Week 9: Android Lesson

[Refer to the Lesson 3 Lecture Note]

Agenda

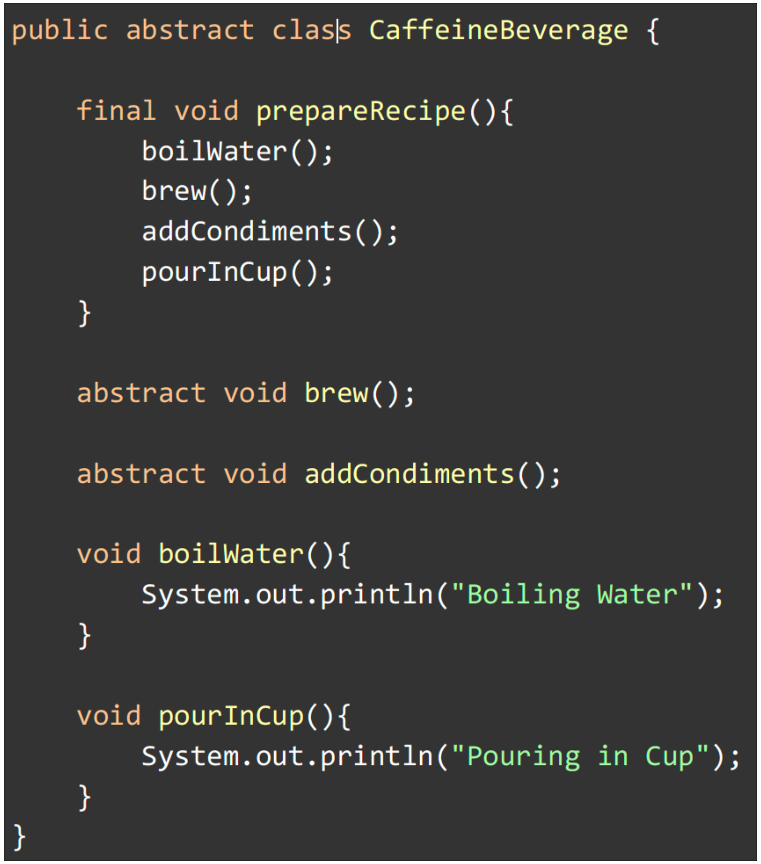
- Java: Template Method Design Pattern
- Java: Generic
- Concurrent Programming (Brief Introduction)
- Executor and Runnable Interface
- Sharing Data Between Main Thread and Child Thread
- Running Asynchronous Task in Android
- Building URL

- Cohort Class: Build XKCD Comic Reader App

Java: Template Method Design Pattern

Used when there is an algorithm or procedures with a fixed structure, but the implementation of some steps are left to the subclasses

Example:



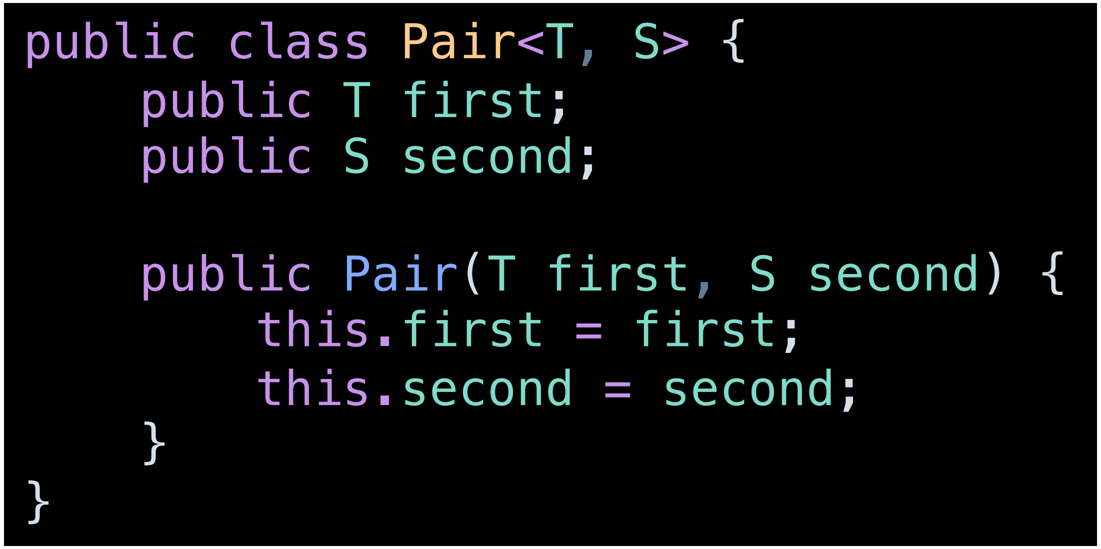
```java
public abstract class CaffeineBeverage {

    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");
    }
}
```

