CMPT 435/835 Tutorial 1

Actors Model & Akka

Ahmed Abdel Moamen

PhD Candidate

Agents Lab

ama883@mail.usask.ca

Department of Computer Science

UNIVERSITY OF SASKATCHEWAN

Content

Actors Model

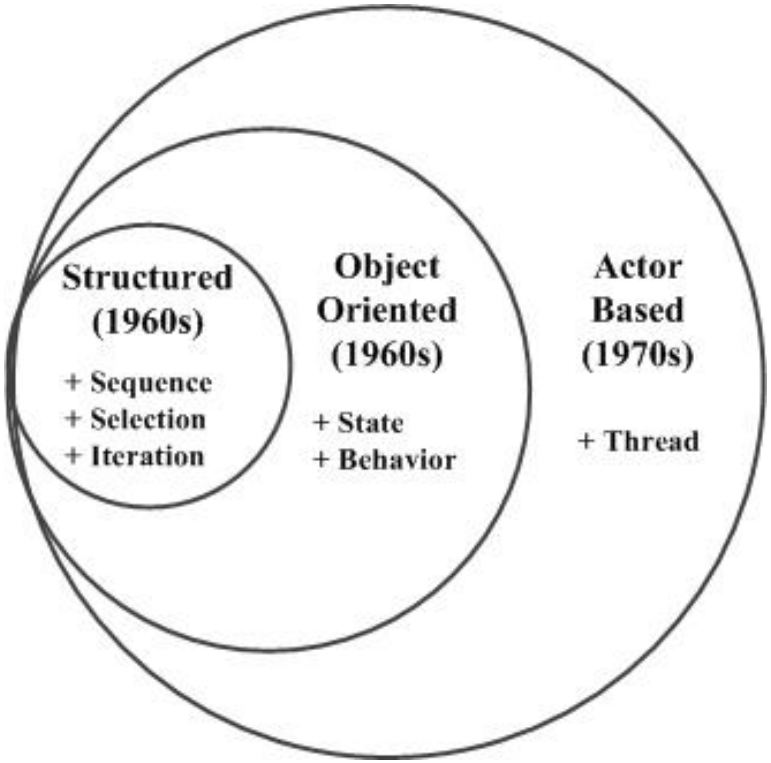
Akka (for Java)

Actors Model (1/7)

- Definition
  - A general model of concurrent computation for developing parallel, distributed, and mobile systems.

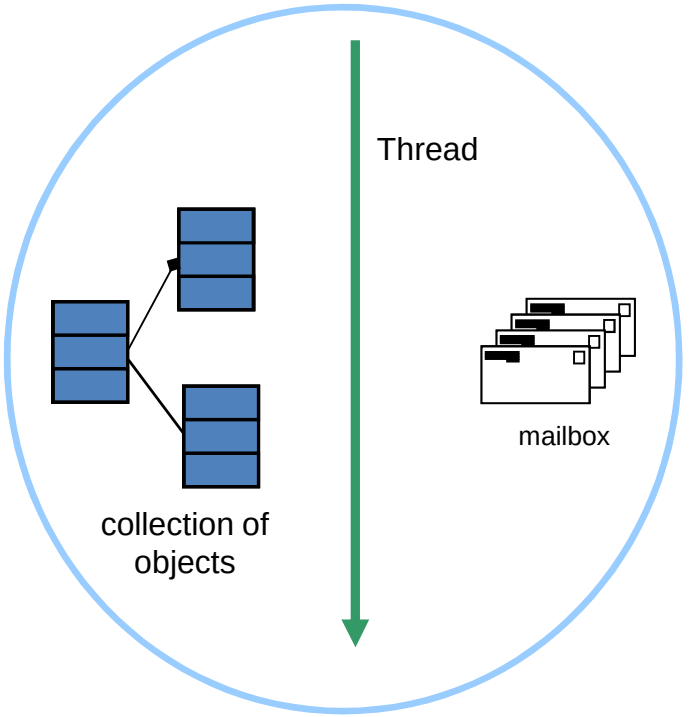
Actors Model (2/7)

- Structured programming considers all programs as composed of three control structures.
- Objects encapsulate data (state) and behavior, separating the interface of an object (what an object does) from its representation (how it does it).
- Actors extend the advantages of objects to concurrent computation by separating control (where and when) from the logic of a computation.



