Concurrency in Java and Actor-based concurrency using Scala

By,

Srinivas Panchapakesan

Concurrency

- Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel.
- Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network.
- The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes.

Advantages of Concurrency

- Almost every computer nowadays has several CPU's or several cores within one CPU. The ability to leverage theses multi-cores can be the key for a successful high-volume application.
- Increased application throughput - parallel execution of a concurrent program allows the number of tasks completed in certain time period to increase.
- High responsiveness for input/output-intensive applications mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task.
- More appropriate program structure - some problems and problem domains are well-suited to representation as concurrent tasks or processes.

Process vs Threads

- **Process**: A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.
- **Threads**: Threads are so called lightweight processes which have their own call stack but an access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data, when this happens in Java will be explained in Java memory model part of this article.

Concurrency in Java

- A Java program runs in its own process. Java supports within one program the usage of threads.
- Java supports threads as part of the Java language. Java 1.5 also provides improved support for concurrency with the in the package java.util.concurrent.
- There are also a few non-traditional models in Java to achieve concurrency. The Interop between Java and Scala is one of them. We will be discussing this later in this presentation.

Threads – What are they for?

- To enable cancellation of separable tasks
- To maintain responsiveness of an application during a long running task
- Some problems are intrinsically parallel
- Some APIs and systems demand it. Like Swing.
- To monitor status of some resource like data bases.

What happens when we run a Thread?

- When we execute an application:
  1. The JVM creates a Thread object whose task is defined by the `main()` method
  2. The JVM starts the thread
  3. The thread executes the statements of the program one by one
  4. After executing all the statements, the method returns and the thread dies

More on Threads….

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap) .
- Two different threads can act on the same object and same static fields concurrently

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

void yield()

- Causes the currently executing thread object to temporarily pause and allow other threads to execute
- Allow only threads of the same priority to run

void sleep(int *m*) or sleep(int *m*, int *n*)

- The thread sleeps for *m* milliseconds, plus *n* nanoseconds

t.join()

- If t is a Thread object whose thread is currently executing. T.join() causes the current thread to pause execution until t's thread terminates.

Life Cycle of A Thread

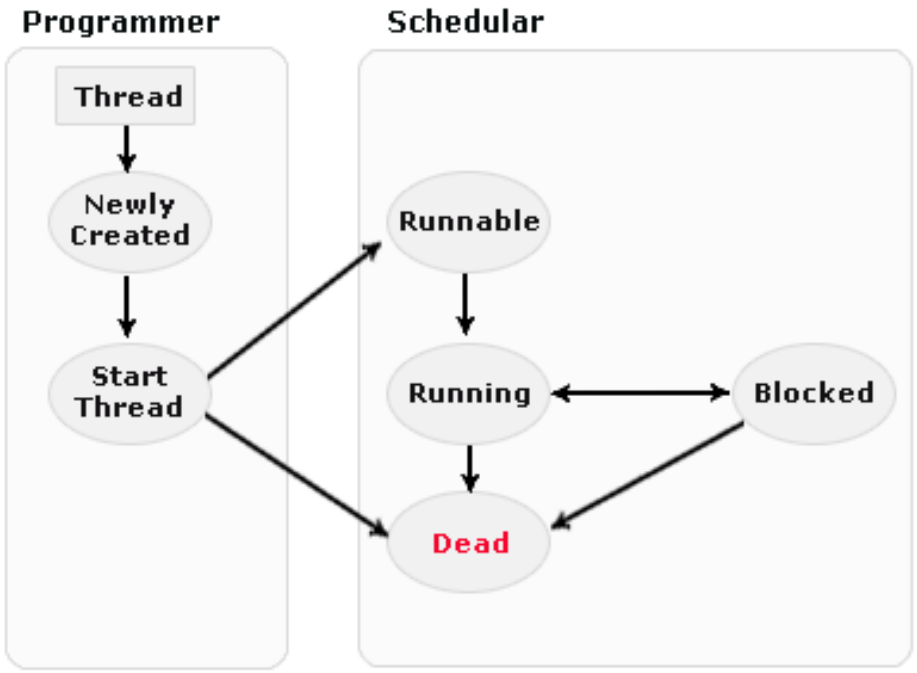


Image from: http://www.roseindia.net/java/thread

Contd…

- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Creating Threads

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

Let's take a look at some code.