Java: Generic

Why generic?

to allow a type or method to operate on objects of **various types** while providing **compile-time type safety**.

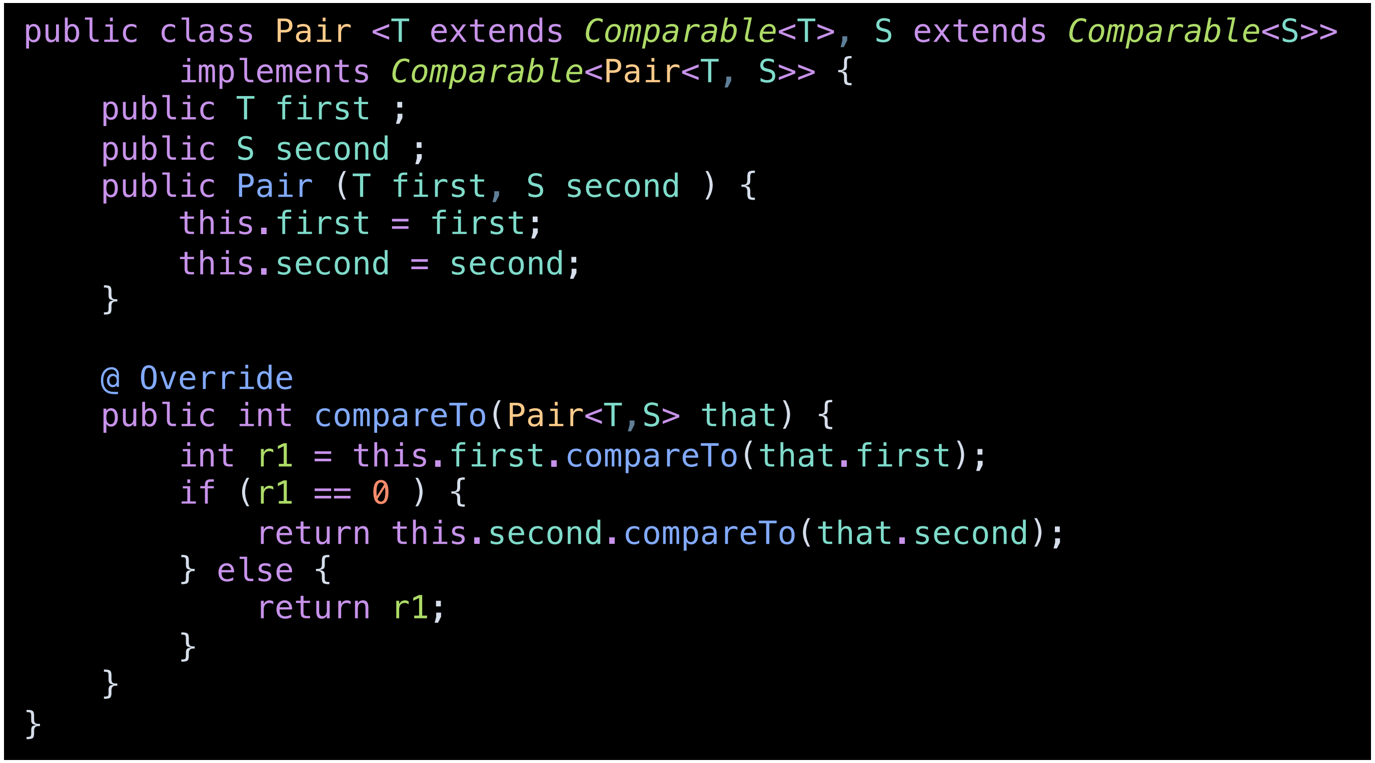
Example



```
public class Pair<T, S> {
    public T first;
    public S second;

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

Java: Bounded Type Parameters

- Suppose we would like to modify the above **Pair** class such that when two **Pair** objects are compared, we compare the **first** items, then if there is a tie, we go on to compare the **second** items.
