SOFT3202

Adam Ghanem

April 14, 2022

Contents

1 Tutorial 7 EXPORT

1.1 Concurrency

1.1.1 Introduction

The idea of concurrent program is that we can execute units in an order that allows parallel execution. This is not meant to be a pure primer into concurrency but this tutorial establishes the foundations necessary to work with some design patterns and expand upon the major project. Just like caching, concurrency is an insanely difficult topic when expanded upon and high performance computing is a very active research area which makes heavy use of parallelism. Some of you may have realised but concurrency is apparent in the majority of the programs you are running. Even the Operating System

you are running is managing concurrent operations, **processes** running in the foreground and background. Moreover, **threads** are the smallest unit of instructions that can be managed by our scheduler. In the context of threads we have, **user threads** and **kernel threads**. The difference between these two is in the area of their implementation (whether kernel or user) and their visibility (kernel threads cannot see userspace threads). The difference between threads and processes is a bit larger and involves faster context switching, memory and address spaces including their independence and also how we can connect and communicate between them (pipes etc).

However in the case of what we will be learning this is excessive in understanding the concurrency we will be doing in Java. In most scenarios we can accelerate programs heavily through the use of concurrency but we can also make use of it to handle tasks in the background for GUI programs such that it is still repsonsive e.g. scaling, minimising and clicking even though a task is running. But the main thing I want to emphasise is the how interesting and powerful concurrency can be, for instance one of the largest machine learning models available to us right now has over a **trillion parameters** [1] and this would not be possible without being able to execute across GPU devices. In this example particularly, the model was run over thousands of GPUs and succinct partitioning and parallelism schemes had to be used in order to get this to synchronise correctly.

1.1.2 Threads in Java

The resources we will need to understand concurrency in Java:

- `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html`
- `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html`
- `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Callable.html`

In Java we have a typical concept of a thread and although I prefer to learn concurrency the old-fashioned way with `C`, Java provides a very smooth transition into the world of concurrency. Java's thread can be created by calling the `Thread` class, this is a functional interface which contains a `run()` method

which we need to override. The `run()` method can be thought of as the main method of our threads.

Let us take a look at an example:

```
public class TestThread {

  public static void main(String[] args) {
    // Make sure to use lambdas here
    Thread thread = new Thread(() -> {
        for (int i = 0; i < 3; i++) {
          System.out.println("Hello World!");
        }
    });

    // Make sure to start.
    thread.start();
  }
}
```

If we do not want to use lambdas all the time and reuse some form of this then we can revert to `Runnable`. The `Runnable` interface is porvided by java and has the same `run()` method which must be overrided. Just an aside, if we wanted to return some value from our workers, make use of the `Callable<?>` class, but due to the fact they may take some time to process and return their data, then `Future<?>` is necessary to collect the value.

```
public class Worker implements Runnable {
  public void run() {
    System.out.println("Running from a worker!");
    }
}

public class ThreadExample {

  public static void main(String[] args) {
    Thread worker = new Thread(new Worker());
    worker.start();

  }
```

```
}
```

1.1.3 Threading Example

Before confusing those who have never touched concurrency and apologies if I already have. Yet let us take a look at the key mechanisms of concurrency first. Usually we spawn threads that are working on some problem, it could be the same problem on different subsets of data or they could work on different tasks altogether. In the case of a GUI one thread could be fetching data from a request to loads available loans while the other is updating account information.

