

# Testing Actors

# Lecture Question

## **Task: Create and test an actor class to track funds in a bank account**

- This question is similar to Monday's in that your actor will only track a single value. Through this question you will practice writing a test suite for actors
- In a package named bank create a class named BankAccount that extends Actor
- Create the following case class/objects in the bank package that will be used as messages
  - A case class named Deposit that takes an Int in its constructor
  - A case class named Withdraw that takes an Int in its constructor
  - A case object named CheckBalance
  - A case class named Balance that takes an Int in its constructor
- The BankAccount class must:
  - Initially contain no funds
  - When it receives a Deposit message, increases its funds by the amount in the message
  - When it receives a Withdraw message, decrease its funds by the amount in the message only if the account has enough money to cover the withdrawal. If it doesn't have enough money, no action is needed
  - When it receives a CheckBalance message, sends its current funds back to the sender in a Balance message
- In a package named tests, write a class named TestBankAccount as a test suite that tests the BankAccount functionality

# Traits and Mixins

- We've seen inheritance by extending abstract classes
- What if we want to extend multiple classes?
  - Example: You want an object class that extends `PhysicalObject` to have physics applied and extend `Actor` to run concurrently
  - This is not allowed
- Can avoid this need by using composition
  - Make a class that extends `actor` and stores a `PhysicalObject` in an instance variable

# Traits and Mixins

- Traits
  - Similar to abstract classes
  - **Cannot have a constructor**
- No limit to the number of traits that can be extended

```
trait Database {  
  def playerExists(username: String): Boolean  
  def createPlayer(username: String): Unit  
  def saveGameState(username: String, gameState: String): Unit  
  def loadGameState(username: String): String  
}
```

# Traits and Mixins

- Mixins
  - When extending multiple traits, we use the term mixin (ie. The traits are mixed into the class)
- Extend any one class/abstract class/trait using "extends"
- Add any number of Traits using "with"
- Must implement all abstract methods from all inheritances

```
class ActorAndDatabase extends Actor with Database {  
  override def receive: Receive = {  
    case _ =>  
  }  
  override def playerExists(username: String): Boolean = {  
    false  
  }  
  override def createPlayer(username: String): Unit = {}  
  override def saveGameState(username: String, gameState: String): Unit = {}  
  override def loadGameState(username: String): String = {  
    ""  
  }  
}
```

# Testing Actors

- We've seen our first actor system where multiple actors can run concurrently
- **But how would we test such a program?**
- Use a FunSuite like we have all semester?
  - FunSuite starts the actor system
  - Creates actors
  - Sends messages
  - Runs some asserts that likely execute before the first message is received by an actor
- Can wait with `Thread.sleep` to wait for messages, but the FunSuite can't receive messages from the actors
  - Now way of gaining information from the actors

# Testing Actors - Library

- Lets pull in a new library to help us out
- The testing library we'll use is the Akka testkit
- Make sure the version number matches your version of the Akka actor library

```
<dependency>  
  <groupId>com.typesafe.akka</groupId>  
  <artifactId>akka-actor_2.12</artifactId>  
  <version>2.5.25</version>  
</dependency>
```

```
<dependency>  
  <groupId>com.typesafe.akka</groupId>  
  <artifactId>akka-testkit_2.12</artifactId>  
  <version>2.5.25</version>  
</dependency>
```

# Testing Actors - Setup

- Using this library, we'll setup a new type of TestSuite using the TestKit class
- This setup can is directly from the test kit documentation and can be reused whenever you setup a test suite for actors

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```



# Testing Actors - Setup

- Import classes/traits from the libraries
- The test kit is build on top of scalatest so we'll use both libraries
- Import the duration package from Scala to use 100.millis syntax
  - Important: This must be manually added. IntelliJ will not suggest importing this and you will have an error on your millis

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors - Setup

- Extend the TestKit class which has a constructor that takes an ActorSystem
- Name and Create a new ActorSystem directly in the constructor

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors - Setup

- Mixin traits for additional functionality
- These traits contain defined methods
  - No methods that need to be implemented

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors - Setup

- From the BeforeAndAfterAll train, we inherited the afterAll method which is called after all our tests complete
- By default, afterAll is implemented but does nothing
- We override afterAll to properly shutdown the actor system

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors - Setup

- Finally, we can setup our tests
- Instead of a test name as we had for unit testing, the test kit uses behavioral tests
  - Name tests by the expected behavior of your code

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors - Setup

- Methods "must" and "in" are inherited from WordSpecLike and are called inline
- Uses a string wrapper class (ie. "A value actor" is implicitly converted to the wrapper class which contains the must method)
- All this to achieve more human readable syntax

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }
}
```

# Testing Actors

- With the test suite setup, we're ready to start testing our actors
- We have access to the actor system we created in the constructor in the system variable
- Use this system to start your actor(s)

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```



# Testing Actors

- Send messages to the actor using the `!` method

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```



# Testing Actors

- **New:** Use the `expectNoMessage` method (inherited from `TestKit`) to wait for messages to resolve before testing
- The test suite will wait for the specified amount of time
  - If there's an error on millis, don't forget to import Scala's duration package

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Testing Actors

- Send a message to an actor that is expecting a response
- The test suite is part of the actor system and can receive the response for testing

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Testing Actors

- **New:** When you expect to receive a message, call `expectMsgType` with the message type you expect
- The time provided is the maximum amount of time to wait before failing the test
- If we don't receive a message of type `Value` within 1 second, this test fails

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Testing Actors

- When the message is received, use asserts to make sure it contains the expected values

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Testing Actors

- Caution: When testing message types with undefined variable names **do not access the variables for testing**
- For today's lecture question, the names of the Int variables are not defined and it is unlikely that we will use the same names
- If we use different names => error in AutoLab

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Testing Actors

- Instead of accessing message variables
  - Create a new message of that type and check for equality
  - Recall that case classes have an equals method that compares the values of its variables instead of comparing references

```
"A value actor" must {  
  "track a value" in {  
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))  
  
    valueActor ! Increase(5)  
    valueActor ! Increase(5)  
  
    expectNoMessage(100.millis)  
  
    valueActor ! GetValue  
    val value: Value = expectMsgType[Value](1000.millis)  
  
    assert(value == Value(20))  
  }  
}
```

# Lecture Question

## **Task: Create and test an actor class to track funds in a bank account**

- This question is similar to Monday's in that your actor will only track a single value. Through this question you will practice writing a test suite for actors
- In a package named bank create a class named BankAccount that extends Actor
- Create the following case class/objects in the bank package that will be used as messages
  - A case class named Deposit that takes an Int in its constructor
  - A case class named Withdraw that takes an Int in its constructor
  - A case object named CheckBalance
  - A case class named Balance that takes an Int in its constructor
- The BankAccount class must:
  - Initially contain no funds
  - When it receives a Deposit message, increases its funds by the amount in the message
  - When it receives a Withdraw message, decrease its funds by the amount in the message only if the account has enough money to cover the withdrawal. If it doesn't have enough money, no action is needed
  - When it receives a CheckBalance message, sends its current funds back to the sender in a Balance message
- In a package named tests, write a class named TestBankAccount as a test suite that tests the BankAccount functionality