- However, that would imply that the generic **T** and **S** are *not as generic* as before, because in order for **Pair** to be **Comparable**, you would need **T** and **S** to be **Comparable** too. Therefore, we have to place constraints on the type parameters (**Bounded Type Parameters**)

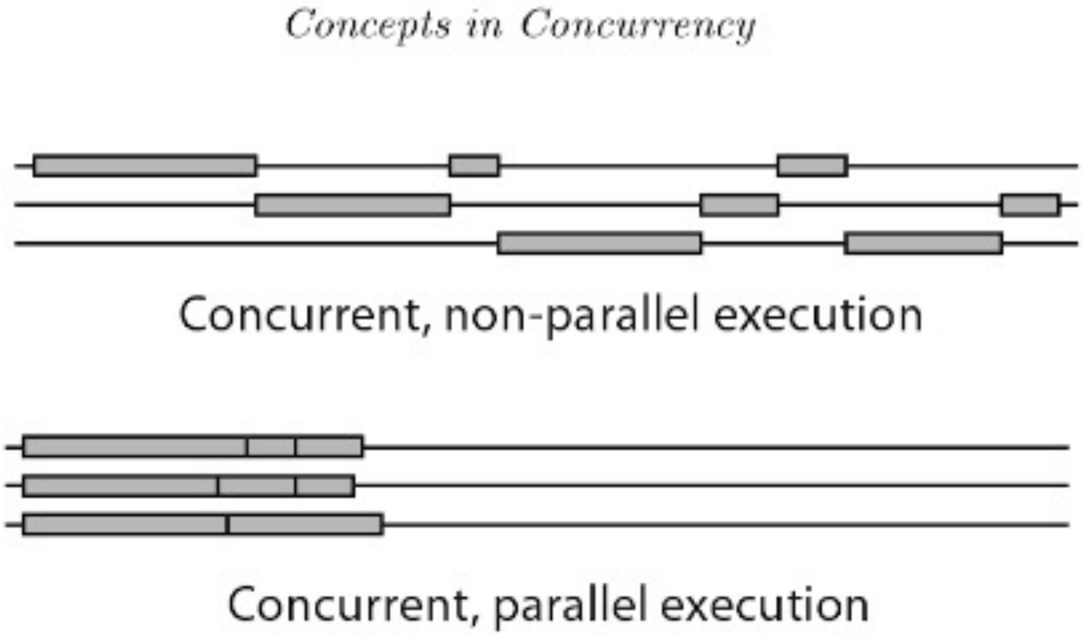


```java
public class Pair <T extends Comparable<T>, S extends Comparable<S>>
        implements Comparable<Pair<T, S>> {
    public T first ;
    public S second ;
    public Pair (T first, S second ) {
        this.first = first;
        this.second = second;
    }

