Mobile programming

Omid Jafarinezhad

Android 01

Effective Java for Android

- Builder Pattern
- Static Factory Methods
- Static Inner Classes
- Return Empty Collection
- Use StringBuilder
- Force No Instantiation
- Avoid Mutability
- Concurrency and Multithreading

Builder Pattern

When you have an object that requires more than ~3 constructor parameters, use a builder to construct the object. It might be a little more verbose to write but it scales well and it's very readable. If you are creating a value class, consider AutoValue.

```java
class Movie {
    static class Builder {
        String title;
        Builder withTitle(String title) {
            this.title = title;
            return this;
        }
        Movie build() {
            return new Movie(title);
        }
    }

    private Movie(String title) {
    [...]
    }
}
// Use like this:
Movie matrix = new Movie.Builder().withTitle("The Matrix").build();
```

Static Factory Methods

```
class Movie {
    [...]
    public static Movie create(String title) {
        return new Movie(title);
    }
}
```

Static Inner Classes

If you define an inner class that does not depend on the outer class, don't forget to define it as static. Failing to do so will result in each instance of the inner class to have references to the outer class.

```
class Movie {
    [...]
    static class MovieAward {
        [...]
    }
}
```

Return Empty Collection

When having to return a list/collection with no result avoid null. Returning an empty collection leads to a simpler interface .

Prefer to return the same empty collection rather than creating a new one.

```
List<Movie> latestMovies() {
    if (db.query().isEmpty()) {
        return Collections.emptyList();
    }
    [...]
}
```

emptyList() might not create a new object with each call. It returns an immutable list, i.e., a list to which you cannot add elements

```
public static final <T> List<T> emptyList() {
    return (List<T>) EMPTY_LIST;
}
```

Use StringBuilder

Having to concatenate a few Strings, + operator might do. Never use it for a lot of String concatenations; the performance is really bad. Prefer a StringBuilder instead.

```java
String list[]=new String[]{"A","C","D","F"};

public String print(String arr[]){

    StringBuilder sb = new StringBuilder();
    String prefix = "";
    for (String str : arr)
    {
        sb.append(prefix);
        prefix = ",";
        sb.append(str);
    }

    System.out.println("WithCommas"+sb);
    return sb;
}
```

Force No Instantiation

If you do not want an object to be created using the new keyword, enforce it using a private constructor. Especially useful for utility classes that contain only static functions.

```
class MovieUtils {
    private MovieUtils() {}
    static String titleAndYear(Movie movie) {
        [...]
    }
}
```

Avoid Mutability

Immutable is an object that stays the same for its entire lifetime. All the necessary data of the object are provided during its creation. There are various advantages to this approach like simplicity, thread-safety and shareability.

```
class Movie {
    [...]
    Movie sequel() {
        return Movie.create(this.title + " 2");
    }
}
// Use like this:
Movie toyStory = Movie.create("Toy Story");
Movie toyStory2 = toyStory.sequel();
```

Concurrency and Multithreading

Multithreading Benefits

- Better resource utilization
- Simpler program design in some situations
- More responsive programs

Multithreading Costs

- More complex design
- Context Switching Overhead

Thread

```
public class MyThread extends Thread {

    public void run(){
        // Process.myTid() is the linux thread ID
        // Thread.getId() is a simple sequential long number
        Log.i(MainActivity.TAG, msg: "MyThread ]]>> " +
                " pid: " + android.os.Process.myPid()+
                " tid: " + android.os.Process.myTid()+
                " id: " + Thread.currentThread().getId());
    }
}
```

```java
public class MainActivity extends AppCompatActivity {

    protected static final String TAG = "prj1-thread";

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

        ThreadSubclass();
//        ThreadRunnable();
//        AnonymousRunnable();
//        LambdaRunnable();
//        RaceCondition();
//        SynchronizedThread();
    }
```

```java
void ThreadSubclass() {
    Log.i(TAG, msg: "ThreadSubclass, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());

    MyThread myThread = new MyThread();
    myThread.start();

    Log.i(TAG, msg: "ThreadSubclass, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());
}
```

Emulator Nexus_5X_API_23 A | edu.sharif.prj01 (7755) | Verbose | prj1-thread | Regex | Show only selecte

```
02-03 07:12:58.290 7755-7755/edu.sharif.prj01 I/prj1-thread: ThreadSubclass, Main Thread ]]>> pid: 7755 tid: 7755 id: 1
02-03 07:12:58.291 7755-7769/edu.sharif.prj01 I/prj1-thread: MyThread ]]>>  pid: 7755 tid: 7769 id: 431
02-03 07:12:58.291 7755-7755/edu.sharif.prj01 I/prj1-thread: ThreadSubclass, Main Thread ]]>> pid: 7755 tid: 7755 id: 1
```

Runnable

```java
public class MyRunnable implements Runnable {

    public void run(){
        Log.i(MainActivity.TAG, msg: "MyRunnable ]]>> " +
                " pid: " + android.os.Process.myPid()+
                " tid: " + android.os.Process.myTid()+
                " id: " + Thread.currentThread().getId());
    }
}
```

```java
void ThreadRunnable(){
    Log.i(TAG, msg: "ThreadRunnable, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());

    MyRunnable myRunnable = new MyRunnable();
    Thread thread = new Thread(myRunnable);
    thread.start();

    Log.i(TAG, msg: "ThreadRunnable, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());
}
```

Anonymous Runnable

```
void AnonymousRunnable(){
    Thread thread = new Thread(new Runnable(){
        public void run(){
            Log.i(MainActivity.TAG, msg: "AnonymousRunnable ]]>> " +
                    " pid: " + android.os.Process.myPid()+
                    " tid: " + android.os.Process.myTid()+
                    " id: " + Thread.currentThread().getId());
        }
    });
    thread.start();

    Log.i(TAG, msg: "AnonymousRunnable, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());

}
```

