

Actors

History

- Carl Hewit et al. 1973: Actors invented for research on artificial intelligence
- Gul Agha, 1986: Actor languages and communication patterns
- Ericsson, 1995: first commercial use in Erlang/OTP for telecommunications platform
- Philipp Haler, 2006: implementation in Scala standard library
- Jonas Bonér, 2009: creation of Akka
- José Valim, 2012: Creation of Elixir

Traditional Synchronization

Multiple threads stepping on each others' toes:

- demarcate regions of code with "don't disturb" semantics
- make sure that all access to shared state is protected

Synchronization

Multiple threads stepping on each others' toes:

- demarcate regions of code with "don't disturb" semantics
- make sure that all access to shared state is protected

Primary tools: lock, mutex, semaphore

In Scala every object has a lock: `synchronized {...}`

```
class BankAccount {  
    private var balance = 0  
  
    def deposit(amount: Int): Unit = synchronized {  
        if (amount > 0) balance = balance + amount  
    }  
  
    def withdraw(amount: Int): Int = synchronized {  
        if (0 < amount & amount <= balance) {  
            balance = balance - amount  
            balance  
        } else throw new Error("insufficient funds")  
    }  
}
```

What is an Actor?

The Actor Model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

An Actor:

- is an object with identity
- has a behavior
- only interacts using *asynchronous* message passing

The Actor Trait

```
// Define a type alias named 'Receive'
// It represents a function that handles messages sent to an Actor
// - It is a PartialFunction, meaning it may not handle every possible input
// - It takes a message of any type (`Any`) and returns nothing (`Unit`), like `void` in Java

type Receive = PartialFunction[Any, Unit]
```

```
// Define a trait named Actor
// A trait in Scala is similar to an interface in Java

trait Actor {
  // This abstract method must be implemented by the Actor
  // It defines the message-handling logic (what to do when a message is received)
  def receive: Receive
  // etc
}
```

A Simple Actor

```
// class 'Counter' extends the 'Actor' trait and implements the 'receive' method

class Counter extends Actor {

  // A mutable variable (the current count)
  var count = 0

  // The 'receive' method defines how this actor handles incoming messages
  def receive = {
    // If the message is the string "incr", increment the count
    case "incr" => count += 1
    // Any other message will be ignored
    // since this PartialFunction doesn't define a case for them
  }
}
```


Exposing the State

```
class Counter extends Actor {  
  var count = 0  
  
  // Add the "get" message  
  def receive = {  
    case "incr" => count += 1  
  
    // `customer` here is an ActorRef:  
    // `get` responds sending the current count back to the `customer` actor  
  
    case ("get", customer: ActorRef) =>  
      customer ! count // '!' is the "send message" operator in Akka  
  }  
}
```

Actors can send messages to addresses (ActorRef) they know.

Example using the class above

```
// A simple actor that receives the count and prints it
class Printer extends Actor {
  def receive = {
    case count: Int => println("Printer received count: " + $count)
  }
}
```

Usage:

```
// Initialize the actor system

counter ! "incr"
counter ! "incr"
counter ! "incr"

// Ask the counter to send its current value to the printer
counter ! ("get", printer)
```

How Messages are Sent

```
trait Actor {  
  // 'self' is an implicit reference to the actor instance itself (its ActorRef)  
  // It allows the actor to refer to its own address without passing it explicitly  
  implicit val self: ActorRef  
  
  // 'sender' gives access to the sender of the current message being processed  
  // This is useful when replying to messages – you can do: sender ! reply  
  def sender: ActorRef  
}
```

What is an **ActorRef** ?

It representing an addressable actor reference

Think of ActorRef as the actor's "mailbox address"

```
abstract class ActorRef {  
    // The '!' method (read: "bang") is the primary way to send a message to another actor  
    // - 'msg: Any': you can send any type of message  
    // - 'implicit sender': the sender is passed implicitly, so the recipient knows who sent it  
    def !(msg: Any)(implicit sender: ActorRef): Unit  
  
    // 'tell' is just an alias for '!'  
    // It makes the call more explicit when passing both the message and the sender  
    def tell(msg: Any, sender: ActorRef): Unit = this.!(msg)(sender)  
}
```

Sending a message from one actor to the other picks up the sender's address implicitly.

