scala

new material

Scala Tutorial | What is Scala Programming Language?

1. Objective of Scala Introduction

This Scala Introduction tutorial will help you in understanding basics of Scala programming language. You will learn about Introduction to Scala, History of Scala, Scala features, why Scala is popular and preferred over other programming languages, differences between Java and Scala and best books to master Scala language. So lets begin our journey with the Scala Tutorial.

2. Scala Programming Introduction

Scala is comparatively new to the programming scene, but has become popular very quickly. Below statements with big names show the Scala popularity in the industry:

- If I had to select a language to use other than Java, it would be Scala. – James Gosling, creator of Java
- If I would have seen 'Programming in Scala' book back in 2003, I'd probably have never created Groovy – James Strachan, creator of Groovy
- None other than Scala can be seen as being 'replacement of Java' , and the momentum behind Scala is now unquestionable – Charles Nutter , co-creator of JRuby

Scala is a general purpose language that combines concepts of object-oriented and functional programming languages. It was developed to overcome the problems faced by other languages and can easily be integrated into existing code. If you face any Query in Scala Tutorial, Please Comment. Now Lets see a video Scala Tutorial for your better understanding.

3. Video Scala Tutorial

4. Scala History

Scala was first conceived in 2001 at École Polytechnique Fédérale de Lausanne by Martin Odersky, who was co-creator of Generic Java, javac, and EPFL's Funnel programming language. First public release of Scala came in 2004 which was followed by version 2.0 in March 2006. In 2012 it was awarded the winner of the ScriptBowl contest at the JavaOne conference.

5. Scala Features

Some of the key characteristics of Scala include:

- It is an object-oriented language that supports many traditional design patterns being inherited from existing programming languages.
- It supports functional programming that enables it to handle concurrency and distributed programming at fundamental level.
- Scala is designed to run on JVM platform that helps in directly using Java libraries and other feature rich APIs.
- Scala is statically typed that prevents it from problems of dynamically typing.
- Scala is easy to be implemented into existing java projects as Scala libraries can be used within Java code.
- There is no need to declare variables in Scala as Scala compiler can infer most types of variables.
- Multiple traits can be designated for a class and then their interface and behaviour can be combined.
- It supports first-class objects and anonymous functions.

6. Why Scala Is Popular

One of the key reasons for Scala's success is its close integration with Java. Scala source code is designed in a manner that its compiler can interpret Java classes, and can fully utilize Java libraries, frameworks, and tools. After compilation, Scala programs can run on Java virtual machines and Android. For web-based development projects, Scala can even be compiled to JavaScript.

However, Scala is far beyond an alternative to Java. It is a more concise language that utilizes simple, easy-to-read syntax, and requires just a fraction of the lines of code when compared to a typical Java program. This makes Scala coding faster and provides easier testing.

Due to the above reasons, companies like Linkedin, Twitter, Coursera, Foursquare etc have ported majority of their code bases to Scala. Many open source projects like Apache Spark, Apache Kafka etc use Scala for their Core. The famous Play framework is also developed using Scala.

7. Prerequisites to learn Scala

Scala Programming is based on Java, so if you have knowledge of Java syntax, then it's pretty easy to learn Scala. But if you do not know Java but are aware of any other programming language like C, C++ or Python then also you can learn Scala very quickly.

Any doubt yet in the Scala Tutorial? Please Comment.

8. Scala Vs Java

Scala has set of features that make it completely different from Java language. Refer Guide for Comparison between Java and Scala to learn about the differences in Scala and Java language to understand how Scala is different from Java.

9. Books to learn Scala

There are not as many books on Scala as compared to Java or C, but there are still few books to get you started, regardless of your previous programming experience.

Refer this Comprehensive Guide for Scala books.

This is all on Scala Tutorial.

10. Summary

If you're new to programming, or looking for a fast, modern language that is a combination of object-oriented and functional programming, Scala must be your choice that is definitely a language worth learning. Hope you like the Scala Tutorial. If you like the Tutorial on Scala, Please Comment.

