Actors
Overview

“If it hurts, stop doing it”

“If **it** hurts, stop doing **it**”

In concurrent programming, shared mutability is “**it**”.

Return to the Basic Problem

- Concurrency can improve responsiveness, reliability, utilization, etc.
- The problem, however, is *shared* access to *mutable* state.
- Eliminate mutability:
  - The focus on functional approaches.
  - Single assignment variables, function values, and recursion.
  - However, eventually something must change (though we can reduce this a lot).
- Eliminate sharing:
  - Divy up work among different concurrent actors (implemented via threads).
  - Every mutable object belongs to *exactly one* actor.
  - Actors interact by sending *immutable messages* to each other.

Actors

- Actors were defined in a 1973 paper by Carl Hewitt and were popularized by the Erlang language (1986), later in Scala.
- Actors “act upon” a message they receive
- Actors encapsulate state and behavior into a lightweight process/thread.
- Like OO objects but with a major semantic difference; they *do not* share state with any other Actor
- Only have impact on other Actors by sending ***messages*** to them.
  - Messages are sent asynchronously and are non-blocking
  - Each Actor has a mailbox (ordered message queue) in which incoming messages are processed one by one.
  - Messages must be immutable (but this is not enforced!). One of the risks is to accidentally share mutable state between actors

http://doc.akka.io/docs/akka/2.1.0/general/actors.html

Designing Actor Systems

- Provides a higher level of abstraction for writing concurrent and distributed systems.
- Don’t have to deal with explicit locking and thread management.
- Actors are typically organized hierarchically :
  - Think of a human organization with workers and supervisors
  - Each actor has exactly one supervisor, the actor that created it.
  - If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help.

http://doc.akka.io/docs/akka/2.1.0/general/actor-systems.html

Programming Actor Based Concurrency

- Akka
  - Scala based solution with a Java API
  - Also used for Software Transactional Memory (STM)

- Java API capabilities:
  - Create Actors
  - Send / Receive Messages
  - Coordinating Multiple Actors
  - Typed Actors
  - Transactions Support (STM) – we'll cover with STM
  - Remote Actors (Distributed Systems)

Actors in AKKA

- First we define the message classes – what messages will we exchange between actors.
  - Objects in the class must be (though this can't be enforced) immutable.
- Second we define the Actors that exchange messages:
  - For now, our actors extend **UntypedActor**.
  - They define one public method:
    void onReceive(Object message)
  - The message class is used to determine the message type.
- Actors are wrapped in a context via **actorOf**
  - actorOf returns an ActorRef
  - The actor is launched via the **start()** method
  - Given a reference to an actor, we send a message via **tell()**.
  - **poisonPill()** messages terminate actors.

Declaring Actors with akka

```
import akka.actor.* ;

public class PCDemo {

  /*
   * Two producers; one consumer
   */
  static private ActorRef [] producer = new ActorRef[2] ;
  static private ActorRef consumer ;

  static class Producer extends UntypedAbstractActor {…}

  static class Consumer extends UntypedAbstractActor {…}
```

Instantiating Actors with and without Constructors

```
ActorSystem system = ActorSystem.create ();

producer[0] = system.actorOf(Props.create
(Producer.class,"Pete"));

consumer = system.actorOf(Props.create(Consumer.class));


// You may see the following in the book/examples
consumer = system.actorOf (
        Props.create (
                new Creator<Consumer>(){
                        public Consumer create () {
                                return new Consumer ();
                }
}));
```

Instantiating Actors with and without Constructors

```
ActorSystem system = ActorSystem.create ();

producer[0] = system.actorOf(Props.create
(Producer.class,"Pete"));

consumer = system.actorOf(Props.create(Consumer.class));
```

```
// It is deprecated, so don't use it
consumer = system.actorOf (
        Props.create (
                new Creator<Consumer>(){
                        public Consumer create () {
                                return new Consumer ();
                }
}));
```

Starting and Stopping Actors

```
 producer[0].start() ;
 producer[1].start() ;
 consumer.start() ;

// Have actors do stuff

 producer[0].stop() ;
 producer[1].stop() ;
 consumer.stop() ;
```

This is the v1.0 way, it doesn't work anymore.

Starting and Stopping Actors

```
~~producer[0].start() ;~~
~~producer[1].start() ;~~
~~consumer.start() ;~~

~~// Have actors do stuff~~

~~producer[0].stop() ;~~
~~producer[1].stop() ;~~
~~consumer.stop() ;~~
```

Instead, use the ActorSystem which auto-starts actors upon creation.

```
ActorSystem system = ActorSystem.create ();

// Add actors to the system
// Actors will start as soon as they are added

// Do stuff

System.terminate () // Graceful shutdown
```

Sending (tell) and Receiving (onReceive) Messages

```
for ( int i = 1 ; i < 10 ; i++ ) {
    producer[(i % 2)].tell("Message #" + i, ActorRef.noSender());
}


// Consumer message receive
public void onReceive(Object message) {
    String s = (String) message ;
    System.out.println("  Consumer receives " + s) ;
}
```

Use `instanceOf` to identify message type if needed.

