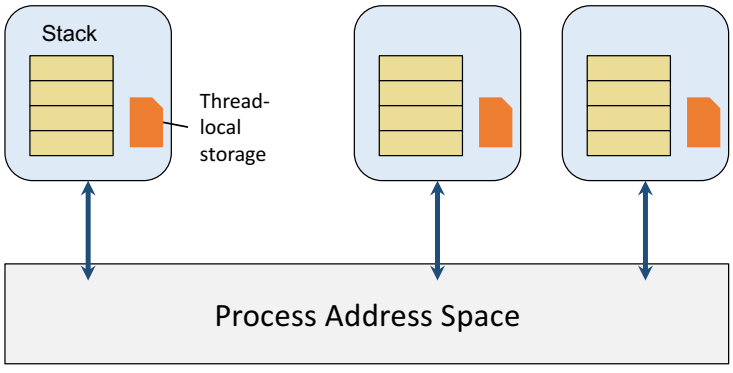
# Introducing Actors with Akka

## Traditional Threading Model

- Multiple threads sharing a single address space



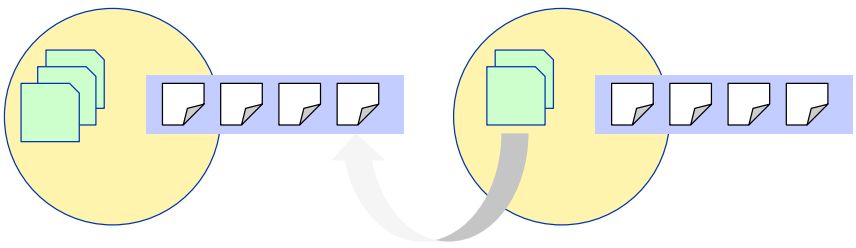
# Issues With The Traditional Model

- Threads no longer viewed as lightweight
  - stack size 512K to 2MB
  - limits number of threads that can be created
- Protection of shared mutable state is hard
  - locking very difficult to get right
  - based on notion of blocking and context switching
  - many problems are timing related
- Much boiler plate needed
  - low level constructs need management



# Actors

- An alternative approach to concurrency and distribution
- Actor is a small, self-contained processing unit
  - contains state, behaviour and mailbox
- Actors communicate by sending messages
  - asynchronously



# Actors

- Should not share any mutable state
  - can have mutable state internally but nothing exposed
- Should communicate using immutable messages
- Should communicate asynchronously
- Behave reactively
  - Only perform calculations in response to messages
- Can exist within one process or across processes
  - also across machines
- Should provide a safe model for handling failures



# A Simple Example

- Two Actors implementing "TickTock" example
- Message types
  - usually defined as Algebraic Data Type

```
import akka.actor._

sealed abstract class Message

case class StartTicking ( tocker: ActorRef ) extends Message
case object TickMessage extends Message
case object TockMessage extends Message
```

# A Simple Example

- The Actors

```
import akka.actor._
class TickActor extends Actor with ActorLogging {
  log.info("Creating Tick Actor")
  override def receive = {
    case StartTicking(tocker) => log.info("Starting... Tick");
                                 tocker ! TockMessage
    case TickMessage => log.info("Tick");
                        Thread.sleep(500); sender ! TockMessage
  }
}
class TockActor extends Actor with ActorLogging {
  log.info("Creating Tock Actor")
  override def receive = {
    case TockMessage => log.info("Tock");
                        Thread.sleep(500); sender ! TickMessage
  }
}
```



# A Simple Example

- The driver application

```
object ActorApp extends App {

  val ttSystem = ActorSystem("TickTock")

  val ticker = ttSystem.actorOf( Props[TickActor] )
  val tocker = ttSystem.actorOf( Props[TockActor] )

  ticker ! StartTicking(tocker)

  Thread.sleep(5000)
  ttSystem.shutdown
}
```