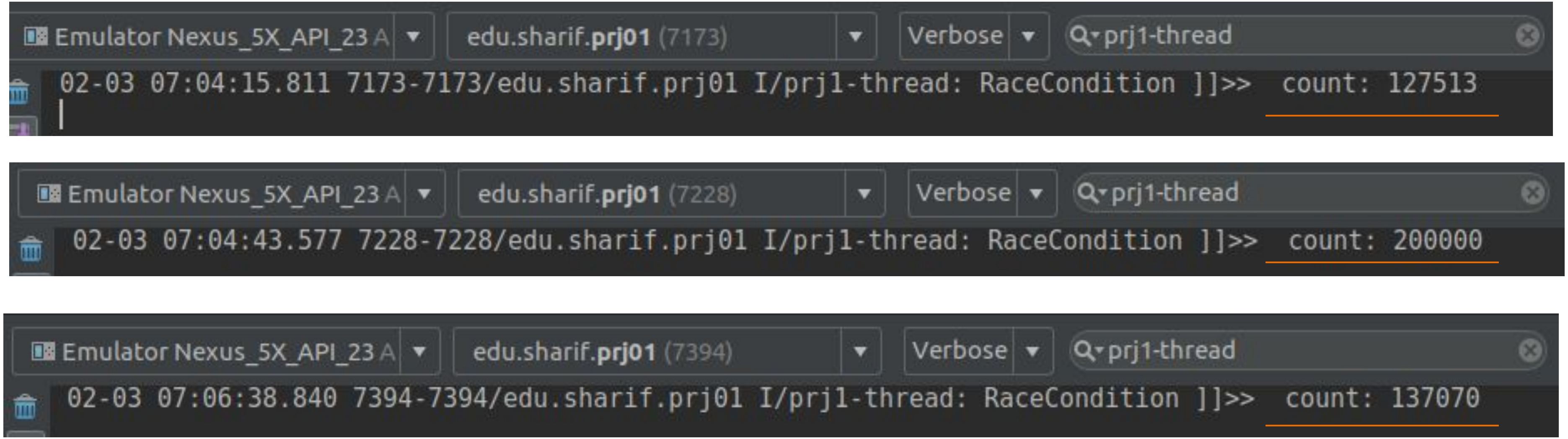
Lambda Runnable

```java
void LambdaRunnable(){
    Runnable lambda = () ->
            Log.i(MainActivity.TAG, msg: "LambdaRunnable ]]>> " +
                    " pid: " + android.os.Process.myPid()+
                    " tid: " + android.os.Process.myTid()+
                    " id: " + Thread.currentThread().getId());

    Thread thread = new Thread(lambda);
    //Thread thread = new Thread(() -> Log.i(MainActivity.TAG, ""));
    thread.start();
    Log.i(TAG, msg: "LambdaRunnable, Main Thread ]]>>" +
            " pid: " + android.os.Process.myPid() +
            " tid: " + android.os.Process.myTid() +
            " id: " + Thread.currentThread().getId());
}
```

Race conditions

Race conditions occur only if **multiple threads are accessing the same resource, and one or more of the threads write to the resource**. ***If multiple threads read the same resource race conditions do not occur,***

```java
public class Counter {
    protected long count = 0;

    public void doWork(){
        for (int i = 0; i<100000;i++){
            this.count = this.count + 1;
        }
    }

    public void safeDoWork(){
        synchronized(this){
            for (int i = 0; i<100000;i++){
                this.count = this.count + 1;
            }
        }

//        for (int i = 0; i<100000;i++){
//            synchronized (this){
//                this.count = this.count + 1;
//            }
//        }
    }
}
```


Emulator Nexus_5X_API_23 A
edu.sharif.prj01 (7173)
Verbose
prj1-thread
02-03 07:04:15.811 7173-7173/edu.sharif.prj01 I/prj1-thread: RaceCondition]]>> count: 127513
Emulator Nexus_5X_API_23 A
edu.sharif.prj01 (7228)
Verbose
prj1-thread
02-03 07:04:43.577 7228-7228/edu.sharif.prj01 I/prj1-thread: RaceCondition]]>> count: 200000
Emulator Nexus_5X_API_23 A
edu.sharif.prj01 (7394)
Verbose
prj1-thread
02-03 07:06:38.840 7394-7394/edu.sharif.prj01 I/prj1-thread: RaceCondition]]>> count: 137070

Preventing Race Conditions

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

```java
public class Counter {
    protected long count = 0;

    public void doWork(){
        for (int i = 0; i<100000;i++){
            this.count = this.count + 1;
        }
    }

    public void safeDoWork(){
        synchronized(this){
            for (int i = 0; i<100000;i++){
                this.count = this.count + 1;
            }
        }

//          for (int i = 0; i<100000;i++){
//              synchronized (this){
//                  this.count = this.count + 1;
//              }
//          }
    }
}
```

```java
    void SynchronizedThread(){
        Counter c = new Counter();

        Thread thread1 = new Thread(() -> c.safeDoWork());
        Thread thread2 = new Thread(() -> c.safeDoWork());

        thread1.start();
        thread2.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Log.i(MainActivity.TAG, msg: "SynchronizedThread ]]>> " +
                " count: " + c.count);
    }
}
```

Critical Section Throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section

```java
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;

    public void add(int val1, int val2){
        synchronized(this){
            this.sum1 += val1;
            this.sum2 += val2;
        }
    }
}
```

```java
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;

    private Integer sum1Lock = new Integer( value: 1);
    private Integer sum2Lock = new Integer( value: 2);

    public void add(int val1, int val2){
        synchronized(this.sum1Lock){
            this.sum1 += val1;
        }
        synchronized(this.sum2Lock){
            this.sum2 += val2;
        }
    }
}
```

Synchronized Blocks in Static Methods

```
public class MyClass {

  public static synchronized void log1(String msg1, String msg2){
     log.writeln(msg1);
     log.writeln(msg2);
  }


  public static void log2(String msg1, String msg2){
     synchronized(MyClass.class){
        log.writeln(msg1);
        log.writeln(msg2);
     }
  }
}
```

Thread Safety

- Local Variables
- Local Object References
  - If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects **as long as none of these methods or objects make the passed object available to other threads**.

```java
public void LocalVariables(){
    long threadSafeInt = 0;
    threadSafeInt++;
}
```

```java
public void LocalObjectReferences(){
    LocalObject localObject = new LocalObject();
    method2(localObject);
    Log.i(MainActivity.TAG, msg: "val:" + localObject.getVal());
}

public void method2(LocalObject val){
    val.setVal(200);
}

static class LocalObject{
    int val;

    void setVal(int v){
        val = v;
    }

    int getVal(){
        return val;
    }
}
```

See ThreadSafe class
in edu.sharif.prj01

Thread Safety - Object Member Variables

Object member variables (fields) are stored on the heap along with the object.

Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method ***is not thread safe***

```java
public class NotThreadSafe{
    StringBuilder builder = new StringBuilder();

    public void add(String text){
        this.builder.append(text);
    }

    public static class MyRunnable implements Runnable{
        NotThreadSafe instance = null;

        public MyRunnable(NotThreadSafe instance){
            this.instance = instance;
        }

        public void run(){
            this.instance.add("some text");
        }
    }
}
```

```java
void ObjectMemberVariablesNotThreadSafe(){
    NotThreadSafe sharedInstance = new NotThreadSafe();

    Thread t1 = new Thread(new NotThreadSafe.MyRunnable(sharedInstance));
    Thread t2 = new Thread(new NotThreadSafe.MyRunnable(sharedInstance));
    t1.start();
    t2.start();
}
```

Thread Safety and Immutability

- Race conditions *occur only if multiple threads are accessing the same resource, and one or more of the threads write to the resource*

- **If multiple threads read the same resource race conditions do not occur**
    - make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe

```java
// Notice how the value for the ImmutableValue instance
// is passed in the constructor.
// Notice also how there is no setter method. Once an
// ImmutableValue instance is created you
// cannot change its value. It is immutable.
public class ImmutableValue{

    private int value = 0;

    public ImmutableValue(int value){
        this.value = value;
    }

    public int getValue(){
        return this.value;
    }
}
```

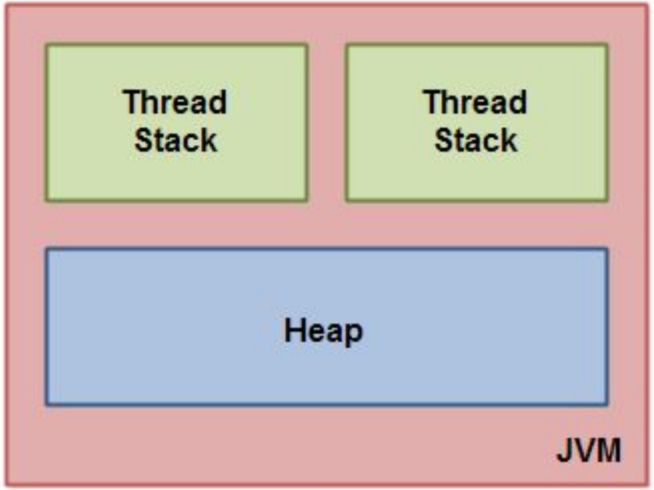
```java
// Notice how the value for the ImmutableValue instance
// is passed in the constructor.
// Notice also how there is no setter method. Once an
// ImmutableValue instance is created you
// cannot change its value. It is immutable.
public class ImmutableValue{

    private int value = 0;

    public ImmutableValue(int value){
        this.value = value;
    }

    public int getValue(){
        return this.value;
    }

    // If you need to perform operations on the ImmutableValue
    // instance you can do so by returning a new instance with
    // the value resulting from the operation.
    public ImmutableValue add(int valueToAdd){
        return new ImmutableValue(this.value + valueToAdd);
    }
}
```

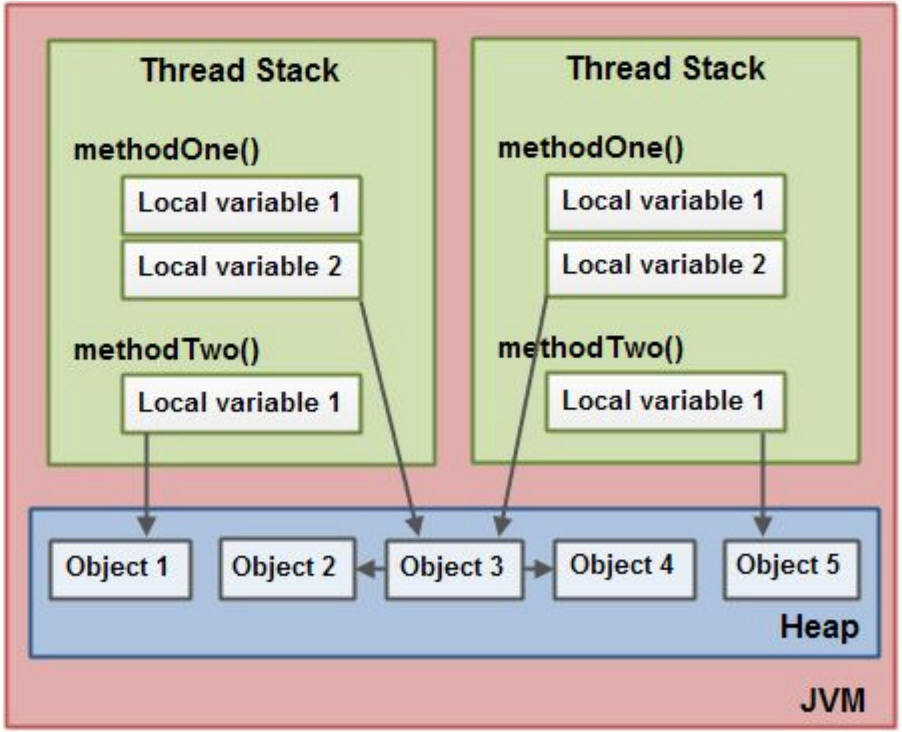
Immutability - The Reference is not Thread Safe!

It is important to remember, that even if an object is immutable and thereby thread safe, **the reference to this object(immutable object) may not be thread safe**

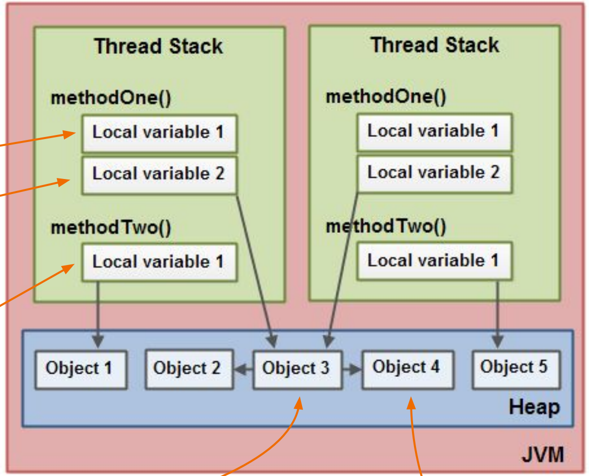
```java
// The class holds a reference to an
// ImmutableValue instance. Notice how it is possible
// to change that reference through both the setValue()
// and add() methods. Therefore, even if the Calculator
// class uses an immutable object internally, it is not
// itself immutable, and therefore not thread safe.
// In other words: The ImmutableValue class is thread safe,
// but the use of it is not.
public class ImmutableReference {
    private ImmutableValue currentValue = null;

    public ImmutableValue getValue(){
        return currentValue;
    }

    public void setValue(ImmutableValue newValue){
        this.currentValue = newValue;
    }

    public void add(int newValue){
        this.currentValue = this.currentValue.add(newValue);
    }
}
```

