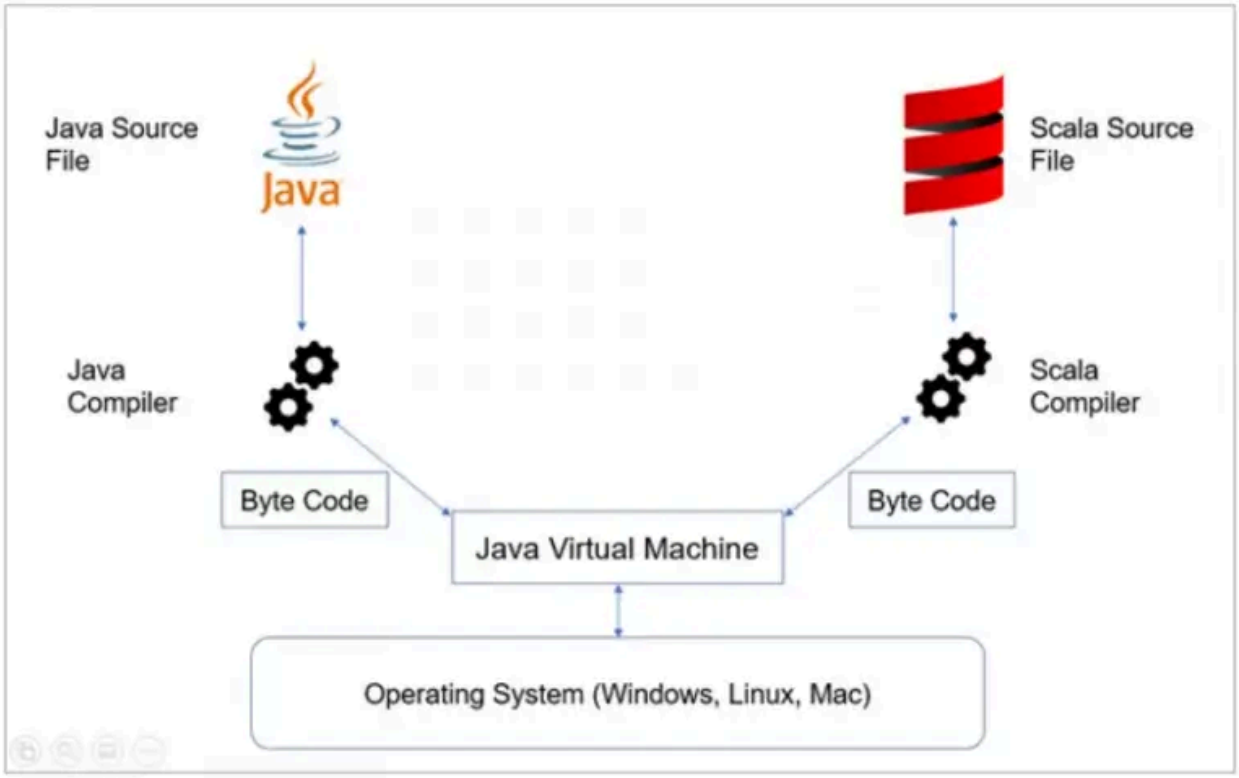
Scala

1. An overview of functional programming



Functional programming (FP) is a programming paradigm that emphasizes the use of pure **functions** and **immutable** data to create predictable, maintainable, and bug-resistant software. Its key features include:

- **Pure Functions**: Functions that always produce the same output for the same input and have no side effects (they do not modify any external state). This makes them easier to **test, debug,** and **reuse**.
- **Immutability**: Data is immutable, meaning once created, it cannot be changed. This avoids shared state and unexpected changes, which improves safety and supports parallelism.
- **First-Class and Higher-Order Functions**: Functions are treated like **values**; they can be assigned to **variables, passed as arguments**, or returned from other functions. Higher-order functions take functions as inputs or return them as outputs, enabling powerful composition.
- **Recursion**: **Instead** of traditional **loops**, FP uses **recursion** to perform repeated computations.

- **Declarative Style**: FP focuses on *what* to solve rather than ***how*** to solve it. Programs are built by applying and composing small, pure functions.