Create and initialise the actors

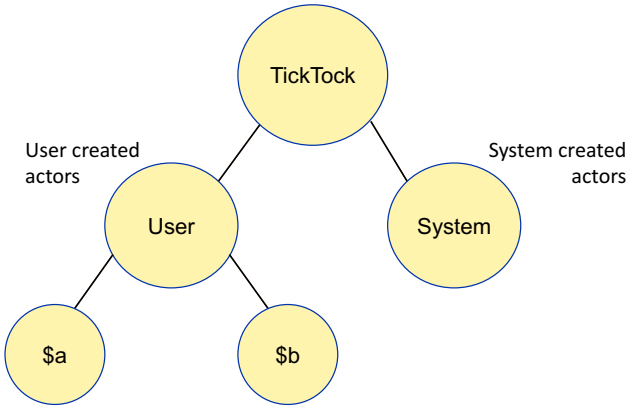
Send start message

Wait 5 seconds then shut down

```
[INFO] [06/25/2013 18:18:48.893] … [akka://TickTock/user/$a] Creating Tick Actor
[INFO] [06/25/2013 18:18:48.897] … [akka://TickTock/user/$b] Creating Tock Actor
[INFO] [06/25/2013 18:18:48.898] … [akka://TickTock/user/$a] Starting… Tick
[INFO] [06/25/2013 18:18:48.898] … [akka://TickTock/user/$b] Tock
[INFO] [06/25/2013 18:18:49.397] … [akka://TickTock/user/$a] Tick
…
```

# Actor Application Structure and Naming

- Actors exist in a hierarchy
  - Important for error handling and recovery
- Pathname identifies individual actors






# Request/Response Operation

- Actor communication encouraged to be asynchronous
  - "fire and forget"
  - no implicit reply
- Request/response communications possible
  - use `ask` method rather than `tell` method
  - `?` rather than `!`
- Leverages Futures for handling replies

# Request/Response Example

- Actor generates and sends a random `Int` value between 0 and 100

```
import akka.actor._

case object GetRandomInt

class RandomNumActor extends Actor with ActorLogging {

  log.info("Creating the Random Number Generator Actor")
  val rGen = new scala.util.Random

  override def receive = {
    case GetRandomInt => sender ! Math.abs(rGen.nextInt) % 100
  }
}
```

Message



# Request/Response Example

- Send request and handle response as `Future[Int]`

```
import akka.actor._
import akka.pattern.ask
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object RNActorApp extends App {

  val rnSystem = ActorSystem("RandomNumbers")
  val rand = rnSystem.actorOf(Props[RandomNumActor], "RandomNumGen")

  implicit val timeout = Timeout(1 seconds)
  val rNumFuture = (rand ? GetRandomInt).mapTo[Int]

  rNumFuture onSuccess {
      case i => println(s"=> $i")
  }
  rnSystem.shutdown
}
```

# Request/Response Example

- Demonstrating async nature of calls

```
// Setup as before …
  1 to 5 foreach { n =>
         (rand ? GetRandomInt).mapTo[Int].onSuccess {
             case i => println(s"$n => $i")
           }
  }
…
```

```
[INFO] [06/25/2013 19:18:49.923] … [akka://RandomNumbers/user/RandomNumGen]
                                    Creating the Random Number Generator Actor
2 => 0
5 => 78
1 => 38
3 => 26
4 => 58
```



# Request/Response Example

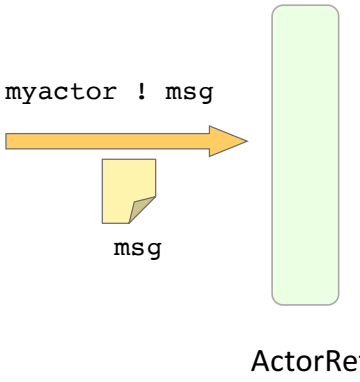
- Blocking on each request until response arrives

```
// Setup as before …
  1 to 5 foreach { n =>
      val rn: Int = Await.result(
                        (rand ? GetRandomInt).mapTo[Int], 1 second)
    println(s"$n => $rn")
  }
…
```

```
[INFO] [06/25/2013 19:22:45.109] … [akka://RandomNumbers/user/RandomNumGen]
                                    Creating the Random Number Generator Actor
1 => 11
2 => 5
3 => 51
4 => 86
5 => 22
```