Using the Sender

For example. If I want to send a message back to the actor that invokes 'get'

```
class Counter extends Actor {  
  var count = 0  
  
  def receive = {  
    case "incr" => count += 1  
    case "get"  => sender ! count  
  }  
}
```

Another example

Expand the Actor Printer

```
class Printer extends Actor {  
  // The 'receive' method defines how this actor handles incoming messages  
  def receive = {  
    // If the message is an Int (the current count)  
    case count: Int =>  
      // Print the received count  
      println(s"[${self.path.name}] received count: $count")  
  
      // Acknowledge the sender by replying back  
      // 'sender' gives access to whoever sent this message  
      sender ! s"Acknowledged count $count from ${self.path.name}"  
    }  
  }
```

Interacting with Printer

```
class CounterClient(printer: ActorRef) extends Actor {  
  def receive = {  
    // This actor receives the acknowledgment from Printer  
    case ack: String =>  
      println(s"[${self.path.name}] got reply: $ack")  
  }  
  
  override def preStart(): Unit = {  
    // Send a number to the printer, using '!' (asynchronous fire-and-forget)  
    // 'self' will be implicitly used as the sender  
    printer ! 42  
  
    // Send the same number explicitly using 'tell'  
    printer.tell(99, self)  
  }  
}
```

The Actor's Context

In the Actor model, the `ActorContext` represents the execution context or environment in which an actor runs.

It provides services and tools that the actor can use to:

- Create other actors
- Change its own behavior dynamically
- Access self and sender references
- Stop itself or other actors

The `Actor` type describes the behavior, the execution is done by its `ActorContext`.

In code

```
// Trait that defines the context in which an Actor runs
trait ActorContext {

  // Change the current message handler (i.e., behavior) of the actor
  // - 'behavior' is a new Receive function (PartialFunction[Any, Unit])
  // - 'discardOld': if true, the current behavior is replaced entirely;
  //   if false, it can be stacked (useful for temporarily overriding behavior)
  def become(behavior: Receive, discardOld: Boolean = true): Unit

  // Revert to the previous behavior (used if become() was stacked with discardOld = false)
  def unbecome(): Unit

  // Other useful context methods typically include:
  // - actorOf(...) to spawn child actors
  // - stop(...) to stop an actor
  // - self, sender, parent, children
}
```

Every Actor has an implicit reference to its ActorContext

Example

```
class ToggleActor extends Actor {  
  def on: Receive = {  
    case "switch" =>  
      println("Turning off...")  
      context.become(off)  
  }  
  
  def off: Receive = {  
    case "switch" =>  
      println("Turning on...")  
      context.become(on)  
  }  
  
  // Initial behavior  
  def receive = on  
}
```

Another example

Define a class 'Counter' that extends the 'Actor' trait.

Functional style (no mutable variables)

```
class Counter extends Actor {  
  // Define a method 'counter' that takes the current count `n` as a parameter  
  // It returns a new Receive behavior for that count  
  def counter(n: Int): Receive = {  
    // When receiving "incr", the actor changes its behavior to handle count n + 1  
    // This replaces the current behavior with a new one where n is incremented  
    case "incr" => context.become(counter(n + 1)) // behavior change = state change  
  
    // When receiving "get", the actor replies to the sender with the current count  
    case "get" => sender ! n  
  }  
  
  // Initial behavior of the actor: start with count = 0  
  def receive = counter(0)  
}
```

Creating and Stopping Actors

Define the ActorContext trait, which provides the execution environment for an Actor. This trait exposes methods that an actor can use to interact with the actor system:

- to create new actors
- to stop existing actors (including itself)

```
trait ActorContext {  
  
  // Create (spawn) a new child actor under the current actor.  
  // - 'p': a Props object, defines the actor type and constructor parameters  
  // - 'name': a unique name for the new actor within the current context  
  def actorOf(p: Props, name: String): ActorRef  
  
  // Stop (terminate) an actor.  
  def stop(a: ActorRef): Unit  
}
```

