Scala Important Topics

1. Right Associative Operators
2. String Interpolation
3. Lambdas
4. Partially Applied Functions
5. Higher Order Functions
6. Function Currying
7. Partial Functions
8. Pattern Matching
9. Higher Kinded Types
10. Implicits Typeclasses
11. For Comprehensions
12. Algebraic Data Types (ADTs)
13. Case Classes
14. Top Level Definitions
15. Exports

✅ 1. Right-Associative Operators

In Scala, any method that ends with `:` is **right-associative**.

```
// Example
val list = 1 :: 2 :: 3 :: Nil  // :: is right-associative

// is interpreted as:
```

```
val list = 1 :: (2 :: (3 :: Nil))

println(list) // List(1, 2, 3)
```

Explanation:

- By default, operators in Scala associate to the left (left-associative).
- But if an operator's name ends with `:`, it is treated as right-associative.
- This affects how expressions are parsed and evaluated.

How it works:

For an expression like:

```
a :: b :: c :: Nil
```

Since `::` ends with a colon, it associates from the right:

```
a :: (b :: (c :: Nil))
```

Under the hood:

- The Scala compiler transforms `a :: b` as `b.::(a)` for right-associative operators.
- Instead of `a.::(b)` as would be typical for left-associative methods.
- That means the method is invoked on the right operand with the left operand passed as parameter.

Summary:

- Operators ending with `:` are right-associative.
- Their methods are called on the right operand, with the left operand as an argument.
- Useful for collection-building operators like `::` in lists.

✅ 2. String Interpolation

Allows embedding expressions inside string literals.

```
val name = "Shivam"
val age = 30

println(s"My name is $name and I am $age years old.")
println(f"My age is $age%.2f years") // formatted string
```

String Interpolation in Scala

String interpolation is a convenient way to embed expressions directly inside string literals. Instead of concatenating strings and variables manually, you can write expressions inside a string with a neat syntax and Scala will evaluate and substitute them.

Types of String Interpolators in Scala

1. `s` interpolator (most common)
   Embeds variables or expressions inside a string by prefixing the string with `s`.

```
val name = "Shivam"
val age = 21
println(s"My name is $name and I am $age years old.")
```

2. Output:
   `My name is Shivam and I am 21 years old.`
   - You can embed simple variables like `$name`.
   - For expressions, use `${...}`, e.g. `${age + 5}`.
3. `f` interpolator
   Similar to `s`, but allows formatted output like `printf` style.

```
val height = 1.75
println(f"$name%s is $height%2.2f meters tall")
```

4. Output:
   `Shivam is 1.75 meters tall`
   Format specifiers are used after `%`:
   - `%s` for strings
   - `%d` for integers
   - `%f` for floating points with precision like `%2.2f` (2 digits width, 2 decimals)
5. `raw` interpolator
   Prints the string literally without interpreting escape sequences like `\n`.

```
println(raw"This \n is not a newline")
```

6. Output:
   `This \n is not a newline` (literally with backslash-n)

Why use String Interpolation?

- Cleaner syntax compared to concatenation:

```
"Hello, " + name + "!"
```

- Enables embedding expressions easily:

```
s"2 + 2 = ${2 + 2}"
```

- Combines expression evaluation and string formatting elegantly.
- Improves readability and reduces common errors in string construction.

More Examples

```
val x = 10
val y = 20

// Simple variables inside string
println(s"x = $x, y = $y")

// Expression inside string
println(s"Sum = ${x + y}")

// Formatted output
println(f"Pi approx = ${Math.PI}%.3f")

// Raw string example
println(raw"This is a tab:\t and not tabulation")
```

Outputs:

```
x = 10, y = 20
```

```
Sum = 30
Pi approx = 3.142
This is a tab:\t and not tabulation
```

✅ 3. Lambdas (Anonymous Functions)

```
val add = (x: Int, y: Int) => x + y
println(add(2, 3)) // Output: 5
```

In Scala, a lambda (also called an anonymous function) is a concise way to define a function without giving it a name. It can be used whenever a function value is needed, especially as arguments to higher-order functions like `map`, `filter`, or `reduce`.

Key Points about Lambdas in Scala:

- Syntax: `(parameters) => expression`
- The arrow `=>` separates the input parameters on the left from the function body on the right.
- Parameters can have explicitly declared types or rely on type inference.
- Lambdas are expressions and can be stored in variables or passed directly to functions.
- Scala encourages using lambdas to write **concise** and **readable** code.

Simple Examples:

```
// Lambda that squares a number
val square = (x: Int) => x * x
println(square(5))  // Output: 25

// Lambda with two parameters
val add = (a: Int, b: Int) => a + b
println(add(3, 4))  // Output: 7

// Lambda with no parameter
```

```scala
val greet = () => "Hello, Scala!"
println(greet())  // Output: Hello, Scala!
```

Lambdas with Collections (Higher-Order Functions):

```scala
val numbers = List(1, 2, 3, 4, 5)

// Using lambda in map to square each element
val squares = numbers.map(x => x * x)
println(squares)  // Output: List(1, 4, 9, 16, 25)

// Using lambda in filter to keep even numbers
val evens = numbers.filter(x => x % 2 == 0)
println(evens)  // Output: List(2, 4)
```

Using Placeholder Syntax for Even Shorter Lambdas:

If the parameters occur only once, Scala lets you use underscores `_` as placeholders:

```scala
val squaresShort = numbers.map(_ * _)
println(squaresShort)  // Output: List(1, 4, 9, 16, 25)
```

Why Use Lambdas?

