Concurrency

© 2008 Haim Michael (Scala, Concurrency)

Introduction

- The Scala programming language support for threads is based on the Java programming language.
- Using the Actors System provided by the Akka framework the development of complex concurrent applications is simplified.

© 2008 Haim Michael (Scala, Concurrency)

The `Runnable` Trait

- The `Runnable` interface we know in Java is available as the `Runnable` trait. It includes one abstract method. The `run()` method, that returns no value.
- We should instantiate a class that extends `Runnable` and overrides its abstract method, and pass over its reference to the `Thread` constructor.

© 2008 Haim Michael (Scala, Concurrency)

The `Runnable` Trait

- Once the `Thread` object is created we should invoke the `start()` method on it. As a result, the `run()` method will start running in a separated thread.
- Once the `start()` method was invoked we cannot invoke it again.

© 2008 Haim Michael (Scala, Concurrency)

The `Runnable` Trait

```
object Main {

  def main(args:Array[String]):Unit = {

    val thread : Thread = new Thread(
      new Runnable {
        def run(): Unit = {
          for(num <- 1 to 10) {
            println(num)
            Thread.sleep(500)
          }
        }
      }
    )

    thread.start

  }
}
```

© 2008 Haim Michael (Scala, Concurrency)

The `Executors` Object

- The `Executors` object includes the definition for various methods that can get us an `ExecutorService` object.
- One of the methods we can invoke on the `Executors` object is the `newFixedThreadPool` method that returns an `ExecutorService` object.

© 2008 Haim Michael (Scala, Concurrency)

The `Executors` Object

- The `ExecutorService` object returned by the `newFixedThreadPool` method represents a pool of threads.
- We can invoke the `execute` method on the `ExecutorService` object we got and pass over a `Runnable` object we want its `run` method to be executed on a separated thread.

© 2008 Haim Michael (Scala, Concurrency)

The `Executors` Object

```
object Main {

  def main(args:Array[String]):Unit = {

    var runnerA = new Runnable {
      override def run(): Unit = {
        for(num <- 1 to 5) {
          println("A")
          Thread.sleep(100)
        }
      }
    }

    var runnerB = new Runnable {
      override def run(): Unit = {
        for(num <- 1 to 5) {
          println("B")
          Thread.sleep(100)
        }
      }
    }
```

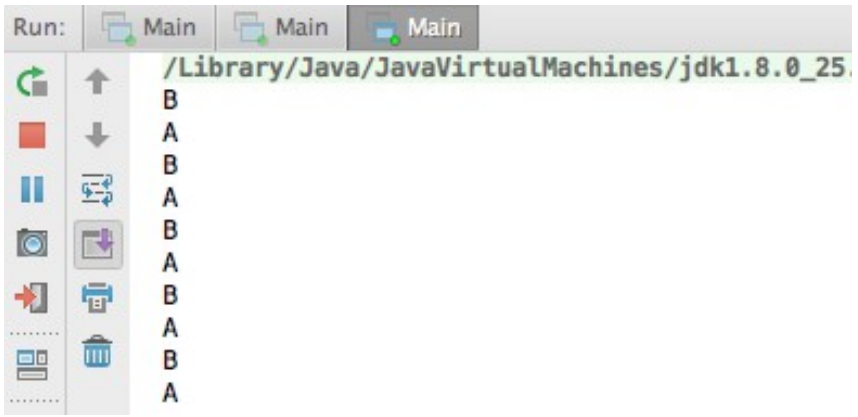
© 2008 Haim Michael (Scala, Concurrency)

The `Executors` Object

```
    val pool = Executors.newFixedThreadPool(3)

    pool.execute(runnerA)
    pool.execute(runnerB)

  }
}
```



© 2008 Haim Michael (Scala, Concurrency)

The `Future` Trait

- The `Future` trait represents an asynchronous computation. When we invoke the `get()` function on a `Future` object we will get the result of the asynchronous computation it represents. If the computation still hasn't ended we will be blocked.

© 2008 Haim Michael (Scala, Concurrency)

The `Future` Trait

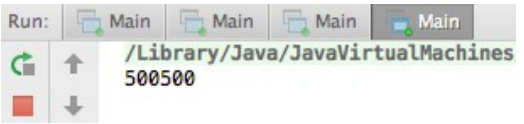
```
object Main {

  def main(args:Array[String]):Unit = {

    val future = new FutureTask[Int](
      new Callable[Int] {
        override def call(): Int = {
          var sum:Int = 0
          for(num <- 1 to 1000) {
            sum += num
            Thread.sleep(10)
          }
          sum
        }
      }
    )

    val pool = Executors.newFixedThreadPool(4)
    pool.execute(future)
    println(future.get)

  }
}
```



© 2008 Haim Michael (Scala, Concurrency)

Synchronization

- Scala allows us to create a synchronized block by using the `synchronized` keyword followed with the block we want to synchronize.

© 2008 Haim Michael (Scala, Concurrency)

Synchronization

```
package com.lifemichael.samples

import java.util.concurrent._


object Main {

  def main(args:Array[String]):Unit = {

      val stack:MyStack = new MyStack
      stack.push(12)
      stack.push(32)
      stack.push(42)
      print("stack.pop()... "+stack.pop)

  }
}
```

© 2008 Haim Michael (Scala, Concurrency)

Synchronization

```
object MyStack {

  private val numbers:Array[Int] = Array[Int](10)
  private var index:Int = 0

  def pop:Int = {
    this.synchronized {
      index -= 1
      this.numbers(index)
    }
  }

  def push(num:Int) = {
    this.synchronized {
      numbers(index) = num
      index += 1
    }
  }

}
```

© 2008 Haim Michael (Scala, Concurrency)