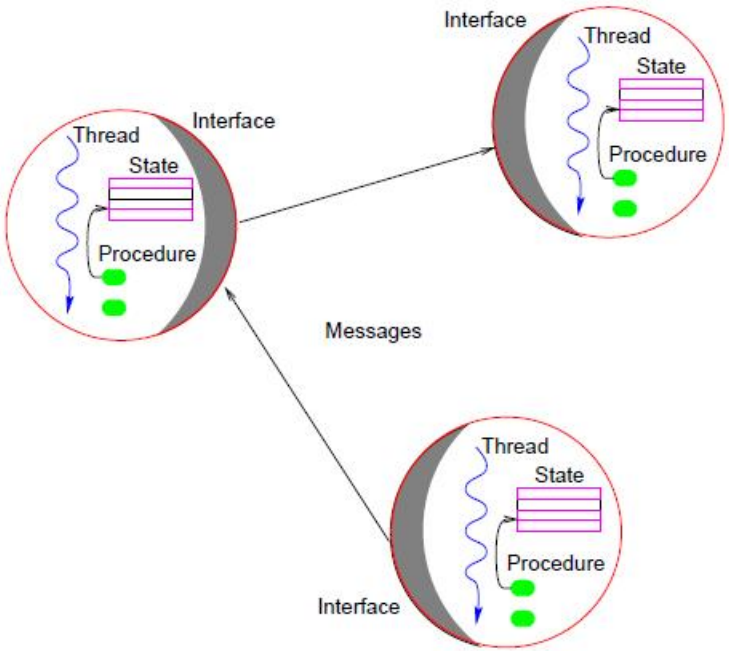
Actors Model (3/7)

- An Actor encapsulates a thread of execution with a collection of objects.
- Only the actor's thread may access its objects directly and change their state.
- Actors communicate by sending messages to each other.
- Messages are sent asynchronously.
- Messages are not necessarily processed in the order they are sent or received.



Actors Model (4/7)

- Each actor is a computational entity, that encapsulates a thread and a state, can concurrently:
  - ***send*** a finite number of messages to other actors;
  - ***create*** a finite number of new actors;
  - designate a new behavior to process subsequent messages (***become***).



Actors Model (5/7)

- The key semantic properties of the pure (standard) Actors model:
  - Encapsulation of state ;
  - Atomic execution of a method in response to a message;
  - Fairness in scheduling actors and in the delivery of messages;
  - Location transparency enabling distributed execution and mobility.

Actors Model (6/7)

- Message passing
  - Recipients of messages are identified by addresses (actor names);
  - Addresses cannot be guessed, which means an actor can only communicate with actors whose addresses it has;
  - Direct and asynchronous communication;
  - No requirement on order of message arrival;
  - Fairness in the delivery of messages.

Actors Model (7/7)

- Actors model has been applied in both research and industrial fields.
  - Actor languages: ErLang, Scala, SALSA, ABCL, etc.
  - Actor libraries: ActorFoundry (Java), Akka (Java & Scala), Pulsar (Python), CAF (C++), etc.
- In practice, Actors model is usually extended for ease of use.

Content

Actors Model

Akka (for Java)

Akka – General Information (1/18)

- Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM.
- About the name
  - A palindrome of letters **A** and **K** as in **A**ctor **K**ernel;
  - The name of a Swedish mountain (where does the logo come from).
- The current version of Akka supports both Java and Scala.
  - Scala started to use Akka as its default actor library from version 2.10.0.

