Akka for Java Developers

Unit 1. What is Akka?

Akka for Java Developers

What is Akka?

Definition from http://akka.io:

– Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on JVM.

Akka is written in Scala but has API for Java.

Akka's source is open.

Toolkit and runtime

Akka gives you both API and small-footprint runtime.

Footprint is less that 10MB.

Highly concurrent applications

- Modern applications handle multiple requests in parallel. Traditional parallel computing has many issues.
- Akka implements Actor Model for concurrency.
- It abandons shared memory model therefore removes the need for blocking.
- Unfortunately in Java we can abandon shared memory only by convention. There is no language support like in Erlang.

Fault-tolerant applications

- Modern applications are deployed in complex environments and communicating with many data sources and external systems.
- Too many points of failure. It is hard to make your application stable enough.
- Akka gives a well-proved mechanism of supervision that allows you to build self-healing systems.

“Let it crash” is the main idea behind this.

Event-driven applications

Akka gives you a simple asynchronous model of communication which resembles how things and people interact in real world.

Distributed applications

- Akka effectively implements Location Transparency for each building block.
- Distribution and scaling is very easy and transparent.
- Id does not require any major changes in code for Akka application to become distributed.

Applications for JVM

Akka was created in Scala but has Java API.

This makes it useable on any JVM language.

Unit 2. Foundation

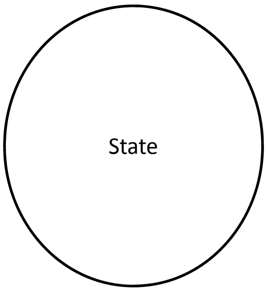
Akka for Java Developers

Problem of Concurrency

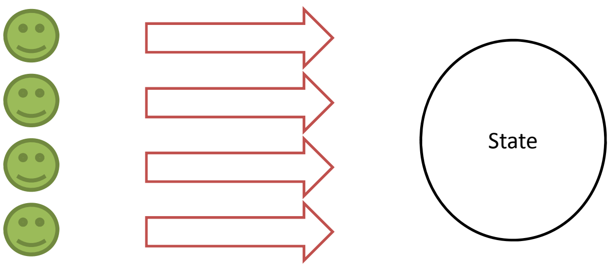
Main problem:

Competition over the mutable state

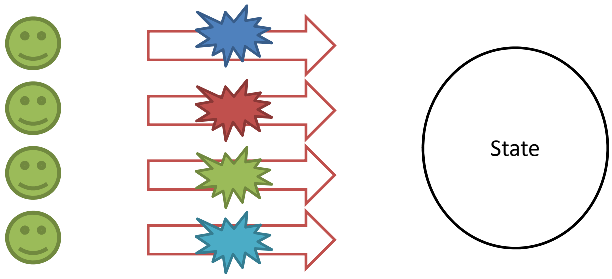
This is your mutable shared state



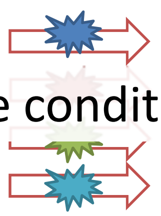
Multiple tasks would like to access it



They do this in parallel

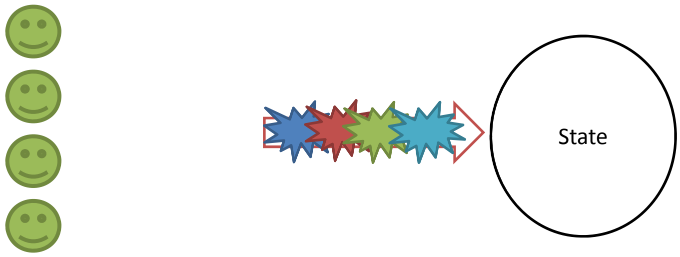


Problem 1



Race conditions

To preserve consistency these updates are applied sequentially

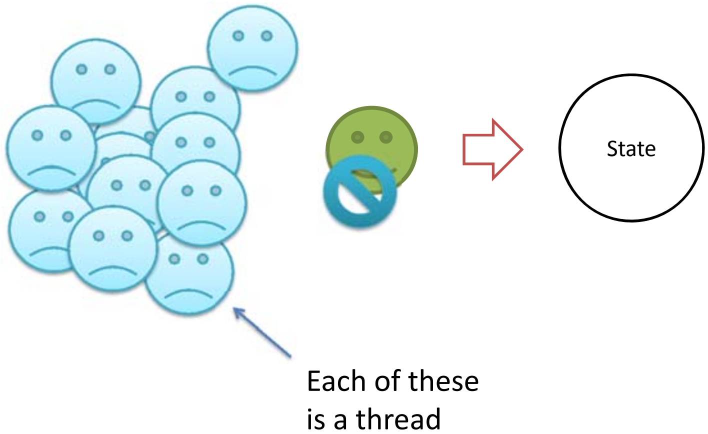


By placing everyone in a queue

First one to get a lock wins

Other tasks are waiting in a queue

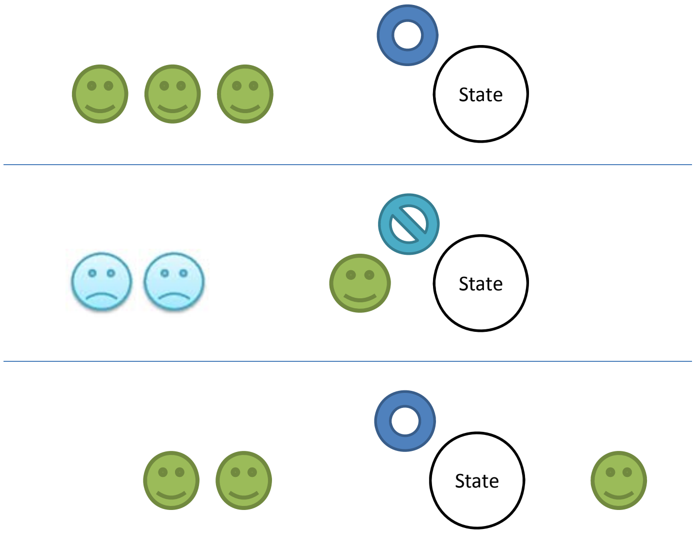
Which is not a queue



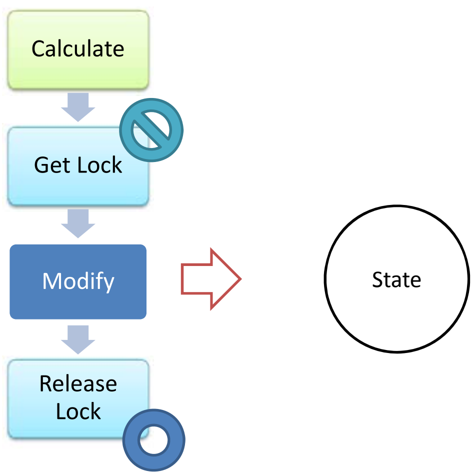
Problem 2

Thread starvation

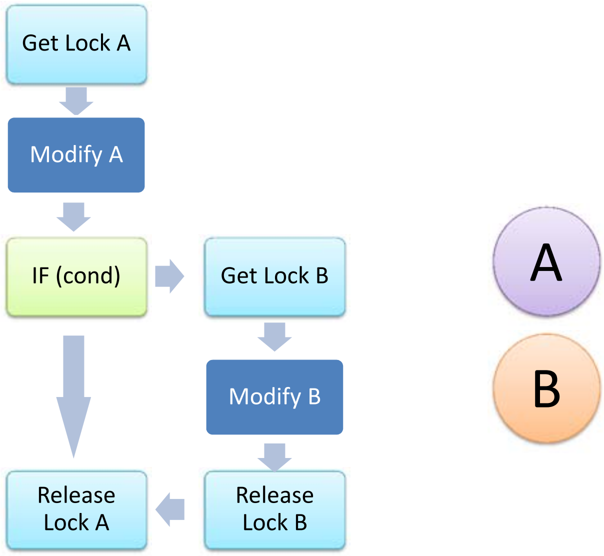
Locks?



