- *Unscaled Value* - an integer of arbitrary precision, that's the **i.**
- *Scale* - the number of digits after the decimal point, that's the **c.**

The **`BigDecimal`** class can be initialized with different types, but we shall stick to using **`String`**. If a **`String`** passed to the constructor does not have a recognizable number e.g. an empty string or a string containing text, a **`NumberFormatException`** is thrown.

Objects of the **BigDecimal** class have methods *add*, *subtract*, *multiply* and *divide*.
As **BigDecima**l is immutable, these methods return a new object.

```
BigDecimal a = new BigDecimal("1");
```

```
BigDecimal b = new BigDecimal("4");
BigDecimal c = a.divide(b);
System.out.println(c);
System.out.println("Scale:" + c.scale() );
System.out.println("Unscaled value:" + c.unscaledValue());
```

If any operation results in an infinite number of decimal places e.g. ⅓ , a **`MathContext`** object is needed, which encapsulates the number of significant figures and the mode of rounding. The following code fragment illustrates.

```
BigDecimal up = new BigDecimal("10");
BigDecimal down = new BigDecimal("7");
int sigFigures = 5;
MathContext mc = new MathContext(sigFigures,RoundingMode.HALF_UP);
BigDecimal result = up.divide(down,mc);
System.out.println(result);
```

There are different rounding modes, which you can read about.

Exceptions

An **exception object** is thrown during events that prevents execution from continuing normally.

You handle these exceptions by putting code in a **`try-catch`** block.

Run the code below and you will see that an **`ArithmeticException`** is caught.

`ArithmeticException` is

- A subclass of **`RuntimeException`**, such exceptions are usually thrown by the JVM
- An **unchecked exception** - the compiler *does not* force you to put the code in a try-catch block (another such exception is **`NumberFormatException`**)

```
public class ExceptionsExample {

   public static void main(String[] args){

       try{
           int a = quotientInt(5,0);
       }catch(ArithmeticException ex){
           ex.printStackTrace();
       }
   }

   public static int quotientInt(int a, int b){
       return a / b;
   }
   //write quotientDouble here later
}
```

Dividing a floating point number by zero does not cause an exception to be thrown.

- Add the following method to the class above double divide by 0 returns "infinity"
- Call it in the try-catch block with b = 0. Is the catch block activated?
- Modify it such that it throws an ArithmeticException if b = 0.

```
public static double quotientDouble(double a, double b){
    return a/b; if (b == 0) {                                   throws ArithmeticException
                  throw new ArithmeticException();
}               }
```

Static Factory Method

A **static factory method** is a static method in a class definition
that returns an instance of that class. (*Attention: this is not the factory design pattern*).

You can overload your constructor to initialize your class with different states,
but you are constrained by Java to have the same name for all constructors.

On the other hand, you can give your static factory method meaningful names to describe what you are doing.

The constructor can be declared private, in which case your class can only be instantiated by calling the static factory methods. Recall that in the singleton design pattern, there is one static method.

```
public class Tea {

   private boolean sugar;
   private boolean milk;

   Tea(boolean sugar, boolean milk){
       this.sugar = sugar;
       this.milk = milk;
   }

   public static Tea teh(){
       return new Tea(true, true);
   }

   public static Tea tehkosong(){
       return new Tea(false, true );
   }
}
```

Hence, you invoke the static factory method like this:

```
Tea tea = Tea.tehkosong();
```

Toasts

You might have seen a message on an android app that disappears after a while.
That is known as a **toast**.
Usually, toasts are displayed to notify users of an event occuring.

The code recipe of a toast is as follows.

- First, call the **static factory method** **`makeText()`** of the toast class and give its required inputs:
    - **`Context`** object.
      A **`Context`** is a super-class of **`AppCompatActivity`**
      (Refer to the AppCompatActivity docs)
      Specifying the context here just means to say on which Activity will your toast be seen.
    - Either a resource id or a String.
      Hard-coded strings are not recommended.
    - Duration of toast. You specify one of two static variables in this:
      `Toast.LENGTH_SHORT` or
      `Toast.LENGTH_LONG`

- **`makeText()`** returns a **`Toast`** object. You can then call the **`show()`** instance method to display the toast. (sometimes I forget this).

Here's an example.

```
Toast.makeText(MainActivity.this,
R.string.warning_blank_edit_text,Toast.LENGTH_LONG).show();
```

Note

If you read the documentation,

- The Toast class constructor is actually public. It is used when you want to customize the design of your toast.
- Most of the time, there is no need to, so **`makeText()`** gives you the standard Toast design.

Logcat

The **Logcat** tab of Android studio displays messages as your app runs.
You may display your own messages to the Logcat using the **`Log`** class.

Messages in the Logcat are divided into one of the following levels and you can filter messages by these levels

- **`d`** for debug
- **`w`** for warning
- **`e`** for error
- **`i`** for info

In addition, every message has a tag for added filtering.
Typically, the apps we do are small apps, so we just stick to one of these levels.

A typical statement to print a message as follows:

```
Log.i(TAG, "Empty String");
```

`TAG` is a String variable that is declared final and static. Here, we are specifying that the message uses the **`i`** level.
Having your app print messages to the Logcat is useful for

- viewing data without having to display it on the UI
- Checking and debugging your code

Question. **`i`** is a static method of the Log class. True/~~False~~

Explicit Intent

An **intent** is a message object that makes a request to the Android runtime system

- to start another specific activity (an **Explicit Intent**), or
- start some other general component in the phone
  e.g. a Map app (an **Implicit Intent**)

Using an intent, you are also able to pass data between the components.

This section is about **Explicit Intents**.

No Data being passed

If you are not passing data between activities, a typical explicit intent is written as follows using the Intent class.

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
startActivity(intent);
```

The constructor of the intent object takes in two inputs

- **`Context`** object specifying the current activity
- **`Class`** object specifying the activity to be started

The intent is then launched by invoking the **`startActivity()`** instance method.

You do not need to write any code in the receiving activity.

Data to pass

If there is data to be passed, we use the **`putExtra()`** method of the intent object as well. Data is stored as key-value pairs.

Step 1. In **MainActivity**:

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
intent.putExtra(KEY,value);
startActivity(intent);
```

The **`putExtra()`** method takes in two inputs

- A final string variable that acts as the key
- The data that is to be stored

Step 2. In **SubActivity**, you then need to obtain the intent object using **`getIntent()`** and retrieve the data using the key using one of the **`get`** methods attached to the intent object. Hence:

```
Intent intent = getIntent();
double value = intent.getDoubleExtra(MainActivity.KEY,
defaultValue);
```

You invoke the appropriate method according to the data type that you want to receive.