Akka - Resources (2/18)

- Official website: http://akka.io/
- A simple step-by-step tutorial (Java): http://doc.akka.io/docs/akka/2.0.2/intro/getting-started-first-java.html
- The reference manual for the latest stable release v2.3.13: http://doc.akka.io/docs/akka/2.3.13/AkkaJava.pdf?_ga=1.187277096.1279128105.1440188263

Akka – Creating Actors (3/18)

- Step 1/3 (Defining)
  - Actors in Java are implemented by extending the *UntypedActor* class and implementing the *onReceive* method.

```
import akka.actor.UntypedActor;
public class MyActor extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        if (message instanceof  String) {
            System.out.println(message);
        } else {
            unhandled(message);
}}}
```

Akka – Creating Actors (cont.) (4/18)

- *Step 2/3 (Configuring)*
  - *Props* is a configuration class to specify options for the creation of actors.

```
import akka.actor.Props;
…
Props props1 = Props.create(MyActorA.class);
Props props2 = Props.create(MyActorB.class, arg1, arg2);
```

Akka – Creating Actors (cont.) (5/18)

- Step 3/3 (Creating)
  - Actors are created by passing a *Props* instance into the *actorOf* factory method which is available on *ActorSystem* and *ActorContex*.
    - Use *ActorSystem* to create top-level actors which are supervised by the actor system's provided guardian actor.
    - Use actor's context (*ActorContex*) to create a child actor.

```
import akka.actor.ActorRef;
Import akka.actor.ActorSystem;
…
final ActorSystem system = ActorSystem.create("MySystem");
final ActorRef myTopLevelActor =  system.actorOf(Props.create(MyActorA.class), "myActor1");
…
class A extends UntypedActor {
      final ActorRef  myChildActor = getContext().actorOf(Props.create(MyActorB.class, "myActor2");
      …
}
```

Akka – Creating Actors (cont.) (6/18)

- *UntypedActor* APIs
  - *getSelf()*, references to the *ActorRef* of the actor.
  - *getSender()*, references to the sender actor of the last received message.
  - *getContext()*, exposes contextual information for the actor and the current message.
  - User-overridable life-cycle hooks
    - *preStart()*, executed after starting the actor (and also by *postRestart()* as default);
    - *preRestart()*
    - *postRestart()*
    - *postStop()*, executed after stopping the actor.

AkkaAkka – Messages (cont.) (7/18)

- Send messages
  - *tell*() sends a message asynchronously and returns immediately.
  - *ask*() sends a message asynchronously and returns a Future representing a possible reply.
  - *tell*() is preferred due to performance concern.
  - For both methods, actors have the option of passing along their own references (*ActorRef*).

Akka – Messages (cont.) (8/18)

- *tell()* (fire-and-forget)
  - No blocking waiting for a message.
  - Giving the best concurrency and scalability characteristics.
  - The sender reference can be retrieved by the receiving actor via its *getSender()* method while processing the message.

```
// The second argument to tell() can be null.
target.tell(message, getSelf());
```

Akka – Messages (cont.) (9/18)

- *ask()* (send-and-receive-Future)
  - Involving actors as well as futures.
  - The receiving actor must reply with
    - *getSender().tell(reply, getSelf())* in order to complete the returned *Future* with a value, or
    - send a *Failure* message to the sender to complete the *Future* with an exception.
  - If the actor does not complete the future, it will expire after the timeout period, specified as parameter to the *ask()* method.

Akka – Messages (cont.) (10/18)

```
import static akka.pattern.Patterns.ask;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.dispatch.Futures;
import akka.util.Timeout;
…
// Sender
Timeout timeout = new Timeout(Duration.create(5, "seconds"));
Future<Object> future = ask(actor, msg, timeout);
String result = (String) Await.result(future, timeout.duration());
…
// Receiver
try {
        String result = operation();
        getSender().tell(result, getSelf());
} catch (Exception e) {
        getSender().tell(new akka.actor.Status.Failure(e), getSelf());
        throw e;
}
```

Akka – Messages (cont.) (11/18)