What is Scala? | A Comprehensive Scala Tutorial

1. Comprehensive Scala Tutorial

We feel immense pleasure in welcoming you to yet another series of tutorials- Scala. What is Scala? What can you do with it? What does it look like? These are some of the questions we will answer today in our comprehensive Scala tutorial. We will discuss what is Scala programming, Scala for beginners, history of Scala, Features of Scala, Frameworks of Scala, Applications of Scala, Companies that use Scala, and technologies that are built on Scala. So, let's begin with the Scala Tutorial and learn what is Scala.

2. What is Scala?

Scala is a portmanteau of ‘scalable’ and ‘language’. It is designed to grow with user demand.

A general-purpose programming language, Scala provides support for functional programming and a strong static type system.

Being much like Java., Scala’s source code compiles into Java byte code. The resulting executable code runs on a JVM (Java Virtual Machine). Actually, it supports language interoperability with Java. So, you can reference libraries written in both languages, in both of them. Other similarities between Java and Scala include an object-oriented nature, and a curly-brace syntax.

However, unlike Java, Scala supports features of functional programming-like currying, type-inference, immutability, lazy evaluation, pattern matching, and nested functions. It also supports higher-order functions, where a function can return another, or take it as a parameter. Scala also has an advanced type system that supports algebraic data types, higher-order types, anonymous types, and covariance and contravariance.

Finally, Scala also allows functionality like operator overloading, raw strings, optional parameters, and named parameters. However, it doesn’t support checked exceptions, like Java does.

After knowing what is Scala, you can read our article on **Why Learn Scala?**

Hope now you are clear with What is Scala. Let us see topic of Scala Tutorial: Scala History.

3. History

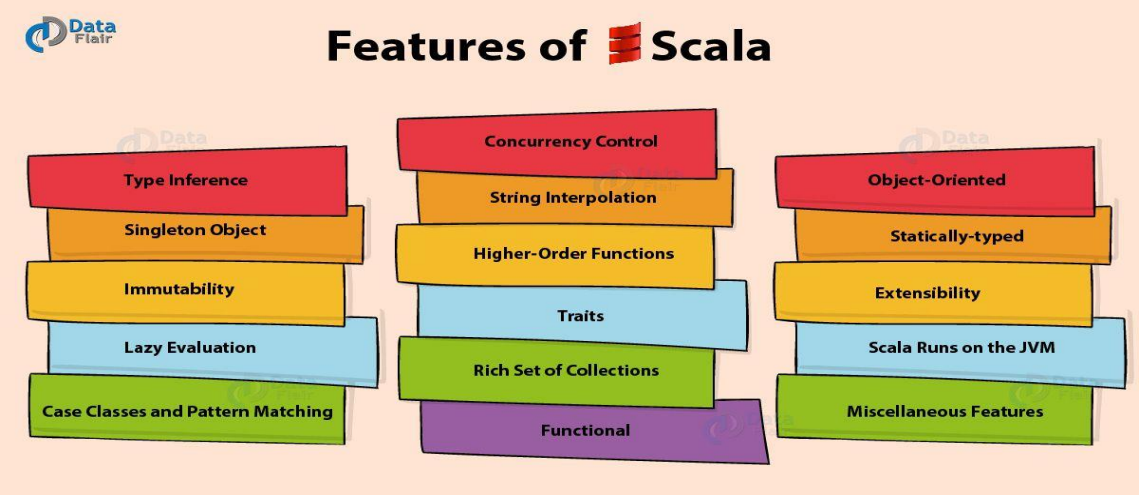
Scala emerged on 20 January 2004, 14 years from now. While its design began in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) (in Lausanne, Switzerland) by Martin Odersky, it saw an internal release in late 2003. In 2004, it released to the public on the Java platform. Further, a second version followed in the March of 2006.

While Java adopted lambda expressions only in 2014, with Java, Scala always supported functional programming fully. On January 17th of 2011, the team for Scala bagged a five-year research grant of over €2.3 million from the European Research Council. Then, on May 12th of 2011, Odersky and collaborators launched Typesafe Inc., which they later renamed to Lightbend Inc. This was to provide commercial support, training, and services for Scala. In 2011, Greylock Partners invested $3 million in Typesafe.