In this case, we want to receive a double value, so we invoke **`getDoubleExtra()`**

This method has two inputs

- The key to retrieve the value
- The default value when there is nothing to be retrieved

Android Activity Lifecycle

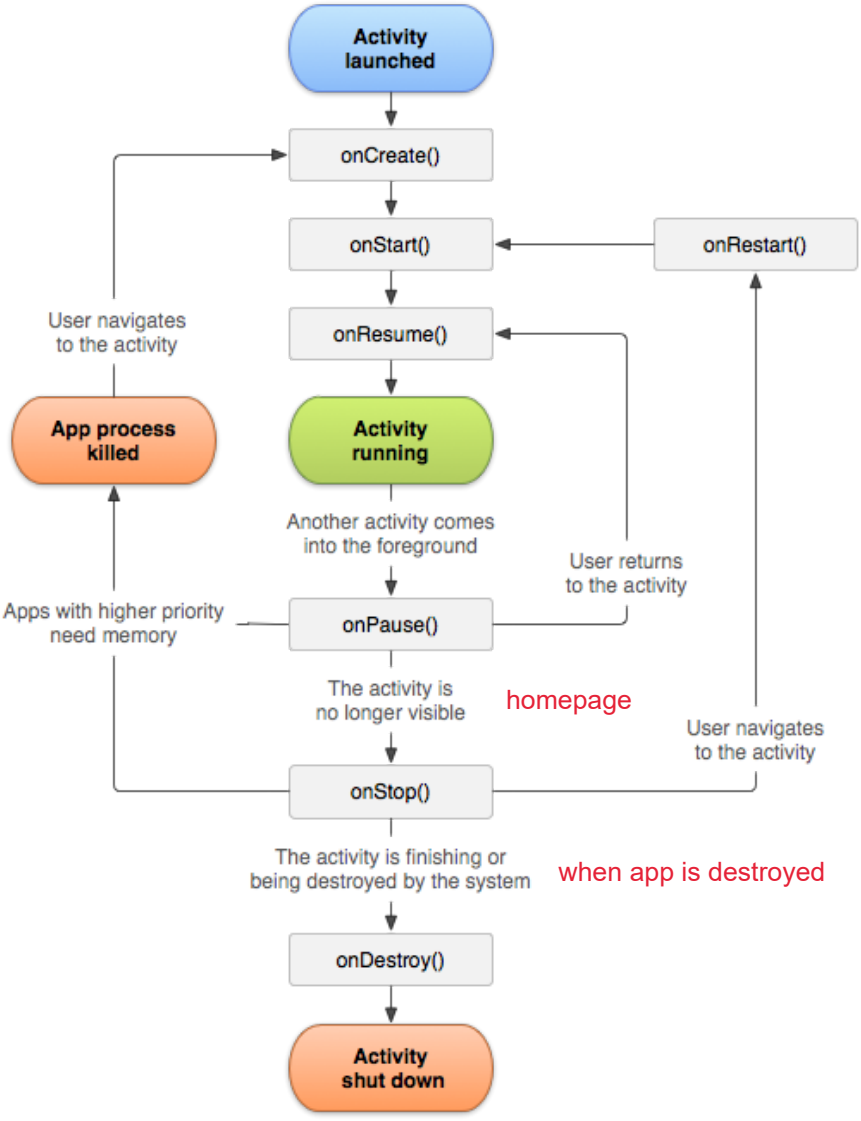
So far, we have been writing code in **`onCreate(),`** which is called when the activity starts.

When interacting with your app, the user may find himself doing some of the following actions

- Access a different activity
- Rotate the screen
- Access a new app
- Closing the app

This changes the state of the current activity.

In doing so, other callbacks are executed. The callbacks are summarized in the following diagram of the **Android Activity Lifecycle**.



Implement the seven callbacks in your app, and write a logcat message in each of them.

- Which methods are called as the activity starts? onCreate, onStart, onResume
- Which methods are called as you navigate from one activity to another? quite a lot
- Which methods are called as you rotate the screen? same as navigating a screen

The next callback that we will write code in is **`onPause()`**;

This is typically done when you want to carry out the following tasks before your activity is destroyed:

- Saving data
- Stopping a timer

Data Persistence with Shared Preferences

As your user interacts with your app, he or she may

- Close the app and restart it
- Rotate the screen (if you allow the screen to be rotated)

In such situations, data entered by your user is not stored.

There are many ways of storing data, which is called **Data Persistence**.

One way of enabling data persistence is through the **`SharedPreferences`** interface.

Information you would like to store is done using **key-value pair**s.

The code recipe is as follows.

1. Declare the filename of your **`SharedPreferences`** object as a final string instance variable. Also, declare a final string variable as a key.
2. In **`onCreate()`**, get an instance of the **`SharedPreferences`** object.
3. In **`onPause()`**, get an instance of the **`SharedPreferences.Editor`** object and store your key-value pairs. Commit your changes using apply.
4. In **`onCreate()`**, retrieve your data using the key. Don't forget to also assign a default value for the situation when no data is stored.

```
private final String sharedPrefFile =
"com.example.android.mainsharedprefs";
public static final String KEY = "MyKey";
SharedPreferences mPreferences;

@Override
protected void onCreate(Bundle savedInstanceState) {
   //other code not shown
   mPreferences = getSharedPreferences(sharedPrefFile,
MODE_PRIVATE);
   String Rate_text = mPreferences.getString(KEY,defaultValue);
}
@Override
protected void onPause() {
   super.onPause();
   SharedPreferences.Editor preferencesEditor =
mPreferences.edit();
   preferencesEditor.putString(KEY, value);
   preferencesEditor.apply();
}
```

Options Menu

When you start a new Android studio project, one option is the **Basic Activity** template. In this template, code for the following UI elements are automatically provided

- **Options menu**
- **Floating Action Bar** (not discussed in this lesson, will talk about it in Lesson 4)

The **Options menu** is a one-stop location for your user to navigate between the activities of the app.
You should see some additional xml tags in the xml files that specify your layout.
You should see the following lines of code in your MainActivity.java:

In **`onCreate()`**, the following code makes the toolbar containing the options menu appear. Don't delete this code.

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

The **`onCreateOptionsMenu()`** method is added to inflate the menu layout and make it appear in the toolbar. Usually, you do not need to add code to this method. The xml layout file is found in **res/menu/menu_main.xml .**

The **`onOptionsItemSelected()`** method is also added. This is where you need to add code to specify what happens when each menu item is clicked. You do this after modifying **menu_main.xml**.

Hence, here are the steps in customizing the options menu.

- add your menu items in **res/menu/menu_main.xml** and remember to give each item a unique ID
- Modify **`onOptionsItemSelected()`** to specify what happens when each menu item is clicked. Very often, you would need to write an intent to bring your user to the other activities in your app.

Builder Design Pattern

When we discussed the Tea class, we saw how static factory methods can be used. Let's expand the tea class to four options.

```
public class TeaTwo {
    private boolean sugar;
    private boolean milk;
    private boolean ice;
    private boolean toGo;
    //code not shown
}
```

If you were to write a constructor or static factory method for each possible combination of options, how many constructors would you have to write?

We may solve this problem by introducing a **static nested class**, usually called a **builder class** that has

- methods to allow the user to specify the options one by one
- One method that returns the actual object

If the constructor is (1) private and (2) takes in an instance of the builder class, the only way to instantiate your object would be to use the builder class.

The code below illustrates this point.

```
public class TeaTwo {
   private boolean sugar;
   private boolean milk;

   private TeaTwo(TeaBuilder teaBuilder){
       this.sugar = teaBuilder.sugar;
       this.milk = teaBuilder.milk;
   }

   static class TeaBuilder{
       private boolean sugar;
       private boolean milk;

       TeaBuilder(){}

       public TeaBuilder setSugar(boolean sugar){
           this.sugar = sugar;
           return this; }

       public TeaBuilder setMilk(boolean milk){
           this.milk = milk;
           return this;}

       public TeaTwo build(){
           return new TeaTwo(this); }
   }
}
```

The builder is then used as follows:

```
TeaTwo teaTwo = new
TeaTwo.TeaBuilder().setSugar(true).setMilk(true).build();
```

Implicit Intents

Recall that in an explicit intent, you specify exactly which activity your user is brought to.

A more general way of using intents is to use an Implicit Intent.
Rather than a specific Activity, In an implicit intent, you specify the type of action you want, provide just enough information and let the android run-time decide.

To illustrate, the code recipe for launching a Map App is given below.

Step 1. You build the URI to specify the location that you want your map app to be. Using this builder helps to avoid errors when hardcoding a URI string.

```
String location = getString(R.string.default_location);
Uri.Builder builder = new Uri.Builder();
builder.scheme("geo").opaquePart("0.0").appendQueryParameter("q",locatio
n);
Uri geoLocation = builder.build();
```

Step 2. You specify the implicit intent by:

- Specify the general action, in this case it is to view data, hence **`Intent.ACTION_VIEW`** is passed to the constructor
- Specify the data that you wish to view, in this case it is the location URI that you build in Step 1.