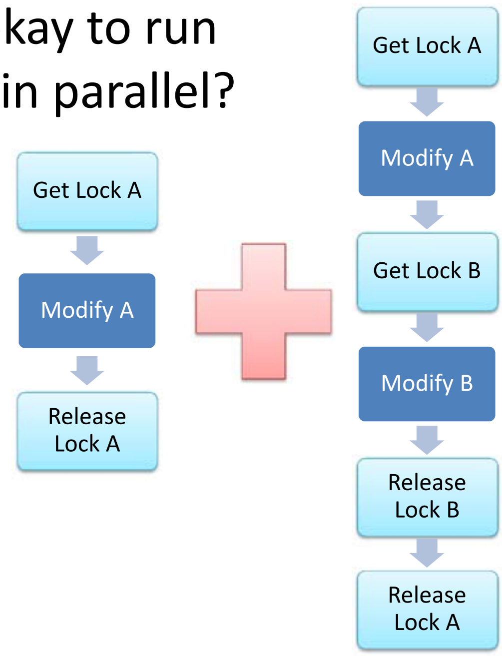
Yes, for very simple cases



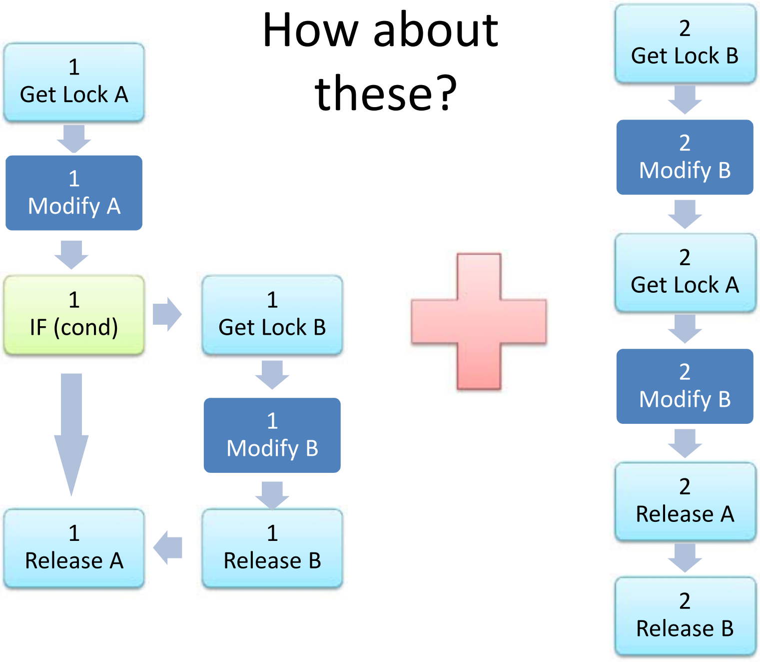
But bad for multiple resources

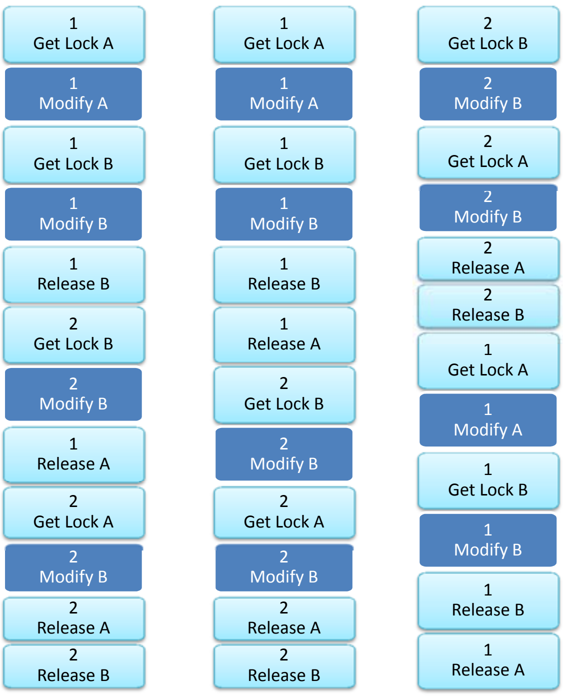


Is it okay to run these in parallel?

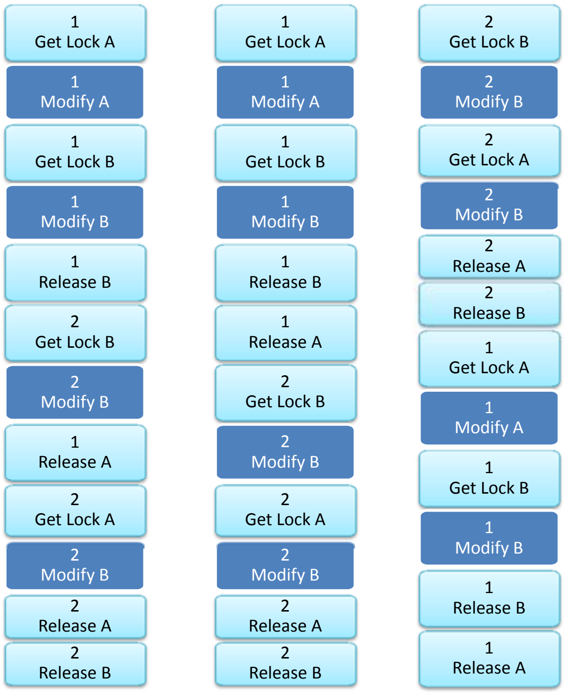


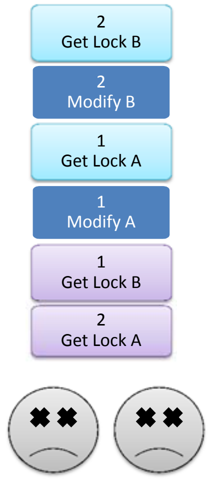
How about these?





