Concurrency in Scala

Runnable/Callable

Runnable has a single method that returns no value.

```
trait Runnable {
  def run(): Unit
}
```

Callable is similar to run except that it returns a value

```
trait Callable[V] {
  def call(): V
}
```

Threads

Scala concurrency is built on top of the Java concurrency model.

On Sun JVMs, with a IO-heavy workload, we can run tens of thousands of threads on a single machine.

A Thread takes a Runnable. You have to call `start` on a Thread in order for it to run the Runnable.

```
scala> val hello = new Thread(new Runnable {
  def run() {
    println("hello world")
  }
})
hello: java.lang.Thread = Thread[Thread-3,5,main]

scala> hello.start
hello world
```

When you see a class implementing Runnable, you know it's intended to run in a Thread somewhere by somebody.

Something single-threaded

Here's a code snippet that works but has problems.

```
import java.net.{Socket, ServerSocket}
import java.util.concurrent.{Executors, ExecutorService}
import java.util.Date

class NetworkService(port: Int, poolSize: Int) extends Runnable {
  val serverSocket = new ServerSocket(port)

  def run() {
    while (true) {
      // This will block until a connection comes in.
      val socket = serverSocket.accept()
      (new Handler(socket)).run()
    }
  }
}

class Handler(socket: Socket) extends Runnable {
```

```
  def message = (Thread.currentThread.getName() + "\n").getBytes

  def run() {
    socket.getOutputStream.write(message)
    socket.getOutputStream.close()
  }
}

(new NetworkService(2020, 2)).run
```

Each request will respond with the name of the current Thread, which is always `main`.

The main drawback with this code is that only one request at a time can be answered!

You could put each request in a Thread. Simply change

```
(new Handler(socket)).run()
```

to

```
(new Thread(new Handler(socket))).start()
```

but what if you want to reuse threads or have other policies about thread behavior?

Executors

With the release of Java 5, it was decided that a more abstract interface to Threads was required.

You can get an `ExecutorService` using static methods on the `Executors` object. Those methods provide you to configure an `ExecutorService` with a variety of policies such as thread pooling.

Here's our old blocking network server written to allow concurrent requests.

```
import java.net.{Socket, ServerSocket}
import java.util.concurrent.{Executors, ExecutorService}
import java.util.Date

class NetworkService(port: Int, poolSize: Int) extends Runnable {
  val serverSocket = new ServerSocket(port)
  val pool: ExecutorService = Executors.newFixedThreadPool(poolSize)

  def run() {
    try {
      while (true) {
        // This will block until a connection comes in.
        val socket = serverSocket.accept()
        pool.execute(new Handler(socket))
      }
    } finally {
      pool.shutdown()
    }
  }
}

class Handler(socket: Socket) extends Runnable {
  def message = (Thread.currentThread.getName() + "\n").getBytes

  def run() {
    socket.getOutputStream.write(message)
    socket.getOutputStream.close()
  }
}

(new NetworkService(2020, 2)).run
```

Here's a transcript connecting to it showing how the internal threads are re-used.

```
$ nc localhost 2020
pool-1-thread-1
```

```
$ nc localhost 2020
pool-1-thread-2

$ nc localhost 2020
pool-1-thread-1

$ nc localhost 2020
pool-1-thread-2
```

Futures

A `Future` represents an asynchronous computation. You can wrap your computation in a Future and when you need the result, you simply call a blocking `get()` method on it. An `Executor` returns a `Future`. If you use the Finagle RPC system, you use `Future` instances to hold results that might not have arrived yet.

A `FutureTask` is a Runnable and is designed to be run by an `Executor`

```
val future = new FutureTask[String](new Callable[String]() {
  def call(): String = {
    searcher.search(target);
}})
executor.execute(future)
```

Now I need the results so let's block until its done.

```
val blockingResult = future.get()
```

See Also Scala School's Finagle page has plenty of examples of using `Future`s, including some nice ways to combine them. Effective Scala has opinions about Futures .