- They avoid defining named methods for **small**, **one-off** functions.
- They support **functional** programming styles by treating functions as first-class citizens.
- Useful when passing behavior to other functions.

✅ 4. Partially Applied Functions

Partially applied functions (PAF) in Scala are functions where you provide **only some** of the **arguments to a function,** leaving the rest to be **supplied later**. This creates a new **function waiting** for the remaining arguments. It's a way to fix some parameters of a function and reuse it with fewer parameters.

```
def multiply(x: Int, y: Int): Int = x * y
val double = multiply(2, _: Int) // partially applied
println(double(5)) // Output: 10
```

Key idea:

- You supply fewer arguments than the function needs.
- The missing arguments are replaced by underscores `_` as placeholders.
- The result is a function that takes the missing arguments as input.

Explanation with Simple Examples

```
// A function taking 3 Ints and multiplying them
def multiply(a: Int, b: Int, c: Int): Int = a * b * c

// Partially applying it by fixing a=2 and b=3, leaving c to be
supplied later
val partialMultiply = multiply(2, 3, _: Int)

// Using the partially applied function by supplying c
println(partialMultiply(4))  // Output: 24 (2*3*4)
```

Here, `partialMultiply` is a new function of type `Int => Int` because only one argument (`c`) remains to be supplied.

Another example: Calculating discounted price

```
def discountPrice(discount: Double, price: Double): Double = (1 -
discount / 100) * price

// Partially apply discount = 20% and leave price to be supplied
later
val discounted = discountPrice(20, _: Double)

println(discounted(1000))  // Output: 800.0
```

```
println(discounted(500))   // Output: 400.0
```

You fix the discount once, and reuse the function for different prices.

Syntax Variations

1. Using underscore placeholder:

```
val add = (a: Int, b: Int) => a + b
val addOne = add(1, _: Int)  // fixes a = 1; b is to be supplied
later

println(addOne(5))  // Output: 6
```

2. Using the function name with underscore to convert to function value without applying any argument (eta-expansion):

```
def multiply(x: Int, y: Int): Int = x * y

val multiplyFunc = multiply _  // function value that takes 2
arguments

println(multiplyFunc(4, 5))  // Output: 20
```

Summary

- Partially applied functions let you fix some arguments and get a new function for the rest.
- Underscore `_` is used as a placeholder for missing arguments.
- Useful in functional programming for creating specialized reusable functions.

Complete Example Program

```
object PartiallyAppliedExample extends App {
  def greeting(greet: String, name: String): String = s"$greet,
$name!"

  // Partially apply greeting fixing the greet argument
```

```
  val sayHello = greeting("Hello", _: String)
  val sayHi = greeting("Hi", _: String)

  println(sayHello("Om"))  // Hello, Om!
  println(sayHi("Nitish"))       // Hi, Nitish!

  // Another example: multiply with 3 parameters
  def multiply(a: Int, b: Int, c: Int): Int = a * b * c
  val multiplyBy6 = multiply(2, 3, _: Int)

  println(multiplyBy6(4))  // 24
}
```

✅ 5. Higher Order Functions

In Scala, Higher Order Functions (HOFs) are functions that either:

- Take one or more functions as parameters, or
- Return a function as a result.

This is possible because in Scala, functions are *first-class values*: you can store them in variables, pass them as arguments, or return them from other functions.
Higher order functions enable flexible, reusable, and expressive code by abstracting behavior, rather than just values.

Takes functions as arguments or returns functions.

```
def applyTwice(f: Int => Int, x: Int): Int = f(f(x))
val result = applyTwice(x => x + 1, 5)
println(result) // Output: 7
```

Explanation

- When a function takes other functions as input, it can customize its behavior by calling them with parameters.
- When a function returns another function, it can create function-producing factories or partially applied functions.

- Many common operations on collections like `map`, `filter`, and `reduce` are implemented as higher order functions.

Simple Scala Examples

1. Function Taking a Function as a Parameter

```
// Define a function that takes a function as an argument
def applyOperation(x: Int, y: Int, op: (Int, Int) => Int): Int =
op(x, y)

val sum = applyOperation(5, 3, (a, b) => a + b)        // sum = 8
val product = applyOperation(5, 3, (a, b) => a * b)    // product
= 15

println(s"Sum: $sum")        // Output: Sum: 8
println(s"Product: $product")// Output: Product: 15
```

Here, `applyOperation` is a higher order function because it accepts a function `op` as its third argument.

2. Function Returning Another Function

```
// Function that returns a function adding a fixed number
def makeAdder(x: Int): Int => Int = {
  (y: Int) => x + y   // anonymous function returned
}

val add10 = makeAdder(10)
println(add10(5))  // Output: 15
```

3. Using Higher Order Functions on Collections

```
val numbers = List(1, 2, 3, 4, 5)
```

```scala
// map is a higher order function: it takes a function to apply to
each element
val squares = numbers.map(x => x * x)

println(squares)  // Output: List(1, 4, 9, 16, 25)

// filter takes a predicate function to select elements
val evens = numbers.filter(_ % 2 == 0)

println(evens)    // Output: List(2, 4)
```

Higher Order Function — A Classic Example

```scala
object HigherOrderFunctionExample extends App {
  // A higher order function that takes a formatter function on
values of type R
  def applyFormatter[R](f: R => String, value: R): String =
f(value)

  // A simple formatter function
  def formatNumber(num: Double): String = s"The number is $num"

  val result = applyFormatter(formatNumber, 42.0)
  println(result)  // Output: The number is 42.0
}
```

Summary

- Higher order functions provide a powerful abstraction by manipulating functions as values.
- They are commonly used for collection processing, event handling, callbacks, and creating function factories.
- Scala standard library is rich with higher order functions, boosting concise and expressive code.