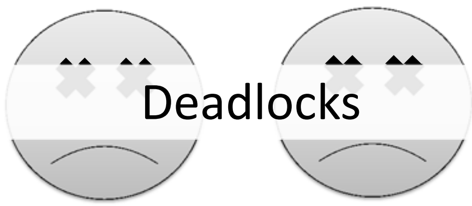
Some of possible scenarios are okay





And some are not

Problem 3



Deadlocks

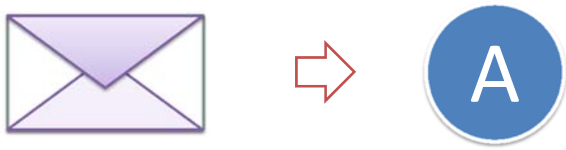
Actor Model

- Everything is an actor
- No shared state
- Message passing

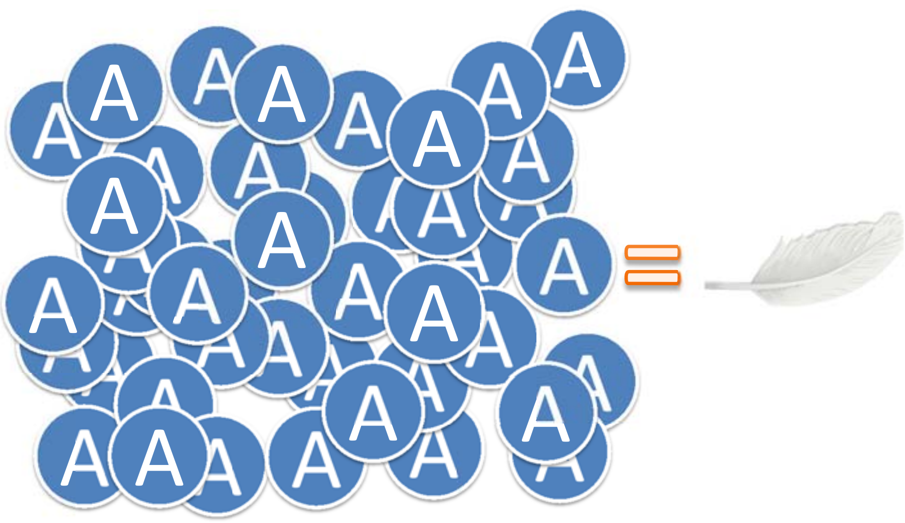
Actor =

mailbox + behavior + state

Actors receive messages



Actors are lightweight



Actors

Actor is the universal primitive
of concurrent computation.

It collects messages from mailbox
and reacts to them.

Local Decisions

It can make local decisions:

- modify private state
- create new actors
- send messages to other actors
- determine how to respond to the next message received.

Modify private state

Create new actor

Send messages to other actors

Determine how to respond to the next message received

Local Effects

- All effects that are produced by actors are local. Actor only can affect on things about which it is aware.

- There is also no simultaneous change in multiple locations. No mutable shared state is permitted.

Function calling vs Message passing

Function calling:

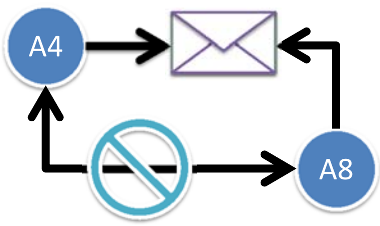
- It requires the reference and the interface.
- This creates unnecessary dependencies and introduces many limits to possible designs.
- It also supports only synchronous communication.

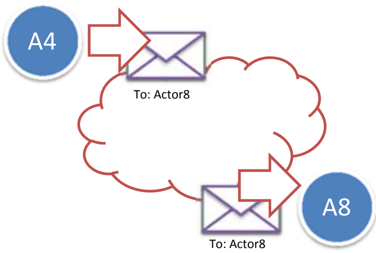
Message passing :

- It requires only the address to which the message is sent.
- There is no need to acquire a reference to the recipient.
- There is no need to interact with recipient directly at all.
- This allows building systems with variable topology.

Message passing

- The only proper way of communication within actor-based applications is through passing messages.
- No mutable messages are permitted.