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(geoLocation);
```

Step 3. Check that the intent is able to be carried out before calling startActivity().

```
if( intent.resolveActivity(getPackageManager()) != null){
      startActivity(intent);
}
```

Other intents

Other common intents, eg. opening the camera, are specified here:
https://developer.android.com/guide/components/intents-common

Implicit Intents in Android SDK API ver. 30+

The following declarations in AndroidManifest.xml are required for the implicit intent to work.

```
<queries>
     <!-- Browser -->
     <intent>
         <action android:name="android.intent.action.VIEW" />
         <data android:scheme="http" />
     </intent>
     <!-- Camera -->
     <intent>
         <action android:name="android.media.action.IMAGE_CAPTURE"
/>
     </intent>
     <!-- Gallery -->
     <intent>
         <action android:name="android.intent.action.GET_CONTENT"
/>
     </intent>
     <!-- Map -->
     <intent>
         <action android:name="android.intent.action.VIEW"/>
         <data android:scheme="geo"/>
     </intent>
 </queries>
```

Unit Testing with JUnit4

Testing your app is one of the phases in development. We'll describe two ways of testing:

- Unit testing
- ~~Instrumented Testing (next section)~~ **NOT TAUGHT IN 2022 Spring, but you are encouraged to read it up by yourself**

A **unit** is the smallest component that can be tested in your software, usually it is a method in a class. **Unit testing** thus ensures that each of these components behave as designed.

Why do unit testing?

- Unit testing validates that your software works,
  even in the face of continual changes in your code
- Writing code for unit testing also forces your program to be modular

In the context of an android app,

- Testing the parts that don't involve the UI **(easy)**: **unit testing**
- Testing the parts that involve the UI (hard): **instrumented testing**

Hence, it is good programming practice to
separate the parts of your code that involve the UI and those that do not.

JUnit4 is a commonly-used open-source framework to conduct unit testing.
To conduct unit testing, you write your tests in a **test class**.

Android studio automatically generates the test class for you.

Question. Since Java comes with a compiler that tries to detect all kinds of anomalies during run-time? Why do we still need to perform testing?

For functional and outcome testing, to ensure that code/logic is written correctly

How to write a unit test

For each unit test,

- you write a method that returns void with the **`@Test`** annotation
- Within this method, use the **`assertEquals()`** method to compare the actual object and the expected object

```
@Test
public void addition_isCorrect() {
   assertEquals(expected, actual);
}
```

There are other assert methods that are available. Take the time to explore them.

In the following unit tests in Examples 1 to 3, suppose that there is a class **`A`** that has a method called **`someFunction()`** that is overloaded and has the following specifications:

- If no parameters are passed to it, it returns a double value of 2.95
- If a string containing a number is passed to it, it returns a **`BigDecimal`** object initialized with that string
- If a string is passed to it and it does not contain a number, it throws an **`IllegalArgumentException`**

Example 1. In the following unit test, we want to write a test to ensure that **`someFunction()`** returns a double value of **`2.95`**. As double objects could have floating point error, the third argument is a tolerance: if abs(*expected* - *actual*) < *tolerance*, then *expected* is treated to be equal to *actual*.

```
@Test
public void someFunctionTest1(){
   assertEquals(2.95, new A().someFunction(),0.001);
}
```

Example 2. In the following unit test, we want to write a test to ensure that **`someFunction()`** returns a **`BigDecimal`** object when a string is passed to it.

```
@Test
public void someFunctionTest2(){
   assertEquals(new BigDecimal("2.95"), new
A().someFunction("2.95") );
}
```

The Android/Java that you need to know

Abstract Classes

A class declared as **`abstract`** cannot be instantiated.

Typically, **abstract classes** have **abstract methods**, which you will recall is a **method signature** that includes the keyword abstract.

Why do we do this?

- **What is to be done** is specified by the abstract methods (and interfaces too)
- **How it is to be done** is specified by their implementation

Because abstract classes can behave as a datatype, it promotes **code reuse**.
Recall the principle of **Program to a supertype**: You may write methods that take in an abstract class as an input, and you are guaranteed that objects passed to it will have the abstract method implemented. (This remark also applies to java interfaces.)

Your abstract class can have constructors specified. One use case is if you would like to force the initialization of any instance variables.
Remember that constructors are not inherited.
Thus, in the constructor of the child classes, you will need to call the superclass constructor using the **`super()`** keyword.

To use an abstract class, sub-class it and implement any abstract methods.

Clicker Question

Given the following

```
abstract class Foo {
   int x = 0;                                  // (i)
   abstract void f() { System.out.println(x); } // (ii)   ii is not ok
   void g();                                   // (iii)
   abstract void h();                          // (iv)
}   abstract methods do not have body
```

Which statement(s) cause(s) compilation errors?

In the example below, complete the class **`Tiger`**.

```
public class TestAbstract {

    public static void main(String[] args){
        Feline tora = new Tiger("Tiger","Sumatran Tiger");
        makeSound(tora);
    }                          this can just be tora.sound();

    public static void makeSound(Feline feline){
        feline.sound();
    }
}

abstract class Feline {

    private String name;
    private String breed;

    public Feline(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    public String getName() {
        return name;
    }

    public String getBreed() {
        return breed;
    }

    public abstract void sound();
}

class Tiger extends Feline{

     //write the constructor and complete the class
}
```

```
Tiger(String name, String breed) {
    super(name, breed);
}

@Override
public void sound() {
    // Implement
}
```

Template Method Design Pattern

One application of an abstract class is in the **template method design pattern**.
In this design pattern, there is an algorithm with a fixed structure, but the implementation of some steps are left to the subclasses. The following example is taken from "Heads First Design Patterns - A brain-friendly guide".

We have a fixed way of brewing caffeine beverages (Coffee, Tea etc), but you'll agree that how you brew it and what condiments to add depends on the beverage.
In the example below,

- The algorithm to make the caffeine beverage, **`prepareRecipe()`** is declared **`final`** to prevent subclasses from altering the algorithm.
- The steps in the algorithm that are common to all beverages are implemented.
- The steps that can vary are declared **`abstract`**.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe(){
        boilWater();
        brew();
        addCondiments();
        pourInCup();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater(){
        System.out.println("Boiling Water");
    }

    void pourInCup(){
        System.out.println("Pouring in Cup");
    }
}
```

We may then have our subclasses of **`CaffeineBeverage`**.

```
class GourmetCoffee extends CaffeineBeverage{
    @Override void brew() {
        System.out.println("Put in Coffee Maker");
    }

    @Override void addCondiments() {
        System.out.println("Adding nothing, because GourmetCoffee");
    }
}
```

We may then brew our coffee:

```
CaffeineBeverage caffeineBeverage = new GourmetCoffee();
caffeineBeverage.prepareRecipe();
```

Generic Classes & Interfaces

You would have used the ArrayList class in the following way::

```
ArrayList<Integer> arrayList = new ArrayList<>();
```

Recall that this design is for **type safety** - an operation cannot be performed on an object unless it is valid for that object.
This means that the code below would cause a compile-time error.

```
arrayList.add("abc");
```

You would also have worked with the Comparable interface:

```
public class Octagon implements Comparable<Octagon>{
    //code not shown
    @Override
    public int compareTo(Octagon octagon) {
        //code not shown
    }
}
```

The **`ArrayList`** class is an example of a **Generic class** where a class takes in an object or objects as a parameter and operates on it.
Similarly, the **`Comparable<T>`** interface is an example of a **Generic interface**.