Recursion Program:

```
object FactorialExample {
 def factorial(n: Int): Int = {
   // Base case: if n is 0, return 1
   if (n == 0) {
     1
   } else {
     // Recursive case: n * factorial of (n-1)
     n * factorial(n - 1)
   }
 }


 def main(args: Array[String]): Unit = {
   val num = 5
   println(s"The factorial of $num is: ${factorial(num)}") // Output: The factorial of 5 is: 120
 }
}
```

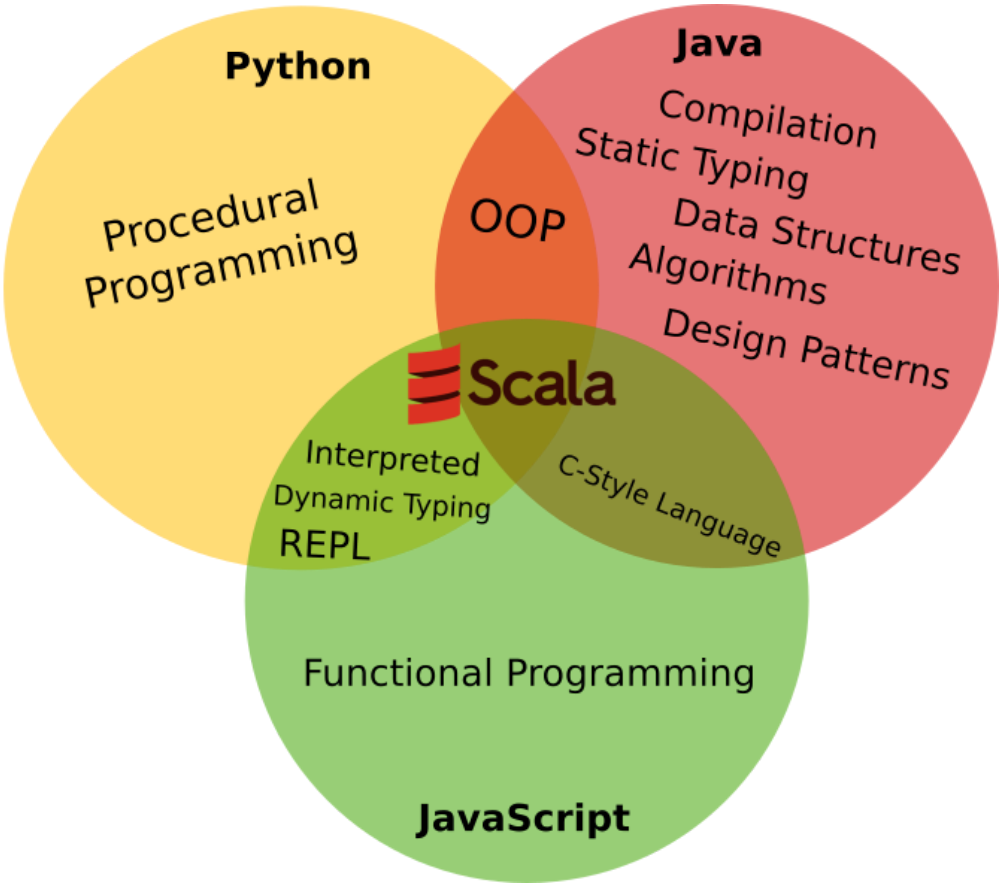
Functional programming contrasts with imperative or object-oriented styles by avoiding mutable state and side effects, which leads to clearer and more predictable code. Scala integrates functional programming with object-oriented features, allowing expressive and concise code well-suited for concurrent and big data applications.
A simple example in Scala demonstrating a pure function:

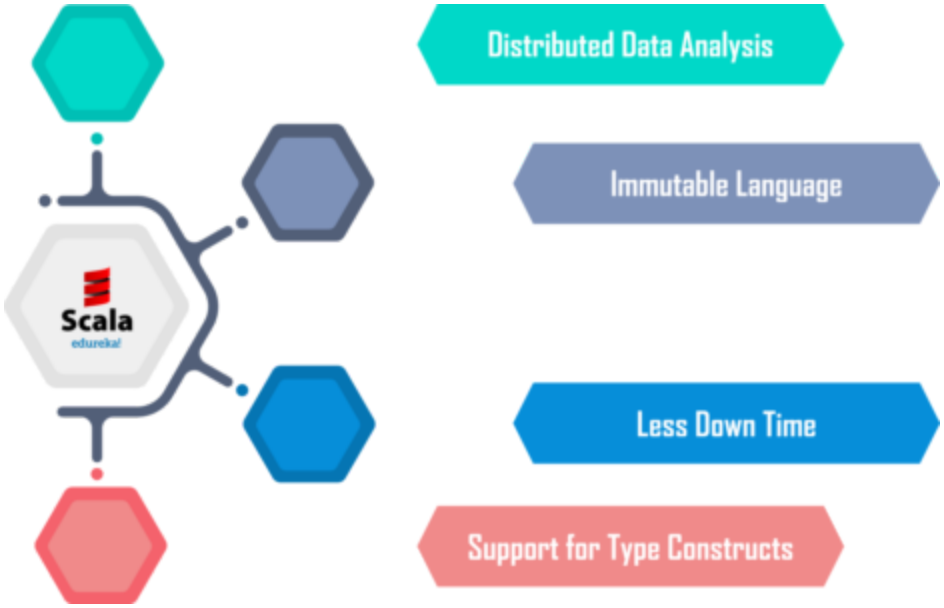
```
def square(x: Int): Int = x * x
println(square(5)) // Outputs 25
```

