## ****1. THREADS AND THE JVM MEMORY****

**SCALA USING JAVA THREADING MODEL**

Scala, being a JVM language, can directly use Java’s threading model:

object BasicThreading extends App {

val thread = new Thread(() => println("Running in a separate thread"))

thread.start()

thread.join() // Wait for the thread to finish

}

## ****2. SCALA FUTURES AND PROMISES****

**SCALA PROVIDES FUTURE IN THE SCALA.CONCURRENT PACKAGE:**

import scala.concurrent.\_

import ExecutionContext.Implicits.global

val future = Future {

Thread.sleep(1000) // Simulate some work

42

}

future.foreach(result => println(s"Got the result: $result"))

#### ****Composing Futures****

val doubled = future.map(\_ \* 2)

val chained = future.flatMap(result => Future(result + 10))

val recovered = future.recover { case e: Exception => 0 }

#### ****Promises****

A Promise is a placeholder for a Future result that is manually completed.

val promise = Promise[Int]()

val futureFromPromise = promise.future

futureFromPromise.foreach(result => println(s"Got: $result"))

promise.success(100) // Completes the

Future with value 100

Use promise.failure(exception) to complete it with an error.

## ****4. ACTORS MODEL WITH AKKA****

### ****Akka Actors Codes****

import akka.actor.\_

class SimpleActor extends Actor {

def receive: Receive = {

case msg: String => println(s"Received message: $msg")

}

}

val system = ActorSystem("MyActorSystem")

val actor = system.actorOf(Props[SimpleActor], "simpleActor")

actor ! "Hello, Actor" // Sending a message to the actor

|  |
| --- |
| **Key Notes** |
| Actors are lightweight and handle messages asynchronously. |
| The ! operator (called **tell**) is used to send messages. |
| Each actor runs in its own lightweight thread. |

## ****Reactive Programming with Akka Streams****

import akka.stream.\_

import akka.stream.scaladsl.\_

import akka.actor.ActorSystem

implicit val system = ActorSystem("StreamSystem")

implicit val materializer = Materializer(system)

val source = Source(1 to 10)

val sink = Sink.foreach[Int](println)

val flow = Flow[Int].map(\_ \* 2)

source.via(flow).to(sink).run()

|  |  |
| --- | --- |
| **Key Notes** | |
| Source | Produces data |
| Flow | Transforms data |
| Sink | Consumes data |