When Generic meets Inheritance

Recall the Pair class

```
public class Pair<T, S> {
  public T first;
  public S second;
  public Pair(T first, S second)
  { this.first = first; this.second = second; }
}
```

And the comparable interface

```
interface Comparable<T> {
   int compareTo(T that);
}
```

Suppose we would like to modify the above `Pair` class such that when two `Pair` objects are compared, we compare the `first` items, if tie, we compare the `second` items. However that would imply that the generic T and S are *not as generic* as before.

```
public class Pair<T extends Comparable<T>, S extends Comparable<S>>
   implements Comparable<Pair<T,S>> {
  public T first;
  public S second;
  public Pair(T first, S second)
  { this.first = first; this.second = second; }
  @Override
  public int compareTo(Pair<T,S> that) {
      int r1 = this.first.compareTo(that.first);
      if (r1 == 0) {
          return this.second.compareTo(that.second);
      } else {
          return r1;
      }
  }
}
```

In the above the type constraint `T extends Comparable<T>` restricts the type variable `T` must be a subtype of `Comparable<T>`. These constraints provide hints to the compiler that it is safe to call `this.first.compareTo(that.first)`. This is known as F-bounded polymorphism.

A glimpse of concurrent programming

In many situations, we want our applications to handle different tasks simultaneously. In this app, we are going to query a web API to retrieve data.

The length of time this task will take depends

- on the internet connection,
- file size, etc

While the data is being retrieved, you want the UI to remain responsive and inform the user that the download is still in progress.

This means that the **download should be done on a separate thread from the UI.**
A thread is a unit of executing instruction sequences in a program. A concurrent program consists of multiple threads and an **executor (scheduler)** which orchestrates the actual execution/scheduling of the threads. In the presence of multiple processing units (CPU cores), the scheduler might be able to execute/schedule multiple threads simultaneously in different processing units, otherwise, the scheduler will interleave the executions of the given set of scheduled threads. This is often abstracted away from the programmers, so as to prevent deadlock or other abnormal outcomes.

Question: Do you know what is the difference between a thread and a process?

Process means a program is in execution, whereas thread means a segment of a process.

Process can have multiple threads, but threads are singular

The Executor class

In Java, we gain access to the executor service / the scheduler service via the Executor class. Through the executor, we are able to access the pool of threads for a program.

The Runnable interface

An instance of the Runnable interface denotes a sequence of instructions to be executed in a thread.

```
// main thread (i.e. UI thread)
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            // some instructions to be executed in the new thread.
        }
});
```

`Executors.newSingleThreadExecutor()` - this instantiates a single thread executor service. Other construction methods are available, e.g. `newFixedThreadPool(int nThreads)`. For details, refer to https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html
`run()` - this is an abstract method defined in the **Runnable** interface. An instance of **Runnable** interface must implement/override this method.

Question: What happens if the Runnable / the sub-routine running in the thread needs to exchange information with the main thread (UI thread)?

Immutable variable

Instructions in the child thread can only access variables from the main thread if they are immutable (i.e. final).

The following is resulting in a compilation error.

```
// main thread (i.e. UI thread)
String s = ...;
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            System.out.println(s); // illegal access
        }
});
```

This is ok.

```
// main thread (i.e. UI thread)
final String s = ...;
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            System.out.println(s); // legal access
        }
});
```

But it implies that shared data cannot be modified in the child thread.

Making use of a generic class

We can introduce a generic class **Container** which serves as a workaround.

```
// a container class
class Container<T> {
    T value;
    Container(T v) { this.value = v; }
    void set(T v) { this.value = v; }
    T get() { return this.value; }
}
```

We put the data to be exchanged between the main and child threads in a **Container** object.

```
// main thread (i.e. UI thread)
String s = …;
final Container<String> cs = new Container<>(s);
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            String s1 = cs.get() + “!”;
            cs.set(s1);
        }
});
```

Question: How can we inform the UI thread when the child thread is done *asynchronously*? In other words, without having the UI thread to constantly check whether the child thread is done.

see below

Looper and Handler

In the Android multithreading library, each thread is associated with a message queue. A message queue is created for the purpose of asynchronous communication. (Just like our messagers, email, WhatsApp, Telegram, for interpersonal communication, without disturbing each other).

There are two additional classes created for the purposes of reading and updating the message queues, namely the **Looper** class and the **Handler** class.

```
// main thread (i.e. UI thread)
ExecutorService executor = Executors.newSingleThreadExecutor();
Looper uiLooper = Looper.getMainLooper(); // get the main looper
Handler handler = new Handler(uiLooper); // get the handler for the main thread
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            // instructions performed in the child thread
            // ...
            handler.post(new Runnable() {
                @Override
                public void run() {
                    //UI Thread will receive and run this
                }
            });
        }
});
```

How to use Executor and Runnable in your Android App

Step 1. Prevent your activity from changing orientation.

- In the Android Activity Lifecycle, recall that the app layout is destroyed and re-created if the screen is rotated.
- If the child thread is running in the background during this process, it will not be able to display the result on the re-created activity. (c.f. Phone number changes)
- Hence, you need to constrain your activity so that it is always in your desired orientation.
- This is done in the android manifest.

Step 2. Define a generic class in MainActivity or some util package which serves as a container.

Make the following decisions to help you decide the generic types:

- What information launches the background task (the child thread) ?
- What information do I want to give the user as the task proceeds?
- What information does the background task provide?
- When the background task is completed, how shall the UI be updated?

Step 3. Decide what jobs are to be run in the two Runnable instances:

- **`run()`** `in the outer` **`Runnable`** always carries out the background task (child thread) e.g. downloading the image data from a URL.
- **`run()`** `in the inner` **`Runnable`** carries out the job to be performed (in the UI thread) after the background task is complete. Suppose the task is to download an image given a URL, this is where you write code to display the image on the UI.

Step 4. Define a method getComic() which implements the background task as well as the handler task (task to be performed upon completion of the background task).

One possible code stump is shown below. In this code stump, **Look at the method signatures and think about why the final type is required. Observe how the generic type containers are used.**

```
void getComic(final String userInput) {
   ExecutorService executor = Executors.newSingleThreadExecutor();
   final Handler handler = new Handler(Looper.getMainLooper());

   executor.execute(new Runnable() {
       @Override
       public void run() {
           //Background work here
           final Container<Bitmap> cBitmap = new Container<>();
           //retrieve the bitmap from xkcd and store it in cBitmap
           handler.post(new Runnable() {
               @Override
               public void run() {
                   //UI Thread work here
                   if (cBitmap.get() != null) {
                       // retrieve the bitmap and display it
                   }
               }
           });
       }
   });
}
```

Step 5. Decide how the user is to execute the background task.

Let's say your user will download the image with a button click. Then you would put the following code in the anonymous class within **`setOnClickListener()`**

```
getComic(userInput);
```

An Alternative - Fun Challenge

`AsyncTask` could be bad and deprecated. Can we define our own?

Can we extract the reusable part from the `getComic()` code?

```
abstract class BackGroundTask<I,O> {
   ExecutorService executor;
   final Handler handler = new Handler(Looper.getMainLooper());
   public BackGroundTask() {
       this.executor = Executors.newSingleThreadExecutor();
   }
   abstract public O runInBackground(I i);
   abstract public void whenDone(O o);

   public void run(final I i) {
       final Container<O> co = new Container<>();
       this.executor.execute(new Runnable() {
           @Override
           public void run() {
               co.set(BackGroundTask.this.runInBackground(i));
               handler.post(new Runnable() {
                   @Override
                   public void run() {
                       if (co.get() != null) {
                           whenDone(co.get());
                       }
                   }
               });
           }
       });
   }
}
```

We may now define a nested class to replace `getComic()` by extending the `BackgroundTask` class.

```
void getComic2 (final String userInput) {
   (new BackGroundTask<String, Bitmap>() {
       @Override
       public Bitmap runInBackground(String userInput) {
           Bitmap bitmap = null;
           // background task here
           return bitmap;
       }

       @Override
       public void whenDone(Bitmap bitmap) {
           if (bitmap != null) {
               // UI thread job here
           }
       }
   }).run(userInput);
}
```

Building a URL

A URL is a URI that refers to a particular website. The parts of a URL are as follows

Component	Example
Scheme	**`https`**
Authority	**`xkcd.com`**
Path	**`info.0.json`** or **`614/info.0.json`**

To prevent parsing errors you may use the **`Uri`** builder before creating a **`URL`** object.