✅ 6. Function Currying

In Scala, Higher Order Functions (HOFs) are functions that either:

- Take one or more **functions as parameters**, or
- **Return a function** as a **result**.

This is possible because in Scala, functions are *first-class values*: you can **store** them in **variables**, **pass** them as **arguments**, or return them from other functions.
Higher order functions enable **flexible, reusable, and expressive code** by **abstracting behavior,** rather than just values.

```
def multiply(x: Int)(y: Int): Int = x * y
val multiplyBy2 = multiply(2) _
println(multiplyBy2(3)) // Output: 6
```

Explanation

- When a function takes other functions as input, it can customize its behavior by calling them with parameters.
- When a function returns another function, it can create function-producing factories or partially applied functions.
- Many common operations on collections like `map`, `filter`, and `reduce` are implemented as higher order functions.

Simple Scala Examples

1. Function Taking a Function as a Parameter

```
// Define a function that takes a function as an argument
def applyOperation(x: Int, y: Int, op: (Int, Int) => Int): Int =
op(x, y)

val sum = applyOperation(5, 3, (a, b) => a + b)        // sum = 8
val product = applyOperation(5, 3, (a, b) => a * b)    // product
= 15

println(s"Sum: $sum")        // Output: Sum: 8
```

```
println(s"Product: $product")// Output: Product: 15
```

Here, `applyOperation` is a higher order function because it accepts a function `op` as its third argument.

2. Function Returning Another Function

```
scala
// Function that returns a function adding a fixed number
def makeAdder(x: Int): Int => Int = {
  (y: Int) => x + y   // anonymous function returned
}

val add10 = makeAdder(10)
println(add10(5))  // Output: 15
```

3. Using Higher Order Functions on Collections

```
val numbers = List(1, 2, 3, 4, 5)

// map is a higher order function: it takes a function to apply to
each element
val squares = numbers.map(x => x * x)

println(squares)  // Output: List(1, 4, 9, 16, 25)

// filter takes a predicate function to select elements
val evens = numbers.filter(_ % 2 == 0)

println(evens)    // Output: List(2, 4)
```

Higher Order Function — A Classic Example

```
object HigherOrderFunctionExample extends App {
```

```
  // A higher order function that takes a formatter function on
values of type R
  def applyFormatter[R](f: R => String, value: R): String =
f(value)

  // A simple formatter function
  def formatNumber(num: Double): String = s"The number is $num"

  val result = applyFormatter(formatNumber, 42.0)
  println(result)  // Output: The number is 42.0
}
```

Summary

- Higher order functions provide a **powerful abstraction** by **manipulating functions** as values.
- They are commonly used for **collection processing, event handling, callbacks,** and **creating function factories**.
- Scala standard library is rich with higher order functions, boosting **concise** and **expressive** code.

✅ 7. Partial Functions

Partial Functions in Scala are functions that are not necessarily defined for every possible input of their input type. Instead, they are defined only for a subset of input values and can explicitly indicate whether they are applicable to a given input or not. This contrasts with total functions that must handle every possible input.

```
val divide: PartialFunction[Int, Int] = {
  case x if x != 0 => 100 / x
}

println(divide.isDefinedAt(0)) // false
println(divide(5))             // 20
```

Key Characteristics:

- Represented by the trait `PartialFunction[A, B]`, where `A` is the input type and `B` is the output type.
- Provide a method `isDefinedAt(x: A): Boolean` to check if the function can handle a particular input.
- Provide an `apply(x: A): B` method to compute the output for defined inputs.
- If applied to an input where the function is not defined, it throws a `MatchError`.
- Often implemented using pattern matching with the `case` keyword.
- Useful for selectively handling input patterns and can be chained together to cover different cases.

Why use Partial Functions?

- They express that a function is only valid for some inputs, improving safety and clarity.
- Cleaner and more concise than manual checks combined with regular functions.
- Commonly used in functional programming idioms and APIs like collections' `collect` method.
- Allow composition via methods like `orElse` to chain multiple partial functions.

Basic Scala Example of a Partial Function

```scala
val partialFunc: PartialFunction[Int, String] = {
  case 1 => "one"
  case 2 => "two"
  case 3 => "three"
}

// Check if the function is defined for some inputs
println(partialFunc.isDefinedAt(2))  // true
println(partialFunc.isDefinedAt(5))  // false

// Apply for defined input
println(partialFunc(2)) // prints "two"

// Applying for undefined input throws error
// println(partialFunc(5)) // Throws scala.MatchError

// Safer way using isDefinedAt
```

```scala
if (partialFunc.isDefinedAt(5))
  println(partialFunc(5))
else
  println("Input not supported")
```