Producer Side of Demo

Producer sends message to consumer
Yields
Sends message a send time

```
Producer Pete receives and passes Message #2-1
Producer Mike receives and passes Message #1-1
Producer Pete repasses Message #2-2
  Consumer receives Message #2-1
Producer Mike repasses Message #1-2
  Consumer receives Message #1-1
Producer Pete receives and passes Message #4-1
  Consumer receives Message #2-2
Producer Mike receives and passes Message #3-1
  Consumer receives Message #1-2
Producer Pete repasses Message #4-2
  Consumer receives Message #4-1
Producer Mike repasses Message #3-2
  Consumer receives Message #3-1
Producer Pete receives and passes Message #6-1
  Consumer receives Message #4-2
Producer Mike receives and passes Message #5-1
  Consumer receives Message #3-2
Producer Pete repasses Message #6-2
  Consumer receives Message #6-1
Producer Mike repasses Message #5-2
  Consumer receives Message #5-1
Producer Pete receives and passes Message #8-1
  Consumer receives Message #6-2
Producer Mike receives and passes Message #7-1
  Consumer receives Message #5-2
Producer Pete repasses Message #8-2
  Consumer receives Message #8-1
Producer Mike repasses Message #7-2
  Consumer receives Message #7-1
Producer Mike receives and passes Message #9-1
  Consumer receives Message #8-2
Producer Mike repasses Message #9-2
  Consumer receives Message #7-2
  Consumer receives Message #9-1
  Consumer receives Message #9-2
```

Typed Actors

- Why can't an Actor be more like an Object?
  - Why do we have to send messages to Actors?
  - Why does the Actor have to be written as an event loop?
  - Why can't we use call / return syntax?
- Well, with **Typed Actors** we can!
  - Typed actors are defined by a Java interface & implementation.
  - When created, work as a standard object in both client *and* provider.
    - Client gets a proxy (also an actor) for the actor of the interface type.
    - Proxy marshalls arguments and sends request to "service actor."
    - Service actor responds to onReceive by unmarshalling arguments.
    - Service actor calls the specified method.
    - If non-void, marshalls results and responds to the proxy.
    - Proxy returns to the client.

Why Typed Actors?

- Typed Actors are nice for bridging between actor systems (the “inside”) and non-actor code (the “outside”), because they allow you to write normal OO-looking code on the outside.
- Typed Actors do have their place, as hybrids between POJO and Actor. For a longer discussion see this blog post.

However….

- Typed Actors can very easily be abused as remote procedure calls (RPC). They have characteristics similar to Java RMI and the accompanying challenges of distributed system design.
- Hence Typed Actors are not what we think of first when we talk about making highly scalable concurrent software easier to write correctly. They have their niche, use them sparingly.

Stack Implemented as a Typed Actor

```
import static akka.actor.Actors.* ;

import akka.actor.* ;
import akka.actor.TypedActor ;
import java.util.* ;

public class Main {
   private static StringStack stack ;

   public static void main(String[] args) throws InterruptedException {

                  stack = (StringStack) TypedActor.newInstance(StringStack.class, StringStackImpl.class) ;
                                                                                  interface          implementation
      for ( Integer i = 0 ; i < 10 ; i++ ) {
         stack.push( "String[" + i + "]" );
      }

      for ( Integer i = 0 ; i < 10 ; i++ ) {                                // push & pop look like method calls, but
                     String v = stack.pop() ;                               // get implemented as messages to the
                     System.out.println("String " + v + " popped") ;       // StringStack typed actor
      }

      Thread.sleep(250) ;

                  TypedActor.stop(stack) ;
   }
}
```

Stack Implemented as a Typed Actor

Interface: StringStack.java

```
public interface StringStack {
   public void push(String s) ;

   public String pop()  ;
}
```

Implementation: StringStackImpl.java

```
import akka.actor.TypedActor ;
import java.util.* ;

public class StringStackImpl extends TypedActor implements StringStack {     // NOTE extension of TypedActor

 private final Deque<String> theStack = new ArrayDeque<String>() ;     // The shared mutable resource

   public void push(String s) {     // "receive" a Push message from the proxy
      System.out.println("Push(" + s + ")" ) ;
      theStack.addFirst(s) ;
   }

   public String pop()     // "receive" a Pop message from the proxy
       String result = theStack.removeFirst() ;
        System.out.println("Pop(" + result + ")" ) ;
      return result ;
   }
}
```

// “receive” a Push message from the proxy

```
public class Push {
   public final String value ;
   public Push(String value) {
      this.value = value ;
   }
}
```

// “receive” a Pop message from the proxy

```
public class Pop {
}
```