- **Subclassing the Thread class**

```
public class MyThread extends Thread {
   public void run () {
      for (int i = 1; i <= 100; i++) {
         System.out.println("Message1");
      }
   }
}
```

- **Implementing the Runnable interface**

```
public class RunnableExample implements Runnable {
   public void run () {
    for (int i = 1; i <= 100; i++) {
                    System.out.println ("Message2");
      }
   }
}
```

Example code

Consider the thread given below.

```
public class PrintThread extends Thread {
    String name;
    public PrintThread(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<100 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }

            System.out.print(name);
        }
}
```

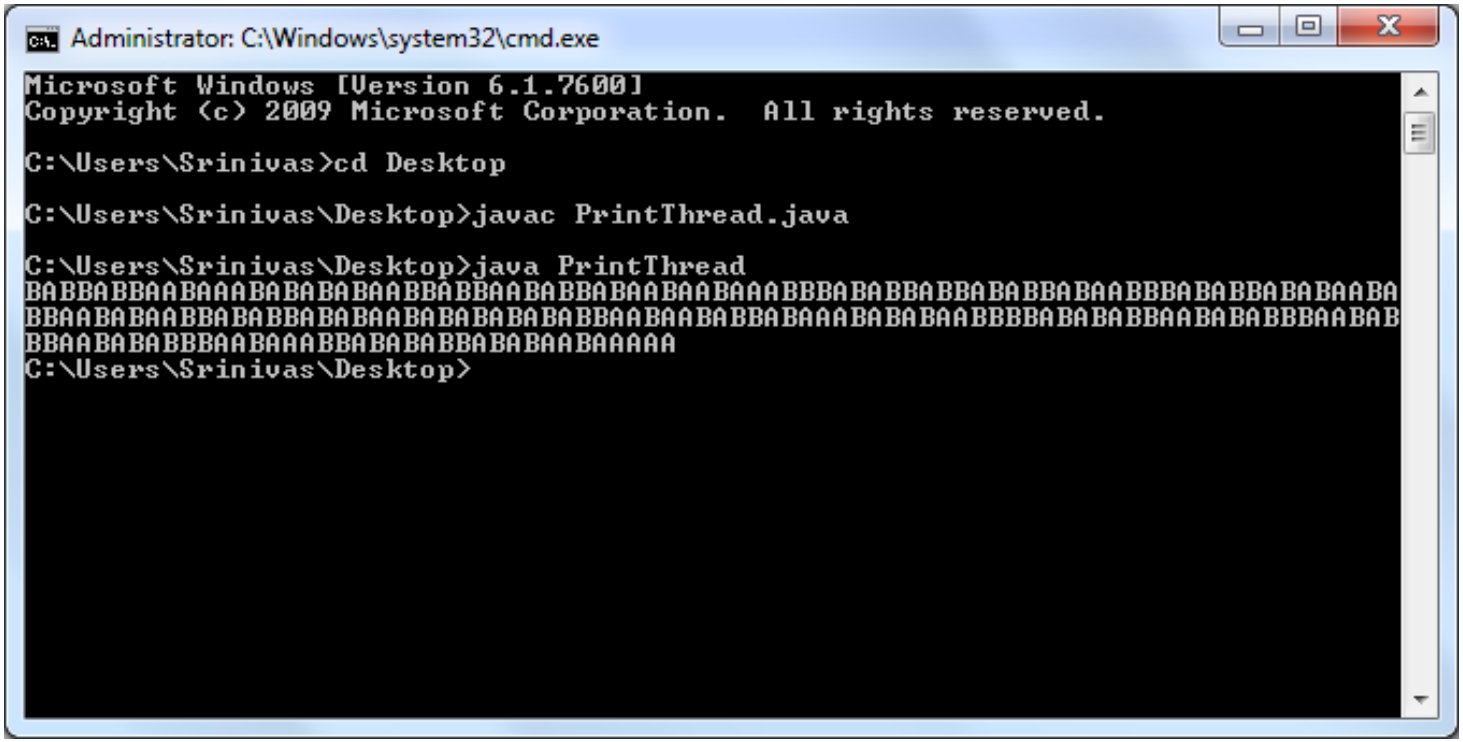
Contd…

We are now running two threads.

```
public static void main(String args[]) {
        PrintThread a = new PrintThread1
  ("A");
        PrintThread b = new PrintThread1
  ("B");

        a.start();
        b.start();
  }
}
```

Example: Output



- We can see in the output above that there is no way to determine the order of execution of threads.

Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either preemptive or nonpreemptive

Scheduling Policies

- Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- Nonpreemptive scheduling – the scheduler never interrupts a running thread
- The nonpreemptive scheduler relies on the running thread to yield control of the CPU so that other threads may execute

Starvation

- A nonpreemptive scheduler may cause starvation (runnable threads, ready to be executed, wait to be executed in the CPU a very long time, maybe even forever)
- Sometimes, starvation is also called a livelock

Time-Sliced Scheduling

- Time-sliced scheduling
  - The scheduler allocates a period of time that each thread can use the CPU
  - When that amount of time has elapsed, the scheduler preempts the thread and switches to a different thread
- Nontime-sliced scheduler
  - the scheduler does not use elapsed time to determine when to preempt a thread
  - it uses other criteria such as priority or I/O status

Java Scheduling

- Scheduler is preemptive and based on priority of threads
- Uses fixed-priority scheduling:
  - Threads are scheduled according to their priority with respect to other threads in the ready queue
- The highest priority runnable thread is always selected for execution above lower priority threads
- Java threads are guaranteed to be preemptive-but not time sliced
- When multiple threads have equally high priorities, only one of those threads is guaranteed to be executing