Java Memory Model



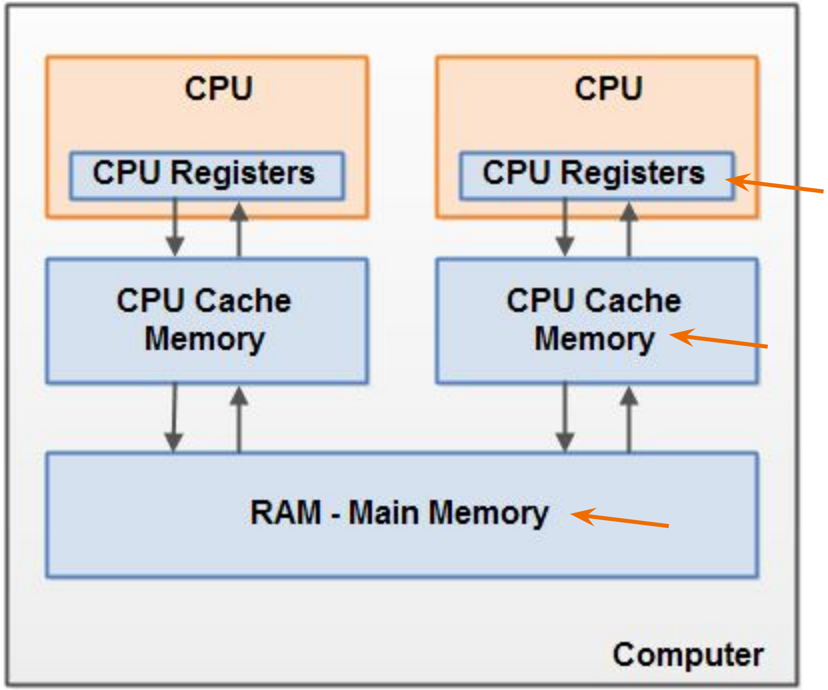


```
public class MyRunnable implements Runnable() {

    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
        Integer localVariable1 = new Integer(99);

        //... do more with local variable.
    }
}
```

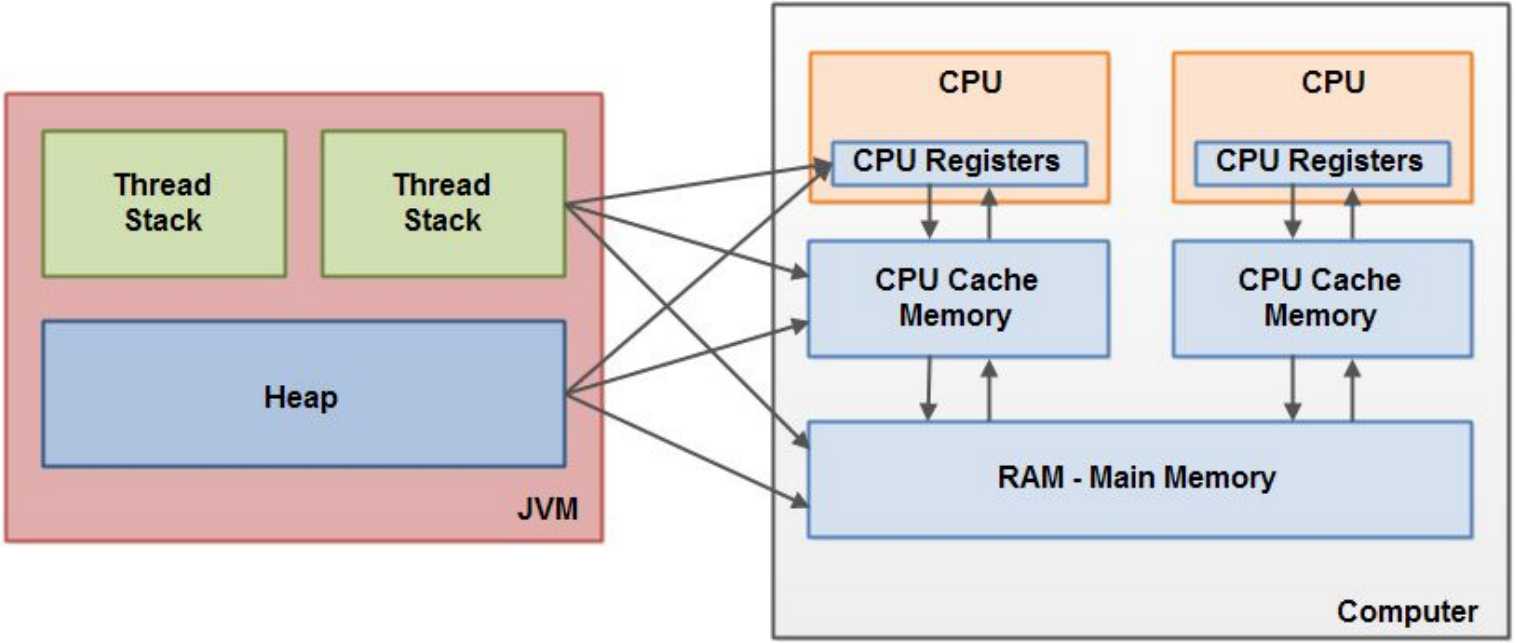



```
public class MySharedObject {

    //static variable pointing to instance of MySharedObject

    public static final MySharedObject sharedInstance =
        new MySharedObject();


    //member variables pointing to two objects on the heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member1 = 67890;
}
```