This function always returns the same output for the same input and does not change any external state.
Languages built around functional programming include **Haskell, Scala, Clojure**, and **Erlang**, while many others like **Python** and **JavaScript** incorporate functional features. Functional programming is particularly beneficial for handling **concurrency**, **simplifying testing**, and improving **modularity** and **code maintainability.**

2. Why Scala?

- Description:
  Scala blends **functional** and **object-oriented programming,** runs on JVM, interoperates with Java, and is ideal for Big Data (Spark, Hadoop).
- Key Points:
  - Scalability & conciseness
  - Functional & OOP features
  - Rich collections library
  - Growing ecosystem for data engineering





3. REPL (Read-Eval-Print Loop)

- Description:
  Scala REPL is an interactive shell allowing rapid experimentation.
- Example:

```
$ scala
Welcome to Scala 2.13. ...
scala> val x = 42
x: Int = 42
scala> println(x + 8)
50
```

- Explanation:
  Show how to start REPL and use it for quick function checks or exploring Scala features.

4. Working with Functions

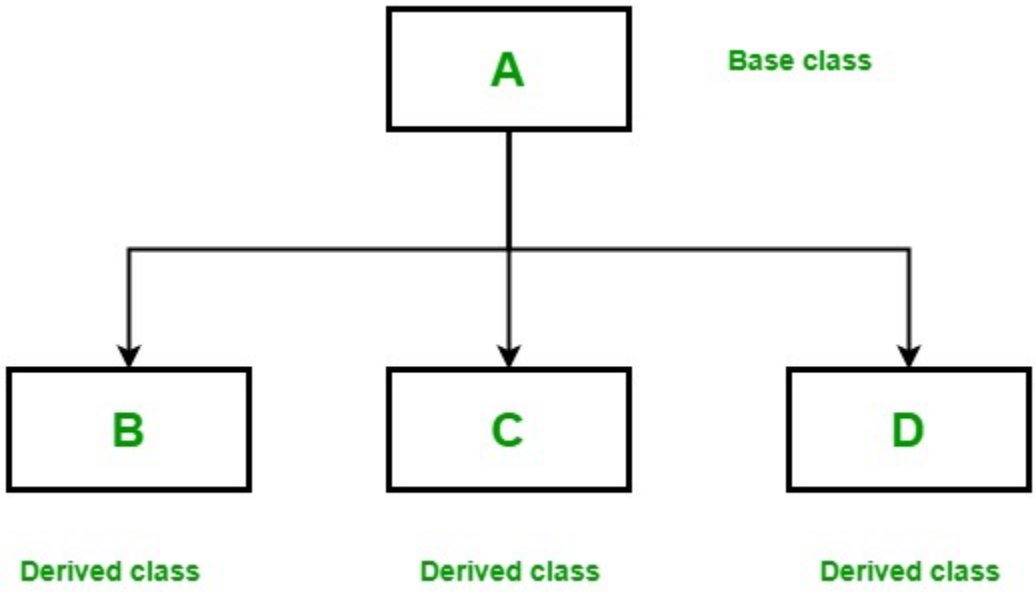
- Description:
  Functions are **first-class citizens**, can be **anonymous**, passed as **arguments**, and support **closures**.
- Example:

```scala
val add = (x: Int, y: Int) => x + y
println(add(10,15))  // 25

def applyFunc(f: Int => Int, v: Int): Int = f(v)
println(applyFunc(x => x * x, 7)) // 49
```

- Explanation:
  Highlight function literals, higher-order functions, and immutability.

5. Objects and Inheritance

- Description:
  Scala supports **object-oriented programming** with **classes**, **objects** (singleton instances), traits (interfaces), and inheritance.