Priority of a Thread

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority
- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

Important concepts: Daemon Threads

- Daemon threads are "background" threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits

Important concepts: Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object
- Both threads "race" to store their value

Important concepts: Critical Section

- The synchronized methods define critical sections
- A critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.
- Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements to solve the errors described above.

Thread Interference

```
class Example{
    private int c = 0;     public void increment() {
        c++;
  }
     public void decrement() {
        c--;
     }
       public int value() {
       return c;
       }
}
```

- In the above class, increment will add 1 to c, and each invocation of decrement will subtract 1 from c.
- if an object of the above class is referenced from multiple threads, interference between threads may prevent this from happening as expected.
- Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Problem with this code

Consider the initial value of c is 0. If there are to threads and Thread 1 invokes increment at about the same time Thread 2 invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

- Thread 1: Retrieve c.
- Thread 2: Retrieve c.
- Thread 1: Increment retrieved value; result is 1.
- Thread 2: Decrement retrieved value; result is -1.
- Thread 1: Store result in c; c is now 1.
- Thread 2: Store result in c; c is now -1.

- Thread 1's result is lost, overwritten by Thread 2. This particular interleaving is only one possibility. Under different circumstances it might be Thread 2's result that gets lost, or there could be no error at all.
- Threads are unpredictable, thread interference bugs can be difficult to detect and fix.

Memory consistency errors

- *Memory consistency errors* occur when different threads have inconsistent views of what should be the same data.
- The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.
- Consider a counter with initial value "0". The counter is shared between two threads, A and B. Suppose thread A increments counter:

```
counter++;
```

- Then, shortly afterwards, thread B prints the counter:

```
System.out.println(counter);
```

- If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B.
- One of the was to solve this problem is by the use of synchronization.