Hardware Memory Architecture

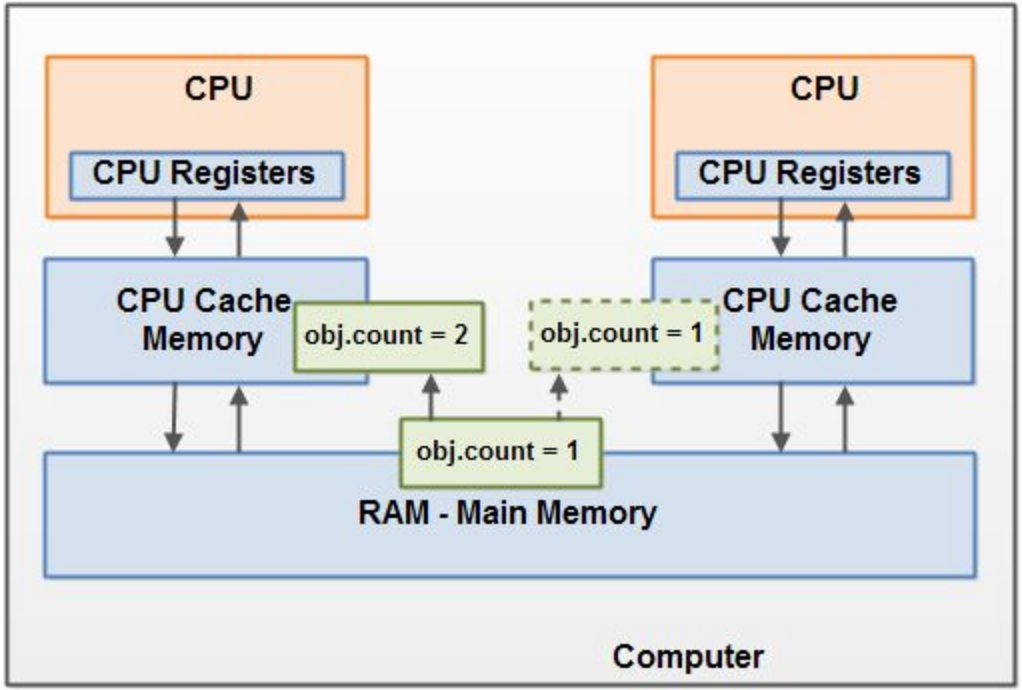


Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

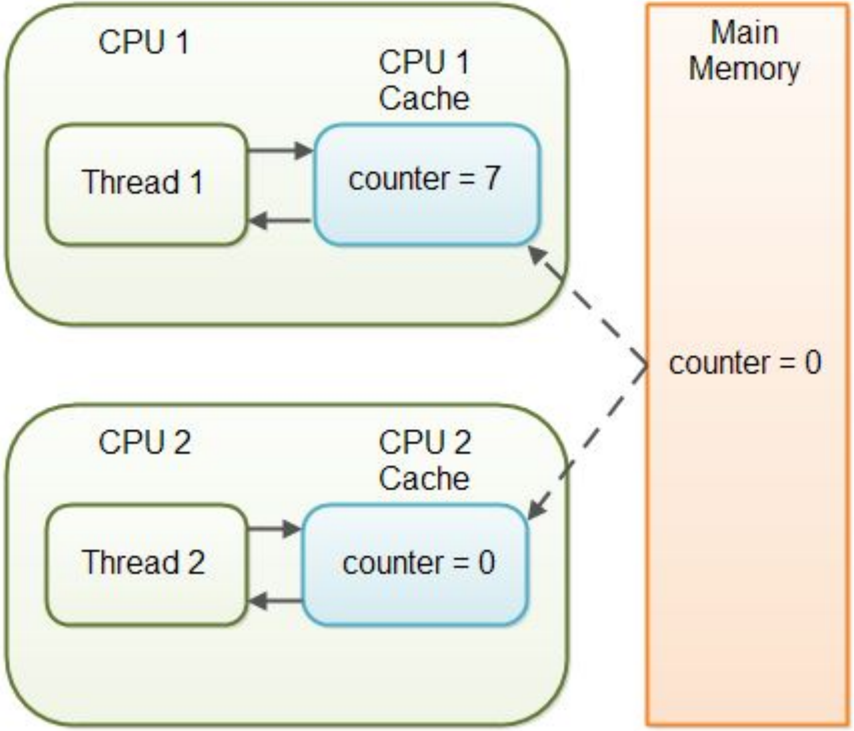


Visibility of Shared Objects

With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems



If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory



Java Volatile Keyword

The Java volatile keyword is intended to address variable visibility problems

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable
- If Thread A reads a volatile variable, then all variables visible to Thread A when reading the volatile variable will also be re-read from main memory

```java
public class MyClass {
    private int years;
    private int months
    private volatile int days;

    public int totalDays() {
        int total = this.days;
        total += months * 30;
        total += years * 365;
        return total;
    }

    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the thread are also written to main memory. That means, that when a value is written to days, the values of years and months are also written to main memory

Notice the totalDays() method starts by reading the value of days into the total variable. When reading the value of days, the values of months and years are also read into main memory. Therefore you are guaranteed to see the latest values of days, months and years with the above read sequence

When is volatile Enough?

if two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use a synchronized in that case to guarantee that the reading and writing of the variable is atomic

In case only **one thread reads and writes** the value of a volatile variable and **other threads only read** the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable

Instruction Reordering Challenges

instruction reordering present a challenge when one of the variables is a volatile variable

```
public void update(int years, int months, int days){
    this.years  = years;
    this.months = months;
    this.days   = days;
}
```

The values of months and years are still written to main memory when the days variable is modified, but this time it happens before the new values have been written to months and years. The new values are thus not properly made visible to other threads

```
public void update(int years, int months, int days){
    this.days   = days;
    this.months = months;
    this.years  = years;
}
```

ThreadLocal

The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables

```java
public class ThreadLocalExample {
    private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>();

    void doWork(){
        threadLocal.set( (int) (Math.random() * 100D) );
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
        }
    }

    Integer getValue(){
        return threadLocal.get();
    }
}
```

```java
void ThreadLocalExampleMethod(){
    ThreadLocalExample sharedInstance = new ThreadLocalExample();

    Thread t1 = new Thread(()-> sharedInstance.doWork());
    Thread t2 = new Thread(()-> sharedInstance.doWork());
    t1.start();
    t2.start();
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    try {
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Log.i(MainActivity.TAG, msg: "ThreadLocalExampleMethod]]>>"+
            sharedInstance.getValue());
}
```

This example creates a single ThreadLocalExample instance which is passed to two different threads. Both threads execute the run() method, and thus sets different values on the ThreadLocal instance. If the access to the set() call had been synchronized, and it had not been a ThreadLocal object, the second thread would have overridden the value set by the first thread.

