Table of Contents

**Java Syllabus1**

What to Learn2

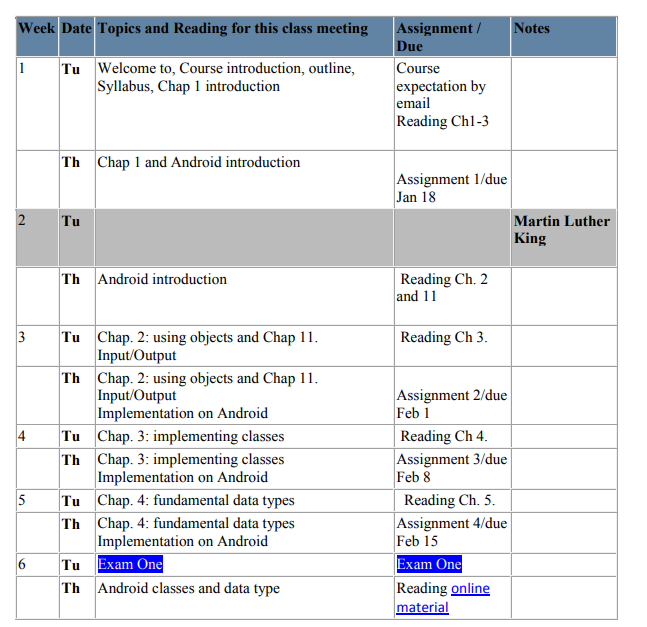
Type chapter title (level 3)3

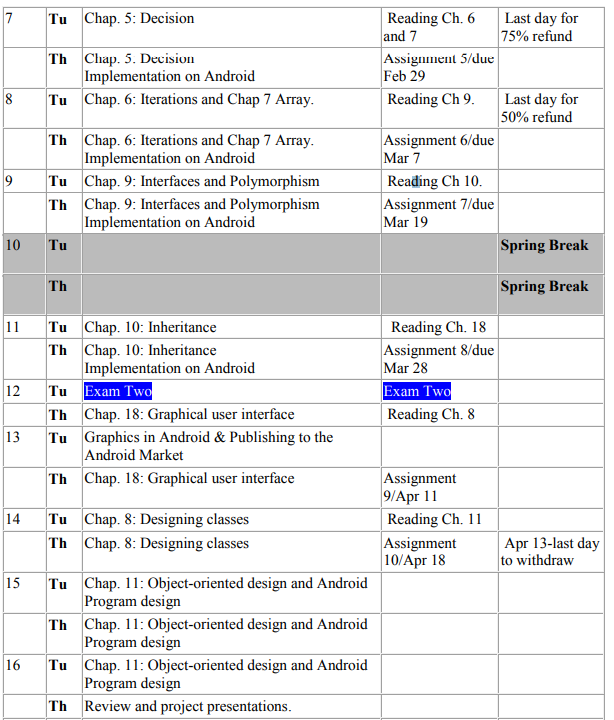
**Type chapter title (level 1)4**

Type chapter title (level 2)5

Type chapter title (level 3)6

**Java learning syllabus**





**What to learn**

1. Handlers
   * Java code gets compiled into a long list of instructions. Programs execute by going down this list.
   * Threads keep track where the execution is.
   * They also can run simultaneously on multi-core devices (on single core parallelism is simulated by 10ms interrupts).
   * Handlers are to put new messages in the event queue of a thread that has an event queue. Take the first message away from the queue, run the appropriate short piece of code for it and then go back to the start.
   * 2 main uses of Handlers
     1. To schedule messages and runnables to be executed at some point in future
     2. To enqueue an action to be performed on a different thread than yours
   * Threads are used to communicate between the threads and main activity
   * There are 2 ways of communication with handler
     1. Messages
     2. Runanables
   * To make a thread looper, in the run() method you can put a looper.prepare() function.
   * There are two main uses for a Handler: (1) to schedule messages and runnables to be executed at some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.
   * The typical syntax for .post is

**mHandler**.post(**new** Runnable() {

@Override

**public void** run() {

**mProgressBar**.setProgress(currentProgressCount);

}

});

1. Context

Putting it simply:

As the name suggests, it's the context of current state of the application/object. It lets newly-created objects understand what has been going on. Typically you call it to get information regarding another part of your program (activity and package/application).