Using Partial Functions with Collections

Partial functions are often used with the `collect` method on collections, which filters and maps elements based on the partial function's domain.

```scala
val nums = List(1, 2, 3, 4, 5, 6)

// Define partial function to double even numbers only
val doubleEvens: PartialFunction[Int, Int] = {
  case x if x % 2 == 0 => x * 2
}

// Apply with collect (only even numbers are transformed)
val doubled = nums.collect(doubleEvens)

println(doubled)  // Output: List(4, 8, 12)
```

Combining Partial Functions with `orElse`

You can chain partial functions to handle more input cases:

```scala
val pf1: PartialFunction[Int, String] = {
  case 1 => "one"
  case 2 => "two"
}

val pf2: PartialFunction[Int, String] = {
  case 3 => "three"
  case 4 => "four"
}
```

```scala
val combined = pf1.orElse(pf2)

println(combined(3)) // "three"
println(combined.isDefinedAt(5)) // false
```

Implementing PartialFunction Trait Manually

```scala
scala
val customPF = new PartialFunction[Int, String] {
  def isDefinedAt(x: Int): Boolean = x > 0 && x < 5
  def apply(x: Int): String = s"Number $x"
}

println(customPF.isDefinedAt(3)) // true
println(customPF(3))             // Number 3
println(customPF.isDefinedAt(6)) // false
// customPF(6) // would throw MatchError if uncommented
```

Summary

Partial Functions:

- Are functions only defined for certain inputs.
- Have `isDefinedAt` to check applicability.
- Support pattern matching style definition.
- Useful for concise, safe handling of subset cases.
- Can be composed and applied to collections effectively.

✅ 8. Pattern Matching

Pattern matching in Scala is a powerful feature that lets you check a value against a series of patterns and execute code based on which pattern matches. It is similar to a switch-case statement in other languages but far more expressive, supporting constants, types, data structures, and even extracting values.

```scala
def greet(person: Any): String = person match {
```

```
  case "Raushan"         => "Hello Raushan!"
  case age: Int if age > 18 => "Adult!"
  case _              => "Unknown"
}

println(greet("Raushan"))
println(greet(20))
```

Basic Syntax

```
valueToMatch match {
  case pattern1 => // code if pattern1 matches
  case pattern2 => // code if pattern2 matches
  ...
  case _ => // default case, matches anything
}
```

- `valueToMatch` is the expression or value you want to match.
- Each `case` specifies a pattern to match against and the code to run if it matches.
- The underscore `_` acts as a wildcard or default catch-all case.

Simple Example

```
val x = 10

val result = x match {
  case 1 => "One"
  case 10 => "Ten"
  case _ => "Other"
}

println(result)  // Output: Ten
```

Patterns can be:

- Constants — matching specific literal values.
- Variables — capture parts of the matched value for use.
- Tuples — match and extract elements from tuples.
- Case Classes — recursively match structured data.
- Sequences and Lists — match on elements with wildcards.
- Types — match based on the type of the value.
- Guards — add conditions using `if` to a case.
- Wildcards — `_` matches anything.

Example: Matching case classes and guards

```
sealed trait Animal
case class Dog(name: String, age: Int) extends Animal
case class Cat(name: String) extends Animal

def animalInfo(animal: Animal): String = animal match {
  case Dog(name, age) if age > 5 => s"$name is an old dog"
  case Dog(name, _) => s"$name is a young dog"
  case Cat(name) => s"$name is a cat"
  case _ => "Unknown animal"
}

println(animalInfo(Dog("Buddy", 8)))   // Buddy is an old dog
println(animalInfo(Dog("Max", 3)))     // Max is a young dog
println(animalInfo(Cat("Kitty")))      // Kitty is a cat
```

Example: Matching tuples

```
val pair = (2, "Scala")

pair match {
  case (1, "Java") => println("Java one")
  case (2, lang) => println(s"Got the number 2 and language
$lang")
  case _ => println("Something else")
```

```
}
```

Example: Matching types

```
def testType(x: Any): String = x match {
  case s: String => s"String: $s"
  case i: Int => s"Int: $i"
  case _ => "Unknown"
}
```

Summary:

Pattern matching lets you concisely branch on data structure shape, value, and type, extracting data as needed. It's widely used in Scala for decomposition of complex data and as an alternative to verbose conditionals.

✅ 9. Higher Kinded Types (HKT)

Types that take type constructors as parameters.

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

// Example for List
object ListFunctor extends Functor[List] {
  def map[A, B](list: List[A])(f: A => B): List[B] =
list.map(f)
}
```

Higher Kinded Types (HKTs) in Scala are a powerful type system feature that allows you to abstract not just over **concrete types**, but over type **constructors—types** that themselves take type parameters. In other words, HKTs let you define generic interfaces or traits that work with **generic containers or**

type constructors like `List`, `Option`, `Seq`, etc., rather than specific concrete types like `List[Int]` or `Option[String]`.

What is a Higher Kinded Type?

- A normal generic type is something like `List[Int]` where `List` is a type constructor and `Int`is a concrete type.
- A higher-kinded type abstracts over the type constructor itself. For example, `F[_]` means "a type constructor that takes one type parameter" (like `List` or `Option`).
- This allows you to write generic code that works uniformly with any container type that is parametrized by another type.

Simple Analogy:

- Normal generic type parameter: `T` (a concrete type such as `Int`, `String`, etc.)
- Higher kinded type parameter: `F[_]` (a type constructor like `List[_]`, or `Option[_]` that itself takes a type parameter)

Example in Scala:

```
// Define a trait that abstracts over type constructors with one
type parameter
trait Container[F[_]] {
  def wrap[A](a: A): F[A]
  def unwrap[A](fa: F[A]): A
}

// Implement Container for Seq
object SeqContainer extends Container[Seq] {
  def wrap[A](a: A): Seq[A] = Seq(a)
  def unwrap[A](fa: Seq[A]): A = fa.head
}

// Use it
val wrappedSeq = SeqContainer.wrap(42)        // Seq(42)
val unwrappedSeq = SeqContainer.unwrap(wrappedSeq)  // 42

println(wrappedSeq)
```

```
println(unwrappedSeq)
```

Here, `F[_]` is a higher-kinded type parameter, representing any generic container type that takes one type argument.

More Advanced Example: Functor with Higher Kinded Types

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

// Functor instance for List
object ListFunctor extends Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f)
}

// Using Functor
val nums = List(1, 2, 3)
val doubled = ListFunctor.map(nums)(_ * 2)
println(doubled)  // Output: List(2, 4, 6)
```

This `Functor` trait abstracts over any type constructor `F[_]` and defines a `map` operation. This pattern enables writing generic functional abstractions.

Summary

- Higher Kinded Types let you abstract over generic type constructors, not just concrete types.
- You write code that can manipulate containers like `List`, `Option`, or custom generic types uniformly.
- The typical syntax involves type parameters of the form `F[_]` or `G[_, _]` for types that take one or more type parameters.
- They unlock powerful functional programming idioms like Functors, Monads, and more.

✅ 10. Implicits

Implicits in Scala is a powerful language feature that allows the compiler to automatically supply certain parameters or perform conversions without the need for explicit invocation by the programmer. It enables cleaner, more concise, and expressive code by letting the compiler fill in certain arguments or conversions "behind the scenes" based on types and scope.

```
implicit val defaultName: String = "Guest"

def greet(implicit name: String): String = s"Welcome,
$name!"

println(greet) // Uses implicit defaultName
```

There are several kinds of implicits in Scala:

1. Implicit Parameters and Arguments:
   Methods can declare one or more parameters as implicit. When you call such methods without explicitly passing those parameters, Scala looks in the current scope for implicit values of the expected type and passes them automatically.
2. Implicit Conversions:
   The compiler can automatically convert one type into another if an implicit conversion function (or implicit class) is in scope. This is useful for adding methods to existing types or converting between types without explicit casts.
3. Implicit Classes:
   Introduced in Scala 2.10, implicit classes provide a convenient syntax for defining implicit conversions that wrap a value inside an enriched class, enabling extension methods.

1. Implicit Parameters and Arguments

You mark parameters as `implicit` in a method definition. When you call such a method without providing those parameters, Scala will search for implicit values in the current scope matching the expected types and supply them.

```
object ImplicitExample extends App {
  implicit val defaultMultiplier: Int = 3

  def multiply(x: Int)(implicit multiplier: Int): Int = x *
multiplier
```

```
  val result1 = multiply(10)          // implicit multiplier=3 is
passed automatically
  val result2 = multiply(10)(5)       // explicitly pass
multiplier=5

  println(result1)  // Output: 30
  println(result2)  // Output: 50
}
```

Explanation:

- `multiply` expects an implicit `Int` parameter.
- Because `defaultMultiplier` is marked as implicit and in the scope, it is passed automatically in `multiply(10)`.
- You can still override by explicitly passing the parameter.

2. Implicit Conversions

You can define an implicit function to convert one type to another. The compiler automatically inserts this conversion when needed.

```
object ImplicitConversionExample extends App {
  implicit def intToString(x: Int): String = s"Number: $x"

  def printString(s: String): Unit = println(s)

  printString(42)  // 42 is implicitly converted to "Number: 42"
}
```

3. Implicit Classes (to add extension methods)

A convenient way to create implicit conversions that add methods to existing types.

```
object ImplicitClassExample extends App {
  implicit class RichInt(val self: Int) extends AnyVal {
    def times(f: => Unit): Unit = {
      for (_ <- 1 to self) f
    }
```

```
  }

  3.times(println("Hello"))  // Prints "Hello" 3 times
}
```

Full Example Combining Implicits

```
object ScalaImplicitsDemo extends App {
  // Implicit value to be used as default multiplier
  implicit val defaultMultiplier: Int = 2

  // Function with implicit parameter
  def multiply(x: Int)(implicit multiplier: Int): Int = x *
multiplier

  // Implicit conversion from Int to a class with extra methods
  implicit class IntOps(val x: Int) extends AnyVal {
    def squared: Int = x * x
  }

  // Using implicit parameter
  println(multiply(5))           // Uses defaultMultiplier = 2,
prints 10

  // Overriding implicit parameter explicitly
  println(multiply(5)(10))       // Prints 50

  // Using implicit class added method
  println(4.squared)             // Prints 16
}
```

Summary

- Implicit parameters simplify method signatures by allowing the compiler to fill common parameters automatically.

- Implicit conversions and implicit classes allow extending existing types or converting types seamlessly.
- Implicits make Scala concise and powerful but should be used judiciously because they can make code harder to read if overused or hidden from the reader.