Read: Pros and Cons of Scala

4. Features of Scala

Every language has some unique features which make them best convenience of a particular type of Project. In this Scala Tutorial we have mentioned 14 such features of Scala Programming Language.



a. Type Inference

Type inference means Scala automatically detects(infers) an expression's data type fully or partially. We don't need to declare it ourselves, and this also lets developers omit type annotations. This has no effect on the type checking.

The compiler checks the types of the subexpressions, or of atomic values like 42(Integer), and true(Bool). It aggregates this information to decide the type for the entire expression.

b. Singleton Object

In Scala, you will see a singleton object instead of static variables and methods. A singleton object is a class with a single object in the source file. To declare this, we use the 'object' keyword:

```
object Main extends App {

println("Hello, World!")

}
```

c. Immutability

Every time you declare a variable in Scala, it is immutable by default. This means you cannot modify it. But if you want, you may declare it as mutable. Then, it is possible to change its value.

This property of Scala helps us with concurrency control.

d. Lazy Evaluation

If declared lazy using the 'lazy' keyword, Scala delays complex computation with evaluating an expression until it absolutely needs it. We also call it call-by-need.

This avoids repeated evaluations. And these are the benefits of the same:

1. It lets us define control flow as abstractions instead of primitives.
2. It lets us define potentially infinite data structures.
3. It also improves performance.

Take an example:

```
lazy val images=getImages()
```

e. Case Classes and Pattern Matching

A regular class that is immutable by default, and is decomposable via pattern matching, is a case class. Its parameters are public and immutable by default. We define a case class with the keywords 'case class', an identifier, and a parameter list which can be empty.

```
case class Book(isbn: String)

val frankenstein = Book("978-0486282114")
```

Pattern matching allows us to compare a value against a pattern. This is like a switch-statement or a series of if-else statements in Java.

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match

{

case 0 => "zero"

case 1 => "one"

case 2 => "two"

case _ => "many"

}
```

Any doubt yet in Scala Programming Tutorial? Please Comment.

f. Concurrency Control

The Actor model from Scala's standard library lets us implement concurrency in code. Scala also has a platform/tool, Akka, which is a separate, open-

source framework for Actor-based concurrency. It is possible to combine/distribute Akka's actors with software transactional memory.

g. String Interpolation

This is a form of template-processing. To interpolate a string is to evaluate a string literal consisting of one or more placeholders to yield a result. The corresponding values replace these placeholders.

String interpolation is observed in Scala since version 2.10.0. Three methods it offers for this are- s, f, and raw.

h. Higher-Order Functions

Such functions can return another function, or take it as a parameter. Scala makes it possible by treating its functions as first-class citizens.

```
val salaries = Seq(20000, 70000, 40000)

val doubleSalary = (x: Int) => x * 2

val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

Here, map is a higher-order function.

Higher-order functions also let us implement function compositions and lambdas.

i. Traits

A trait is a type holding certain fields and methods. We define traits using the 'trait' keyword:

```
trait Greeter

{

def greet(name: String): Unit

}
```

You can think of it like a partially implemented interface. You can create a trait with abstract, and optionally, non-abstract methods, and can also combine multiple traits.

j. Rich Set of Collections

The Scala library has a huge set of collections with classes and traits to help collect data into an immutable or a mutable collection. The scala.collection.immutable package holds all immutable collections. They don't allow us to modify data. Likewise, the scala.collection.mutable package holds all the mutable ones.

k. Functional

Scala is a functional language, and treats its functions as first-class citizens. It will let you create higher-order functions, like we've discussed twice earlier in this article. Other than that, it also supports nested functions and methods, and currying. Currying is the act of translating the evaluation of a function that takes multiple arguments, into evaluating a sequence of functions, each with a single argument.

l. Object-Oriented

Scala is object-oriented while also being functional. In it, every value is an object.

m. Statically-typed

You usually won't need to declare redundant type information in your code. Scala will decide that based on the types of subexpressions, or of atomic values. This is in pertinence to type inference.

n. Extensibility

When you're building domain-specific applications, you need domain-specific language extensions too. Scala delivers a combination of language mechanisms. Overall, it makes it easy to smoothly add new language constructs as libraries. Constructs like implicit classes and string interpolation help us do this; we don't need meta-programming features like macros.