# Inside an Actor

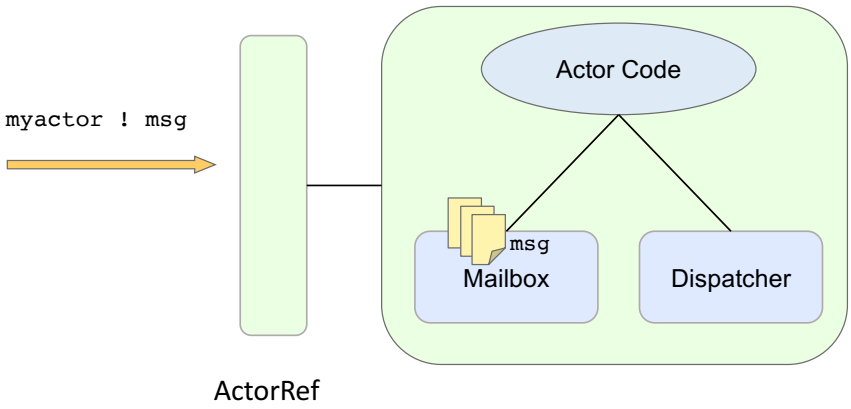
- Outside world communicates with ActorRef
  - hides specific details of actor implementation
  - also hides location





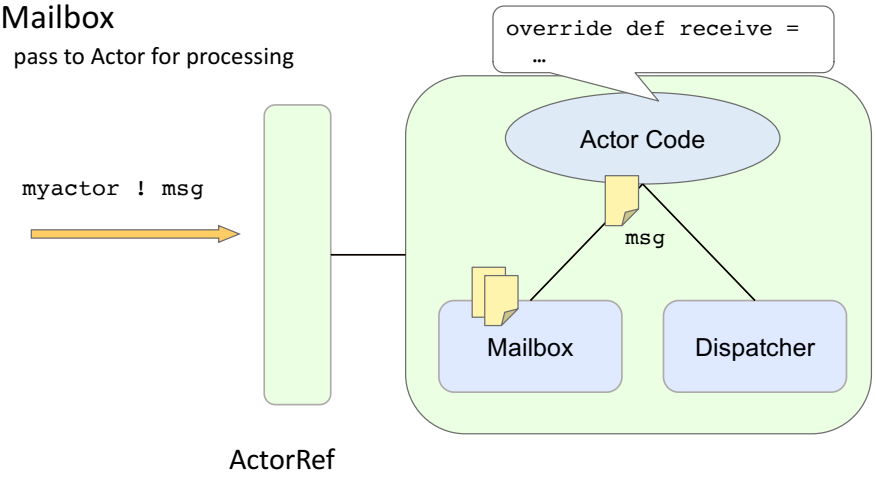

# Inside an Actor

- Messages added to queue (Mailbox)
  - FIFO ordering



# Inside an Actor

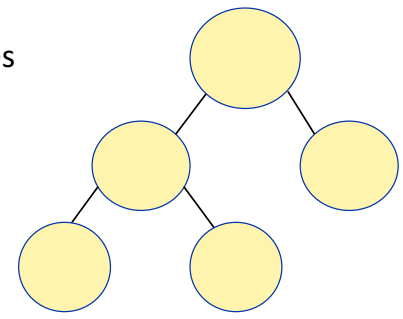
- Dispatcher uses thread from thread pool to remove message from Mailbox
  - pass to Actor for processing






# The Actor System

- Collection of related actors
  - arranged as hierarchy
- Provides context for shared resources
  - base of actor naming
  - configuration data
  - factory for "top level" actors
  - default execution context
  - scheduling service
  - event stream
- Multiple Actor Systems allowed per application (JVM)
  - or per classloader

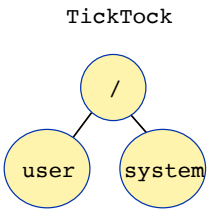


# The Actor System

```
object ActorApp extends App {

 val ttSystem = ActorSystem("TickTock")

 …
}
```




- Set up skeleton actor hierarchy
- `user` subtree for user managed actors
- `system` subtree for system managed actors
- Read and parse configuration



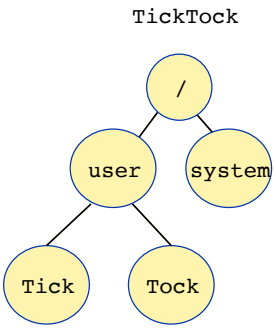
# The Actor System

```
object ActorApp extends App {

 val ttSystem = ActorSystem("TickTock")

 val ticker = ttSystem.actorOf(
                Props[TickActor], "Tick")
 val tocker = ttSystem.actorOf(
                Props[TockActor], "Tock")

 …
}
```