Thread Safety Problem

```
class Person(var name: String) {
  def set(changedName: String) {
    name = changedName
  }
}
```

This program is not safe in a multi-threaded environment. If two threads have references to the same instance of Person and call `set`, you can't predict what `name` will be at the end of both calls.

In the Java memory model, each processor is allowed to cache values in its L1 or L2 cache so two threads running on different processors can each have their own view of data.

Let's talk about some tools that force threads to keep a consistent view of data.

Three tools

synchronization

Mutexes provide ownership semantics. When you enter a mutex, you own it. The most common way of using a mutex in the JVM is by synchronizing on something. In this case, we'll synchronize on our Person.

In the JVM, you can synchronize on any instance that's not null.

```
class Person(var name: String) {
  def set(changedName: String) {
    this.synchronized {
      name = changedName
    }
  }
}
```

volatile

With Java 5's change to the memory model, volatile and synchronized are basically identical except with volatile, nulls are allowed.

`synchronized` allows for more fine-grained locking. `volatile` synchronizes on every access.

```
class Person(@volatile var name: String) {
  def set(changedName: String) {
    name = changedName
  }
}
```

AtomicReference

Also in Java 5, a whole raft of low-level concurrency primitives were added. One of them is an `AtomicReference` class

```
import java.util.concurrent.atomic.AtomicReference

class Person(val name: AtomicReference[String]) {
  def set(changedName: String) {
    name.set(changedName)
  }
}
```

Does this cost anything?

@AtomicReference is the most costly of these two choices since you have to go through method dispatch to access values.

`volatile` and `synchronized` are built on top of Java's built-in monitors. Monitors cost very little if there's no contention. Since `synchronized` allows you more fine-grained control over when you synchronize, there will be less contention so `synchronized` tends to be the cheapest option.

When you enter synchronized points, access volatile references, or deference AtomicReferences, Java forces the processor to flush their cache lines and provide a consistent view of data.

PLEASE CORRECT ME IF I'M WRONG HERE. This is a complicated subject, I'm sure there will be a lengthy classroom discussion at this point.

Other neat tools from Java 5

As I mentioned with `AtomicReference`, Java 5 brought many great tools along with it.

CountDownLatch

A `CountDownLatch` is a simple mechanism for multiple threads to communicate with each other.

```
val doneSignal = new CountDownLatch(2)
doAsyncWork(1)
doAsyncWork(2)

doneSignal.await()
println("both workers finished!")
```

Among other things, it's great for unit tests. Let's say you're doing some async work and want to ensure that functions are completing. Simply have your functions `countDown` the latch and `await` in the test.

AtomicInteger/Long

Since incrementing Ints and Longs is such a common task, `AtomicInteger` and `AtomicLong` were added.

AtomicBoolean

I probably don't have to explain what this would be for.

ReadWriteLocks

`ReadWriteLock` lets you take reader and writer locks. reader locks only block when a writer lock is taken.

Let's build an unsafe search engine

Here's a simple inverted index that isn't thread-safe. Our inverted index maps parts of a name to a given User.

This is written in a naive way assuming only single-threaded access.

Note the alternative default constructor `this()` that uses a `mutable.HashMap`

```
import scala.collection.mutable

case class User(name: String, id: Int)
```

```
class InvertedIndex(val userMap: mutable.Map[String, User]) {

  def this() = this(new mutable.HashMap[String, User])

  def tokenizeName(name: String): Seq[String] = {
    name.split(" ").map(_.toLowerCase)
  }

  def add(term: String, user: User) {
    userMap += term -> user
  }

  def add(user: User) {
    tokenizeName(user.name).foreach { term =>
      add(term, user)
    }
  }
}
```

I've left out how to get users out of our index for now. We'll get to that later.

Let's make it safe

In our inverted index example above, userMap is not guaranteed to be safe. Multiple clients could try to add items at the same time and have the same kinds of visibility errors we saw in our first `Person` example.

Since userMap isn't thread-safe, how we do we keep only a single thread at a time mutating it?