o. Scala Runs on the JVM

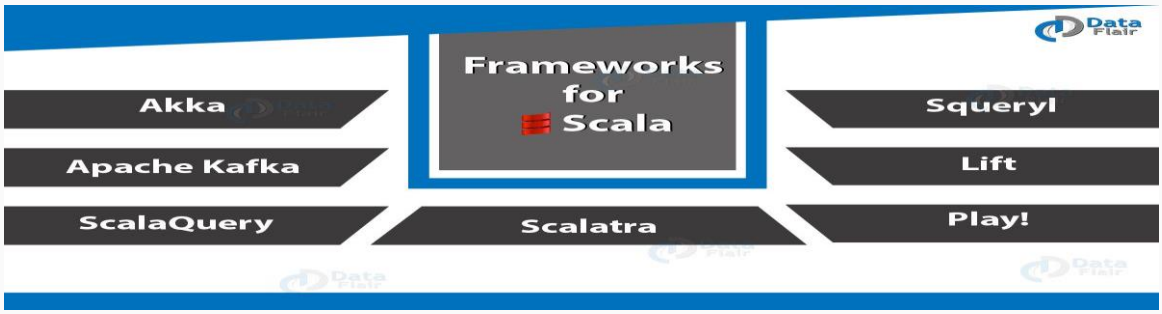
Scala's compiler converts the source into Java byte code that runs on the JVM (Java Virtual Machine).

Read more on Features of Scala.

Next topic in Scala Tutorial is Frameworks for Scala.

5. Frameworks for Scala

Which frameworks and development tools do we have with Scala? Let's see.



a. Akka

https://akka.io/

A free and open-source toolkit and runtime, Akka simplifies constructing concurrent and distributed applications on the JVM. Akka is good with distributed processing.

b. Apache Kafka

https://kafka.apache.org/

Kafka is an open-source stream processing software platform by the Apache Software Foundation. It is written in Scala and Java. From Kafka, you can expect a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka is good with distributed processing too.

c. ScalaQuery

scalaquery.org/

ScalaQuery is a low-level Scala API for database access, composable non-leaky abstractions, and compile-time checking and type-safety. The JDBC API is powerful but verbose.

d. Squeryl

squeryl.org/

Squeryl is a DSL for manipulating database objects from within Scala. It is strongly-typed, declarative, and SQL-like.

e. Lift

https://liftweb.net/

Lift is a free and open-source web framework for Scala. David Pollak created it because he was dissatisfied with some aspects of Ruby on Rails.

f. Play!

https://www.playframework.com/

An open-source web-application framework, Play! is written in Scala.

g. Scalatra

scalatra.org/

Scalatra is yet another free and open-source web application framework that was written in Scala. It is an alternative to the Lift, Play!, and Unfiltered frameworks, and is a port of the Sinatra framework.

Next topic in Scala Tutorial is Applications of Scala.

6. Applications of Scala

So, what can we do with Scala? The following are just some of the things you can build with Scala:

1. Android applications
2. Desktop applications
3. Concurrency and distributed data processing, for instance, Spark
4. Front and back ends of web applications with scala.js
5. Highly concurrent things, like messaging apps, with Akka
6. Distributed computing; because of its concurrency capabilities
7. Scala is used with Hadoop; Map/Reduce programs
8. Big Data and data analysis with Apache Spark
9. Data streaming with Akka
10. Parallel batch processing
11. AWS lambda expression
12. Ad hoc scripting in REPL

7. What Companies Use Scala?

- Apple
- Sony
- Twitter
- Netflix
- LinkedIn
- Tumblr
- Foursquare
- The Guardian
- AirBnB
- Precog
- Klout
- Meetup.com
- Remember the Milk
- The Swiss Bank UBS
- Amazon
- IBM

- Autodesk
- NASA
- Xerox

We have mentioned few top companies that use Scala in this Scala Tutorial, but there are others as well.

8. Hot Technologies That Were Built in Scala

These big names made use of Scala:

Apache Spark

Scalding

Apache Kafka

Apache Samza

Finagle (by Twitter)

Akka

ADAM

Lichess

This was all on Scala Tutorial. Hope you are clear with What is Scala and it's Scala overview

9. Conclusion: Scala Programming Tutorial

So, in this Scala Tutorial we have discussed Scala for beginners, what is Scala programming, Scala for beginners, history of Scala, Features of Scala, Frameworks of Scala, Applications of Scala, Companies that use Scala, and technologies that are built on Scala. This is just the beginning. Walk with us in our journey with Scala; it's going to be fun!