- Top level actors created relative to Actor System

# Creating an Actor

- Actors never created directly
  - use factory method
  - actor constructed "behind" `ActorRef`
- Can create top level actor
  - parent is `user` actor

```
…
val ttSystem = ActorSystem("TickTock")
val ticker = ttSystem.actorOf( Props[TickActor], name = "Tick")
…
```

- Or subordinate actor
  - parent is creating actor

```
public class MyActor extends Actor {
 val worker = context.actorOf( Props[WorkerActor], "Labourer")
…
```



# Creating an Actor

- `Props` type specifies information for the actor factory
  - creation options
  - customisation of Dispatcher, Deployment, Routing
- Simple usage when Actor class has no-arg constructor

```
class TickActor extends Actor with ActorLogging {
  …
}
```

```
// Default use assumes no-arg constructor for actor
val ticker = ttSystem.actorOf(Props[TickActor], "Ticker")
```

# Creating an Actor

- `Props` allows constructor arguments to be passed
  - different mechanisms
  - use `apply()` method

```
class TickActor ( msg: String ) extends Actor with ActorLogging
{
  …
}
```

```
// Need to pass argument to constructor

val tickActor = context.actorOf(Props(classOf[TockActor],
                                      "tick"),
                                "ticker")
```



# Creating an Actor

- Alternative approach based on companion object for actor
  - recommended approach as most flexible

```
class TickActor ( msg: String ) extends Actor with ActorLogging {
  …
}

object TickActor {
  def props( m: String ) = Props( classOf[TickActor], m )
}
```

```
// Create using method from companion object

val tickActor = context.actorOf( TickActor.props("tick"),
                                 "ticker")
```

# Accessing an Existing Actor

- Obtaining `ActorRef` to actor already running
  - rather than creating the actor
- Use `actorSelection` instead of `actorOf`
  - refer to actor using its pathname

```
…
val ttSystem = ActorSystem("TickTock")
val ticker = ttSystem.actorOf( Props[TickActor], name = "Tick")
…
val ticker2 = ttSystem.actorSelection("akka://TickTock/user/Tick")
…
ticker ! TickMessage
ticker2 ! TickMessage
…
```

Both messages sent to same actor



# Configuration

- Sophisticated configuration possible
  - using Typesafe configuration library
- Configuration specified in external file
  - default name `application.conf`
  - syntax is HOCON – superset of JSON
- Read automatically when creating ActorSystem
- Multiple sources of config possible
  - System Properties, `application.conf`, `application.json`, `application.properties`, `reference.conf`

`=` and `:` are interchangeable

```
TickTock {

  howlong = 2

  Ticker {
    message : "Ping"
  }

  Tocker {
    message : "Pong"
  }

}
```

# Configuration

- Using the configuration data

```
object ActorApp extends App {

  val ttSystem = ActorSystem("TickTock")
  val ttSystemConfig = ttSystem.settings.config

  val howLong = ttSystemConfig.getInt("TickTock.howlong")
  println(s"Running for $howLong seconds")

  val tickProps = Props( creator = { () =>
       new TickActor( ttSystemConfig.getString(
                                    "TickTock.Ticker.message"))
                                    } )

  val ticker = ttSystem.actorOf(tickProps, "Ticker")
  …
}
```



# Messages

- Messages should be typed
  - actor can "receive" any type of message
- Messages should be immutable
- Use case classes to allow payload
  - case objects if no parameters
  - Algebraic Data Types useful

```
sealed abstract class Message

case class StartTicking ( tocker: ActorRef ) extends Message
case object TickMessage extends Message
case object TockMessage extends Message
case object DoSomeWork extends Message
```

# Sending Messages

- Messages sent to `ActorRef`
- Two options:
- Fire and forget
  - `tell` or `!` method

```
tickActor ! TickMessage
```

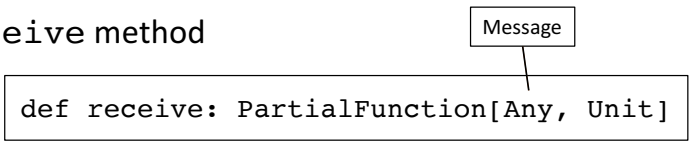
- Request/response
  - `ask` or `?` method
  - returns `Future[Any]` as placeholder for reply
  - more later

```
val result: Future[Any] = someActor ? DoSomethingForMe
```