```
Emulator Nexus_5X_API_23 A   edu.sharif.prj01 (3573)   Verbose   prj1-thread
02-05 09:58:47.921 3573-3573/edu.sharif.prj01 I/prj1-thread: ThreadLocalExampleMethod]]>>null
```

```java
public class ThreadLocalExample {
    private Integer threadLocal = new Integer( value: 0);

    void doWork(){
        threadLocal= (int) (Math.random() * 100D) ;
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
        }
    }

    Integer getValue(){
        return threadLocal;
    }
}
```

```java
void ThreadLocalExampleMethod(){
    ThreadLocalExample sharedInstance = new ThreadLocalExample();

    Thread t1 = new Thread(()-> sharedInstance.doWork());
    Thread t2 = new Thread(()-> sharedInstance.doWork());
    t1.start();
    t2.start();
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    try {
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Log.i(MainActivity.TAG, msg: "ThreadLocalExampleMethod]]>>"+
            sharedInstance.getValue());
}
```

```
02-05 09:56:18.179 3385-3385/edu.sharif.prj01 I/prj1-thread: ThreadLocalExampleMethod]]>>41
```

```
Emulator Nexus_5X_API_23 A    edu.sharif.prj01 (3489)    Verbose    prj1-thread
02-05 09:57:08.777 3489-3489/edu.sharif.prj01 I/prj1-thread: ThreadLocalExampleMethod]]>>26
```

Locks in Java

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this){
      return ++count;
    }
  }
}
```

```
Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();
```

```
lock.lock();
try{
  //do critical section code, which may throw exception
} finally {
  lock.unlock();
}
```

Lock Reentrance

Synchronized blocks in Java are reentrant. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object

Java's synchronized blocks are reentrant

```
public class Reentrant{

  public synchronized outer(){
    inner();
  }

  public synchronized inner(){
    //do something
  }
}
```

Android ReentrantLock

```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  // ...

  public void m() {
    lock.lock();  // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```

wait, notify and notifyAll

A thread that calls wait() on any object becomes inactive until another thread calls notify() on that object.

In order to call either wait() or notify the calling thread must first obtain the lock on that object. In other words, the calling thread must call wait() or notify() from inside a synchronized block

```
public class MonitorObject{
}

public class MyWaitNotify{

  MonitorObject myMonitorObject = new MonitorObject();

  public void doWait(){
    synchronized(myMonitorObject){
      try{
        myMonitorObject.wait();
      } catch(InterruptedException e){...}
    }
  }

  public void doNotify(){
    synchronized(myMonitorObject){
      myMonitorObject.notify();
    }
  }
}
```

```java
public class Message {
    private String msg;

    public Message(String str){
        this.msg=str;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String str) {
        this.msg=str;
    }
}
```

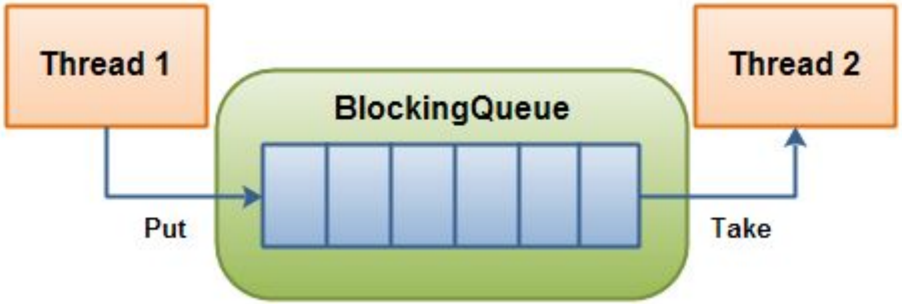
```java
public class Waiter implements Runnable {

    private Message msg;
    final CountDownLatch latch;

    public Waiter(Message m, CountDownLatch latch) {
        this.msg = m;
        this.latch = latch;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        synchronized (msg) {
            try {
                Log.i(MainActivity.TAG, msg: "WaitNotifyTest.Waiter]>> time:"
                        + System.currentTimeMillis());
                msg.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //process the message now
            Log.i(MainActivity.TAG, msg: "WaitNotifyTest.Waiter]>> name:"
                    + name + " processed:" + msg.getMsg());
            this.latch.countDown();
        }
    }
}
```

```java
public class Notifier implements Runnable {

    private Message msg;
    final CountDownLatch latch;

    public Notifier(Message msg, CountDownLatch latch) {
        this.msg = msg;
        this.latch = latch;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        Log.i(MainActivity.TAG, msg: "WaitNotifyTest.Notifier]>> started");
        try {
            Thread.sleep( millis: 1000);
            synchronized (msg) {
                msg.setMsg(name+" Notifier work done");
                msg.notify();
                // msg.notifyAll();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.latch.countDown();
    }
}
```

```java
void WaitNotifyTest(){
    // intialising count down latch by 2,
    // as it will wait for 2 threads to finish execution
    final CountDownLatch latch = new CountDownLatch(2);

    Message msg = new Message( str: "process it");

    Waiter waiter = new Waiter(msg, latch);
    new Thread(waiter, name: "waiter1").start();

    Waiter waiter1 = new Waiter(msg, latch);
    new Thread(waiter1,  name: "waiter2").start();

    Notifier notifier = new Notifier(msg, latch);
    new Thread(notifier,  name: "notifier1").start();

    try {
        latch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Log.i(MainActivity.TAG,  msg: "WaitNotifyTest]>> end");
}
```

Blocking Queues

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely



```
public class BlockingQueue {

  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }


  public synchronized void enqueue(Object item)
  throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }


  public synchronized Object dequeue()
  throws InterruptedException{
    while(this.queue.size() == 0){
      wait();
    }
    if(this.queue.size() == this.limit){
      notifyAll();
    }

    return this.queue.remove(0);
  }

}
```

BlockingQueue Implementations

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

BlockingQueue Example

```
BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);
Consumer consumer = new Consumer(queue);

new Thread(producer).start();
new Thread(consumer).start();

Thread.sleep(4000);
```

BlockingQueue Example (2)

```
public class Producer implements Runnable{

    protected BlockingQueue queue = null;

    public Producer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            queue.put("1");
            Thread.sleep(1000);
            queue.put("2");
            Thread.sleep(1000);
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Consumer implements Runnable{

    protected BlockingQueue queue = null;

    public Consumer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Thread Pools

```
public class ThreadPool {

    private BlockingQueue taskQueue = null;
    private List<PoolThread> threads = new ArrayList<PoolThread>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new BlockingQueue(maxNoOfTasks);

        for(int i=0; i<noOfThreads; i++){
            threads.add(new PoolThread(taskQueue));
        }
        for(PoolThread thread : threads){
            thread.start();
        }
    }

    public synchronized void  execute(Runnable task) throws Exception{
        if(this.isStopped) throw
            new IllegalStateException("ThreadPool is stopped");

        this.taskQueue.enqueue(task);
    }

    public synchronized void stop(){
        this.isStopped = true;
        for(PoolThread thread : threads){
           thread.doStop();
        }
    }

}
```