You can get the context by invoking getApplicationContext(), getContext(), getBaseContext() or this (when in a class that extends from Context, such as the Application, Activity, Service and IntentService classes).

Typical uses of context:

**Creating new objects**: Creating new views, adapters, listeners:

TextView tv = new TextView(getContext());

ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(), ...);

**Accessing standard common resources**: Services like LAYOUT\_INFLATER\_SERVICE, SharedPreferences:

context.getSystemService(LAYOUT\_INFLATER\_SERVICE)

getApplicationContext().getSharedPreferences(\*name\*, \*mode\*);

**Accessing components implicitly**: Regarding content providers, broadcasts, intent

getApplicationContext().getContentResolver().query(Uri, ...);

1. Intent
   * Explicit: Is a messenger that runs between activities, services, or broadcast receivers to send a message to them when sth happens
   * Implicit: Is when you don’t know who to send the message to but you know the functionality, so you implicitly provide the intent
2. Super.myClass

Object oriented languages (of which Java is one), have the idea of "inheritance". You can create a class that inherits functionality from an existing class.

Here's an example. Let's say that you have a class called Car that describes the standard things that you can do with a car. It has methods such as turnLeft(float degrees), turnRight(float degrees), accelerate(), and brake().

Now, let's say that this Car class is very generic, and doesn't do all the stuff you want. For instance, it does all kinds of complicated math to handle turning the steering wheel, but it doesn't use the blinker first. So what you can do is this:

class BetterCar extends Car {

void turnLeft(float degrees) {

turnOnleftBlinker();

super.turnLeft(degrees);

turnOffLeftBlinker();