A simple example dealing with these would be:

```
public class ThreadingExample {
  static int value = 1;
  // In the class/method in which we are performing/launching threads
  // we need to able to catch and deal with these exceptions.
    public static void main(String[] args) throws InterruptedException {

      // See how we make use of lambdas
        Thread t1 = new Thread(() -> {
            while (value < 10) {
                System.out.println("T1: " + value);
                value += 1;
            }
        });

        Thread t2 = new Thread(() -> {
            while (value < 15) {
                System.out.println("T2: " + value);
                value += 1;
            }
        });

        // What does this print out?
        System.out.println(value);

        t1.start(); t2.start();
```

```
        // How about this
        System.out.println(value);

        // Hmm let us try sleeping
        Thread.sleep(1);
        System.out.println(value);
        }
}
```

From an example provided by Josh say we have a bank account managing transactions that occur simultaneously:

```
public class Bank {
  private int account = 100;

  public static void main(String[] args) {
    Bank account = new Bank();
    Thread adding = new Thread(new AddMoney(account));
    Thread subtracting = new Thread(new TakeMoney(account));
    adding.start();
    subtracting.start();
  }

  public void addMoney(int amount) {
    if (amount < 1) {
      System.out.print("Trying to add a negative amount to the account!");
      System.out.println(" Transaction rejected.");
      return;
    }
    int newBalance = account + amount;
    System.out.print("Account balance is " +
                     account +
                     ". Adding " +
                     amount + ".");
    account = newBalance;
    System.out.println("New balance is " + account + ".");
    return;
  }
```

```
public void subtractMoney(int amount) {
  if (amount < 1) {
    System.out.print("Trying to subtract a negative amount from the account!");
    System.out.println(" That's generous, but the transaction is rejected.");
    return;
  }
  int newBalance = account - amount;
  System.out.print("Account balance is " +
                   account +
                   ". Subtracting " +
                   amount + ".");
  account = newBalance;
  System.out.println("New balance is " + account + ".");
  return;
}

class AddMoney implements Runnable {
  private Bank account;

  public AddMoney(Bank account) {
    this.account = account;
  }

  /**
   * Add 1000 to the given account, 60 times
   */
  public void run() {
    for (int i = 0; i < 60; ++i) {
      try {
        Thread.sleep(500);
      } catch (Exception e) {
        System.err.println("Already interrupted.");
      }
      account.addMoney(1000);
    }
  }
}
```

```
  class TakeMoney implements Runnable {
    private Bank account;

    public TakeMoney(Bank account) {
      this.account = account;
    }

    /**
     * Take 1000 from the given account, 60 times
     */
    public void run() {
      for (int i = 0; i < 60; ++i) {
        try {
          Thread.sleep(500);
        } catch (Exception e) {
          System.err.println("Already interrupted.");
        }
        account.subtractMoney(1000);
      }
    }
  }
}

public Lock lock = new ReentrantLock();
// The AddMoney and TakeMoney classes can now 'lock' the Bank
// so only they can use it.
account.lock.lock();
account.addMoney(1000);
account.lock.unlock();
```

Now if we were to run it with these modifications, will our output change?

We have allowed protection from concurrent modification but we have taken what is known as a **coarse-grained locking** approach. This means that we have restricted access to the whole data structure so that threads now have to execute sequentially. It starts now to get difficult to advance past the naive approach.

1.1.4 Synchronisation

The area where it becomes quite tricky to manage is in the form of synchronisation. The idea of concurrency is great and can offer exorbitant performance improvements but at what cost if our data is not synchronised correctly, or if threads conflict with each other. For example, say we have two threads working in the background on a single `String a = "oh"` , one thread concatenating `" hi"` and the other `" mark!"`. In this scenario there is an obvious order we have to maintain, `"oh hi mark!"`. Yet, if we were just to run these threads and Java and get it to return back we end up in a bit of a pickle, one thread could concatenate before the other, we have enforced no particular order! We need to understand how to share resources between our threads and balance modifications to them.