Synchronized Methods

- To make a method synchronized, simply add the synchronized keyword to its declaration:

```
class SynchronizedExample{
    private int c = 0;

public synchronized void increment() {
        c++;
   }
      public synchronized void decrement() {
         c--;
      }
        public synchronized int value() {
        return c;
        }
}
```

Synchronized Methods (cont.)

- Making the methods as synchronized has two effects:

- It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done with the object.

- This automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. The changes to the state of the object are visible to all threads.

- This strategy is effective, but can present problems with deadlock, starvation and livelock.

Monitors or Intrinsic Locks

- Each object has a "monitor" that is a token used to determine which application thread has control of a particular object instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued

Monitors or Intrinsic Locks(cont.)

- Entering a monitor is also referred to as locking the monitor, or acquiring ownership of the monitor
- As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Synchronized Statements

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addItem(String item) {
synchronized(this) {
    lastItem = item;
    itemCount++;
  } itemList.add(item);
 }
```

- In this example, the addItem method needs to synchronize changes to lastItem and itemCount. It also needs to avoid synchronizing invocations of other objects' methods. Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking itemList.add.

Atomic Access

- An *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. It cannot be broken down into other simpler sentences.
- There are actions in java you can specify that are atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
  - Reads and writes are atomic for *all* variables declared volatile (*including* long and double variables).
- Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.
- Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors.

High Level Concurrency Objects: A Brief Overview.

- We have discussed about the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.
- High-level concurrency features were introduced with version 5.0 of the Java platform. Most of these features are implemented in the new java.util.concurrent packages.

- Lock objects support locking idioms that simplify many concurrent applications.
- Executors define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
- Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
- ThreadLocalRandom (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

A look at Java concurrency again

- All concurrency abstractions in Java are based on the shared mutable state paradigm and prone to the vulnerabilities of race conditions, blocking/contention and deadlocks.
- Java 5 gave us better abstractions for coarse-grained concurrency.
- Fork/Join will allow programmers to exploit finer-grained concurrency by allowing independent task components to be parallelized through recursive divide-and-conquer techniques.
- The Fork/Join task will be mapped to a pool of worker threads that initiates and manages their execution using advanced queuing and scheduling techniques.

A look at Java concurrency again (Cont.)

- The shared memory model is a cause of concern in Java. As a model of computation, threads are non-deterministic, and hence not composable
- When multiple threads can potentially invade a piece of critical section concurrently, the programming model becomes hugely complicated.
- Fork/Join points us towards functional decomposition, which becomes most effective, when we can talk about side-effect-free subtasks that can execute in its own thread of execution independent of its other counterparts.
- We should note that not all problems are parallelizable via decomposition. To ensure that all subtasks do not share any mutable state, while programming in an imperative language, is a big ask.

Actor concurrency

The actor model consists of a few key principles:

- No shared state
- Lightweight processes
- Asynchronous message-passing
- Mailboxes to buffer incoming messages
- Mailbox processing with pattern matching