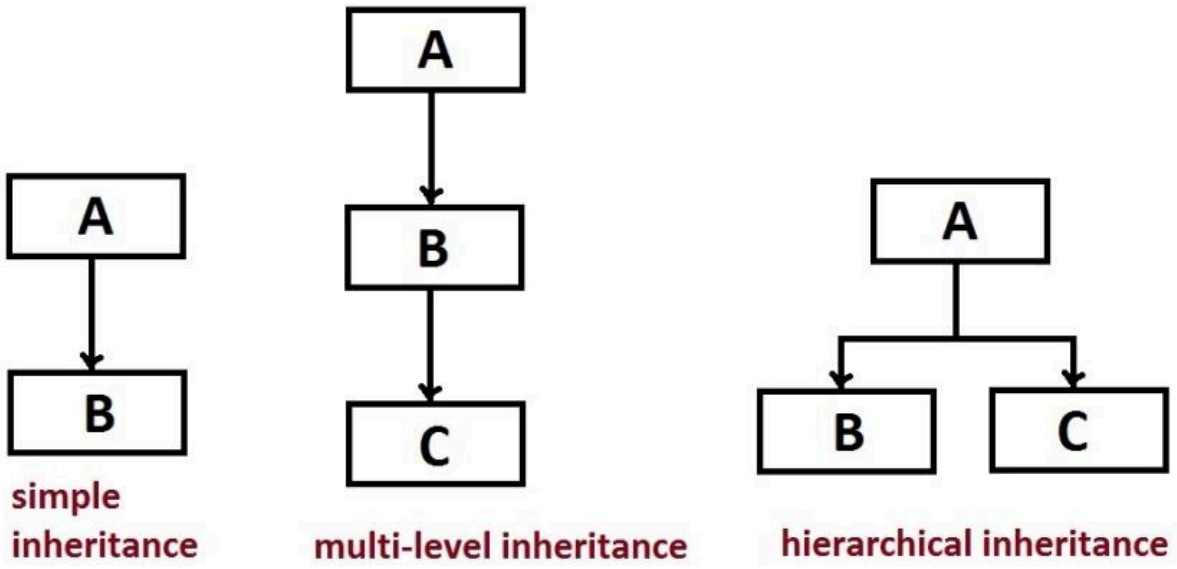


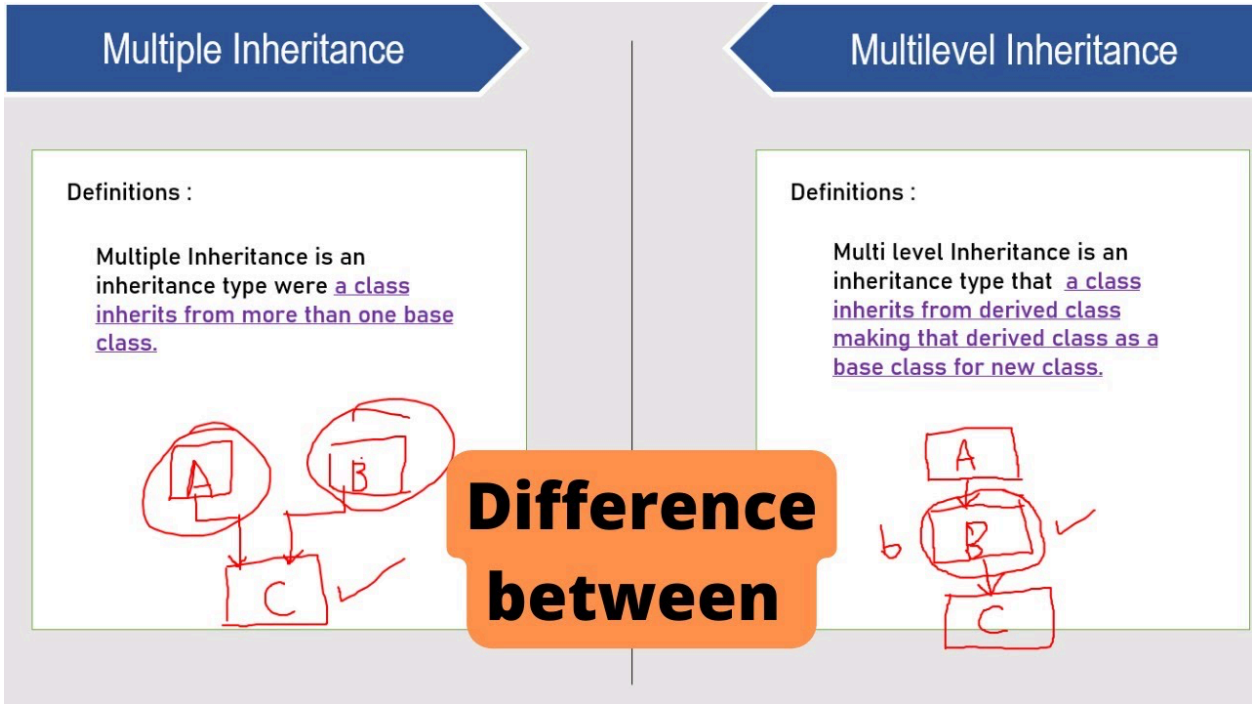
- Example:

```scala
object CRMSystem {

 def main(args: Array[String]): Unit = {

   //Create a instance of the Customer Class and pass the arguments custmoerId,name,email
```

```
   val customer1 = new Customer("SBI001", "Dhivya", "dhivya@gmail.com")
   println(customer1.getCustomerDetails())

   customer1.updateAddress("123 Dhivya Reddy St, Hyderabad")
   println(customer1.getCustomerDetails())

   customer1.sendPromotionalEmail("Special Offer!", "Dear Dhivya, enjoy 20% off
your next purchase!")

   val customer2 = new Customer("SBI002", "Shiva", "Shiva@hotmail.com")
   println(customer2.getCustomerDetails())
 }
}

class Customer(val customerId: String, var name: String, var email: String) {

 private var address: Option[String] = None //opetional variable

   def updateAddress(newAddress: String): Unit = {
     address = Some(newAddress)
     println(s"Address updated for $name to $newAddress")
   }

 def getCustomerDetails(): String = {
   val addr = address.getOrElse("No address provided")
   s"Customer ID: $customerId, Name: $name, Email: $email, Address: $addr"
```

```
  }

  def sendPromotionalEmail(subject: String, body: String): Unit = {
    println(s"Sending email to $email with subject: $subject and body: $body")
    // In a real system, this would integrate with an email sending service
  }
}
```