# Handling Messages

- Core of actor functionality
  - actor only responds to messages
- `receive` method




- Unknown message type message to be published on event stream
- Messages delivered in send order
  - per sender
- Message processing guaranteed thread safe
  - as long as no *shared* mutable state is used

# Handling Messages

```
case class Tick

class Counter extends Actor {
  var counter = 0

  def receive = {
    case Tick =>
      counter += 1
      println(counter)
   case m: Any =>
      println(s"Strange message: $m")  }
  }
}
```

```
val cSystem = ActorSystem("Counter")
val c1 = cSystem.actorOf(Props[Counter])
c1 ! Tick
c1 ! Tick
c1 ! 99
```

```
1
2
Strange message: 99
```



# Handling Messages

- Receive timeout can be set

```
case class Tick

class Counter extends Actor {
  var counter = 0
  context.setReceiveTimeout(1 seconds)
  def receive = {
    case Tick =>
      counter += 1
      println(counter)
    case ReceiveTimeout =>
      println("Nobody talking to me...")
  }
}
```

```
1
Nobody talking to me...
2
```

```
val cSystem = ActorSystem("Counter")
val c1 = cSystem.actorOf(Props[Counter])
c1 ! Tick
Thread sleep 1500
c1 ! Tick
```

# Handling Messages

- `sender` method gives access to message sender
  - `ActorRef`
  - can be used for reply
- Message can include alternative `ActorRef` for reply

```
case class Tick
case class TickTo( recipient: ActorRef )

class Counter extends Actor {
  var counter = 0

  def receive = {
    case Tick =>
      counter += 1; sender ! counter
    case TickTo(replyTo: ActorRef) =>
      counter += 1; replyTo ! counter
  }
}
```



# Handling Messages

- Message may be forwarded to another actor
  - `forward` method
  - original sender information is retained
  - recipient sees original sender through `sender` method

```
case class Tick

class Counter extends Actor {
  var counter = 0

  def receive = {
    case Tick =>
      counter += 1; sender ! counter
    case TickTo(replyTo: ActorRef) =>
      counter += 1;
      replyTo ! Counter
      replyTo forward Counter
  }
}
```

receiver sees this actor as sender

receiver sees original sender as sender

# Stopping an Actor

- `stop` method on `ActorRefFactory`
  - `ActorSystem` for stopping top level actors
  - `ActorContext` for stopping child actors


```
val cSystem = ActorSystem("Counter")
val c1 = cSystem.actorOf(Props[Counter])
c1 ! Tick
...
cSystem.stop(c1)
```

- Actions:
  - complete processing of current message
  - remaining queued messages may be sent to `DeadLetters`
  - call `stop` on all child actors
  - when children all stopped, call `postStop` method
  - notify supervisor (usually parent)



# Stopping an Actor

- Alternative is to send actor `PoisonPill` message
  - handled after other messages in queue
  - effect as for `stop` method
  - now deprecated

- Use `Kill` message to kill actor
  - causes `ActorKilledException` to be thrown
  - effect dependent on supervision strategy
  - more later

# Changing an Actor's Behaviour

- `context.become()`
  - installs new receive behaviour

```
c1 ! Tick
c1 ! Tick
c1 ! Change
c1 ! Tick
```

```
1
2
Changing behaviour
1
```

```
case class Tick
case class Change
class Counter extends Actor {
  var counter = 0
  def receive = {
    case Tick =>
      counter += 1; println(counter)
    case Change =>
      println("Changing behaviour")
      context.become ( {
        case Tick =>
          counter -= 1; println(counter)
      })
    case ReceiveTimeout =>
      println("Nobody talking to me...")
  }
}
```



# Actor Lifecycle Callbacks

- Callback functions available for actor lifecycle
  - `preStart()`
  - `postStop()`
  - `preRestart()`
  - `postRestart()`

} Used with fault handling

- DeathWatch allows actor to register for another actor stopping
  - `context.watch(actorRef)`
  - causes Terminated message to be sent when actor stops

# Additional Akka Features

- Java API
  - completely interoperable with Scala API
- "Let it crash" failure management
  - based on hierarchical actor structure
  - highly flexible recovery
- Dynamic reconfiguration of actors
  - changing behaviour while application is running
- Flexible dispatching of requests to actors
  - "routers"
- Clustering support
  - from 2.2