Actor concurrency (cont.)

- An Actor is a mathematical model of concurrent computation that encapsulate data, code and its own thread of control, and communicate asynchronously using immutable message passing techniques.
- When the basic architecture is shared-nothing, each actor seems to act in its own *process* space.
- The language implement lightweight processes on top of the native threading model.
- Every actor has it's own mailbox for storing messages, implemented as asynchronous, race-free, non-blocking queues.
- Scala actors use Fork/Join as the underlying implementation. And exposes a concurrency model that encourages shared-nothing structures that interact through asynchronous message passing.

Actor concurrency (cont.)

- An actor is a process that executes a function. Here a process is a lightweight user-space thread (not to be confused with a typical heavyweight operating-system process).
- Actors never share state and thus never need to compete for locks for access to shared data.
- Instead, actors share data by sending messages that are immutable. Immutable data cannot be modified, so reads do not require a lock.
- Messages are sent asynchronously and are buffered in an actor's mailbox.
- A mailbox is essentially a queue with multiple producers (other actors) and a single consumer. A particular actor is driven by receiving messages from the mailbox based on pattern matching.

Actor message passing.

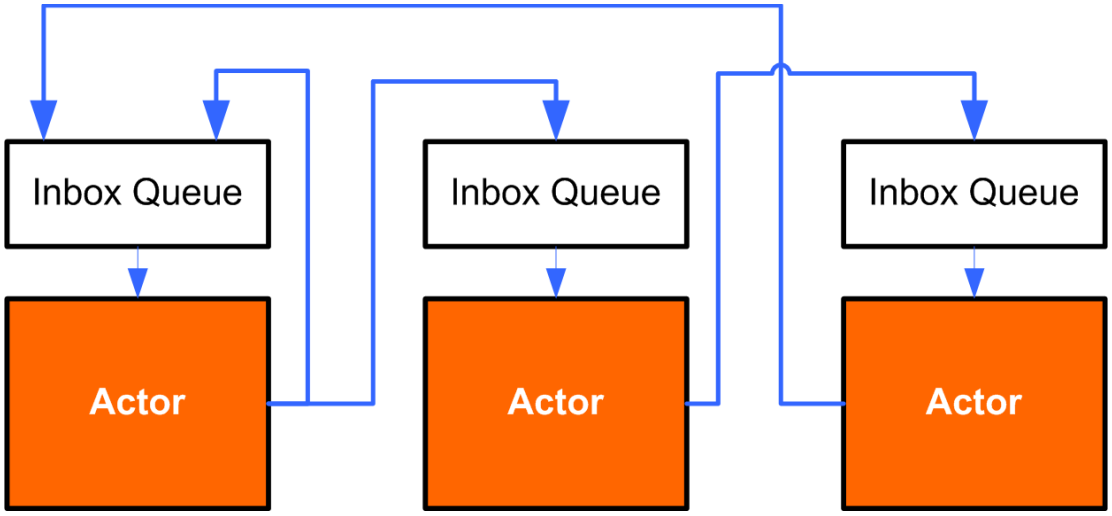


Fig: Message passing between objects, each object has an inbox (a queue) of messages which it works through and objects send messages to the inbox of other objects, where messages are immutable

Ref: http://codemonkeyism.com

- Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way.
- It smoothly integrates features of object-oriented and functional languages and also enables Seamless integration with Java.
- Scala programs run on the Java VM, are byte code compatible with Java so you can make full use of existing Java libraries or existing application code.
- You can call Scala from Java and you can call Java from Scala, the integration is seamless. Moreover, you will be at home with familiar development tools, Eclipse, NetBeans or Intellij for example, all of which support Scala.

Actors in Scala

- Scala is a hybrid language that takes cues from both the object-oriented and functional programming traditions.
- the Scala actors implementation is provided as part of the Scala standard library (equivalent to the JDK in Java), not as part of the language. It replicates much of the Erlang actor model.
- Actors are defined as a *trait* in the Scala standard library.
- In Scala, a trait is a set of method and field definitions that can be mixed into other classes.
- This is similar to providing an abstract superclass in Java but more flexible, particularly in comparison to multiple inheritance.

Pattern Matching

- In Scala, a class is a type definition, and you can create instances with new just as in Java. You can also define a type as an object, which means it is always a true singleton instance.
- To do pattern matching, Scala has a concept of *case classes* and *case objects*. Case classes are defined with a special case modifier that causes the compiler to add some extra code to the class.
- Actors perform pattern matching on messages received from their inboxes, so it is common to use case classes to define messages.

Example: Scala Actor

```
class Player() extends Actor {
 def act() {
 // message handling loop
 }
}
```

- We are creating a simple actor in the above code. The act() method is part of the contract of the Actor trait

Example: Scala Actor (cont.) Scala Message Definitions

```
abstract case class Move
 case object Rock extends Move
 case object Paper extends Move
 case object Scissors extends Move
 case object Start
 case class Play(sender:Actor)
 case class Throw(player:Actor, move:Move)
```

- We are now defining some messages. This defines an abstract Move class and three singleton implementations of that class representing rock, paper, and scissors.

- This also defines a singleton Start message and classes for Play and Throw message classes.

Example: Scala Actor (cont.) The Coordinator actor