✅ 11. Typeclasses

Typeclasses in Scala are a powerful pattern that enables *ad-hoc polymorphism* — allowing you to extend behavior to types without modifying their source code or relying on inheritance. They let you define operations or methods that can work for many different types in a **flexible**, **modular**, and **type-safe** manner.

```
trait Show[A] {
  def show(value: A): String
}

implicit object IntShow extends Show[Int] {
  def show(value: Int): String = s"Int: $value"
}

def printShow[A](value: A)(implicit s: Show[A]): Unit =
  println(s.show(value))

printShow(42) // Output: Int: 42
```

Conceptual Overview

- A typeclass is defined as a *trait* with one or more abstract methods, parameterized on a type `A`.
- You create typeclass instances by providing implementations of that trait for specific types.
- Functions or methods can then declare a parameter that requires an implicit instance of the typeclass, allowing them to operate generically on any type that has an available instance.
- This pattern is similar to interfaces but typeclasses promote *composition* over *inheritance*, and decouple behavior from types.
- Typeclasses leverage implicit values or given instances (Scala 3) to supply implementations transparently.

Minimal Working Example: `Show` Typeclass for Custom String Representation

Step 1: Define the Typeclass Trait

```
trait Show[A] {
  def show(a: A): String
}
```

This declares a typeclass `Show` with a method `show` which converts a value of type `A` to a `String`.

Step 2: Provide Instances for Specific Types

```
implicit val intShow: Show[Int] = new Show[Int] {
  def show(a: Int): String = s"Int($a)"
}

implicit val stringShow: Show[String] = new Show[String] {
  def show(a: String): String = s"String($a)"
}
```

These implicit instances define how to "show" `Int` and `String` values.

Step 3: Use the Typeclass in Generic Functions

```
def printShow[A](value: A)(implicit sh: Show[A]): Unit = {
  println(sh.show(value))
}

// Or using context bound syntax
def printShow2[A: Show](value: A): Unit = {
  val sh = implicitly[Show[A]]
  println(sh.show(value))
}
```

You can call:

```
printShow(123)         // prints: Int(123)
printShow("hello")     // prints: String(hello)
```

Enrichment with Implicit Class for Convenient Syntax

To enable nicer syntax like `value.show`, define an implicit class:

```
implicit class ShowOps[A](value: A)(implicit sh: Show[A]) {
  def show: String = sh.show(value)
}

println(123.show)         // Int(123)
println("world".show)     // String(world)
```

Complete Example Program

```
object TypeclassExample extends App {
  trait Show[A] {
    def show(a: A): String
  }

  implicit val intShow: Show[Int] = (a: Int) => s"Int($a)"
  implicit val stringShow: Show[String] = (a: String) =>
s"String($a)"

  implicit class ShowOps[A](value: A)(implicit sh: Show[A]) {
    def show: String = sh.show(value)
  }

  def printShow[A: Show](value: A): Unit = {
    println(value.show)
  }
```

```
  printShow(42)           // Output: Int(42)
  printShow("Scala")      // Output: String(Scala)

  println(100.show)       // Output: Int(100)
  println("Hello".show)   // Output: String(Hello)
}
```

Why Use Typeclasses?

- Extensibility: Adding new operations or new types without modifying existing code.
- Separation of Concerns: Behavior (the “typeclass”) is separated from data types.
- Ad-Hoc Polymorphism: Like interfaces in OOP but more flexible; you can add typeclass instances for types you don’t own.
- Compile-time Safety: Typeclass instances are resolved at compile time via implicits.
- Modularity: Easy to maintain and test separate instances.

Additional Real-World Uses

- Serialization typeclasses (e.g., JSON encoding)
- Equality comparison abstractions
- Numeric typeclasses for algebraic structures (monoids, functors, monads)
- Logging or printing behavior

✅ 12. For Comprehensions

```
val list1 = List(1, 2)
val list2 = List("a", "b")

val combined = for {
  x <- list1
  y <- list2
} yield s"$x$y"
```

```
println(combined) // List(1a, 1b, 2a, 2b)
```

✅ 13. Algebraic Data Types (ADTs)

Algebraic Data Types (ADTs) in Scala are a way to model data by combining simpler types into more complex ones using two fundamental patterns: **sum types** and **product types**. They are especially powerful when used with pattern matching for **exhaustive and typesafe** handling of data.

What are Algebraic Data Types?

- Product Types ("and" types): Combine multiple values into one.
  In Scala, product types are typically case classes that hold multiple fields.
  Example: A `Point` with an `x` and `y` coordinate.
  The cardinality (number of possible instances) is the product of the cardinalities of the fields.
- Sum Types ("or" types): Define a type that can be one of several alternatives.
  In Scala, sum types are modeled using sealed traits or abstract classes with multiple extending case classes or case objects.
  Example: A `Shape` that can be a `Circle` or a `Rectangle`.
  The cardinality is the sum of the cardinalities of its variant types.

Using ADTs helps make illegal states unrepresentable and provides exhaustive compiler checks when used with pattern matching.