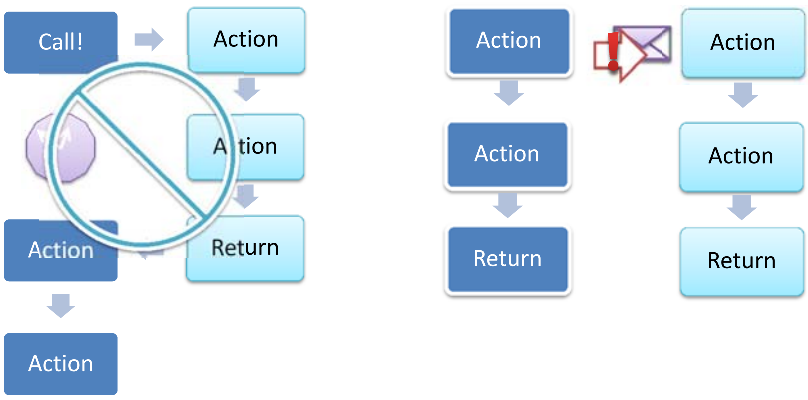




Non-blocking behavior

- Locks and synchronized blocks simulate “time freezes”.
- Actor model allows us to embrace the time.
- In general it is not required for actors to use only non-blocking behavior but in heavy-loaded systems blocking causes issues with performance.
- There almost always is a way to solve the problem in a non-blocking style.

Non-blocking behavior



Akka Actor Example

```
import akka.actor.UntypedActor;

public class SimpleActor extends UntypedActor {

    @Override
    public void onReceive(Object message)
       throws Exception {
         if("Hello!".equals(message)){
             System.out.println("Oh, hi there!");
         }
    }

}
```

Akka Actor Application example

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

public class AkkaExample {
   public static void main(String[] args) throws Exception {

        ActorSystem system = ActorSystem.create("Example");

        Props actorProps = Props.create(SimpleActor.class);
        ActorRef actor = system.actorOf(actorProps);

        actor.tell("Hello!", null);

        Thread.sleep(100);

        system.shutdown();

    }
}
```

ActorSystem

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

- ActorSystem represents the environment in which actors are running. Treat it like a logical application instance.
- Each actor system creates a set of threads and execution contexts on which your application will be executed.
- It is possible to run multiple actor systems but you should know that each one is heavyweight enough so be careful.

Props

```
system.actorOf( Props.create( SimpleActor.class ) )
```

- Props class is a container which describes how the Actor should be created.

- In general it should contain at least the class of the actor, and may also contain constructor arguments and some other information.

ActorRef

```
ActorRef actor = system.actorOf(actorProps);
actor.tell("Hello!", null);
```

- Actor reference is a handle to the actor instance.

- You only can interact with actor through the ActorRef.

- It is **immutable**, **serializable** and **network-aware** so it is safe to pass it to other actors.

UntypedActor

```
public class SimpleActor extends UntypedActor {

public void onReceive(Object message)throws Exception {
                          …
```

- This is a base class for creating actors in Java.
- You have to implement onReceive method.

Starting an actor revisited

```
Props actorProps = Props.create(SimpleActor.class);
ActorRef actor = system.actorOf(actorProps);
```

- Actor can be named.