- Explanation:
  Explain class inheritance, method overriding, and trait usage.

Summary:

Code	Meaning
Some("value")	A present value exists
None	No value is available
Option[String]	A container that may or may not have a String
address = Some(newAddress)	Setting the optional address to a value

Why use `Option` instead of `null`?

Scala discourages using `null`. Instead:

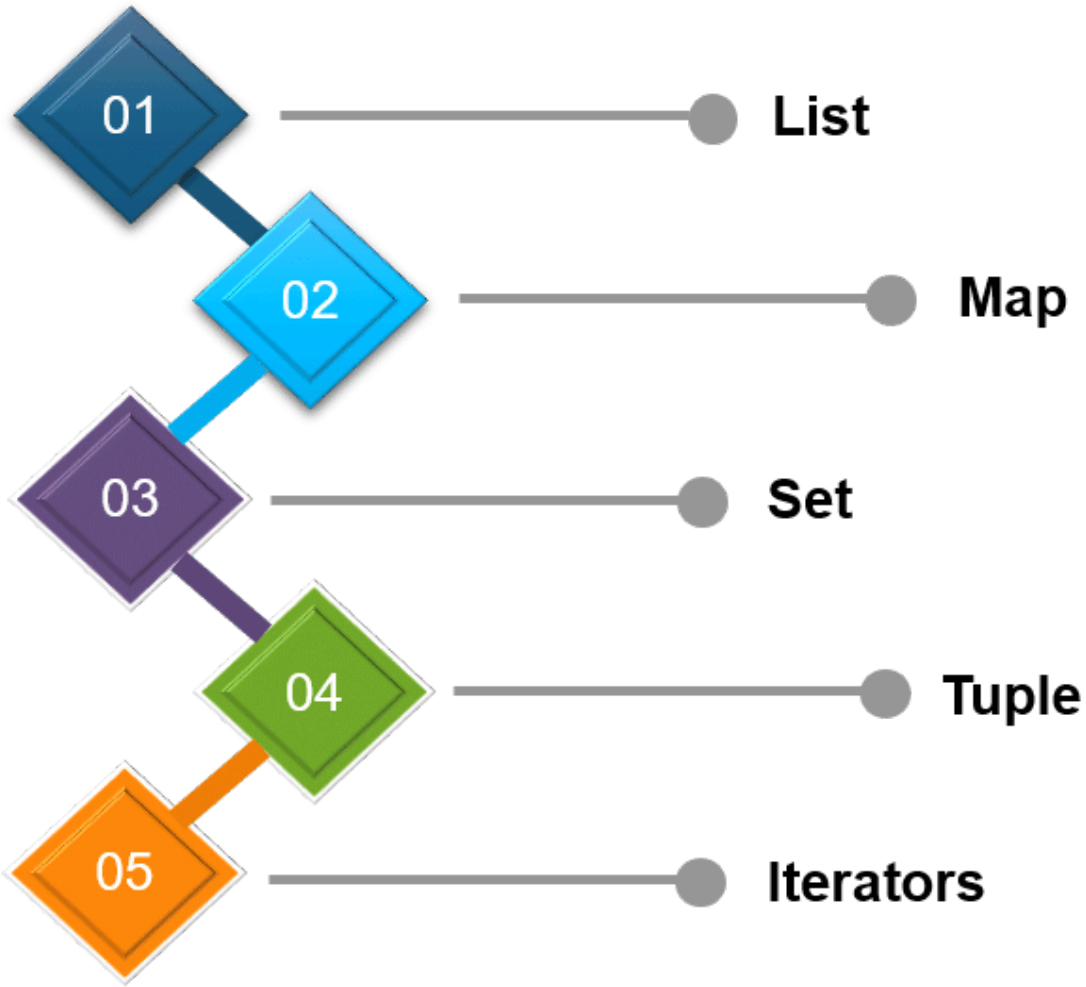
- `Option` forces you to **explicitly handle the absence of a value**.
- `address.getOrElse("No address provided")` safely gets the value or a default if none is set.