An Actor Application

Define the main actor that drives the application logic

```
class CounterMain extends Actor {  
  // Create an instance of the Counter actor as a child of this actor  
  val counter: ActorRef = context.actorOf(Props[Counter], "counter")  
  
  // Send some increment messages to the counter  
  counter ! "incr"  
  counter ! "incr"  
  counter ! "incr"  
  
  // Ask the counter to send its current value back (reply goes to this actor)  
  counter ! "get"  
  
  // This actor handles the reply from the counter  
  def receive: Receive = {  
    case count: Int =>  
      println(s"Count was $count") // Print the count  
      context.stop(self)           // Stop this actor (ends the app)  
  }  
}
```

The Main App


Define the main application entry point

```
object CounterMainApp extends App {  
  // Create the actor system  
  val system = ActorSystem("CounterSystem")  
  
  // Create the main actor that orchestrates everything  
  system.actorOf(Props[CounterMain], "main")  
  
  // The system will shut down after the CounterMain actor stops (not shown here)  
  // For a clean shutdown, you could use CoordinatedShutdown or watch termination manually  
}
```

The Actor Model of Computation

Upon reception of a message, an actor can do any combination of the following:

- **Send messages** — communicate with other actors asynchronously.
- **Create actors** — spawn child actors to delegate work or structure the system hierarchically.
- **Designate the behavior for the next message** — dynamically change its own behavior for future messages.

 Actors encapsulate both state and behavior, allowing safe, lock-free concurrency by reacting to messages.

Actor Encapsulation

Actors are isolated: external code **cannot access their internal state or behavior directly**.

Only interaction: **asynchronous message passing** via known addresses (`ActorRef`):

- every actor knows its **own address** (`self`)
- creating an actor returns its **address**
- addresses can be **shared and passed in messages** (e.g., via `sender`)

 This model enforces isolation and prevents shared-memory issues like race conditions.

Actor Encapsulation

Actors are **fully independent units of execution**.

- Run **locally and concurrently**
- No shared memory, **no global synchronization**
- Communication is **one-way and asynchronous**

 Like people sending emails: each works independently and responds when ready.

Actor-Internal Evaluation Order

Each actor is effectively **single-threaded**:

- Messages are handled **sequentially**
- A call to `context.become` changes behavior **before the next message**
- **Processing one message is atomic** — no interleaving with other actors

✓ This simplifies reasoning: no need for locks inside an actor.

Actor-Internal Evaluation Order

Actors process one message at a time:

- No overlap between message handlers
- Behavior changes apply to the next message
- Atomicity ensures safe local state updates

 This is like `synchronized`, but without blocking — instead, messages queue up in the mailbox.

Actor-Internal Evaluation Order

An actor is effectively single-threaded:

- messages are received sequentially
- behavior change is effective before processing the next message
- processing one message is the atomic unit of execution

This has the benefits of synchronized methods, but blocking is replaced by enqueueing a message.

The Bank Account (revisited)

Good practice: define Actor's messages in companion object.

In this case 4 `case` classes one for each actor message

```
object BankAccount {  
  case class Deposit(amount: BigInt)  
  case class Withdraw(amount: BigInt)  
  case object Done  
  case object Failed  
}
```

The Bank Account (revisited)

```
class BankAccount extends Actor {  
  var balance: BigInt = BigInt(0)  
  
  def receive: Receive = {  
    case Deposit(amount) => // Use pattern matching to extract the amount  
      balance += amount  
      sender ! Done  
    case Withdraw(amount) if amount <= balance => // A guard  
      balance -= amount  
      sender ! Done  
    case _ => sender ! Failed // Fail in other cases  
  }  
}
```

Actor Collaboration

- picture actors as persons
- model activities as actors

Transferring Money (0)

```
object WireTransfer {  
  case class Transfer(from: ActorRef, to: ActorRef, amount: BigInt)  
  case object Done  
  case object Failed  
}
```