Take a look at another of Josh's example:

```
public class HelloConcurrency {
    private static boolean keep_going = true;
    private static int value = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            while (keep_going) {
                value = 1;
                if (value == 1) {
                    check(1, value);
                }
            }
        });

        Thread t2 = new Thread(() -> {
            while (keep_going) {
                value = 2;
                if (value == 2) {
                    check(2, value);
                }
            }
        });
```

```
        Thread t3 = new Thread(() -> {
            while (keep_going) {
                value = 3;
                if (value == 3) {
                    check(3, value);
                }
            }
        });

        t1.start();
        t2.start();
        t3.start();

        Thread.sleep(30);
        keep_going = false;
    }

    public static synchronized void check(int expected, int actual) {
        if (expected != actual) {
            System.out.println("Mistake! " + expected +
                               " does not equal " +
                               actual);
        }
    }
}
```

In Java we can acheive this synchronisation through multiple different methods. Through `Locks` or through `synchronized` blocks. A `Lock` is a synchronisation mechanism that states that, "if you want to enter this region or use this object you need to first have hold of the lock. If the lock is currently held then you must wait and grab it yourself."

We can make use of `ReentrantLock()` in Java:

```
public Lock lock = new ReentrantLock();
// ...
lock.lock();
// if someone wanted to read from your account they can't until you release this l
```

```
account.addMoney(1000);
lock.unlock()
```

Another mechanism is known as the `synchronized` block. This allows a single thread to enter a defined region, the other threads will have to wait until the executing thread leaves this block.

```
// Create a static final object that we will lock
private static final Object lock = new Object();

public void addMoney(double money) {
    // One thread passing through here
    synchronized(lock) {
        this.money -= money;
    }
}
```

1.1.5 Deadlocks

Deadlocks can be thought simply of as a scenario in which one participant (be it a thread, process etc) is waiting for a resource that it can never receive. For instance say we have processes, $P1$ and $P2$ and locks, $L1$ and $L2$. If $P1$ currently is holding $L1$ and needs $L2$ and $P2$ currently has $L2$ but needs $L1$ to release it's lock, then we have it a deadlock. This will stall forever and results in significant wasted resources and in some cases denial of services. A certain environment has to be available for deadlocks to occur and involve all these conditions to be true:

- resources need to be contested in such a way that prevents others from accessing i.e. there needs to be **Mutual Exclusion**.
- resources can only be released by the process holding it - **no preemption**
- a process must want to require another resource which is held by another process - **hold and wait**
- **circular wait** - which is the example I discussed above, one process must be waiting for another resource while it is holding onto one and the other threads must want its resource.

We can see an exampel of this in the below code, specifically with this section:

```
    public static void actionOne() {
        synchronized (lockOne) {
            synchronized (lockTwo) {
                System.out.println("Got inside Action one!");
            }
        }
    }

// Access order is different to the other thread
    public static void actionTwo() {
        synchronized (lockTwo) {
            synchronized (lockOne) {
                System.out.println("Got inside Action two!");
            }
        }
    }
```

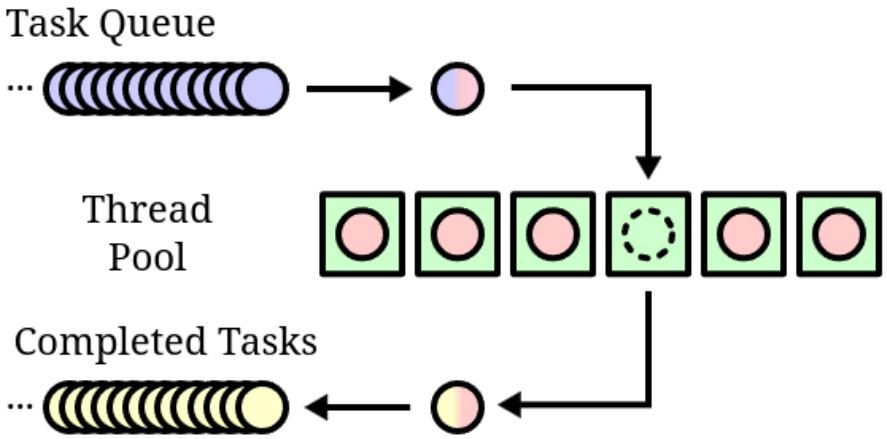
Let us take a look at Josh's full example for Deadlocks and see if you can find a solution to avoiding it:

```
public class HelloDeadlocks {
    private static final Object lockOne = new Object();
    private static final Object lockTwo = new Object();

    private static boolean keep_going = true;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            while (keep_going) {
                actionOne();
            }
        });

        Thread t2 = new Thread(() -> {
            while (keep_going) {
                actionTwo();
            }
```

```
        });

        t1.start();
        t2.start();

        Thread.sleep(30);
        keep_going = false;
    }

    public static void actionOne() {
        synchronized (lockOne) {
            synchronized (lockTwo) {
                System.out.println("Got inside Action one!");
            }
        }
    }

    public static void actionTwo() {
        synchronized (lockTwo) {
            synchronized (lockOne) {
                System.out.println("Got inside Action two!");
            }
        }
    }
}
```

