Concurrency and Actors
Task: Create an Actor class that tracks a single Int

- In a package named actors create a class named StringActor that extends Actor
- Create the following case class/objects that will be used as messages
  - A case class named Append that takes a String in its constructor
  - A case object named GetValue
  - A case class named Value that takes a String in its constructor
- The StringActor class must:
  - Take a String in its constructor. This will be the initial String that it will store
  - When it receives an Append message, append its value to the end of the currently stored value
  - When it receives a GetValue message, sends its current value back to the sender in a Value message
Concurrency

• Most programs we've written execute code sequentially
• Each statement of code is executed in the order they are written
• Can have control flow to decide which statements are executed and in which order

• What if we want multiple pieces of code to execute at the same time?
Concurrency

- We've written 2 types of concurrent software already
- In CSE115, you wrote a web server
  - What if 2 users are visiting your site at the same time?
  - Server waits for requests and handles them as they are received
  - You provide callback functions that are called when a request arrives
- In CSE116, we saw GUIs on each HW assignment
  - GUI runs an update loop to display the current state of the software
  - GUI simultaneously listens for user inputs
  - You provide listener classes with a method that is called when the user takes an action
Concurrency

• For both web servers and GUIs
• We used libraries that hid the concurrency
• What if we want to write concurrent code that is not part of a web server or GUI?
• We'll see how to write concurrent programs using actors
Concurrency - Actors

- The Akka library
- Add to pom.xml and install
- Akka uses actors for concurrency
- We create and instantiate actor classes and each actor runs concurrently
- Actors are based on a message passing system
  - Multiple actors run in the same program at the same time
- Actors pass messages to share information
- Write code that executes in reaction to a message
- Messages are case classes or case objects
Concurrenty - Actors

- Receiving a message is an event

- **Event-Based Architecture**
  - Write code that is executed when an event occurs
  - Create events that cause code to run
Case Class/Object

- Case class
  - A different type of class in Scala
  - Primarily used to store values provided through its constructor
  - Typically have no body
  - Are compared by value, not reference

- Case object
  - Used when no values are stored (no constructor)
  - Can be used to signal that an event has occurred

```scala
case class BuyEquipment(equipmentID: String)
case object Setup
```
Concurrency - Actors

- To define an Actor
- Extend the Actor class
- Implement the receive method to define how the Actor responds to different message types

```scala
import akka.actor._

case object CustomMessageType
case class AnotherMessageType(message: String)

class MyActor extends Actor {

  def receive: Receive = {
    case CustomMessageType => // do something
    case received: AnotherMessageType => received.message // do something
  }
}
```
Concurrency - Actors

- Messages are instances of case classes or case objects
- Use a case statement to make decisions based on the type of the message
- If the message is a case class, declare a variable to access its values

```scala
import akka.actor._

case object CustomMessageType

case class AnotherMessageType(message: String)

class MyActor extends Actor {
  def receive: Receive = {
    case CustomMessageType => // do something
    case received: AnotherMessageType => received.message // do something
  }
}
```
Concurrency - Actors

• Create an actor and add it to actor system
• The actor is now running concurrently with your program
• Send messages using the ! method

```scala
object CounterTest extends App {
  val system = ActorSystem("FirstSystem")

  val actor = system.actorOf(Props(classOf[MyActor]))

  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```
Concurrency - Actors

• Cannot create an Actor using the new keyword

• Use Props (part of the Akka library) and pass the class as an argument

```scala
object CounterTest extends App {
  val system = ActorSystem("FirstSystem")

  val actor = system.actorOf(Props(classOf[MyActor]))

  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```
Concurrency - Actors

- If your Actor class takes a constructor parameters pass them in the Props call

```scala
class MyActor(n: Int) extends Actor {
  def receive: Receive = {
    case CustomMessage => // do something
    case r: AnotherMessageType => r.message // do something
  }
}

object CounterTest extends App {
  val system = ActorSystem("FirstSystem")
  val actor = system.actorOf(Props(classOf[MyActor], 10))
  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```
Counting Example
Actors - Counting Example

• Create an Actor class that counts down from 20 as fast as it can
• Send the actor a Start message to start the countdown
• Start is a case object
• We'll create 3 of these actors and watch them count down concurrently
Actors - Counting Example

- We'll use 4 different message types
  - All are case objects
- Start - Tells a Counter to start its countdown
- IsDone - Sent to a Counter to ask if it's done or not
- Done - Sent from Counter to indicate that it is done counting
- NotDone - Sent from Counter to indicate that it is not done counting

```scala
class Counter(name: String) extends Actor {
  var n = 0

  def countDown(): Unit = {
    if (n >= 0) {
      println(this.name + " - " + n)
      n -= 1
      countDown()
    } else {
      println(this.name + " finished")
    }
  }