Thread Pools(2)

```
public class PoolThread extends Thread {

    private BlockingQueue taskQueue = null;
    private boolean       isStopped = false;

    public PoolThread(BlockingQueue queue){
        taskQueue = queue;
    }

    public void run(){
        while(!isStopped()){
            try{
                Runnable runnable = (Runnable) taskQueue.dequeue();
                runnable.run();
            } catch(Exception e){
                //log or otherwise report exception,
                //but keep pool thread alive.
            }
        }
    }

    public synchronized void doStop(){
        isStopped = true;
        this.interrupt(); //break pool thread out of dequeue() call.
    }

    public synchronized boolean isStopped(){
        return isStopped;
    }
}
```

ExecutorService

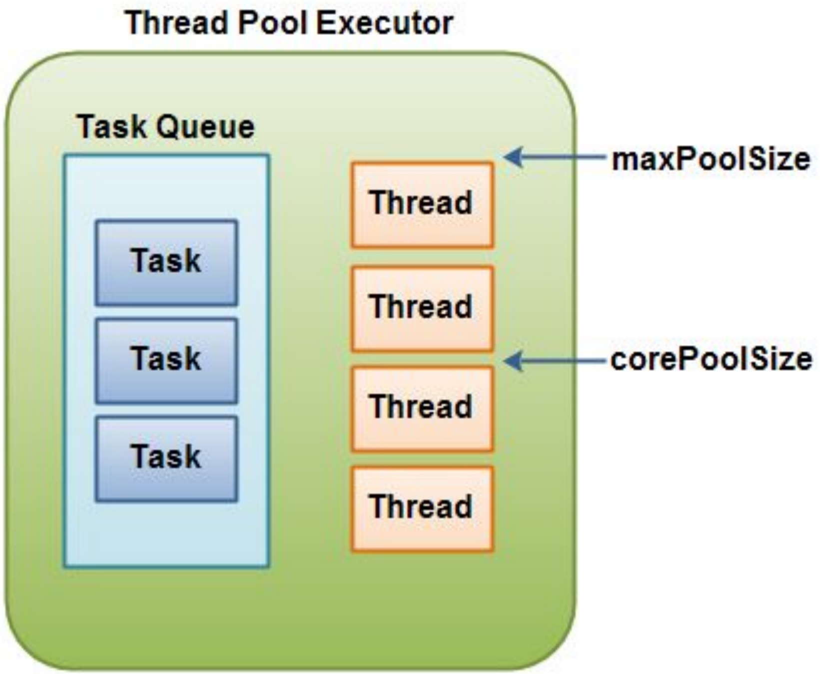
```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

ThreadPoolExecutor

is an implementation of the ExecutorService interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads

```
int  corePoolSize  =    5;
int  maxPoolSize   =   10;
long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =
        new ThreadPoolExecutor(
                corePoolSize,
                maxPoolSize,
                keepAliveTime,
                TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>()
                );
```



ScheduledExecutorService

First a ScheduledExecutorService is created with 5 threads in. Then an anonymous implementation of the Callable interface is created and passed to the schedule() method. The two last parameters specify that the Callable should be executed after 5 seconds

```java
void ScheduledExecutorServiceMethod(){
    ScheduledExecutorService scheduledExecutorService =
            Executors.newScheduledThreadPool( corePoolSize: 5);

    ScheduledFuture scheduledFuture =
            scheduledExecutorService.schedule(
                    new Callable() {
                        public Object call() throws Exception {
                            Log.i(MainActivity.TAG,  msg: "ScheduledExecutorService]>> Executed");
                            return "Called!";
                        }
                    },
                     l: 5,
                    TimeUnit.SECONDS);
}
```

ScheduledExecutorService.scheduleAtFixedRate(...)

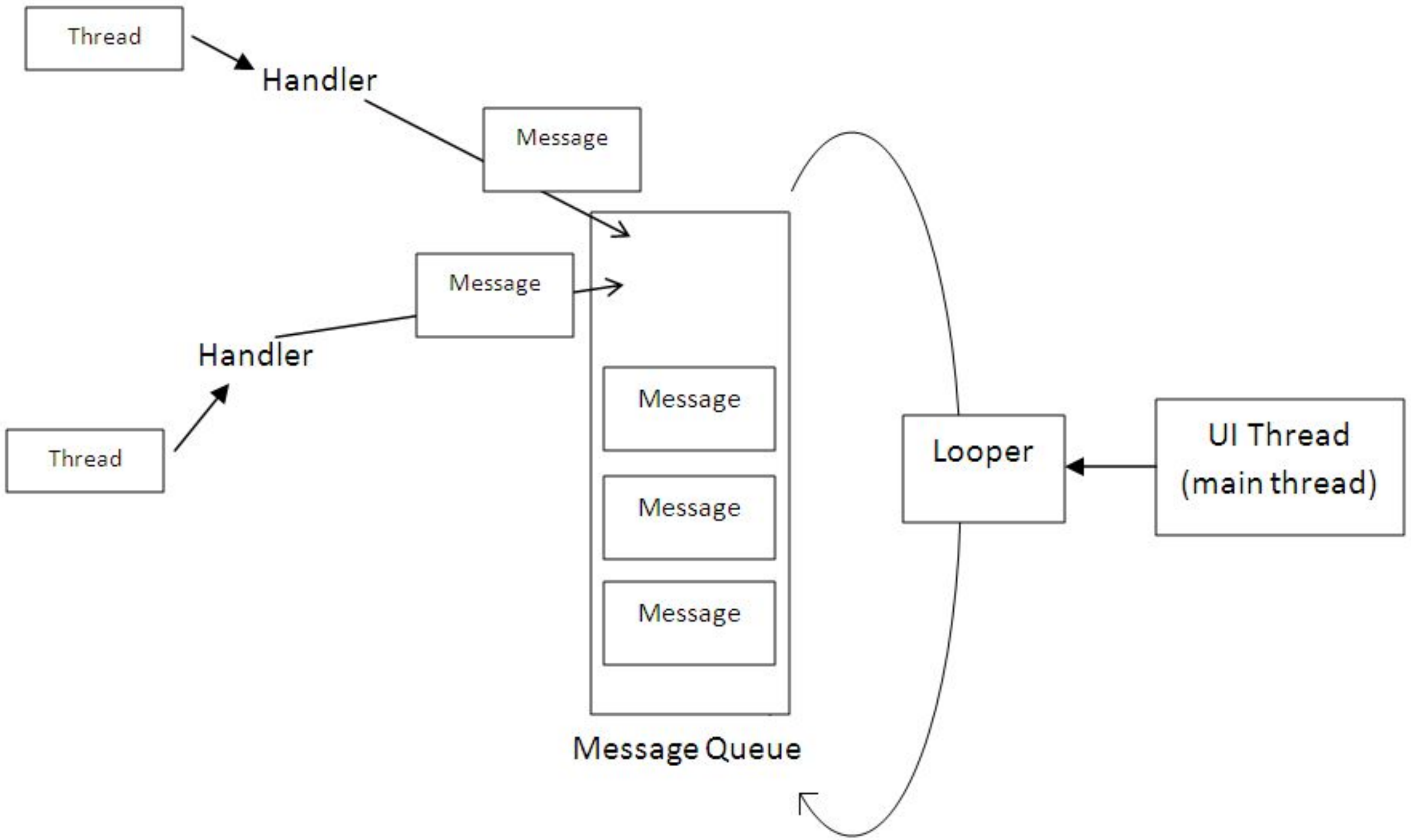
- **scheduleAtFixedRate** (Runnable, long initialDelay, long period, TimeUnit timeunit)
    - This method schedules a task to be executed periodically. The task is executed the first time after the initialDelay, and then recurringly every time the period expires

Android Handler

Android is a single threaded UI framework

A Handler allows communicating back with UI thread from other background thread

- It is the Android way of Thread communication
- *A good example of using a Handler is when you have a Runnable, you do something in the background thread, and then – you want to update UI at some point. In this case, you initialize a Handler as new Handler(Looper.getMainLooper), call handler.post() and do the UI job inside the post()*


Thread
Handler
Message
Message
Handler
Thread
Message
Message
Message
Message Queue
Looper
UI Thread
(main thread)

Executing Code After Delay (in UI thread)