Using such a builder helps to eliminate coding errors.

The string constants may be placed in the **`strings.xml`** file instead.

After you create the **`Uri`** object, use it to make a **`URL`** object.

The constructor of the **`URL`** class throws a **`MalformedURLException`**.

As this is a **checked exception**, you have to put the code in a **`try-catch`** block.

```
final String scheme = "https";
final String authority = "xkcd.com";
final String back = "info.0.json";
URL url = null;

Uri.Builder builder = new Uri.Builder();
builder.scheme(scheme)
       .authority(authority)
       .appendPath(back);

Uri uri = builder.build();

try{
   url = new URL(uri.toString());
}catch(MalformedURLException ex) {
   Log.i(TAG, "malformed URL: " + url.toString());
}
```

Connecting to the internet

With a URL object, you are ready to download data from the internet.
In the starter code, you are provided with a **`Utils`** class.
There are three static methods for you to use.
You need not worry about how they are implemented, although if you are querying an API for your 1D project, you might find the code useful.
You should see that some of these methods print data to the logcat.

The following methods take in the comic number, queries the xkcd API and returns the image URL for that comic. Please note the exceptions thrown by this method.

```
static String getImageURLFromXkcdApi(String comicNo)
```

The following method takes in a URL for an image and returns the image as a Bitmap object. Please note the exceptions thrown by this method.

```
static Bitmap getBitmap(URL url)
```

The following method takes in a Context object and checks if a network connection is available. True is returned if a network connection is available, False if is not. **`Context`** is the superclass of **`AppCompatActivity`**.

```
static boolean isNetworkAvailable(Context context)
```

> If you use the Android Emulator, you may experience difficulties with checking for an internet connection. Please borrow a phone.

```
static boolean isNetworkAvailable(Context context) {
    ConnectivityManager connectivityManager
            = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
    boolean haveNetwork = activeNetworkInfo != null && activeNetworkInfo.isConnected();
    Log.i(UTILS_TAG, "Active Network: " + haveNetwork);
    return haveNetwork;
}
```

Android Manifest

In order to

- Constrain the activity orientation
- Access the internet

The android manifest needs some additional information.

They are shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.norman_lee.comicapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ComicApp"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"
            android:screenOrientation="portrait"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

</manifest>
```

The JSON format

JSON stands for **J**ava**S**cript **O**bject **N**otation and stores data using key-value pairs.

A sample response from the xkcd API is

```
{"month": "11", "num": 2068, "link": "", "year": "2018", "news": "",
"safe_title": "Election Night", "transcript": "", "alt": "\"Even the
blind\u00e2\u0080\u0094those who are anxious to hear, but are not able
to see\u00e2\u0080\u0094will be taken care of. Immense megaphones have
been constructed and will be in use at The Tribune office and in the
Coliseum. The one at the Coliseum will be operated by a gentleman who
draws $60 a week from Barnum & Bailey's circus for the use of his
voice.\"", "img": "https://imgs.xkcd.com/comics/election_night.png",
"title": "Election Night", "day": "5"}
```

Before you make use of this data, you would have to make sense of it. Online JSON viewers are available to help you.

Parsing JSON data using the JSONObject class

Once you are able to make sense of this data, you are ready to parse it. One way is to use the **`JSONObject`** class.

The **`JSONObject`** constructor takes in a string variable that contains the JSON data.

You then retrieve the value using the key and one of the appropriate get methods.

```
JSONObject jsonObject = new JSONObject(json);
String safe_title = jsonObject.getString("safe_title");
```

Another alternative is the GSON library, which many programmers find useful in parsing JSON data. I will leave you to learn it on your own and you won't be tested on it.

The Android/Java that you need to know

Recall Lesson 2 & 3

We might get bogged down with specific details in the code, but we should also gain an overview of the Android framework and some advanced features in Java.

For each feature described, state the part of the android framework that you can implement.

Feature	**Framework component**
Bring the user from one Activity to another Activity	Explicit intent
To validate that your app behaves as it should when a user interacts with it	Instrumented Testing
Bring the user from the app to another app with a specific function	Implicit intent
The series of callbacks as an Activity is created and/or destroyed	onCreate, onDestory, onStart, onRestart, onStop, onPause
To run long tasks in another thread	ExecutorService
Store data to make it available at another point in time	SharedPreferences
To validate that code components (e.g. methods) in your app behaves as it should	Unit Testing
To develop reusable code by subtyping polymorphism	Delegation
To develop reusable code by parametric polymorphism	Generics
To develop reusable code by subtyping polymorphism and parametric polymorphism	Generic Interface (pg 124)

Static Nested Classes

Recall that a class definition can contain **nested classes**.

*InnerClass meant to be static

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.
One reason for having a static nested class is to have a model class to store data.

Static inner classes can be used to model data

The following example is from the app that you will build in the lesson.
`DataSource` is a class that is meant to contain data that will be displayed in the RecyclerView.
A private ArrayList variable is declared to hold instances of **`CardData`**, a static inner class.
Each instance of **`CardData`** is meant to hold information for one image.
Bear in mind that such static classes cannot access non-static variables of the enclosing class.

e.g. DataSource and using static inner CardData

Recall the Singleton design pattern

Recall that the singleton design pattern allows only one instance of a class to exist.

This is done by

- Making the constructor private
- The sole instance is stored in a private static variable
- Using a static factory method to return an instance

```
public class Singleton{

    private static Singleton singleton;

    private Singleton(){
         //any tasks you need to do here
    }

    public static Singleton getInstance(){

        if(singleton == null){
            singleton = new Singleton();
        }

        return singleton;
     }

     //other methods in your class

}
```

Explicit Intent - startActivityForResult()

Step 1. Declare your request code, a final static integer variable that contains a unique integer that identifies your particular intent.

This is necessary as your activity could have more than one call to **`startActivityForResult()`**

```
final int REQUEST_CODE_IMAGE = 1000;
```

Step 2. Declare an explicit intent in the usual way.

Then invoke **`startActivityForResult()`** with two arguments

- The intent
- The request code

```
Intent intent = new Intent(MainActivity.this, DataEntry.class);
startActivityForResult(intent, REQUEST_CODE_IMAGE);
```

Step 3. In the destination activity, the user should interact with it.

Step 4. A user action (e.g. clicking a button) brings the user back to the origin activity.

The following code initiates this process.

```
Intent returnIntent = new Intent();
returnIntent.putExtra(KEY, value); //optional
setResult(Activity.RESULT_OK, returnIntent);
finish();
```

The first argument of **`setResult()`** is either

- **`Activity.RESULT_OK`** if the user has successfully completed the tasks
- **`Activity.RESULT_CANCELED`** if the user has somehow backed out

Hence, this code may be written twice, one for each scenario described above.

If you have data to transfer, you are reminded that you can use the **`putExtra()`** method above. (Recall Lesson 2).

Step 5. Back in the origin activity, override the callback `onActivityResult()` to listen out for the result and carry out the next task. Note the sequence of if-statements.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {

   if (requestCode == REQUEST_CODE_IMAGE) {
       if(resultCode == Activity.RESULT_OK){

           //if you use putExtra in Step 4, then you need this step
           double value = data.getDoubleExtra(DataEntry.KEY,
defaultValue);
           Toast.makeText(this,"Message",Toast.LENGTH_LONG).show();
       }
       if (resultCode == Activity.RESULT_CANCELED) {
           //Write your code if there's no result
       }
   }
}
```

If you have data transferred from step 4, you may retrieve this data on the intent object passed to this callback using the **`getDoubleExtra()`** method or other suitable methods(Recall Lesson 2).