- Forward messages
  - A message can be forward from one actor to another.
  - The original sender reference is maintained.
  - The context of the “mediator” actor should be passed along as well.
  - This can be useful when writing actors that work as routers, load-balancers, etc.

```
target.forward(result, getContext());
```

Akka – Messages (cont.) (12/18)

- Receive messages
  - When an actor receives a message, it is passed into the *onReceive()* method, which is an abstract method on the *UntypedActor* base class that needs to be defined.
  - The *UntypedActorContext setReceiveTimeout()* defines the inactivity timeout after which the sending of a *ReceiveTimeout* message is triggered.

Akka – Messages (cont.) (13/18)

```
import akka.actor.ActorRef;
import akka.actor.ReceiveTimeout;
import akka.actor.UntypedActor;
import scala.concurrent.duration.Duration;

public class MyReceiveTimeoutUntypedActor extends UntypedActor {
      public MyReceiveTimeoutUntypedActor() {
      getContext().setReceiveTimeout(Duration.create("30 seconds"));
}
public void onReceive(Object message) {
      if (message.equals("Hello")) {
            …
      } else if (message instanceof ReceiveTimeout) {
            …
      } else {
            unhandled(message);
}}}
```

Akka – Messages (cont.) (14/18)

- Reply to messages
  - getSender() can be used to get a handle of the sender for replying to a message.
  - The handle returned by getSender() can also be stored for later using.
  - If the sender is not provided, the reply is sent to a "dead-letter" actor.

```
public void onReceive(Object message) {
        Object result = caculateResult().
        …
        getSender().tell(result, getSelf());
}
```

Akka – Hotswapping (15/18)

- The getContext().become() method can be called with an actor to hotswap the actor's message loop.

```
import akka.japi.Procedure;

public class HotSwapActor extends UntypedActor {
  Procedure<Object> angry = new Procedure<Object>() {
    public void apply(Object message) {
      if (message.equals("bar")) {
        getSender().tell("I am already angry?", getSelf());
      } else if (message.equals("foo")) {
        getContext().become(happy);
  }};

  Procedure<Object> happy = new Procedure<Object>() {
    public void apply(Object message) {
      if (message.equals("bar")) {
        getSender().tell("I am already happy :-)", getSelf());
      } else if (message.equals("foo")) {
        getContext().become(angry);
  }};
```

```
  public void onReceive(Object message) {
    if (message.equals("bar")) {
      getContext().become(angry);
    } else if (message.equals("foo")) {
      getContext().become(happy);
    } else {
      unhandled(message);
  }}
}
```

Akka – Stopping Actors (16/18)

- Actors are stopped by invoking the stop method of a *ActorRefFactory*.
  - *ActorContext*, stopping child actors;
  - *ActorSystem*, stopping top-level actors.
- Termination of an actor:
  - suspends its mailbox processing;
  - sends a stop command to all its children;
  - keeps processing the internal termination notifications from its children until the last one is gone;
  - terminating itself.
- Upon *ActorSystem.shutdown()*, the whole system will be terminated in the fore-mentioned manner.

Akka - Logging (17/18)

- *LoggingAdapter* has different methods for handling *error*, *warning*, *info*, and *debug* information, respectively.
- The logging options can be configured in the application configuration file (%AKKA_HOME%\config\application.conf).

```
import akka.event.Logging;
import akka.event.LoggingAdapter;

class MyActor extends UntypedActor {
   LoggingAdapter log = Logging.getLogger(getContext().system(), this);
   public void preStart() {
      log.debug("Starting");
   }
   public void preRestart(Throwable reason, Option<Object> message) {
      log.error(reason, "Restarting due to [{}] when processing [{}]",
         reason.getMessage(), message.isDefined() ? message.get() : "");
   }
}
```

Akka – Sample Programs (18/18)

- HelloWorld.
- Pi (The “first tutorial” of Akka).
- How to run Akka programs from command-line.

THANKS!

Q & A