1.1.6 Thread Pools and Executors

Thread pools are a essentially a design pattern for managing multiple threads and tasks. A thread pool consists of a queue of tasks which are completed by worker threads. The key distinction here between simply launching threads, is that these workers are reusable which allows us to significantly improve performance since we do not have to manage constantly **creating and deleting threads, which can have significant overhead**. Moreover, within these tutorial we make use of **fixed thread pools** which have a specified number of threads. The reason is that if a worker is created for every new task then we can effectively drain the resources of a system. To highlight thread pools, Wikipedia is our friend here:



Now `Executor` is our fancy Java mechanisms for creating thread pools through factory methods that allow us to efficiently manage asynchronous applications. `ExecutorService` and `Executors` are the classes specifically responsible for helping to create and work with thread pools, allowing itself to reuse previous workers. As the Java docs imply, instead of having to run individual `new Thread(new Worker()).start()` we can use our `Executor` to manage our threads. A general process we follow when doing this is to create some tasks that are `Runnable`, then make an `ExecutorService` that spawns some number of threads. We than execute the tasks to this pool and shutdown when we are finished.

```
Executor executor = anExecutor();
 executor.execute(new Worker());
 executor.execute(new Worker());
 ...
```

Our `Executor` manages launching tasks, our `ExecutorService` manages the lifecycle of these tasks and the scheduled version of this which supports future or repetitive tasks, `ScheduledExecutorService`.

```
import java.util.concurrent.*;

public class HelloExecutors {
    private static final int NUM_THREADS = 4;
    private static final ExecutorService pool = Executors.newFixedThreadPool(2);
    private static final Integer[] value = {0};
```

```
public static void main(String[] args) throws
  InterruptedException, ExecutionException {
    // Above we create an ExecutorService thread pool.
    // A thread pool is a design pattern for concurrency
    // that allows the overhead of creating threads to be managed.
    // There are a lot of different kinds
    // depending on the performance needs of the application.

    // We will use that thread pool to execute Tasks.
    //You can mostly consider Tasks to be the things you
    // might otherwise give to a Thread via dependency injection
    // for it to then run (the lambda functions
    // in the previous exercises). Notice how the
    // performance changes as you change nThreads.
    //
    // Remember that while the ExecutorService APIs
    // will help you create and manage your threads, it does
    // nothing to help you manage your concurrent execution
    // - synchronization and avoiding race conditions,
    // deadlocks etc. is entirely your problem!

    Runnable myRunnable = () -> {
        try {
          /*
          A runnable is something that can be run,
          but for which direct returned output is not
          This doesn't mean it can't have any effects
          - but it will operate on existing data via references.
          Be careful with this - a lot of race conditions are
          caused because of unexpected side effects due
          to threads modifying data by reference instead of a callable
          working on its own copy and returning a result.
          */
            value[0] = value[0] + 1;
            Thread.sleep(300);
            System.out.println("Value was " + value[0]);
        } catch (InterruptedException e) {
```

```
        e.printStackTrace();
    }
};

Callable<String> myCallableString = () -> {
    Thread.sleep(300);
    return "This is a string";
};

Callable<Integer> myCallableInteger = () -> {
    Thread.sleep(300);
    return value[0];
};

// Notice the different ways of running tasks.
// In particular the difference between Callables with a return
// type, Runnables without, but the option to provide a default