```
ActorRef actor = system.actorOf(actorProps, "Name");
```

- Name should not start with ‘$’ sign and must be unique within one level of hierarchy.

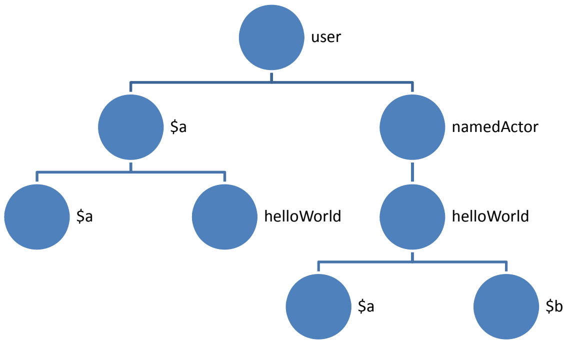
More Tools for Actors

UntypedActorContext	• This is an object which gives you more controls for actor's lifecycle and access to various ActorSystem facilities. • This object has its own actorOf methods which gives you an opportunity to launch new actors from within your actor. • You can get your actor's context object by calling getContext().
ActorContext	• Is a more general form of ActorContext. • It is the same object but has only the essential Scala API. • You can get it by calling context().

ActorSystem structure

- Actors form hierarchies.
- Your actor is always created as child of another actor.
- When starting actor system creates the root of hierarchy – guardian actor (named "user").

Actor Hierarchies



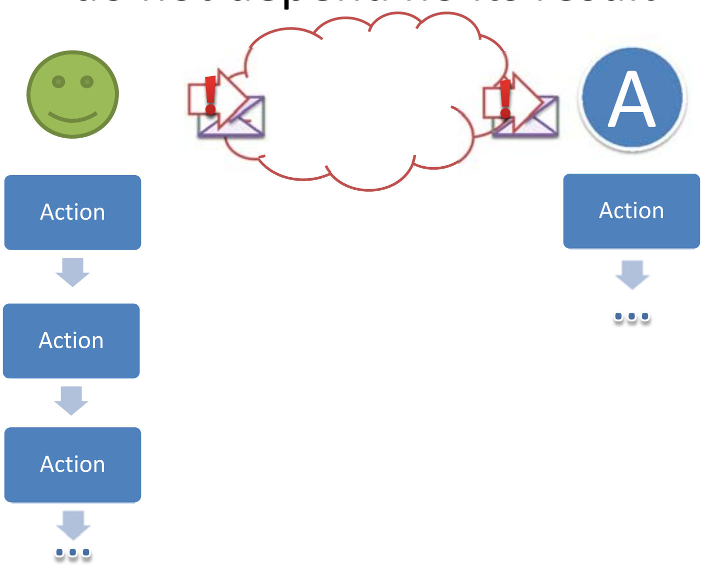
Actor Hierarchies

- **ActorSystem.actorOf** creates your actors as children of the root actor.
- Your actors can create their own children by calling **ActorContext.actorOf** methods.
- Actor names which you optionally supply to **actorOf** method must be unique only among sibling actors.

Unit 3. Futures

Akka for Java Developers

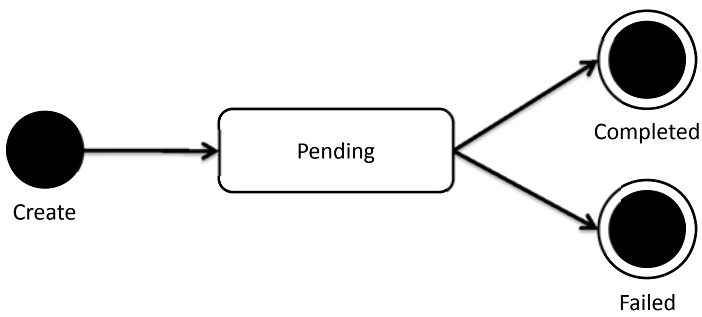
It is easy to launch parallel task if you do not depend no its result



What is a Future?

- Future is a simple way to efficiently perform simultaneous operations.
- It is a container which contents will normally become available sometime after creation.
- Future may contain a result of computation or an exception raised during computation.

Future Lifecycle



How Future Works

Future is a collection of 0 or 1 elements.

Future can be created:

– **Empty** → It is pending for a value to appear
– **With value** → future is complete upon creation
– **With exception** → future is failed upon creation

Futures are essentially non-blocking.

Launching a parallel task with Akka

```
import akka.actor.ActorSystem;
import scala.concurrent.Future;
import akka.dispatch.Futures;
import java.util.concurrent.Callable;

 final ActorSystem system = ActorSystem.create();

 Future<String> pending = Futures.future(
               new Callable<String>() {
                   @Override
                   public String call() throws Exception {
                       return "Hello from the Future!";
                   }
               }, system.dispatcher());
```

Futures API basics

- scala.concurrent.Future is a Scala interface.
  - You can get objects with this interface from many places. From client perspective they all behave the same but they may work differently inside.
  - There is also an “old” interface in scala.parallel package. Try not to get confused.

- akka.dispatch.Futures is a Akka’s Java-friendly interface for creating futures.

Creating Future

```
Future<String> pending =
            Futures.future(callable, dispatcher);


Future<String> successful =
            Futures.successful("hello");


Future<String> failed =
            Futures.failed(exception);
```

Consuming Future Values

Future values can be consumed **asynchronously** or **synchronously**.

- Though asynchronous consumption is encouraged, there are cases when it is not possible to use it.
- Synchronous consumption may cause a current thread to block.

Consuming asynchronously

Asynchronous consumption is similar to consuming events. It requires an event handler.

Event subscription	Result is delivered to
`Future<T>.onSuccess`	`OnSuccess<T>.onSuccess(T value)`
`Future<T>.onFailure`	`OnFailure.onFailure(Throwable e)`
`Future<T>.onComplete`	`OnComplete<T>.onComplete(Throwable e, T value)`

You can attach multiple event handlers to the Future

Consuming asynchronously

```
pending.onSuccess(new OnSuccess<String>() {
            @Override
            public void onSuccess(String result)
                    throws Throwable {

                System.out.println("Yay! I got " + result);

            }
        }, system.dispatcher());
```

Consuming synchronously

- **Await** class helps in waiting for Future results.
- **It blocks.** But it never blocks forever.
- With this model deadlocks are not possible.

Consuming synchronously

```
import scala.concurrent.Await;
import scala.concurrent.duration.Duration;
import java.util.concurrent.TimeUnit;

Duration duration = Duration.create(5, TimeUnit.SECONDS);

String value = Await.result(pending, duration);

//Other ways to create durations
Duration duration2 = Duration.create(5, "seconds");
Duration duration3 = Duration.create("5 seconds");
```

Programming the Future

- If the result of parallel task must be transformed after being received, use **mapping**.
- If there are many parallel operations but they are all needed to calculate the final result, use **sequence**, **fold** or **reduce**.
- You can chain these operations.

Mapping the result

```
Future<String> pending = Futures.future(callable, dispatcher);

Future<Integer> mapped = pending.map(

                new Mapper<String, Integer>() {
                    @Override
                    public Integer apply(String value) {

                        return value.length();

                    }
                }, system.dispatcher());
```

Sequence and Reduce Futures

```
List<Future<String>> listOfFutureStrings= ...