}

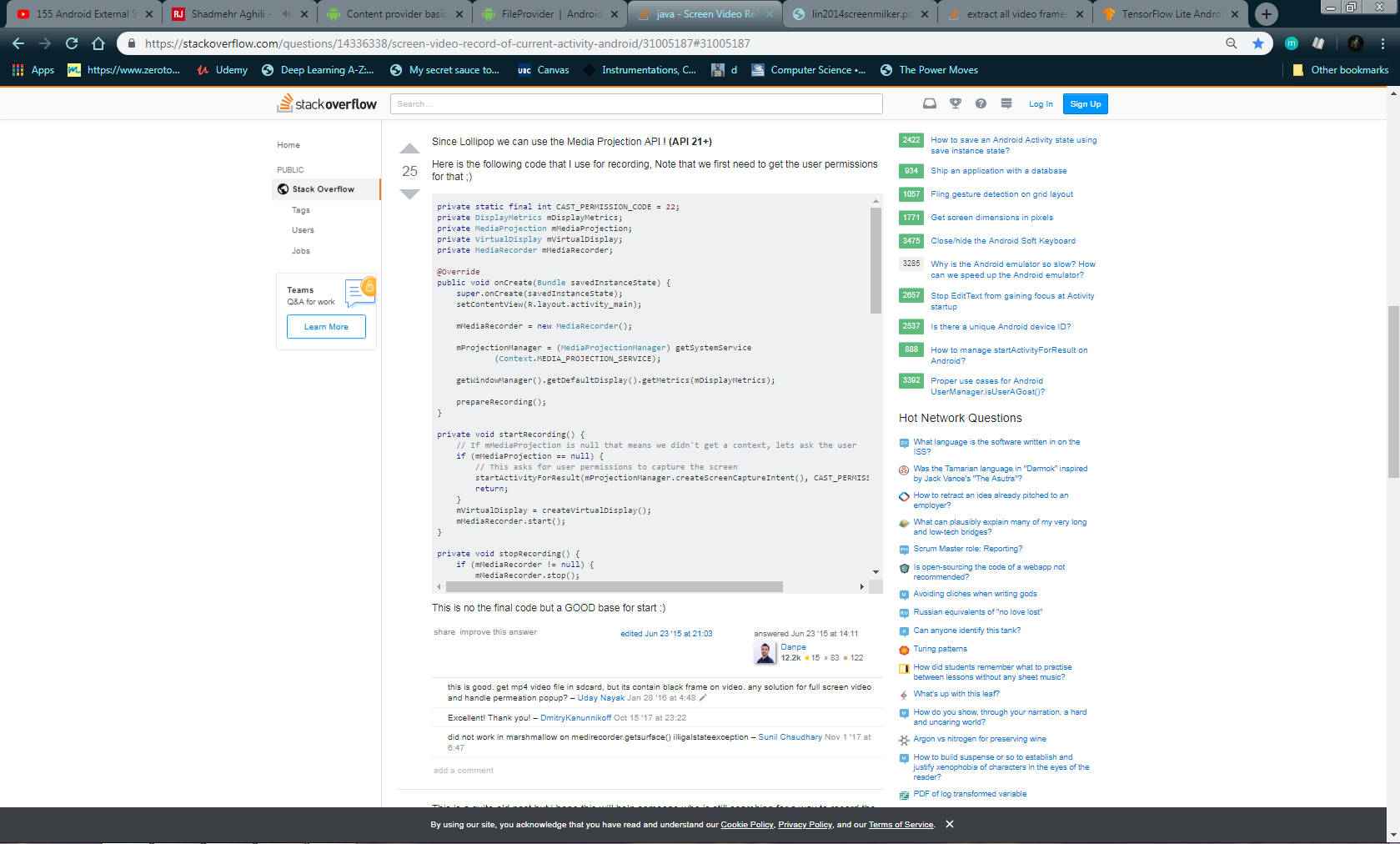
}

What you're doing here is creating a class called BetterCar that can do all the things that Car can do (it extends Car). You're also overriding the turnLeft method to do more stuff. But you still want to be able to access the functionality of turnLeft from the class that you're extending (the super class). You do this by using the key word "super", which references the instance of the super class that the current class is extending.

So your new turnLeft method turns on your blinker, executes the turnLeft method that you've inherited from Car, and then turns off the blinker

1. Extends and inheritance in general
2. Try catch block
   * Surrounds your code so that in case of an error or throwable, you know exactly where to look at.
3. Static meaning
   * It’s a way of grouping classes together. No top-level static classes allowed. Only inner or nested.
4. Media Manager
5. View V connection to the buttons
   * It is sent so that you have a handle on the View V.
   * If a button is pressed, using V you can handle the operation related to that button such as getting the id etc.
6. Layout inflator
   * Inflation is converting the xml file into a view in java
   * Layout inflator needs to use getSystemService() to create an object of layoutinflater
   * Only precompiled xml files by aapt (the compiler) can be inflated +
7. onImageListener/onImageClickListener
8. Intent Filters
   * Remember the messenger/mailman analogy.
   * The intent is the messenger which takes messages from the one activity/service/Broadcast Receiver
   * Intent filter is like a guard that makes sure you can pass through with your message.
   * Three things must match to let the intent through.
     1. Action
     2. Category
     3. Data

Possible solutions to screen buffering issue (This got fixed in the MNIST app)­­

* Save to the content provider storage
* SQLite storage
* Framebuffer
  + Seems reasonable reading the name :D
* Capturing a video (30fps possible) and extracting the frames for TF lite (code available for capturing)
* Grabbing bitmaps and exporting to tensorflow lite
* ffmpeg
* Research about screen milker
* The code in the picture below allows you to screen record with 30fps and you can add code to actually extract frames

1. Runnables

* A Runnable is basically a type of class (Runnable is an Interface) that can be put into a thread, describing what the thread is supposed to do.