    @ 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;
        }
    }
}
```

Brief Introduction to Concurrent Programming

- **Asynchronous** = **Non-blocking** -> A new task can be started without having to wait until the previous task is finished.
- **Synchronous** = Execute tasks **sequentially** -> Cannot start another task until the current task is finished.
- **Concurrent** = **Multiple tasks** are executed at the same time, but not necessarily more than 1 active threads.
- **Parallel** = **Multiple tasks** (or active threads) are executed at the **same time** by **multiple cores**
- **Thread** = smallest sequence of programmed instructions (task) that can be managed independently by a **scheduler** (executor)

Brief Introduction to Concurrent Programming



Concepts in Concurrency

Concurrent, non-parallel execution

Concurrent, parallel execution

https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism

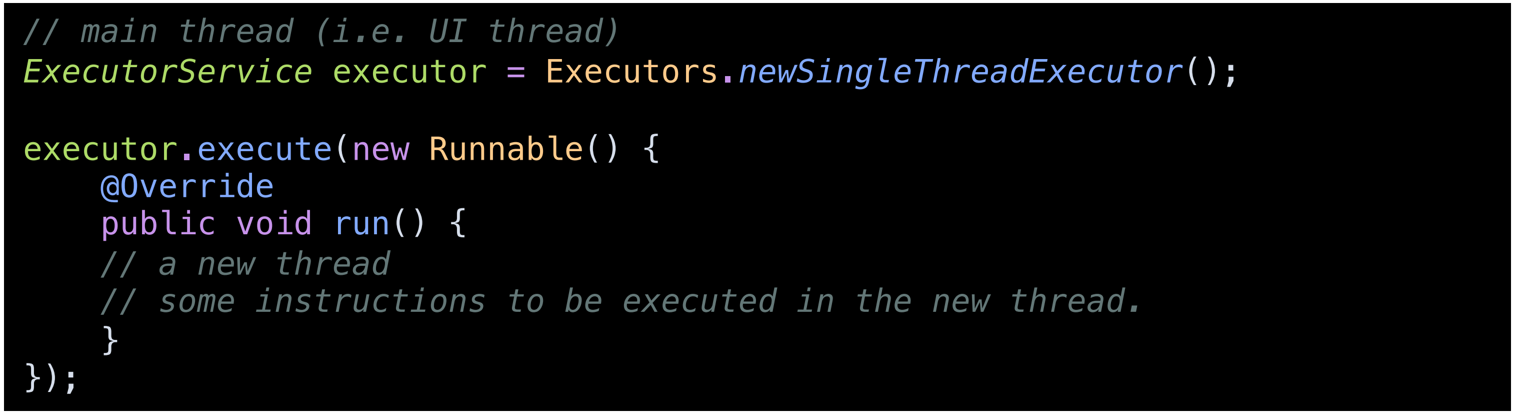
Concurrent Programming in Android

- By default, everything that your app does is executed in a **single thread** called *main thread* or *UI thread*
- Performing **long operations** in the UI thread, such as network access or database queries will **block** the main thread, thus making the app unresponsive during the operation
- **Solution**: create a *background thread* (also called *worker thread*) to execute the background tasks
- Java provides package for concurrent programming: java.util.concurrent

Executor and Runnable Interface

- **Executor** class allows **access** to the pool of **threads**
- **Runnable** interface denotes the **tasks to be executed** as a thread

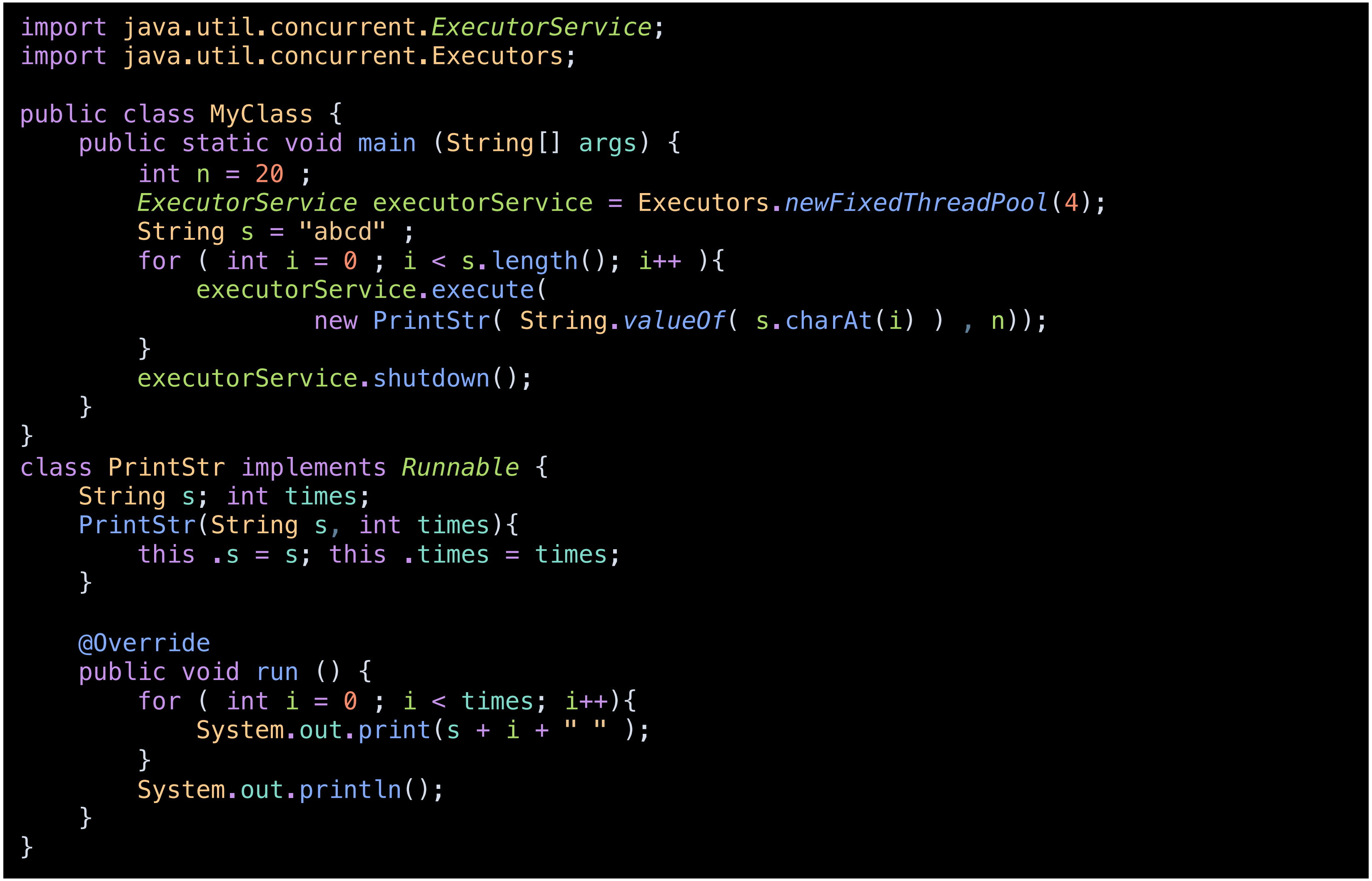
Executor and Runnable Interface