//- see the ExecutorPool API for the details.
pool.execute(myRunnable);
Future<String> resultFromRunnable =
  pool.submit(myRunnable, "Finished Ok");
pool.execute(myRunnable);
Future<String> resultStringOneTask =
  pool.submit(myCallableString);
Future<String> resultStringTwoTask =
  pool.submit(myCallableString);
Future<Integer> resultIntegerOneTask =
  pool.submit(myCallableInteger);
Future<Integer> resultIntegerTwoTask =
  pool.submit(myCallableInteger);

// The sleep here is to reasonably ensure that
// we are in the middle of resultFromRunnable when it
// is cancelled - demonstrating the InterruptedException.
// If it hasn't started yet the cancellation will work,
// but no InterruptedException will be raised.
// If it is finished the cancellation won't work.
Thread.sleep(100);
```

```
boolean cancelled = resultFromRunnable.cancel(true);
System.out.println("Cancellation returned " + cancelled);

// This will probably be false - aside from the
// 100ms sleep this main thread is likely to be way ahead of
// the task threads, which need to wait for the ExecutorPool
// to be ready, assign them to a thread, THEN be
// executed, and even then wait 300ms internally
System.out.println("Result String One isDone returns: "
                   + resultIntegerOneTask.isDone());

// But when we execute this call we then sit and wait until the task is do
String resultStringOne = resultStringOneTask.get();
System.out.println("ResultStringOne is "
                   + resultStringOne);

// So now this will be true, right? Nope - there's cleanup
// that occurs after the return value
// is generated. If isDone is true, the result value will
// always have been created and returned
// (assuming no exception or cancellation), but the reverse is not true.
System.out.println("Result String One isDone returns: "
                   + resultIntegerOneTask.isDone());

System.out.println("ResultStringTwo is "
                   + resultStringTwoTask.get());
System.out.println("ResultIntegerOne is "
                   + resultIntegerOneTask.get());
System.out.println("ResultIntegerTwo is "
                   + resultIntegerTwoTask.get());

// Now however we can be reasonably sure it is done
// (this will change depending on the number of threads
// available - if all the threads can run at once you might still get fals
System.out.println("Result String One isDone returns: "
                   + resultIntegerOneTask.isDone());

// Notice that unless you uncomment
```

```
        // the following the execution never stops:
        // See the ExecutorService API for an example of how to
        // do this shutdown more cleanly where
        // you might have tasks with infinite loops or long-running tasks.
        pool.shutdown();
    }
}
```

I suggest taking a look at the documentation to determine how to properly shutdown, since there could be tasks running in the background (possibly zombie processes) which drain resources. An ExecutorService can be used to solve this (from Java Docs):

```
void shutdownAndAwaitTermination(ExecutorService pool) {
   pool.shutdown(); // Disable new tasks from being submitted
   try {
     // Wait a while for existing tasks to terminate
     if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
       pool.shutdownNow(); // Cancel currently executing tasks
       // Wait a while for tasks to respond to being cancelled
       if (!pool.awaitTermination(60, TimeUnit.SECONDS))
           System.err.println("Pool did not terminate");
     }
   } catch (InterruptedException ex) {
     // (Re-)Cancel if current thread also interrupted
     pool.shutdownNow();
     // Preserve interrupt status
     Thread.currentThread().interrupt();
   }
 }
```

They first stop receiving any incoming tasks and then shutdown the remaining tasks.

1.1.7 JavaFX

With JavaFX we will be dealing with `Service`, `Worker`, and `Task`. A `Task` you can think of as an implemented `FutureTask`, in comparison to a `Callable` or `Runnable` it is merely wrapping it in a `Future` and allowing you to access,

cancel the task and more.