The [Runnable Interface](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) requires of the class to implement the method run() like so:

public class MyRunnableTask implements Runnable {

public void run() {

// do stuff here

}

}

And then use it like this:

Thread t = new Thread(new MyRunnableTask());

t.start();

If you did not have the Runnable interface, the Thread class, which is responsible to execute your stuff in the other thread, would not have the promise to find a run() method in your class, so you could get errors. That is why you need to implement the interface.

### **Advanced: Anonymous Type**

Note that you do not need to define a class as usual, you can do all of that inline:

Thread t = new Thread(new Runnable() {

public void run() {

// stuff here

}

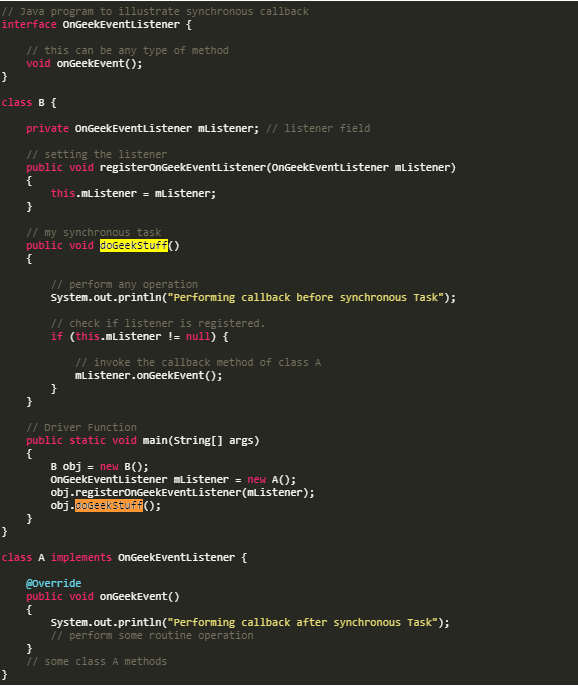
});

t.start();

This is similar to the above, only you don't create another named class.

* In Nathan’s code for demo app, there is no .run() function

1. Callback
   * The idea here is that, since Java does not have a function pointer, it needs a way of calling other methods and let them know that a running task (Synchrounus or Async) has been finished and the previous function can now continue operation as normal
     1. Example, when you get a lawnmower to cut the grass, you don’t have to wait there till the job is done. You give them your phone number and wait for them to call you telling it’s done



Output:

Performing callback before synchronous Task

Performing callback after synchronous Task

From what I understood in the above example, first the main function defines a new object from class B. This class has a private mListener. Then the main function creates a new object of type onGeekEventListener which is essentially an object of type A which can implement GeekEventListener.

Then the main function calls obj.registerOnGeekEventListener, which sends the mListener to the object B and calls this.mListener = mListener.

Now obj, has mListener as its listener

Then the doGeekStuff is called

1. Interface

n its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

interface Bicycle {

// wheel revolutions per minute

void changeCadence(int newValue);

void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);

}

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

class ACMEBicycle **implements** Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

// The compiler will now require that methods

// changeCadence, changeGear, speedUp, and applyBrakes

// all be implemented. Compilation will fail if those

// methods are missing from this class.

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void speedUp(int increment) {

speed = speed + increment;

}

void applyBrakes(int decrement) {

speed = speed - decrement;

}

void printStates() {

System.out.println("cadence:" +

cadence + " speed:" +

speed + " gear:" + gear);

}

}

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

1. ViewTreeObserver
   * The ViewTreeObserver is a way of measuring the size of a view before it is even drawn on the screen
   * It also allows you to listen to gestures and events happening globally
   * Any object in your layout files can be used for this purpose.
   * In
2. this

16

It refers to the current instance of a particular object, so you could write something like

public Object getMe() {

return this;

}

A common use-case of this is to prevent shadowing. Take the following example:

public class Person {

private final String name;

public Person(String name) {

// how would we initialize the field using parameter?

