# Java Concepts for MAD

Updated: 6th August, 2019

This document describes a range of Java language features that are routinely used in Android development.

## 1 Lists

Java has certain in-built data structures (making up the "collections framework"). One of these is the `ArrayList` class, which is basically a wrapper around an array.

`ArrayList` implements the `List` interface, and typically you create an empty one like this:

```
List<MyClass> myList = new ArrayList<>(); // A list of 'MyClass' objects.
```

Why not simply an array? Mostly because `List`/`ArrayList` can *add and remove* items:

```
myList.add(new MyClass());     // Append to the end.
myList.add(0, new MyClass());  // Prepend to the start.
myList.add(10, new MyClass()); // Insert at index 10.
myList.addAll(anotherList);    // Add everything from another list.

myList.remove(10);  // Remove 10th element.
myList.remove(obj); // Search for 'obj' and remove it.
myList.clear();     // Remove everything.
```

We can retrieve information from a list like this:

```
int length = myList.size();   // Get current list size.
MyClass obj = myList.get(10); // Retrieve object at index 10.
```

We can iterate through all list elements with a "for-each" loop:

```
for(MyClass obj : myList) // Get each MyClass object in turn.
{
    System.out.println(obj.toString());
}
```

# 2 Nested Classes and Interfaces

In Java, you can declare classes and interfaces *inside* other classes/interfaces. We call these (the ones inside) *nested* classes/interfaces.

In particular, you will quickly encounter the nested interface `View.OnClickListener`, whose declaration looks something like this:

```
public abstract class View extends ... implements ...
{
    ... // View's constructors, methods and fields.
    public static interface OnClickListener // Nested interface
    {
        void onClick(View v);
    }
}
```

We can use this interface just like any other interface. We just have to remember to prefix it with the name of its containing class. Nested classes and interfaces are created for organisational reasons – to have related code in the same place.

## 2.1 Inner Classes

An inner class is a kind of nested class, where each inner class object is "owned" by a particular outer class object (not just the outer class itself). Inner classes are non-static.

```
public class OuterClass
{
    private int data = 42;

    private class InnerClass          // Inner class (has access to
    {                                 // an OuterClass object)
        void display()
        {
            System.out.println(data);
        }
    }

    public void doSomething()
    {
        InnerClass ic = new InnerClass();
        ic.display();
    }
}
```

We can access the `OuterClass` object's fields from within an inner class. We wouldn't be able to do this in a *static* nested class.

If you need to refer to the outer class object itself from within the inner class, you can write "`OuterClass.this`". (By itself, "`this`" inside an inner class only refers to the inner class object.)

## 2.2 Anonymous Classes

An anonymous class is a special inner class, which you define and instantiate all in one go. It's written as an *expression*, starting with the "`new`" keyword, followed by the name of its *super*class/interface (and its constructor parameters, if any), and then a pair of braces enclosing a set of methods and fields.

When you make an anonymous class, you're basically saying, "I need a *single object*, right here, that inherits from something and overrides one or more methods."

Here's a simple example:

```
public interface MyInterface
{
    int getValue();
}

public class OuterClass
{
    public MyInterface createObj()
    {
        MyInterface obj = new MyInterface()      // Anonymous class
        {                                        // expression
            @Override
            public int getValue() { return 42; }
        };
        return obj;
    }
}
```

You would not normally be able to write "`new MyInterface()`", of course, because interfaces cannot be directly instantiated. However, this is actually creating an anonymous class that implements `MyInterface` and provides a concrete implementation of `getValue()`.

This is done in Java/Android to handle events, particularly button presses. You can provide a `View.OnClickListener` object to a `Button` (or any other `View`), and your object's `onClick()` method will get called when the button is pressed.

How do you create a `View.OnClickListener` object? Usually as an anonymous class:

```
public class MyActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle b)
    {
        ...
        Button myButton = findViewById(R.id.theButton);
        myButton.setOnClickListener(
            new View.OnClickListener()      // Anonymous class that
            {                               // implements the nested
                @Override                   // interface OnClickListener.
                public void onClick(View v)
                {
                    ... // Take action
                }
            }
        );
    }
}
```

## 3 Singletons

A singleton is a class that is (a) designed to only have one instance, and (b) keeps track of its own single instance using a `static` field, which can be retrieved via a static method. They generally look like this:

```
public class MyData                                         // Singleton
{
    private static MyData instance = null;

    public static MyData getInstance() // Retrieve the singleton instance.
    {
        if(instance == null) // "Lazily" create the instance when needed.
        {
            instance = new MyData();
        }
        return instance;
    }

    ... // All the class's normal fields and methods.
}
```

(Some sources will insist that you also make the constructor private, to prevent the outside

world creating another instance, but this isn't generally that important.)

In general, singletons are problematic, because they make unit testing more difficult, and you should look for alternatives whenever possible. *However,* in Android development, they are extremely useful in providing a way for multiple activities to share data.

When one activity sends an `Intent` to start another activity (or to return a result) it goes through the OS. Any data transferred this way must be serialisable (able to be represented as a string, and converted back afterwards). This is easy and appropriate when the data is simple; e.g. individual numbers and strings.

However, it's awkward to transfer a whole *dataset* this way – a potentially large amount of complex data that, in principle, you should only need one copy of. If you rely on `Intents` for transferring datasets, then each activity would need to redundantly store its own copy, and you would need to re-send and update it whenever one of the activities changes it.

Instead, a dataset can be represented by (or accessed through) a singleton class. Each activity can then get a direct reference to it by calling `MyData.getInstance()`. The first activity to do so (it doesn't matter which) triggers the singleton's creation. There is only one copy of the data, and so no serialisation or resending is required.

You only need *one* singleton though. If you need to share more than one object between activities, you can simply have one singleton that aggregates all the objects to be shared – a single central access point. Multiple singletons can lead to confusion.

# 4 Class Literals

At various points in Android development, you will see the syntax "`MyClass.class`"[1]. This is called a *class literal*, and it's a way of referring to a class at runtime.

For instance, when creating an `Intent` for starting a new activity, you provide it a class literal that indicates which activity to start.

(FYI, class literals refer to actual objects of type `Class`. There are other ways to get these objects too, and there are some nifty things you can do with them, but that's beyond the scope of the unit.)

**End of Worksheet**

[1] This is also a filename, but here we're talking about an piece of the source code.