Scala Features – A Comprehensive Guide

1. Introduction to Scala Features

In this Scala programming tutorial, you will learn what is Scala programming language, what is Scala language used for, how scala is both object oriented and functional, how scala is statically typed and extensible and other special Scala features that makes it the choice of programmers and creates Comparison between Java vs Scala.

The name Scala is the abbreviation of the term SCAlable Language. Martin Odersky and his group created the language in 2003 to provide a high-performance, concurrent-ready environment for functional programming and object-oriented programming on the java virtual machine(JVM) platform. Scala is modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant and type-safe way. It smoothly integrates features of object-oriented and functional language. The main features of Scala are as follows –

2. Scala is Object-Oriented

Scala is pure object-oriented language in the sense that every value is an object. Type and behaviour of the object are described by classes and traits.

3. Scala is functional

Scala is also a functional language in the sense that every function in scala is a value and every value is an object, which makes every function an object.

- Scala provides a lightweight syntax for defining anonymous functions, also called Function literals. Function literals are nameless functions. Their

concept and use of Arrow syntax have many names like – lambda expressions, lambdas, function0, function1, function2…..

- Scala supports higher order functions, a function which take function as input or function as a return type is called higher order function.
- Scala allows functions to be nested and Currying.
- Scala case classes are instantiable classes that include several automatically generated methods. All these methods are based on the class parameter lists.
- Scala's built-in supports for pattern matching model algebraic types used in many functional programming languages.
- Singleton objects provide a convenient way to group functions that aren't members of a class.
- Sequence comprehensions are useful for formulating queries.

4. Scala is Statically Typed

Scala, unlike some of other statically typed languages (c,pascal,rust,etc.), does not expect users to provide redundant type information. They need not specify a type in most cases, and certainly don't have to repeat it as well.

Abstractions are used in a safe and coherent manner. In particular, the type system supports:-

- **Generic classes:** Scala has built in support for classes parameterized with types. This can restrict the reuse of the class abstraction. Such generic classes are particularly useful for the development of collection classes.
- **Polymorphic methods:** Methods in scala can be parameterized by type or values.

```
Ex-  def dup[T](x: T, n: Int): List[T] = {

if (n == 0 ) Nil

else

x :: dup(x, n – 1)

}

println(dup[Int](3, 4))

println(dup("three", 3))
```

Method dup is parameterized with type T and value parameters x : T and n : Int. Scala type system automatically infers the type of the parameter according to the value.

- **Bounded Type:** It can be a specific class or its subtype or base type.

1. **Upper Bound:** An upper bound restricts a type to only that type or one of its subtypes. This means that an upper bound defines what a type must be and accepts subtypes through polymorphism. Use upper-bound relation operator (<:) to specify an upper bound for a type.
2. **Lower Bound**: A lower bound restricts a type to only that type or else one of the base types it extends. Use the lower-bound relation operator (>:) to specify a lower bound for a type.

- **Type variance**: Whereas adding upper or lower bounds will make type parameters more restrictive, we can add type variance to make type parameters less restrictive. Type variance specifies how a type parameter may adapt to meet a base type or subtype. Here are three generic classes that use each of the variance types.

```
class InVar[T]   { override def toString = "InVar" }
class CoVar[+T]              { override def toString = "CoVar" }
class ContraVar[-T]  { override def toString = "ContraVar" }
/************ Regular Assignment ************/
val test1: InVar[String] = new InVar[String]
val test2: CoVar[String] = new CoVar[String]
val test3: ContraVar[String] = new ContraVar[String]
```

The '+' denotes covariance and '-' denotes contravariance with respect to the type parameter. With respect to type parameters, the class is invariant without a plus or minus. When the type parameters are the same on both sides, the assignments work fine.