Transferring Money (1)

```
class WireTransfer extends Actor {  
  def receive: Receive = {  
    case Transfer(from, to, amount) =>  
      from ! BankAccount.Withdraw(amount)  
      context.become(awaitWithdraw(to, amount, sender))  
  }  
  
  def awaitWithdraw(to: ActorRef, amount: BigInt, client: ActorRef): Receive = ???  
}
```

Transferring Money (2)

```
class WireTransfer extends Actor {  
  // ...  
  
  def awaitWithdraw(to: ActorRef, amount: BigInt, client: ActorRef): Receive = {  
    case BankAccount.Done =>  
      to ! Deposit(amount)  
      context.become(awaitDeposit(client))  
    case BankAccount.Failed =>  
      client ! Failed  
      context.stop(self)  
  }  
  
  def awaitDeposit(client: ActorRef): Receive = ???  
}
```

Transferring Money (3)

```
class WireTransfer extends Actor {  
  // ...  
  
  def awaitDeposit(client: ActorRef): Receive = {  
    case BankAccount.Done =>  
      client ! Done  
      context.stop(self)  
  }  
}
```

Message Delivery Guarantees

- all communication is inherently unreliable
- delivery of a message requires eventual availability of channel & recipient

Message Delivery Guarantees

- all communication is inherently unreliable
- delivery of a message requires eventual availability of channel & recipient

Delivery guarantees:

- **at-most-once:** sending once delivers $[0, 1]$ times
- **at-least-once:** resending until acknowledged delivers $(1, \infty)$ times
- **exactly-once:** processing only first reception delivers 1 time

Reliable Messaging

Messages support reliability:

- all messages can be persisted
- can include unique correlation IDs
- delivery can be retries until successful

Reliable Messaging

Messages support reliability:

- all messages can be persisted
- can include unique correlation IDs
- delivery can be retries until successful

Reliability can only be ensured by business-level acknowledgement.

Making the Transfer Reliable

- log activities of WireTransfer to persistent storage
- each transfer has a unique ID
- add ID to Withdraw and Deposit
- store IDs of completed actions within BankAccount

Message Ordering

If an actor sends multiple messages to the same destination, they will not arrive out of order (this is Akka-specific).

Recap

Actors are fully encapsulated, independent agents of computation.

Messages are the only way to interact with actors.

Explicit messaging allows explicit treatment of reliability.

The order in which messages are processed is mostly undefined.

Designing Actor Systems

Starting Out with the Design

- Imagine giving the task to a group of people, dividing it up.
- Consider the group to be of huge size.
- Start with how people with different tasks will talk with each other.
- Consider these “people” to be easily replaceable.
- Draw a diagram with how the task will be split up, including communication lines.

Let It Crash: The Philosophy

Embrace failure rather than prevent it.

- Errors are expected in distributed systems.
- Defensive programming leads to complexity and rigidity.
- The Actor model **isolates failures**: actors crash and restart without affecting others.

★ In Erlang/Elixir: "fail fast, recover quickly"

Why Let It Crash Works

- Each actor is isolated: a crash affects **only that actor**.
- When an actor fails, its supervisor can restart or handle it.
- No need for complex error handling inside each actor.

 Simpler, more resilient systems emerge from letting small parts fail.

Supervision Trees

Actors can supervise child actors:

- Supervisors detect failures and apply **restart strategies**.
- Failures don't propagate chaos; they're **contained**.
- The structure forms a **supervision hierarchy** (tree).

```
val child = context.actorOf(Props[Worker], "worker")
```



Trees reflect modularity and control scope of recovery.

Supervision Strategies

Common strategies include:

- **Restart** : recreate the actor fresh.
- **Resume** : ignore failure and continue.
- **Stop** : terminate the actor.
- **Escalate** : let the failure bubble up.

```
override val supervisorStrategy =  
  OneForOneStrategy() {  
    case _: ArithmeticException => Resume  
    case _: NullPointerException => Restart  
    case _: Exception => Stop  
  }
```


Designing for Resilience

Design tips:

- Compose the system from **small, crashable actors**.
- Assign supervision clearly: **who is responsible for whom?**
- Avoid complex local try/catch blocks — rely on supervision.
- Structure follows failure boundaries.

 Resilience is an architectural choice, not an afterthought.