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(w: Double, h: Double) extends Shape

def area(shape: Shape): Double = shape match {
  case Circle(r)      => math.Pi * r * r
  case Rectangle(w,h) => w * h
}
```

Simple Scala Example Illustrating ADTs

Let's create a small domain of shapes and commands using ADTs, and explain them:

```
scala
// Sum Type: Shape can be either Circle or Rectangle
```

```scala
sealed trait Shape

case class Circle(radius: Double) extends Shape  // Product type:
has radius
case class Rectangle(width: Double, height: Double) extends Shape
// Product type: width and height

// Function to calculate area using pattern matching
def area(shape: Shape): Double = shape match {
  case Circle(r) => Math.PI * r * r
  case Rectangle(w, h) => w * h
}

// Sum Type: Command can be Move or Rotate, each with parameters
specific to the command
sealed trait Command

case class Move(distance: Int) extends Command
case class Rotate(angle: Int) extends Command

// Process command using pattern matching
def execute(cmd: Command): Unit = cmd match {
  case Move(d) => println(s"Moving forward $d meters")
  case Rotate(a) => println(s"Rotating $a degrees")
}

// Examples of usage
val c = Circle(2.0)
val r = Rectangle(3.0, 4.0)

println(area(c))   // Output: 12.566370614359172
println(area(r))   // Output: 12.0

val commands: List[Command] = List(Move(10), Rotate(90))
commands.foreach(execute)
// Output:
```

```
// Moving forward 10 meters
// Rotating 90 degrees
```

Explanation of the Program

- Sum Type with Sealed Trait:
  The `sealed trait Shape` means all possible subtypes (`Circle`, `Rectangle`) must be declared in the same file, enabling compiler exhaustiveness checking. A `Shape` instance can be *either* a `Circle` *or* a `Rectangle`.
- Product Types with Case Classes:
  `Circle` contains one piece of data: radius (a `Double`); `Rectangle` contains two: width and height. These group related data together, representing a single value with multiple components (hence "product").
- Pattern Matching on ADTs:
  The `area` function uses pattern matching to handle each case of `Shape`. The compiler can warn if a case is missing because `Shape` is sealed.
- Command ADT Example:
  The `Command` trait is a sum type with two concrete variants `Move` and `Rotate`, each product types with relevant parameters. Pattern matching on a `Command` lets you handle all variations cleanly.

Why Use ADTs?

- Expressiveness: Clearly model complex domains where data can take different forms.
- Exhaustiveness Checking: The compiler verifies that all cases are handled, reducing bugs.
- Eliminates Invalid States: By modeling only valid combinations, illegal states become unrepresentable.
- Functional Style: Works very naturally with pattern matching—a fundamental Scala technique.

✅ 14. Case Classes

Used to model immutable data with pattern matching support.

```
case class Person(name: String, age: Int)

val p = Person("Dani", 40)
println(p.name)
```

Case classes in Scala are a special type of class designed primarily for modeling immutable data. They provide a concise syntax to define classes that hold data and come with several powerful features generated by the compiler automatically, reducing boilerplate code and improving code clarity and pattern matching.

Concept and Features of Case Classes:

- Immutable by default: Parameters of a case class are by default `val` (immutable).
- No `new` keyword needed: You can instantiate a case class without the `new` keyword, thanks to an auto-generated `apply` method in the companion object.
- Automatic `toString`, `equals`, and `hashCode`: Makes debugging and comparing objects easy.
- Pattern matching support: Auto-generated `unapply` method makes case classes compatible with Scala's powerful pattern matching.
- Copy method: Provides a convenient way to clone an object with some modified fields, without mutating the original.
- Constructor parameters become public fields: You can access them directly with dot notation.

Example Program with Explanation

```
// Define a simple case class Person with two fields
case class Person(name: String, age: Int)

object CaseClassDemo extends App {
  // Instantiate case classes without `new`
  val person1 = Person("Dhivya", 30)
  val person2 = Person("Nikil", 25)

  // Access fields directly
  println(s"Person1's name is ${person1.name} and age is
${person1.age}")

  // toString method prints readable format
  println(person1)  // prints: Person(Dhivya,30)

  // equals and hashCode allow content comparison
  val person3 = Person("Dhivya", 30)
```

```
  println(person1 == person3)  // true (compares field values, not
reference)

  // Copy method to create a new instance with some changes
  val olderDhivya = person1.copy(age = 31)
  println(olderDhivya)  // Person(Dhivya,31)

  // Pattern matching with case classes
  def greet(p: Person): String = p match {
    case Person("Dhivya", age) => s"Hello Dhivya, age $age!"
    case Person(name, _)       => s"Hi $name!"
    case _                     => "Who are you?"
  }

  println(greet(person1))  // Hello Dhivya, age 30!
  println(greet(person2))  // Hi Nikhil!
}
```

Explanation:

- We define a `Person` case class with two fields: `name` (String) and `age` (Int).
- Creating instances is concise; no `new` keyword is required because the compiler generates an `apply` method automatically.
- Fields `name` and `age` are accessible directly since case class constructor parameters are `val`by default.
- Printing an instance invokes the auto-generated `toString` giving a clean readable output.
- Equality (`==`) checks the content, not reference identity.
- The `copy` method creates a new instance with changes (here, increased age), maintaining immutability.
- Pattern matching is straightforward because the compiler generates an `unapply` method enabling extraction of constructor fields.

Case classes are widely used in Scala, especially in functional programming, for modeling immutable data records, database rows, messages, and Sum/Algebraic Data Types combined with sealed traits.

✅ 15. Top Level Definitions

Since Scala 3, you can define methods and variables outside objects.