6. Working with Lists and Collections

- Description:
  Scala collections are powerful and immutable by default. Lists and sequences support many higher-order methods like map, filter, reduce.
- Example:

```
val nums = List(1,2,3,4,5)
val doubled = nums.map(_ * 2)
println(doubled)  // List(2, 4, 6, 8, 10)

val evens = nums.filter(_ % 2 == 0)
println(evens)    // List(2, 4)
```

- Explanation:
  Discuss immutability, transformation functions, and collection types.


01
List
02
Map
03
Set
04
Tuple
05
Iterators


Immutability
Ordering
List in Scala
Functional Operations
Thread Safety
Heterogeneous

7. Core Big Data and Scala Topics

- Description:
  Cover fundamental Big Data tooling with Scala, including setup and common workflows.
- Key Sections:
  - Scala vs Java: similarities and differences
  - Functional Programming vs traditional programming (Scala vs Python)
  - Configuring Scala with IntelliJ IDEA
  - Scala design patterns
  - Scala 2 classes and objects
  - Asynchronous programming
  - Environment setup: Scala, Hadoop, Hive, Spark
  - Hadoop HDFS commands overview
  - Apache Spark 2 with Scala: DataFrames, SQL, transformations, joins, windowing

8. Scala vs Java – Similarities & Differences

- Description:
  Scala syntax is concise; it combines FP and OOP, whereas Java is mainly OOP.
- Example:
  Java-style:

```
public int add(int x, int y) {
  return x + y;
}
```

Scala-style:

```
def add(x: Int, y: Int): Int = x + y
```

- Explanation:
  Highlight type inference, concise syntax, and built-in FP support.

9. Functional Programming vs Traditional Programming (Scala vs Python)

- Description:
  Scala encourages immutability and purity, while Python supports procedural and OO styles with some functional features.
- Example:
  Scala immutable list:

```scala
val list = List(1, 2, 3)
val newList = list.map(_ * 2)
```

Python list mutability example:

```python
lst = [1, 2, 3]
lst = [x * 2 for x in lst]
```

- Explanation:
  Discuss implications for concurrency and parallelism.

10. Setup Scala with IntelliJ IDEA (including environment)

- Description:
  Guide for installing Scala plugin on IntelliJ, creating a Scala project, setting up sbt build tool.
- https://docs.scala-lang.org/getting-started/intellij-track/getting-started-with-scala-in-intellij.html

11. Scala Design Patterns

- Description:
  Focus on common Scala patterns like companion objects, case classes, traits, and pattern matching.

12. Working Examples on Scala and Spark

- Provide code snippets for Spark data processing tasks from your list, with explanations and sample datasets.

Additional Recommended Sections

- Error Handling in Scala (try-catch, Either, Option)
- Pattern Matching and **Case** Classes (powerful alternatives to Java-style switches)
- Asynchronous Programming in Scala (Futures and Promises)
- Testing Scala Code (ScalaTest or Specs2 introduction)
- Best Practices & Resources for learning Scala and Big Data tools