```java
// We need to use this Handler package
import android.os.Handler;

// Create the Handler object (on the main thread by default)
Handler handler = new Handler();
// Define the code block to be executed
private Runnable runnableCode = new Runnable() {
    @Override
    public void run() {
      // Do something here on the main thread
      Log.d("Handlers", "Called on main thread");
    }
};
// Run the above code block on the main thread after 2 seconds
handler.postDelayed(runnableCode, 2000);
```

Execute Recurring Code with Specified Interval

```java
// We need to use this Handler package
import android.os.Handler;

// Create the Handler object (on the main thread by default)
Handler handler = new Handler();
// Define the code block to be executed
private Runnable runnableCode = new Runnable() {
    @Override
    public void run() {
      // Do something here on the main thread
      Log.d("Handlers", "Called on main thread");
      // Repeat this the same runnable code block again another 2 seconds
      // 'this' is referencing the Runnable object
      handler.postDelayed(this, 2000);
    }
};
// Start the initial runnable task by posting through the handler
handler.post(runnableCode);
```

```java
// Removes pending code execution
handler.removeCallbacks(runnableCode);
```

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    progressBar = (ProgressBar) findViewById(R.id.progressBar);
    startProgress = (Button) findViewById(R.id.start_progress);
    textView = (TextView) findViewById(R.id.textView);
    progressBar.setMax(MAX);

    thread1 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int  i = 0;i<100;i++){
                Log.d("I",":"+i);
                progressBar.setProgress(i);
                try{
                    Thread.sleep(1000);
                }
                catch (InterruptedException ex){
                    ex.printStackTrace();
                }
                Message message = new Message();
                message.what = UPDATE_COUNT;
                message.arg1 = i;
                mHandlerThread.sendMessage(message);
            }
        }
    });
```

```java
@Override
protected void onResume() {
    super.onResume();
    mHandlerThread = new Handler(){
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            if (msg.what == START_PROGRESS){
                thread1.start();
            }
            else if(msg.what == UPDATE_COUNT){
                textView.setText("Count"+msg.arg1);
            }
        }
    };
}
```

```java
public final class Message /*implements Parcelable*/ {
    /**...*/
    public int what;
    /**...*/
    public int arg1;
    /**...*/
    public int arg2;
    /**...*/
    public Object obj;
    /**
     * Optional Messenger where replies to this message can be sent.  The
     * semantics of exactly how this is used are up to the sender and
     * receiver.
     */
//    public Messenger replyTo;
    /**...*/
    public int sendingUid = -1;
    /**...*/
    /*package*/ static final int FLAG_IN_USE = 1 << 0;
    /**...*/
    /*package*/ static final int FLAG_ASYNCHRONOUS = 1 << 1;
    /**...*/
    /*package*/ static final int FLAGS_TO_CLEAR_ON_COPY_FROM = FLAG_IN_USE;
    /*package*/ int flags;
    /*package*/ long when;
    /*package*/ Bundle data;
    /*package*/ Handler target;
    /*package*/ Runnable callback;
    // sometimes we store linked lists of these things
    /*package*/ Message next;
    /**...*/
```

```java
public final class Looper {

    private static final String TAG = "Looper";
    // sThreadLocal.get() will return null unless you've called prepare().
    static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
    final MessageQueue mQueue;
    final Thread mThread;
    public static void loop() {
        final Looper me = myLooper();
        if (me == null) {
            throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
        }
        final MessageQueue queue = me.mQueue;

        for (;;) {
            Message msg = queue.next(); // might block
            if (msg == null) {
                // No message indicates that the message queue is quitting.
                return;
            }
            //...
            try {
                msg.target.dispatchMessage(msg);
            } finally {

            }
            // ...
            msg.recycleUnchecked();
        }
    }
```

```java
public class Handler {
    /*
     * Set this flag to true to detect anonymous, local or member classes
     * that extend this Handler class and that are not static. These kind
     * of classes can potentially create leaks.
     */
    private static final boolean FIND_POTENTIAL_LEAKS = false;
    private static final String TAG = "Handler";
    private static Handler MAIN_THREAD_HANDLER = null;
    /**...*/
    public interface Callback {
        /**...*/
        public boolean handleMessage(Message msg);
    }

    public final boolean sendMessage(Message msg)
    {
        return sendMessageDelayed(msg,  delayMillis: 0);
    }
    public final boolean post(Runnable r)
    {
        return  sendMessageDelayed(getPostMessage(r),  delayMillis: 0);
    }
    private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
    }
    public final boolean sendMessageDelayed(Message msg, long delayMillis)
    {
        if (delayMillis < 0) {
            delayMillis = 0;
        }
        return sendMessageAtTime(msg,  uptimeMillis: SystemClock.uptimeMillis() + delayMillis);
    }
    public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
        MessageQueue queue = mQueue;
        if (queue == null) {
            RuntimeException e = new RuntimeException(
                    this + " sendMessageAtTime() called with no mQueue");
            return false;
        }
        return enqueueMessage(queue, msg, uptimeMillis);
    }
```