```
def add(x: Int, y: Int): Int = x + y
val msg = "Top-level function"
```

Top Level Definitions in Scala (especially Scala 3) allow you to write all kinds of definitions—such as values (`val`), variables (`var`), methods (`def`), types (`type`), classes, objects, and extensions—directly at the top level in source files, without needing to wrap them inside an enclosing object, class, or package object.

Explanation

Traditionally in Scala 2, you had to place methods and values inside objects or package objects to organize code and enable JVM compatibility. For example, top-level functions or values outside of classes were not allowed; you needed something like:

```
package mypackage

object Utils {
  def greet(name: String): String = s"Hello, $name"
}
```

Scala 3 relaxes this requirement by allowing you to write:
scala

```
package mypackage

def greet(name: String): String = s"Hello, $name"
```

Or even omit the package and write definitions directly in a source file, so that values, methods, type aliases, and classes appear at the top level naturally.
Internally, the Scala 3 compiler generates a synthetic wrapper object for such top-level definitions (named after the source file), but this is transparent to the user and doesn't affect usage.

Benefits

- Reduced boilerplate—no need for enclosing objects when not needed.
- Cleaner, more natural code organization.

- Ability to mix top-level values, methods, types, and classes freely in source files.
- Simplified organization compared to Scala 2 package objects (which are now deprecated).

Example Program Using Top Level Definitions (Scala 3 style)

```scala
// File: Utilities.scala

// A type alias defined at the top level
type Labelled[T] = (String, T)

// A top-level value
val count: Labelled[Int] = ("count", 42)

// A top-level method
def getCount: Int = count._2

// A case class defined at top level
case class Point(x: Double, y: Double)

// An extension method for Point (top-level extension)
extension (p: Point)
  def distanceToOrigin: Double = Math.sqrt(p.x * p.x + p.y * p.y)

// Using the above top-level definitions in a main method at top
level
@main def runExample(): Unit =
  println(s"Count value: $getCount")

  val p = Point(3.0, 4.0)
  println(s"Point: $p")
  println(s"Distance to origin: ${p.distanceToOrigin}")
```

Explanation of the Program

- `type Labelled[T]` is a type alias declared at the top level.
- `val count` is a top-level immutable value describing a labelled integer.

- `def getCount` is a top-level method returning the integer inside the labelled tuple.
- `case class Point` defines a simple 2D point at the top level.
- An extension method `distanceToOrigin` on `Point` is declared at the top level using Scala 3's extension syntax.
- The `@main def runExample()` defines the program entry point at the top level, printing the count and demonstrating usage of the point and its extension method.

No wrapping objects or classes are needed to hold these definitions, which simplifies file structure and improves readability.
If you use this file with Scala 3, you can compile and run it as-is, and the compiler handles packaging and generation of wrapper objects transparently.

✅ 16. Exports (Scala 3)

Re-export members of inner objects or traits.

What are Exports in Scala?

In Scala 3, `export` clauses help you re-expose or alias selected members of an object or class so that those members appear as if they were directly defined in the exporting class or object. This allows you to avoid writing boilerplate forwarding methods.
In simpler terms, instead of writing many forwarding methods manually that delegate calls to an encapsulated object, you can simply export those members. This feature greatly improves code clarity and reduces redundancy.

Key Points about `export`:

- An `export` statement defines aliases for members of some expression (usually an object or class field).
- The syntax looks like imports but uses the keyword `export`.
- You can export single members, multiple members, all members (`*`), or rename (`as`) or hide (`_`) members during export.
- Exported members behave as if they are part of the exporting class/object from the outside.
- This feature is useful for delegation patterns, wrapping APIs, or adapters.

Simple Example Program Demonstrating `export`

```
// File: ExportExample.scala
```

```
class Printer:
  def print(msg: String): Unit = println(s"Printing: $msg")
  def status: String = "Printer ready"

class Scanner:
  def scan(): String = "Scanned document"
  def status: String = "Scanner ready"

// Copier combines Printer and Scanner capabilities
class Copier:
  private val printer = new Printer
  private val scanner = new Scanner

  // Export selected members from printer and scanner
  export printer.print
  export scanner.scan

  // Hide the status from printer; only expose scanner's status
  export printer.status as _
  export scanner.status

// Usage demonstration
@main def runExportExample(): Unit =
  val copier = new Copier

  // We can call exported methods directly on Copier
  copier.print("Hello World!")   // Delegated to printer.print
  val scanned = copier.scan()    // Delegated to scanner.scan
  println(scanned)

  // Copier exposes scanner's status only
  println(s"Copier status: ${copier.status}")
```

Explanation of the Program:

1. Classes `Printer` and `Scanner`:
   Each defines some methods. Both have a method named `status` with different meanings.
2. Class `Copier`:
   Contains private instances of `Printer` and `Scanner`. Instead of manually writing forwarding methods like:
3. `scala`

```scala
def print(msg: String) = printer.print(msg)
def scan() = scanner.scan()
def status = scanner.status
```

4. The class uses the `export` clause:

```scala
export printer.print
export scanner.scan
export printer.status as _    // hides Printer's status
export scanner.status        // exposes Scanner's status
```

5. This automatically creates forwarding methods to `printer.print` and `scanner.scan`, but only exposes `scanner.status` (hides the `printer.status`).
6. Usage:
   - `copier.print` calls `printer.print` under the hood.
   - `copier.scan` calls `scanner.scan`.
   - `copier.status` returns `scanner.status`.
7. From outside, the `Copier` acts as if it defines those methods directly, but the forwarding is generated automatically by the compiler.

Benefits of `export`:

- Eliminates manual boilerplate forwarding methods.
- Makes delegation/composition patterns cleaner.
- Provides fine-grained control over which members to expose, hide, or rename.
- Maintains encapsulation while providing external interfaces.