Strategy Design Pattern

In the strategy design pattern, parts of the behaviours of an object are handed over to other objects. This is known as **delegation**.

This provides flexibility at run-time as you can change those behaviours.

```
public abstract class Duck {

   private FlyBehavior flyBehavior;
   private QuackBehavior quackBehavior;
   String name;

   public Duck(){
   }

   public Duck(String name){
       this.name = name;
   }

   public void setFlyBehavior(FlyBehavior flyBehavior) {
       this.flyBehavior = flyBehavior;
   }

   public void setQuackBehavior(QuackBehavior quackBehavior) {
       this.quackBehavior = quackBehavior;
   }

   public void performFly(){
       flyBehavior.fly();
   }

   public void performQuack(){
       quackBehavior.quack();
   }

   public abstract void display();
}
```

In the abstract class above, the delegation happens as follows

- The flying behaviour is delegated to a **`FlyBehavior`** object
- The quacking behaviour is delegated to a **`QuackBehavior`** object

For the FlyBehavior, we implement different objects that represent different behaviour.

```
interface FlyBehavior {
   void fly();
}
```

```
class FlapWings implements FlyBehavior {
   @Override
   public void fly() {
       System.out.println("Flapping my Wings");
   }
}
```

Implement a class **`CannotFly`** that implements **`FlyBehavior`**.

The **`fly()`** method prints out "**`I cannot fly :(`**"

```
class CannotFly implements FlyBehavior {
  @Override
  public void fly() {
    System.out.println("I cannot fly :(");
  }
}
```

Similarly, for **`QuackBehavior`** objects:

```
public interface QuackBehavior {
   void quack();
}
```

```
public class LoudQuack implements QuackBehavior {
   @Override
   public void quack() {
       System.out.println("QUACK");
   }
}
```

Finally, we subclass Duck with our own object.

```
public class MallardDuck extends Duck {

   MallardDuck(String name){
       super(name);
   }

   @Override
   public void display() {
       System.out.println("I am " + name + ", the Mallard Duck");
   }
}
```

And we can run our **`MallardDuck`** object and set their behaviours at run-time:

```
public class TestDuck {

   public static void main(String[] args){

       Duck duck = new MallardDuck("Donald");
       duck.setFlyBehavior(new FlapWings());
       duck.setQuackBehavior(new LoudQuack());
       duck.display();  I am Donald the Mallard Duck
       duck.performFly();  Flapping my Wings
       duck.performQuack();  QUACK
   }
}
```

We see here the **flexibility of composition over inheritance**.

We develop our duck behaviours independent of the type of duck.

The behaviour of the duck is delegated to separate objects.

You assemble your specific duck at run-time,

which gives you the flexibility to change its behaviour if needed.

Adapter Design Pattern

The word **interface** is an overloaded word

- in Java terminology it would mean a type of class with method signatures only
- it could also mean the set of methods that a class allows you to access
  (think of ‘user interface’)

An adapter design pattern converts the interface of one class into another that a client class expects.

Adapter Design Pattern Example

You have an interface **`Duck`** and a class **`MallardDuck`** that implements this interface.

```
public interface Duck {
   void quack();
   void fly();
}
```

```
public class MallardDuck implements Duck {
   @Override
   public void quack() {
       System.out.println("Mallard Duck says Quack");
   }

   @Override
   public void fly() {
       System.out.println("Mallard Duck is flying");
   }
}
```

Then you have a client that loops through all ducks and makes them fly and quack.

```
import java.util.ArrayList;
public class DuckClient {

   static ArrayList<Duck> myDucks;

   public static void main(String[] args){
       myDucks = new ArrayList<>();
       myDucks.add( new MallardDuck());
       makeDucksFlyQuack();
   }

   static void makeDucksFlyQuack(){
      for(Duck duck: myDucks){
               duck.fly();
               duck.quack();
      }
   }
}
```

Now you have a **Turkey** interface.

```
public interface Turkey {

   public void gobble();
   public void fly();
}
```

How might we allow **Turkey** objects to be used by the same client?

Adapter design pattern = make some object compatible with another

We write an adapter class that

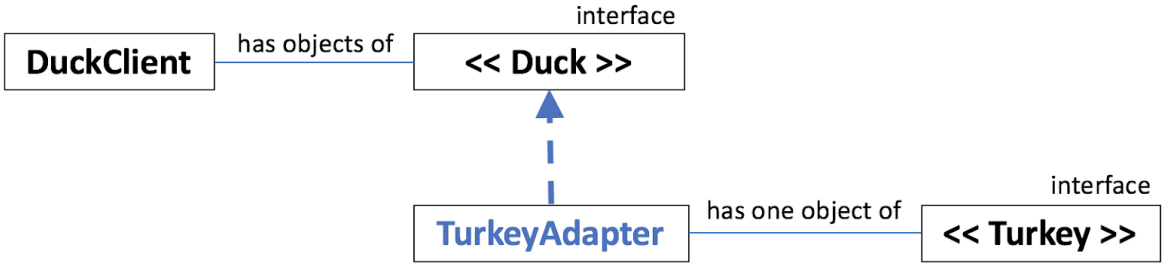
- has the same Duck interface and
- takes in a Turkey object

TurkeyAdapter is a Duck now

```
public class TurkeyAdapter implements Duck {

   Turkey turkey;

   TurkeyAdapter(Turkey turkey){
       this.turkey = turkey;

   }
   @Override
   public void quack() {
       //implement this
   }    turkey.gobble()

   @Override
   public void fly() {
       //implement this
   }   turkey.fly()
}
```

This material was taken from "HeadFirst-Design Patterns"

Explaining the Duck/Turkey Adapter Example




```
TurkeyAdapter turkeyAdapter = new TurkeyAdapter(new Turkey () {
  @Override
  public void gobble() {
  }

  @Override
  public void fly() {
  }
}

* This is like an anonymous class from Android Lesson 1
```

```
public class ClientMain {

    public static void main(String[] args) {
        Duck duck = new MallardDuck();
        Turkey turkey = new WildTurkey();

        Duck turkeyAdapter = new TurkeyAdapter(turkey);
        System.out.println("----Turkey");
        turkey.gobble();
        turkey.fly();

        System.out.println("----Duck");
        testIt(duck);

        System.out.println("----Turkey Adapter");
        testIt(turkeyAdapter);
    }

    public static void testIt(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

*WildTurkey here is presumably a class like MallardDuck

How to implement RecyclerView

To use RecyclerView in your app,

Step 1. ensure that you have the following dependency in your module-level gradle file

```
implementation 'androidx.recyclerview:recyclerview:1.0.0'
```

Step 2. Include the following widget tag in the Activity layout where you want to have the recyclerView.

```
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/charaRecyclerView"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

Step 3. In AndroidManifest.xml, add the attribute **android:exported=”true”** to the Activity element in which the RecyclerView is used. (Compulsory for SDK 31+)

Step 4. Assuming each data item is stored in a CardView, design the layout of each data item. CardView is an XML file!

Step 5. Decide the source of your data:

- Stored in the res folder
- SQLiteDatabase
- Cloud Database
- etc

This lesson shows you how to use data from a local SQLiteDatabase.