Future<Iterable<String>> futureListOfStrings=
        Futures.sequence(
                listOfFutureStrings,
                system.dispatcher());


Future<Integer> futureInteger = Futures.reduce(
        listOfFutureStrings,
        new akka.japi.Function2<Integer, String, Integer>() {
             @Override
             public Integer apply(Integer arg1, String arg2) {
                 return arg1 + arg2.length();
             }
         }, system.dispatcher());
```

Unit 4. Creating Actors

Akka for Java Developers

Akka Actor Example

```
import akka.actor.UntypedActor;

public class SimpleActor extends UntypedActor {

    @Override
    public void onReceive(Object message)
       throws Exception {

        if("Hello!".equals(message)){
            System.out.println("Oh, hi there!");
        }

    }
}
```

Starting Actors

Actors are started at some point of hierarchy:

- `ActorSystem.actorOf(Props props)`
  Will start a root actor.

- `ActorContext.actorOf(Props props)`
  Will start a child actor.

Props tell how actor is created

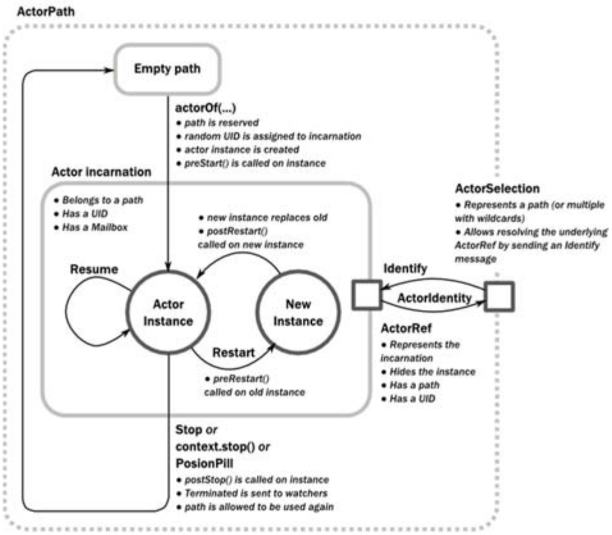
Props are describing how to create an actor:

```
Props props = Props.create(MyUntypedActor.class)

Props props = Props.create(
            MyUntypedActor.class, arg1, arg2)

Props props = Props.create(
      new Creator<MyActor>() {
            @Override public MyActor create() {
                  return new MyActor("...");
            }
      });
```

Actor Lifecycle



Lifecycle methods

You can override these methods:

Method	Purpose
`preStart()`	`To perform operations after constructor fired and exactly before actor can receive its first message.`
`postStop()`	`Is called after the actor stops receiving messages and before the actor is detached from its path.`
`preRestart()`	`Is called on failed instance before it becomes replaced.`
`postRestart()`	`Is called on new instance in the same moment as preStart().`

onReceive method

- Messages from actor’s mailbox are delivered to **onReceive** method one by one.
- Actor may react on messages.
- It may detect message type or identity by using instanceof or equals().
- There are few messages that are processed by actor itself and never reach onReceive.

Special messages

- Sending **PoisonPill** will result in stopping the recipient actor.
- Upon receiving an **Identify** message actor responds with an **ActorIdentity** message. It can be used to find ActorRefs in system.

Actor’s state

- Actor’s **onReceive** method never runs in parallel. It is very safe to change actor’s private state without locks or synchronize blocks.

- You can create an initial state in the constructor or in **preStart()** method.

- You must never share the state.

Communicating to Actors

There is only one way to communicate with an actor: ActorRef.tell(message, sender)

sender parameter allows you to attach a return address to your message so recipient can send an answer. You may pass **null** value if you are not expecting any reply.

Communicating to Actors

Ask is a convenient pattern for cases when you are expecting a reply from an actor.

```
Future<Object> future = Patterns.ask(actor, msg, timeout);
```

Timeout indicates for how long this Future will wait for a reply before failing. You never wait forever.

How actor replies

Actor replies by telling to a **ActorRef** from **getSender()**.

This **ActorRef** points to a dead letters mailbox if there is no sender information.