```
class Coordinator() extends Actor {
  def act() {
    val player1 = new Player().start
    val player2 = new Player().start
    loop {
      react {
        case Start => {
          player1 ! Play(self)
          player2 ! Play(self)
     }
 case Throw(playerA, throwA) => {
    react {
       case Throw(playerB, throwB) => {
       announce(playerA, throwA, playerB, throwB)
       self ! Start
       }
      }
     }
    }
   }
  }
```

Example: Scala Actor (cont.)
The Coordinator actor

- The *react* behavior, provides a lightweight actor that is not bound to a thread.
- The *receive* behavior creates an actor that is actually bound to ajava.lang.Thread
- Every actor has an internal mailbox where messages are received from other actors and consumed within the actor.
- In the Coordinator actor, we see our case classes being used during message matching.
- Messages are sent asynchronously with the ! Operator.

Scala Conclusion

- The Scala support for actors is quite strong, building firmly on the Erlang model.
- Writing actors code is a natural part of writing Scala code, and the Scala library has full support not just for actor definition, pattern matching, and flexible message send, but also for many of the Erlang-style error-handling primitives.

Other actor frameworks for Java

- Moving on to Java libraries.
- Scala and can create lightweight actors by relying on scheduling over a thread pool of actors defined by partial functions or closures.
- In Java, these techniques don't exist as part of the language.
- Most of the Java actor libraries rely on bytecode modification either at compile time or runtime.

Kilim

- Kilim is an actor framework that relies on the two concepts of Task and Mailbox.
- To create an actor, your class should extend Task, but an actor does not have a mailbox created as part of the Task.
- We must create and provide Mailboxes between actors as needed by the application.
- Mailboxes use generics, and you must define messages passed through mailboxes as classes. It must be initialized with its inbox.

Kalim Example

```
public class Player extends Task {
  private static final Random RANDOM = new Random();
  private final String name;
  private final Mailbox<PlayMessage> inbox;

  public Player(String name, Mailbox<PlayMessage> inbox) {
    this.name = name;
    this.inbox = inbox;
  }

  public void execute() throws Pausable {
    while(true) {
      PlayMessage message = inbox.get();
      message.getResponseMailbox().putnb(new ThrowMessage(name, randomMove()));
    }
  }

  private Move randomMove() {
    return Move.values()[RANDOM.nextInt(Move.values().length)];
  }
}
```

- The above snippet displays the Player actor in Kilim. It must be initialized with its inbox. The Coordinator's mailbox is passed inside the Play Message for a reply.

Other Frameworks:

ActorFoundry:

ActorFoundry is a different take on actors in Java. It piggybacks on Kilim by reusing the Kilim compile-time weaver. However, it provides a different API and a different runtime execution environment.

Actors Guild

Actors Guild is a Java actor framework that relies on runtime bytecode modification with the ASM library and thus is easier to integrate into your project than Kilim or ActorFoundry.

Other Frameworks (Cont.)

Jetlang

- This technically does not provide an actor framework but rather a set of building blocks from which actor-like functionality.
- Jetlang is based on the older Retlang library for .NET.
- Jetlang has three primary concepts:
  - Fiber - a lightweight thread-like construct
  - Channel - a way to pass messages between Fibers
  - Callback - a function to be invoked when a message arrives on aChannel in the context of a Fiber

Conclusion

- Concurrency is an important part of programming these days. More and More focus will be made on Parallel programming in the near future. It is important to learn the different ways in which we can achieve parallelism in a program.
- We have talked about actor based concurrency as well as Java threads. Though threads seem difficult to manage, there are plenty of applications to it. There is no right or wrong approach to write a parallel program. Different applications will have different demands.

Reference

- http://download.oracle.com/javase/tutorial
- http://www.javaworld.com
- http://codemonkeyism.com
- http://apocalisp.wordpress.com/2008/07/28/threadless-concurrency-with-actors/