```
// 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)**.
- ***run()*** - this is an abstract method defined in the **Runnable** interface. An instance of **Runnable** interface must implement/override this method.

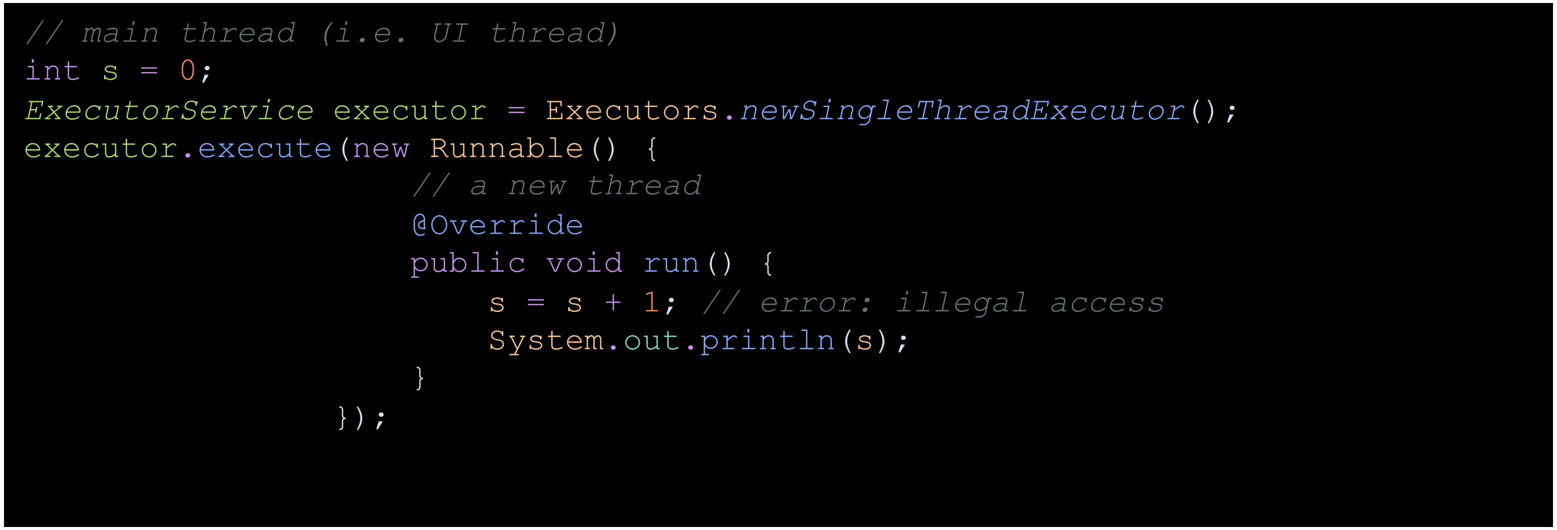
Executor and Runnable Interface Example



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MyClass {
    public static void main (String[] args) {
        int n = 20 ;
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        String s = "abcd" ;
        for ( int i = 0 ; i < s.length(); i++ ){
            executorService.execute(
                    new PrintStr( String.valueOf( s.charAt(i) ) , n));
        }
        executorService.shutdown();
    }
}
class PrintStr implements Runnable {
    String s; int times;
    PrintStr(String s, int times){
        this .s = s; this .times = times;
    }

    @Override
    public void run () {
        for ( int i = 0 ; i < times; i++){
            System.out.print(s + i + " " );
        }
        System.out.println();
    }
}
```

Immutable and Generic Class

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

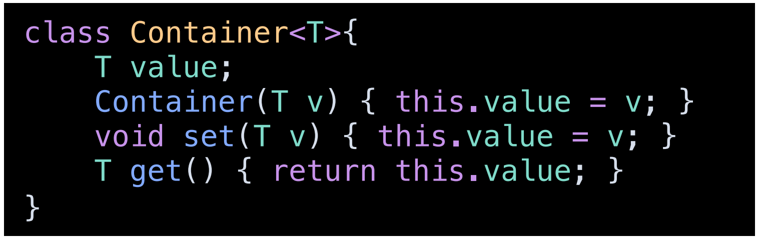