```
public void onReceive(Object message) {
    if("Hello!".equals(message)){
       getSender().tell("Oh, hi there!“, getSelf());
    }
}
```

Dead Letters Mailbox

ActorSystem has a special mailbox to which all undeliverable messages are redirected.

ActorSystem.deadLetters() gives you a reference to this mailbox.

Stopping Actors

Three ways:

- `ActorSystem.stop(ActorRef ref)`
- `ActorContext.stop(ActorRef ref)`
- `ActorRef.tell(PoisonPill.getInstance(), null);`

Stopping Actors

Stop procedure is performed asynchronously:

- Current message processing completes.
- Actor suspends mailbox.
- All next messages are gone to Dead Letters.
- Actor sends stop commands to its children.
- It awaits for all children to stop.
- It announces its termination to supervisors.
- It finally stops.

Stop() vs PoisonPill

- stop() gives the actor a chance to process current message and then shuts it down
- PoisonPill is placed to the end of the mailbox queue. Actor will shut down after it reaches this message.

Anyway, it takes time to shut down an Actor.

Unit 5. Wiring Actors

Akka for Java Developers

How actors discover each other

Actor only can communicate with another actor if it knows something about it:

- ActorRef
- Path

References and Paths

- **Actor Path** represent an address in some ActorSystem which may or may not be inhabited by an actor at some point of time.

- **Actor Reference** points to a specific incarnation of an actor which temporarily occupies some Path.
  – When actor stops the reference becomes invalid and messages will not be delivered to the new incarnation on the same Path.

Obtaining ActorRef

ActorRef may be obtained:

- Via constructor parameter
  - Pass it through the Props instance
- By receiving in message
  - ActorRefs are immutable and serializable and can be passed freely or transferred over a network
- By creating a child actor
  - You get ActorRef and then can obtain a list of children from getContext().children()

Discover Actor by its Path

ActorSystem and ActorContext can convert Path to ActorSelection.

- ActorSelection allows communicating with an actor behind a Path.
- It does not tied to any actor incarnation.
- The actor corresponding to the selection is looked up when delivering each message.

Invalid destinations

Messages to invalid destinations such as:

- an ActorRef to a dead Actor
- an ActorSelection to an unoccupied path

are delivered to Dead Letters Mailbox.

Unit 6. Messages and State

Akka for Java Developers

Java is Java

- There is no language support for non-shareable state.
- There is no language support for enforcing state immutability
- There is a language support for sharing and breaking every piece of state possible.

Immutable messages

Design messages to be immutable.

- Use final fields
- Hide set-methods
- Use builders

Why immutable messages?

- Mutable messages are a form of shared state.
- Without locks there is a chance of receiving message in inconsistent state.
- Also you never know how this message will travel:
  - It may go through the network
  - It may go through variable set of intermediaries

Immutable state

- If your actor has a state it may need to expose it through sending it in messages
- Sending a state in messages directly is an **act of sharing**.
- If you want to send your state in messages, it **should be immutable**.
- If you cannot design your state immutable, then you **must send copies** of your state.

State and Parallel tasks

If you are using Futures for processing something in actors, then you must be careful.

- Parallel tasks often are executed, well, in parallel.
- Futures launched during processing one message may complete after the actor moved on to a different messages.

Watch your captures!

Beware!

```
public void onReceive(Object message){
    if (message instanceof SomeMessage) {
        //Do something
        Futures.future(new Callable<Object>() {
            public SomeResult call() {
                  //do something in parallel
                  getSender().tell(something, getSelf());
                  return null;
            }
        }, context().system().dispatcher());
     }
 }
```

Save all that might change

```
public void onReceive(Object message){
    if (message instanceof SomeMessage) {
        //Do something
        final ActorRef sender=getSender();
        Futures.future(new Callable<Object>() {
            public SomeResult call() {
                        //do something in parallel
                  sender.tell(something, getSelf());
                        return null;
                 }
          }, context().system().dispatcher());
     }
 }
```

Just don't.

```
int counter=0;

public void onReceive(Object message){
    if (message instanceof SomeMessage) {
        //Do something
        Futures.future(new Callable<Object>() {
            public SomeResult call() {
              //do something in parallel
              counter++;
              return null;
                }
          }, context().system().dispatcher());
     }
 }
```

Possible solution

```
int counter=0;

public void onReceive(Object message) throws Exception {
    if (message instanceof SomeMessage) {
       Futures.future(new Callable<Object>() {
         public SomeResult call() {
              //do something in parallel
            getSelf().tell(new IncrementCounter(),null);
            return null;
          }
       }, context().system().dispatcher());
     }
     if(message instanceof IncrementCounter){
        counter++;
     }
 }
```