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
    case IsDone =>
      if (n <= 0) {
        sender() ! Done
      } else {
        sender() ! NotDone
      }
  }
}

case object Start
case object IsDone
case object Done
case object NotDone
```
We define actors just like any other class

Can have constructor, variables, methods

This class:

Take a String in it's constructor

Initializes a variable n to 0

Has a countDown method to start a countdown and print the progress along the way

class Counter(name: String) extends Actor {
  var n = 0
  def countDown(): Unit = {
    if (n >= 0) {
      println(this.name + " - " + n)
      n -= 1
      countDown()
    } else {
      println(this.name + " finished")
    }
  }
  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
    case IsDone =>
      if (n <= 0) {
        sender() ! Done
      } else {
        sender() ! NotDone
      }
  }
}

case object Start
case object IsDone
case object Done
case object NotDone
Since we extend Actor, we must implement Receive

Use case syntax to react differently to different message types

Whenever this actor receives a message of type Start, it resets its counter to 20 and starts a countdown

```scala
class Counter(name: String) extends Actor {
  var n = 0
  def countDown(): Unit = {
    if (n >= 0) {
      println(this.name + " - " + n)
      n -= 1
      countDown()
    } else {
      println(this.name + " finished")
    }
  }

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
    case IsDone =>
      if (n <= 0) {
        sender() ! Done
      } else {
        sender() ! NotDone
      }
  }
}
```

```scala
case object Start
case object IsDone
case object Done
case object NotDone
```
Actors - Counting Example

- When this actor receives a message of type IsDone
- Uses the sender() method to send a message back to whatever actor sent the message
- Send Done or NotDone based on the status of the countdown
- In this way, actors can communicate by passing messages

```scala
class Counter(name: String) extends Actor {
  var n = 0

  def countDown(): Unit = {
    if (n >= 0) {
      println(this.name + " - " + n)
      n -= 1
      countDown()
    } else {
      println(this.name + " finished")
    }
  }

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
    case IsDone =>
      if (n <= 0) {
        sender() ! Done
      } else {
        sender() ! NotDone
      }
    }
  }
}
```
To use the Actor we'll create 3 objects of this type with different names.

Send each Actor the Start message so they count down.

```scala
class Counter(name: String) extends Actor {
  ...
  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
  }
}
```

```scala
object CounterTest extends App {
  val system = ActorSystem("CountingSystem")

  val one = system.actorOf(Props(classOf[Counter], "1"))
  val two = system.actorOf(Props(classOf[Counter], "2"))
  val three = system.actorOf(Props(classOf[Counter], "3"))

  one ! Start
  two ! Start
  three ! Start
}
```
A Actors - Counting Example

- All three counter countdown concurrently
- No way to know which will finish first

```scala
class Counter(name: String) extends Actor {
  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
  }
}
```

```scala
object CounterTest extends App {
  val system = ActorSystem("CountingSystem")

  val one = system.actorOf(Props(classOf[Counter], "1"))
  val two = system.actorOf(Props(classOf[Counter], "2"))
  val three = system.actorOf(Props(classOf[Counter], "3"))

  one ! Start
  two ! Start
  three ! Start
}
```
Actors - Counting Example

- Let's create another Actor that will communicate with the three counters
- This actor will "ask" each counter if it's done or not
- Once all counters are done, it will print a message to the screen

```scala
class Supervisor(counters: List[ActorRef]) extends Actor {

  var total: Int = counters.size
  var completed: List[ActorRef] = List()

  def receive: Receive = {
    case Update =>
      counters.foreach((actor: ActorRef) => actor ! IsDone)
    case Done =>
      if (!completed.contains(sender())){
        completed ::= sender()
        if (completed.size == this.total) {
          println("All counters complete")
        }
      }
    case NotDone =>
      println("A counter is not done yet")
  }
}
```
• Use the ActorRef class to send messages to other actors
• sender() returns the ActorRef of the sender of a message

```scala
class Supervisor(counters: List[ActorRef]) extends Actor {
  var total: Int = counters.size
  var completed: List[ActorRef] = List()

  def receive: Receive = {
    case Update =>
      counters.foreach((actor: ActorRef) => actor ! IsDone)
    case Done =>
      if (!completed.contains(sender())){
        completed ::= sender()
        if (completed.size == this.total) {
          println("All counters complete")
        }
      }
    case NotDone =>
      println("A counter is not done yet")
  }
}
```
Actors - Counting Example

- Add the supervisor to the system and have it update twice per second
- Use a scheduler to repeatedly send a message

```scala
object CounterTest extends App {
  val system = ActorSystem("CountingSystem")

import system.dispatcher

val one = system.actorOf(Props(classOf[Counter], "1"))
val two = system.actorOf(Props(classOf[Counter], "2"))
val three = system.actorOf(Props(classOf[Counter], "3"))

val supervisor = system.actorOf(Props(classOf[Supervisor], List(one, two, three)))

one ! Start
two ! Start
three ! Start

system.scheduler.schedule(0.milliseconds, 500.milliseconds, supervisor, Update)
}
```
Counting Example Demo
Lecture Question

Task: Create an Actor class that tracks a single Int

- In a package named actors create a class named StringActor that extends Actor
- Create the following case class/objects that will be used as messages
  - A case class named Append that takes a String in its constructor
  - A case object named GetValue
  - A case class named Value that takes a String in its constructor
- The StringActor class must:
  - Take a String in its constructor. This will be the initial String that it will store
  - When it receives an Append message, append its value to the end of the currently stored value
  - When it receives a GetValue message, sends its current value back to the sender in a Value message