```
// main thread (i.e. UI thread)
int s = 0;
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
                    // a new thread
                    @Override
                    public void run() {
                        s = s + 1; // error: illegal access
                        System.out.println(s);
                    }
                });
```

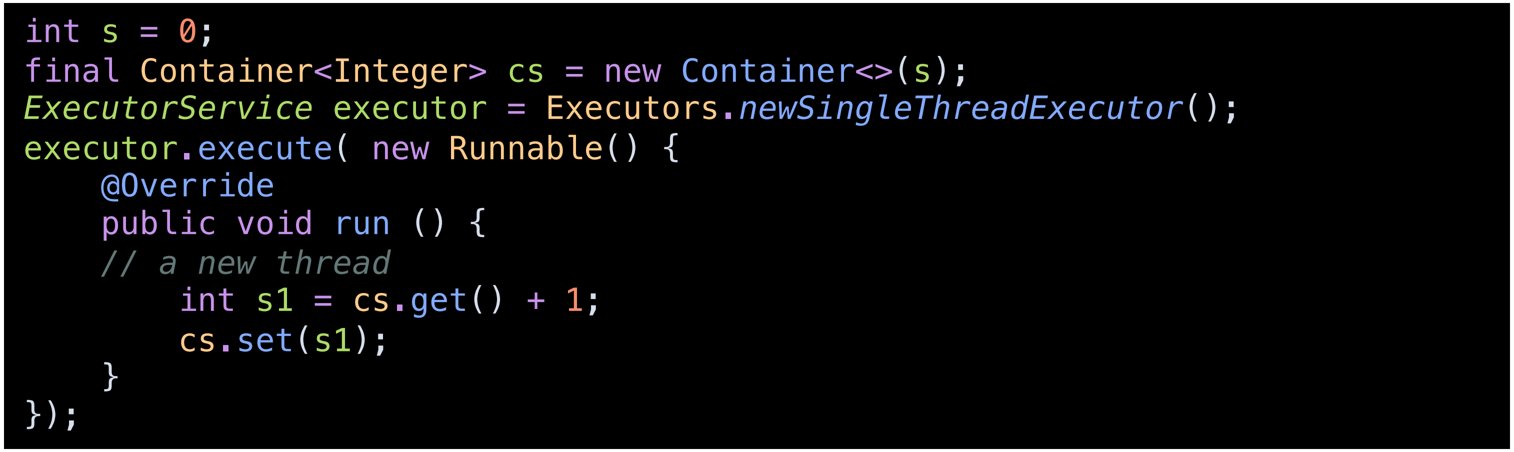
Immutable and Generic Class

- **Problem:**
  Shared data cannot be modified in child class

- **Solution:**
  Making use of **generic** as a container of an object



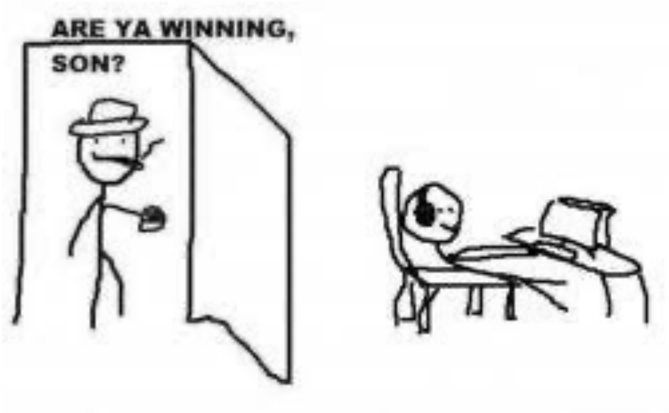
```
class Container<T>{
    T value;
    Container(T v) { this.value = v; }
    void set(T v) { this.value = v; }
    T get() { return this.value; }
}
```