You would have written a Database Helper class.

Step 6. Write an Adapter class that extends the **`RecyclerView.Adapter<VH>`** class.

This class takes in your data source and is called by the Android runtime to display the data on the RecyclerView widget. This class also references the data item that you designed in step 3.

This will be explained in the next section.

Step 7 continues on the next page …

Step 7. In the java file for your activity, write code for the following

- Get a reference to the recyclerView widget using findViewById()
- Get an instance of an object that points to your dataSource
- Instantiate your Adapter
- Attach the adapter to your recyclerView widget
- Attach a Layout manager to your recyclerView widget. A LayoutManager governs how your widgets are going to be displayed. Since we are scrolling up and down, we will just need a LinearLayoutManager.

The sample code is here. You will need to adapt the code a little.

```
recyclerView = findViewById(R.id.charaRecyclerView);
dataSource = ??? ;
charaAdapter = new CharaAdapter(this, dataSource );
recyclerView.setAdapter(charaAdapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

This way of coding shows you how **delegation** is performed.

Delegation is the transferring of tasks from one object to a related object.

The **`RecyclerView`** object delegates

- the role of retrieving data to the **`RecyclerView.Adapter`** object.
- the role of managing the layout to the **`LinearLayoutManager`** object

Thus the RecyclerView object makes use of the Strategy Design Pattern.

A **`GridLayoutManager`** is also available.

Writing The RecyclerView Adapter - Static Inner Class

The RecyclerView adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Your RecyclerView Adapter should extend the **`RecyclerView.Adapter<VH>`** class.
`VH` is a generic class that subclasses **`RecyclerView.ViewHolder`**.
This is an **abstract class** without abstract methods.
Hence, Android is forcing you to subclass this class to use its methods.

This class is meant to hold references to the widgets in each data item layout.
Typically, we will write such a class as an inner class within the recyclerView adapter.

Hence, the classes are declared in the following way.

```
public class CharaAdapter extends
RecyclerView.Adapter<CharaAdapter.CharaViewHolder>{

   //code not shown
   static class CharaViewHolder extends RecyclerView.ViewHolder{
        //code not shown
   }    No override here
}       Has a constructor
```

Having designed your CardView layout for each data item, **`CharaViewHolder`** will contain instance variables that are meant to hold references to the widgets on the layout.
The references are obtained by calling **`findViewById()`** within the constructor.

Writing the RecyclerView Adapter - write the constructor and override three methods

The **constructor** should take in

- a Context object
- Object for your data source

The context object is used to get a layout inflator object to be used in **`onCreateViewHolder()`**.

`RecyclerView.Adapter<VH>` is an abstract class and you have to override three abstract methods.

`onCreateViewHolder()` is called by the run-time each time a new data item is added.
In the code recipe below:

- The CardView layout is inflated
- A reference to the layout in memory is returned **`itemView`**
- This reference **`itemView`** is passed to the constructor of **`CharaViewHolder`**
- **`CharaViewHolder`** uses this reference to get references to the individual widgets in the layout

Here's a typical recipe:

```
public CharaViewHolder onCreateViewHolder(@NonNull ViewGroup
viewGroup, int i) {
   View itemView = mInflater.inflate(R.layout.layout, viewGroup,
false);
   return new CharaViewHolder(itemView);
}
```

`onBindViewHolder()` is meant to

- get the appropriate data from your data source
- attach it to the widgets on each data item, according to the adapter position.

Hence, the data on row 0 of a table goes on position 0 on the adapter and so on.

`getItemCount()` is meant to return the total number of data items. Hence, if you return 0, nothing can be seen on the RecyclerView.

Seeing the connection

The RecyclerView adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Compare the RecyclerView implementation with the Duck/Turkey example

Example	RecyclerView component
Duck and DuckClient	RecyclerView
TurkeyAdapter	RecyclerView.Adapter<VH>
Turkey	ViewHolder

ViewHolder = charaViewHolder = item that contains each individual cardView

RecyclerView.Adapter<VH> = charaAdapter = adapt the information passed for flexibility

RecyclerView = recyclerView = client that loops through to set a certain stuff
(Uses strategy design pattern to manage the things)

Getting Each Item to Respond to Clicks

This is not in the list of TODOs, but it would be instructive to think about how it can be done.

Since we extend **`RecyclerView.ViewHolder`**, we have access to the parent class' methods. One method is **`getAdapterPosition()`**, which displays the ViewHolder's position on the RecyclerView.
Use this method to display a toast when each ViewHolder is clicked.

Option 1. Since a reference to the CardView layout is passed to the ViewHolder class, then you may call **`setOnClickListener`** on this reference within the constructor, and pass to it an anonymous class in the usual way.

Option 2. **`CharaViewHolder`** class can implement the **`View.OnClickListener`** interface. Then **`onClick`** has to be implemented as an instance method.
You still need to call **`setOnClickListener`** on the reference to the CardView layout.
What object do you pass to **`setOnClickListener`**?

Swiping To Delete

This is also not in the list of TODOs.

However, it would be instructive to consider how we might delete entries from the RecyclerView.

We are able to write code to delete a particular ViewHolder when it is swiped left/right.

First, add a method in **`DataSource`** to delete any particular entry, if one is not present.

```
void removeData(int i){
    dataArrayList.remove(i);
}
```

The code recipe is to create an instance of **`ItemTouchHelper`** and attach the RecyclerView instance to it.

```
ItemTouchHelper itemTouchHelper
              = new ItemTouchHelper(simpleCallback);
itemTouchHelper.attachToRecyclerView(recyclerView);
```

The constructor takes in an object that extends the **`ItemTouchHelper.SimpleCallback`** abstract class.

To use this class,

- Pass the direction of swiping that you want to detect to its constructor
- Override **`onSwipe()`**

From the documentation, the directions are specified via constants.

As you are going to use this object only once,

an acceptable practice is to use an anonymous abstract class.

As we are coding for swiping, we do not write any other code in **`onMove()`**.

```
ItemTouchHelper.SimpleCallback simpleCallback = new
ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT |
ItemTouchHelper.RIGHT ) {
   @Override
   public boolean onMove(@NonNull RecyclerView recyclerView,
@NonNull RecyclerView.ViewHolder viewHolder, @NonNull
RecyclerView.ViewHolder viewHolder1) {
       return false;
   }
```

```
    @Override
    public void onSwiped(@NonNull RecyclerView.ViewHolder
viewHolder, int i) {
            //code to delete the view goes here
    }
}
```

Two parameters are passed to **`onSwiped()`**:

- an instance of the ViewHolder that is currently being swiped
- the direction (change the variable name of the autogenerated code …)

Within **`onSwiped()`**, the tasks are

- **Downcast** the ViewHolder object so that you can use the instance variables or methods that you have defined
- Call your dataSource with the required information to delete the particular row in the database
- Display any other UI message e.g. a toast message saying a deletion has been happening
- Notify the RecyclerView adapter that the database has an item removed
  (Where did the **`getAdapterPosition()`** method come from?)

```
CharaAdapter.CharaViewHolder charaViewHolder
                    = (CharaAdapter.CharaViewHolder) viewHolder;
int position = charaViewHolder.getAdapterPosition();
dataSource.removeData(position);
//Write code to display a Toast if you'd like
charaAdapter.notifyDataSetChanged();
```

For your reference, the entire sequence of code is shown below.