- https://openjfx.io/javadoc/17/javafx.graphics/javafx/concurrent/Worker.html
- https://openjfx.io/javadoc/17/javafx.graphics/javafx/concurrent/Task.html
- https://openjfx.io/javadoc/17/javafx.graphics/javafx/concurrent/Service.html

Now why is concurrency useful in GUI programming? Well if I have a request to button that is fetching thousands of items to shop from, do I really want the user to be looking at a frozen screen? No, you want to have these long drawn out processes (unless necessary otherwise) to be executed in the background. Take a look at the example Josh has provided and try to break down what is going down here:

```
import javafx.application.Application;
import javafx.concurrent.Task;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

public class HelloJavaFXWorkers extends Application {
    private Button taskBtn;
    private Button cancelBtn;
    private Button stateBtn;
    private Label statusLbl;

    private Task<Integer> task;
```

```
private final ExecutorService pool =
  Executors.newFixedThreadPool(2, runnable -> {
    Thread thread = new Thread(runnable);
    thread.setDaemon(true);
    return thread ;
});


@Override
public void start(Stage primaryStage) {
    taskBtn = new Button("Long operation");
    cancelBtn = new Button("Cancel operation");
    stateBtn = new Button("Get state");
    statusLbl = new Label("Current value: Not started yet");
    // Try having another button here that ping the typicode URI that
    // we had before. Let a user basically get a post and view it.

    cancelBtn.setDisable(true);

    taskBtn.setOnAction((event) -> {runTask();});
    cancelBtn.setOnAction((event) -> {cancelTask();});
    stateBtn.setOnAction((event) -> {checkState();});

    HBox hBox = new HBox(taskBtn, cancelBtn, stateBtn);
    VBox vBox = new VBox(hBox, statusLbl);

    Scene scene = new Scene(vBox);
    primaryStage.setScene(scene);
    primaryStage.setTitle("JavaFX App");
    primaryStage.show();
}

private void runTask() {
    taskBtn.setDisable(true);
    cancelBtn.setDisable(false);

    task = new Task<>() {
```

```
        final Random rand = new Random();

        @Override
        protected Integer call() {
            updateMessage("Task One just started");
            setStatusLabel("Current value: just started");
            int value = 0;
            for (int i = 0; i < 100; i++) {
                if(isCancelled()) {
                    setStatusLabel("Current value: cancelled");
                    updateMessage("Task was cancelled");
                    break;
                }
                setStatusLabel("Current value: " + value);
                updateMessage("Value is currently: " + value);
                updateProgress((i + 1), 100);
                try {
                    Thread.sleep(250);
                } catch (InterruptedException ignored) {
                    if (isCancelled()) {
                        setStatusLabel("Current value: cancelled");
                        updateMessage("Task was cancelled");
                        break;
                    }
                }

                value = rand.nextInt(1000);
            }

            return value;
        }
    };

    pool.execute(task);
}

private void setStatusLabel(String status) {
    // statusLbl.setText(status);
```

```
    }

    private void cancelTask() {
        task.cancel();
        taskBtn.setDisable(false);
        cancelBtn.setDisable(true);
    }

    private void checkState() {
        if (null != task) {
            String taskMessage = task.getMessage();
            double taskProgress = task.getProgress();
            String status = String.format("Task message: %s\n" +
                                          "Task progress: %.0f%%",
                                          taskMessage, taskProgress * 100);

            Alert a = new Alert(Alert.AlertType.NONE, status, ButtonType.CLOSE);
            a.show();
        }
        else {
            Alert a = new Alert(Alert.AlertType.NONE,
                                "You didn't start the task yet!", ButtonType.CLOSE
            a.show();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Commented in the above code, `setStatusLabel` would constantly print off error messages since it is dealing with all the threads not just the JavaFX Application thread. This is thrown because the JavaFX thread is the only one that can manage changes to the scene graph. You have many options from here, you can:

- check whether the current thread is the JavaFX application thread

through `isFxApplicationThread`

- and you can perform `platform.runLater(Runnnabe runnable)` which specifically executes the `Runnable` on the javafx application thread at some time in the future.