```
int s = 0;
final Container<Integer> cs = new Container<>(s);
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute( new Runnable() {
    @Override
    public void run () {
    // a new thread
        int s1 = cs.get() + 1;
        cs.set(s1);
    }
});
```

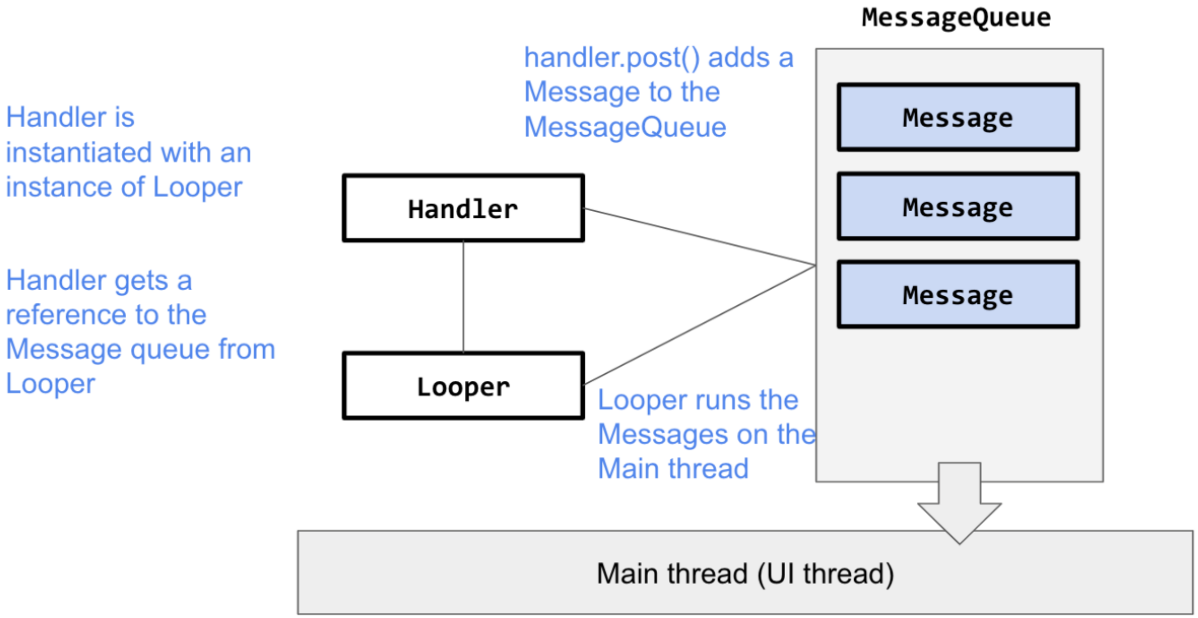
Running Asynchronous Task in Android

- Problem:
  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



Running Asynchronous Task in Android

- ***Main thread*** executes all task/events (***Message***) from ***MessageQueue***
- ***Looper*** manages ***MessageQueue*** and is abstracted away from us
- ***Message*** can be ***Runnable*** object
- ***Handler*** object allows you to send ***Message*** to ***MessageQueue***


MessageQueue
handler.post() adds a Message to the MessageQueue
Handler is instantiated with an instance of Looper
Handler
Message
Message
Message
Handler gets a reference to the Message queue from Looper
Looper
Looper runs the Messages on the Main thread
Main thread (UI thread)

Running Asynchronous Task in Android

```
// main thread (i.e. UI thread)
ExecutorService executor = Executors.newSingleThreadExecutor();
Looper uiLooper = Looper.getMainLooper(); // get the main looper
final Handler handler = new Handler(uiLooper); // get the handler for the main
thread

executor.execute( new Runnable() {
    @Override
    public void run () {
    // instructions performed in the child thread
    // ...
        handler.post( new Runnable() {
            @Override
            public void run () {
            //UI Thread will receive and run this
            }
        });
    }
});
```

There are two ***Runnable*** objects. Why?

Building URL

- A URL is a URI that refers to a particular website
- URL consists of 3 components: **Scheme, Authority, and Path**
- You can create URL object by passing a hardcoded string. But in Java, we have URI builder to help us writing the string without having to worry with all the symbols between the 3 components
- For example, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html has the following components:
  - Scheme: https
  - Authority: docs.oracle.com
  - Path: javase/8/docs/api/java/util/concurrent/Executor.html