```
charaAdapter = new CharaAdapter(this, dataSource);
recyclerView.setAdapter(charaAdapter);
recyclerView.setLayoutManager( new GridLayoutManager(this,3) );

ItemTouchHelper.SimpleCallback simpleCallback
= new ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT) {
    @Override
    public boolean onMove(@NonNull RecyclerView recyclerView,
        @NonNull RecyclerView.ViewHolder viewHolder,
        @NonNull RecyclerView.ViewHolder viewHolder1) {
        return false;
    }

    @Override
    public void onSwiped(@NonNull RecyclerView.ViewHolder viewHolder, int i) {

        CharaAdapter.CharaViewHolder charaViewHolder = (CharaAdapter.CharaViewHolder) viewHolder;
        int position = charaViewHolder.getAdapterPosition();
        dataSource.removeData(position);
        charaAdapter.notifyDataSetChanged();

    }
};

ItemTouchHelper itemTouchHelper = new ItemTouchHelper(simpleCallback);
itemTouchHelper.attachToRecyclerView(recyclerView);
```

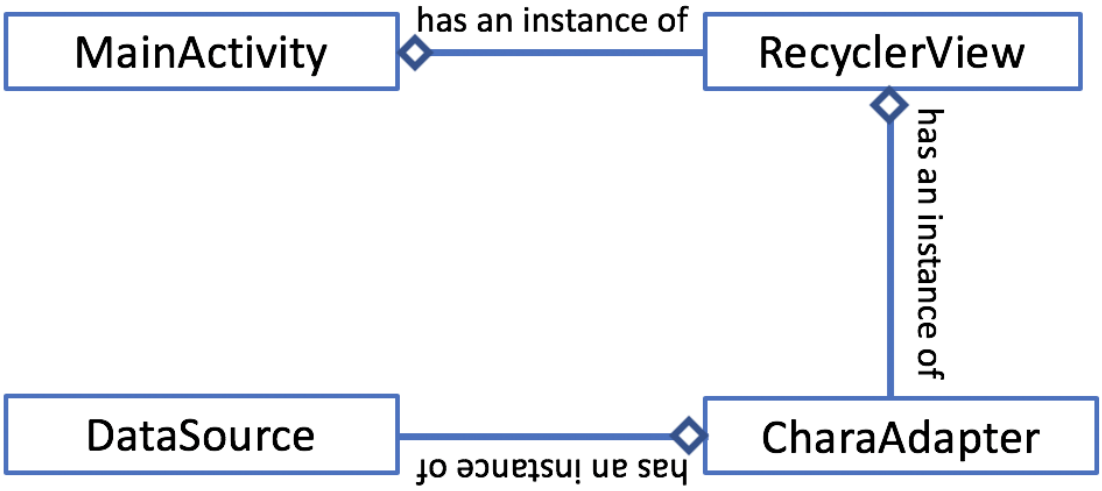
Building the RecyclerView (TODOs 11.x)

Before You Begin

- ensure that the res/drawables folder has a sufficient number of images (six or more)
- Examine the **`DataSource`** class WHERE?
- Examine the **`CharaAdapter`** class
- Examine the **`Utils.firstLoadImages`** method
- Examine **`pokemon.xml`** in res/layout

The relationship between the classes

The diagram below is simplified to show you the main ideas in this section.



Revision Questions (Lessons 1 - 3)

The code below has error(s). Suggest how to rectify it.

```
class A{

    class B{
        static void methodB(){
            System.out.println("Method B");
        }
    }
}
```

seems right, can use A.B.methodB()

How would you access the variable quack?

```
class Pond{

    static class Duck{
        int quack = 111;
    }
}
```

new Pond.Duck().quack

What is wrong with this code?

```
class A{

    String s = "AAA";

    void methodA(){
        System.out.println(s);
    }

    static class B{

        void methodB(){
            methodA(); methodA() MUST be static
        }
    }
}
```

static inner class cannot access enclosing class instance variables/methods

What are the statements that will be printed out?

```
public class TryCatchMystery {
   public static void main (String[] args) {
       try {
           method1();
           method2();
       } catch (IllegalArgumentException e) {
           System.out.println("main IllegalArgumentException");
       } catch (RuntimeException e) {
           System.out.println("main RuntimeException");
       }
   }
   public static void method1() {
       System.out.println("entered method1");
       try {
           method2();
       } catch (IllegalArgumentException e) {
           System.out.println("method1 IllegalArgumentException");
       }
       System.out.println("exited method1");
   }
   public static void method2() {
       System.out.println("entered method2");
       throw new IllegalArgumentException();
   }
}
```

"entered method1"
"entered method2"
"method1 IllegalArgumentException"
"exited method1"

"entered method2"
"main IllegalArgumentException"

Study the following code.

```
class Pond{

   private double size;
   private String type;

   private Pond(double size, String type){
       this.size = size;
       this.type = type;
   }
}
```

Without changing the code above (i.e. you may add code but not modify existing code), suggest **two** ways in which you could still allow Pond to be instantiated.

```
public static Pond makePond(double size, String type){
   return new Pond(size, type);
}


static class PondBuilder{
   private double size;
   private String type;

   PondBuilder() {
   }

   public PondBuilder setSize(double size){
      this.size = size;
      return this;
   }

   public PondBuilder setType(String type){
      this.type = type;
      return this;
   }

   public Pond build(){
      return new Pond(size, type);
   }
}
```

Coding Practice (Static Inner Class, Static Factory Method)

Write a class **`QuadraticEquation`** whose objective is to generate a quadratic equation with positive integer coefficients that have two real roots. The quadratic equation is represented by an object of the class **`QuadraticCoefficient`** that is a static inner class of **`QuadraticEquation`**.

The class **`QuadraticEquation`** has two private integer fields, **`aMax`** and **`cMax`** that represent the largest value of the coefficients a and c.
These values are initialized by the constructor, which is private.
The only way users instantiate your class is by a static factory method called **`getEquationGenerator()`**.
This static factory method allows the user to specify the values of **`aMax`** and **`cMax`**.

There is one public instance method **`getTwoRoots()`** that returns an instance of the inner class **`QuadraticCoefficient.`**
This method

- Generates a random value of a between 1 and **`aMax`**, both values inclusive
- Generates a random value of c between 1 and **`cMax`**, both values inclusive
- Generates a random value of b less than 100 such that the quadratic equations will have two real roots.
- Returns an instance of **`QuadraticCoefficient`** with the values of a, b and c generated

It has a static inner class **`QuadraticCoefficient`** that has

- Three private integer fields **`a`**, **`b`** and **`c`**
- Getter methods for these fields
- toString is overridden to display the following:
  When a = 7, b = 24 and c = 19 toString() returns the string
  “`y = 7x^2 + 24x + 19`”
  `(the quotes are not displayed on the screen)`

```
public class QuadraticEquation {
    private Integer aMax;
    private Integer cMax;

    private QuadraticEquation(Integer aMax, Integer cMax) {
        this.aMax = aMax;
        this.cMax = cMax;
    }

    public static QuadraticEquation getEquationGenerator(Integer aMax, Integer cMax) {
        return new QuadraticEquation(aMax, cMax);
    }

    public QuadraticCoefficient getTwoRoots() {
        Random random = new Random();
        Integer aMax = random.nextInt(1, this.aMax + 1);
        Integer cMax = random.nextInt(1, this.cMax + 1);

        Integer b2 = (int) Math.ceil(Math.sqrt(4 * aMax * cMax + 1));

        Integer b = random.nextInt(b2, 100);

        return new QuadraticCoefficient(aMax, b, cMax);
    }

    static class QuadraticCoefficient {
        private Integer a;
        private Integer b;
        private Integer c;

        public Integer getA() {
            return a;
        }
        public Integer getB() {
            return b;
        }
        public Integer getC() {
            return c;
        }

        private QuadraticCoefficient(Integer a, Integer b, Integer c){
            this.a = a;
            this.b = b;
            this.c = c;
        }

        @Override
        public String toString() {
            return "y = " + a + "x^2 + " + b + "x + " + c;
        }
    }
}
```