You might consider locking on userMap while adding.

```
def add(user: User) {
  userMap.synchronized {
    tokenizeName(user.name).foreach { term =>
      add(term, user)
    }
  }
}
```

Unfortunately, this is too coarse. Always try to do as much expensive work outside of the mutex as possible. Remember what I said about locking being cheap if there is no contention. If you do less work inside of a block, there will be less contention.

```
def add(user: User) {
  // tokenizeName was measured to be the most expensive operation.
  val tokens = tokenizeName(user.name)

  tokens.foreach { term =>
    userMap.synchronized {
      add(term, user)
    }
  }
}
```

SynchronizedMap

We can mixin synchronization with a mutable HashMap using the SynchronizedMap trait.

We can extend our existing InvertedIndex to give users an easy way to build the synchronized index.

```
import scala.collection.mutable.SynchronizedMap

class SynchronizedInvertedIndex(userMap: mutable.Map[String, User]) extends InvertedIndex(userMap) {
  def this() = this(new mutable.HashMap[String, User] with SynchronizedMap[String, User])
}
```

If you look at the implementation, you realize that it's simply synchronizing on every method so while it's safe, it might not have the performance you're hoping for.

Java ConcurrentHashMap

Java comes with a nice thread-safe ConcurrentHashMap. Thankfully, we can use JavaConverters to give us nice Scala semantics.

In fact, we can seamlessly layer our new, thread-safe InvertedIndex as an extension of the old unsafe one.

```
import java.util.concurrent.ConcurrentHashMap
import scala.collection.JavaConverters._

class ConcurrentInvertedIndex(userMap: collection.mutable.ConcurrentMap[String, User])
    extends InvertedIndex(userMap) {

  def this() = this(new ConcurrentHashMap[String, User] asScala)
}
```

Let's load our InvertedIndex

The naive way

```
trait UserMaker {
  def makeUser(line: String) = line.split(",") match {
    case Array(name, userid) => User(name, userid.trim().toInt)
  }
}

class FileRecordProducer(path: String) extends UserMaker {
  def run() {
    Source.fromFile(path, "utf-8").getLines.foreach { line =>
      index.add(makeUser(line))
    }
  }
}
```

For every line in our file, we call `makeUser` and then `add` it to our InvertedIndex. If we use a concurrent InvertedIndex, we can call add in parallel and since makeUser has no side-effects, it's already thread-safe.

We can't read a file in parallel but we *can* build the User and add it to the index in parallel.

A solution: Producer/Consumer

A common pattern for async computation is to separate producers from consumers and have them only communicate via a `Queue`. Let's walk through how that would work for our search engine indexer.

```
import java.util.concurrent.{BlockingQueue, LinkedBlockingQueue}

// Concrete producer
class Producer[T](path: String, queue: BlockingQueue[T]) extends Runnable {
  def run() {
    Source.fromFile(path, "utf-8").getLines.foreach { line =>
      queue.put(line)
    }
  }
}

// Abstract consumer
abstract class Consumer[T](queue: BlockingQueue[T]) extends Runnable {
  def run() {
    while (true) {
      val item = queue.take()
      consume(item)
    }
  }

  def consume(x: T)
}

val queue = new LinkedBlockingQueue[String]()
```

```
// One thread for the producer
val producer = new Producer[String]("users.txt", q)
new Thread(producer).start()

trait UserMaker {
  def makeUser(line: String) = line.split(",") match {
    case Array(name, userid) => User(name, userid.trim().toInt)
  }
}

class IndexerConsumer(index: InvertedIndex, queue: BlockingQueue[String]) extends Consumer[String](queue) with UserMaker {
  def consume(t: String) = index.add(makeUser(t))
}

// Let's pretend we have 8 cores on this machine.
val cores = 8
val pool = Executors.newFixedThreadPool(cores)

// Submit one consumer per core.
for (i <- i to cores) {
  pool.submit(new IndexerConsumer[String](index, q))
}
```

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;