Concurrency and Actors

Lecture Task

- Genetic Algorithm: Lecture Task 6 -

Functionality: In the Playlist **object** implement the following method:

- A method named “costFunction” that:
  - Takes a Map of Strings to Ints as a parameter representing the rating of a specific user. The keys of this map will be youtubeIds and the values are the ratings from this user
  - Returns a function that takes a Playlist as a parameter and returns a Double. This function will compute the cost of a Playlist for this specific user based on their ratings. The function will compute the cost as follows:
    - Compute a “raw cost” as the sum of the cost of all the songs. Remember that you can call your Song.costFunction method to get a cost function for songs and call this function to compute these costs. Sum up all the costs and that’s the raw cost of the playlist
    - Compute a cost multiplier that is:
      - 1000.0 if the playlist contains a duplicate song (Check if the playlist contains any youtubeIds multiple times). This large multiplier will all but certainly prevent any playlist from containing duplicate songs
      - Compute the standard deviation of the averageEnergyRating of all songs
        - If the standard deviation is < 0.5, set the cost multiplier to 10.0
        - If the standard deviation is >= 0.5, set the cost multiplier to the inverse of the standard deviation (eg. 1.0 divided by the standard deviation)
    - Returns the raw cost times the cost multiplier

Testing: In the tests package, complete the test suite named LectureTask6 that tests this functionality.

Reminder

- You will not be required to **write** code using Concurrency, Actors, or WebSockets (The topics for this week of lecture)
- You will have interview/quiz questions on these topics
  - You are expected to **explain** these concepts
  - You are expected to **read** code that uses Actors

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
  - GUI runs an update loop to display the current state of the software
  - GUI simultaneously listens for user inputs and runs the logic of your program
  - 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

Concurrency - 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

```
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

```
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

```
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

```
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

```
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 any constructor parameters, pass them in the Props call

```
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

- 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

```
case object Start
case object IsDone
case object Done
case object NotDone
```

```
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
      }
  }

}
```

Actors - Counting Example

- We define actors just like any other class
  - Can have constructor, variables, methods
- This class:
  - Takes a String in the constructor
  - Initializes a variable n to 0
  - Has a countDown method to start a countdown and print the progress along the way

```
case object Start
case object IsDone
case object Done
case object NotDone
```

```
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
      }
  }

}
```

Actors - Counting Example

- 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

```
case object Start
case object IsDone
case object Done
case object NotDone
```

```
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
      }
  }

}
```

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

```
case object Start
case object IsDone
case object Done
case object NotDone
```

```
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
      }
  }

}
```

Actors - Counting Example

- 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

```
class Counter(name: String) extends Actor {

  ...

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
  }
}
```

```
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

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

```
class Counter(name: String) extends Actor {

  ...

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
  }
}
```

```
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
}
```

Counter Demo

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

```
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

- Use the ActorRef class to send messages to other actors
  - sender() returns the ActorRef of the sender of a message

```
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

```
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)
}
```

Supervisor Counter Demo

Stock Trader Example

- Simulate stocks changing prices and a trader purchasing stocks

- Stock
  - Receives a Tick message and changes its price
    - Price changes are random for this simulation
    - Tick messages are sent to all stocks at regular intervals
  - Receives a GetPrice message and responds with its current price

- Trader
  - Knows each stock's ticker symbol and actor reference
  - Receives a CheckStocks message and checks the price of all known stocks
  - Receives Price messages from stocks and decides [randomly] whether or not to buy some shares

To the Code

Lecture Task

- Genetic Algorithm: Lecture Task 6 -

Functionality: In the Playlist **object** implement the following method:

- A method named “costFunction” that:
  - Takes a Map of Strings to Ints as a parameter representing the rating of a specific user. The keys of this map will be youtubeIds and the values are the ratings from this user
  - Returns a function that takes a Playlist as a parameter and returns a Double. This function will compute the cost of a Playlist for this specific user based on their ratings. The function will compute the cost as follows:
    - Compute a “raw cost” as the sum of the cost of all the songs. Remember that you can call your Song.costFunction method to get a cost function for songs and call this function to compute these costs. Sum up all the costs and that’s the raw cost of the playlist
    - Compute a cost multiplier that is:
      - 1000.0 if the playlist contains a duplicate song (Check if the playlist contains any youtubeIds multiple times). This large multiplier will all but certainly prevent any playlist from containing duplicate songs
      - Compute the standard deviation of the averageEnergyRating of all songs
        - If the standard deviation is < 0.5, set the cost multiplier to 10.0
        - If the standard deviation is >= 0.5, set the cost multiplier to the inverse of the standard deviation (eg. 1.0 divided by the standard deviation)
    - Returns the raw cost times the cost multiplier

Testing: In the tests package, complete the test suite named LectureTask6 that tests this functionality.