- **Compound Types:** Sometimes it is necessary to express that the type of an object is a subtype of several other types. In Scala this can be done by compound types, which are intersections of object types. We can specify the type of obj to be both One and Two. This compound type is written like this in Scala:

```
def OneAndTwo(obj: One with Two): Cloneable = {
//…
}
```

Compound types can consist of several object types and they may have a single refinement which can be used to narrow the signature of existing object members. The general form is: A with B with C … { refinement }

- **Implicit Parameters and conversions**: What if the function executes even if all the parameters are not specified? For the function to operate correctly, the missing, unspecified parameters would have to come from somewhere. One approach would be to define default parameters for your function, but for this, the function must know what the correct values for the missing parameters should be.

Another approach is to use implicit parameters. Here the caller provides the default value in its own namespace. Implicit parameter can be defined as a separate parameter group from the other non-implicit parameters. Local value can be invoked as implicit so it can be used as the implicit parameter. When the function is invoked without specifying a value for the implicit parameter, the local implicit value is then picked up and added to the function invocation. Implicit keyword has to be used to mark a value, variable or function parameter as implicit. An implicit value or variable may be used to fill in for an implicit parameter in a function invocation.

Another implicit feature in Scala is implicit conversions with classes. An implicit class is a type of class that provides an automatic conversion from another class. By automatic conversion from Instances of type A to type B, an instance of type A can appear to have fields and methods as if it were an instance of type B but there are some restrictions about how you can define and use them:

1. An implicit class must be defined within another object, class, or trait. Fortunately, implicit classes defined within objects can be easily imported to the current name-space.
2. They must take a single non implicit class argument.
3. The implicit class's name must not conflict with another object, class, or trait in the current namespace. Thus, you cannot use a case class as an implicit class because its automatically generated companion object would break this rule.

5. Scala is Extensible

The development of domain specific application often requires domain specific language extensions. Scala makes it easy to smoothly add new language in the form of libraries:-

- Any methods could be used as an infix or postfix operator.
- Depending on the expected type, closures are constructed automatically.

A join use of both features facilitated the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

6. Scala runs on the JVM

Scala is compiled into java byte code. And execute on the Java virtual machine (JVM) which means that java and scala has the common execution platform. The scala compiler compiles scala code into java byte code, which is then executed by the 'scala' command that is similar to the java command.

7. Scala can Execute Java Code

Scala allows you to use all the classes of the java SDK and also your own custom java classes, or your java open source projects.

8. Scala can do Concurrent & Synchronize processing

Scala programming allows you to express general programming patterns effectively which reduces the number of lines and helps the programmer to code in a type-safe way. It allows you to write codes in an immutable manner, which makes it easy to apply concurrency and parallelism.

Why Scala? | 16+ Reasons to Why Learn Scala

1. Why Scala?

Okay, we see all the hype about **Scala**, you also must have heard about Scala, you want to learn it but you are waiting for a solid reason to learn Scala. Why Scala? This article answers you question "Why should I learn Scala Programming Language?" Let see why Scala is a beneficiary language to learn and what it offers that you. You can share your own reasons to why Scala with us.

2. Type Inference

Scala automatically infers(detects) the data type of an expression partially or fully. This means that we don't need to declare it ourselves. Such a facility lets the programmer leave out type annotations while still allowing type checking.

So, how does this work? The compiler takes in the types of subexpressions, or those of atomic values like 42(Integer), true(Bool), and so on. It then aggregates this information, and then decides the type for an expression.

Read: Features of Scala

3. Singleton Object

Scala has no static variables or methods. Instead, it makes use of a singleton object. This is essentially a class with only a single object in the source file. We do this using the 'object' keyword, and not the 'class' keyword. For instance:

```
1. object Main extends App {
2. println("Hello, World!")
3. }
```

4. Immutability

In Scala, every variable you declare is immutable by default. You cannot change it. However, you can also explicitly declare it to be mutable. You can change the value of such a variable.

Immutable data helps us manage concurrency control, as you would imagine. Once we create an immutable object, we cannot modify its state.

5. Lazy Evaluation

We discussed this when we talked Scala vs Java. Lazy evaluation is when Scala delays evaluating an expression until it absolutely needs it. This is also called call-by-need.

This also avoids repeated evaluations. Lazy evaluation has the following benefits:

1. We can define control flow as abstractions instead of primitives.
2. We can define potentially infinite data structures.
3. Lazy evaluation improves performance.