// we can't do: name = name;

}

}

In the above example, we want to assign the field member using the parameter's value. Since they share the same name, we need a way to distinguish between the field and the parameter. this allows us to access members of this instance, including the field.

public class Person {

private final String name;

public Person(String name) {

this.name = name;

}

}

1. Double Colon (::)
   * The **double colon (::) operator**, also known as **method reference operator** in Java, is used to call a method by referring to it with the help of its class directly. They behave exactly as the lambda expressions. The only difference it has from lambda expressions is that this uses direct reference to the method by name instead of providing a delegate to the method.
   * Syntax

<Class name>**::**<method name>

* + More Details Later

1. Synchronized methods
   * When a method is synchronized, it means that if a thread accesses and locks that method, another thread will not be able to access that and must wait for the first thread to finish the operation
2. From / .from
   * Complete this tomorrow, and add everything you learnt from the customview app
3. Observable
   * The most flexible mechanism to have the object Observable
     1. Includes 2 methods, addOnPropertyChangedCallback(addOnPropertyChangedCallback c) and removeOnPropertyChangedCallback(OnPropertyChangedCallback c)
     2. The OnPropertyChangredCallback must be notified when your fields have changed.
     3. **public abstract void** onPropertyChanged(Observable sender, **int** propertyId);
     4. The class must also be Bindable
4. @Bindable annotation
   * This tells the data binding framework that the property is observable and also generates an identifier in the BR class.
5. BR class
   * Similar to R class with the difference that it holds the binding resources

**Priorities**

First I need to understand how TensorFlow lite works. By this I mainly mean

* What sort of input does it accept for image classification
  + The classification of the images are done by taking bitmaps of the running camera and getting the pixels of the images. The bitmap is converted to byte buffer to do its job
  + If you’re able to capture the bitmap the rest of the code for the classifier is already written down.

Along with

* How to import TF lite to the app
* How to train models based on our new data
* How to test the model

RxJava stuff

* Operators: Stuff that transform one observable into another
* Find what each operator does from the link below.

<http://reactivex.io/documentation/operators.html>

**Slidenerd Videos pointers**

**Challenges of the Threads**

* Main Thread should be available to handle events from different Views and components otherwise the app becomes irresponsive to clicks
* ActivityManager and the WindowManager are 2 apps that monitor the app for responsiveness
* More than 5 seconds of irresponsive on main Thread behaviour and the Raptors will lose :D

**Void vs void**

* void is a return type for methods
* Void is a reference for functions that can only return void.
* Not too much difference but useful for Async tasks
* When using Async tasks, the only method that runs on other threads is the doInBackground

# **ML SECTION OF THE APP**

**What I learned form the TF Image classification app**

* <color name="semi\_transparent">#66000000</color>
  + The color code above is a semi-transparent color
* When making a tool bar
  + //In the activity's onCreate() method, call the activity's setSupportActionBar() method, and pass the activity's toolbar. This method sets the toolbar as the app bar for the activity
* When sth gets deprecated … (what then)
* @Overrride simply means that methods in the child class following the @override statement will override the methods in the parent class in case they have the same name and arguments.
  + Not necessary but useful to prevent unwanted errors
* RectF keeps data for four floating points that corresponds to 4 corners of a rectangle
* tflite and GPUdelegate need closing. (read below)
  + Both should call “.close” method and set equal to null
* Camera2 API is a stronger camera api in android that improves the capturing rates and the allows you to see whether you’re on front camera or back camera
* onImageAvailable: Callback that is called when a new image is available from ImageReader.

**General Knowledge**

* AI, a subset of machine learning, is composed of two modes: Training models with data, predictably called training; and using those models to make a prediction, called inference.
* Next, add the TensorFlow Lite dependency in your app’s build.gradle file.

dependencies {  
 implementation 'org.tensorflow:tensorflow-lite:+'  
}

This downloads the latest stable version, but typically you’ll want to give your library a set version number for stable builds. Since TensorFlow Lite is in active development, you might want to use the nightly builds when you’re testing things out.

* You can get constructor, getter, or setter for all you classes in Android studio using right click -> Generate-> Look at the list