In Scala, we can have a lazy declaration:

lazy val images=getImages()

Read: Control Structures in Scala

6. Case Classes and Pattern Matching

A case class is a regular one that is immutable by default, and is decomposable via pattern matching. All its parameters are immutable and public by default.

Defining a case class takes the keywords 'case class', an identifier, and a parameter list(can be empty).

case class Book(isbn: String)

val frankenstein = Book("978-0486282114")

Pattern matching lets us check a value against a pattern. You can use this in place of a switch-statement or a series of if-else statements in Java.

```
import scala.util.Random
val x: Int = Random.nextInt(10)
x match
{
case 0 => "zero"
case 1 => "one"
case 2 => "two"
case _ => "many"
}
```

7. Concurrency Control

Scala's standard library includes the Actor model. Using this, you can implement concurrency in your code. Apart from this, it also has a platform/tool, Akka. It is a separate, open-source framework providing for Actor-based concurrency. You can combine or distribute Akka's actors with software transactional memory.

8. String Interpolation

String interpolation is the act of evaluating a string literal, consisting of one or more placeholders, to yield a result. In this result, corresponding values replace the placeholders. This is a kind of template-processing.

Since version 2.10.0, Scala offers string interpolation. The three methods it offers for this are- s, f, and raw.

Read: Tuples in Scala

9. Higher-Order Functions

A higher-order function is one that takes another as a parameter, or returns it. This is possible in Scala only because it treats functions as first-class citizens.

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

Here, map is a higher-order function.

Such functions also let us implement function compositions, and lambdas.

10. Traits

A trait, in Scala, is a type containing certain fields and methods. You can combine multiple traits.

To define a trait, we use the 'trait' keyword:

trait Greeter

```
{

def greet(name: String): Unit

}
```

So, a trait is like a partially implemented interface. You can create a trait with abstract, and optionally, non-abstract methods.

Not yet convinced Why Scala is good for you? We have more reasons.

11. Rich Set of Collections

Scala has a huge set of collections in its library. This has classes and traits to help you collect data into a mutable or immutable collection. The scala.collection.mutable package holds all mutable collections. Making use of this package, you can add, remove, and update data. Likewise, the scala.collection.immutable package holds all immutable collections; they don't let us modify data.

Read: Best Books for Scala

12. Functional

Scala is a language of a functional paradigm. It treats its functions as first-class citizens. This is why it also lets us create higher-order functions- ones that can return a function, or take it as a parameter. It also supports nesting of functions, and currying(translating evaluation of a function, that takes

multiple arguments, into evaluating a sequence of functions, each with a single argument).

13. Object-Oriented

This language also supports object-oriented programming, purely. Every value is an object. We'll have more on this later.

14. Statically-typed

In most cases, you won't need to specify redundant type information to Scala; it will figure it out on its own. This is in relevance to type inference.

Read: Scala vs Java

15. Extensibility

Developing domain-specific applications needs domain-specific language extensions. In a way, Scala puts forward a combination of language mechanisms, making it easy to smoothly add new language constructs as libraries. We can do this using facilities like implicit classes and string interpolation. You don't necessarily need meta-programming features like macros.

16. Scala Runs on the JVM

The Scala compiler turns the source code into byte code. This runs on the JVM (Java Virtual Machine). This makes it a language similar to Java and Groovy.

17. Miscellaneous Features

Some of Scala's quirks confuse Java programmers:

1. Where Java mandates that all import statements must be at the top of the source, in Scala, they can be almost anywhere within your whole program.
2. In Scala, operators are just methods.
3. Definitely everything looks like an object in Scala.
4. Here, methods may have multiple parameter lists. Wow! Let's discuss this in a later lesson, this seems really exciting for a programmer stuck with similar languages.
5. It is possible to nest methods.
6. A class' body is its constructor.
7. In Scala, we don't have static methods; we have objects.

This was all about Why Scala article. Hope you like the Why Scala article and have now got the answers to "why should I learn Scala".

18. Conclusion

Scala Has many features that makes it special as we saw in the above why Scala post. Wasn't this a step ahead in your journey with Scala? But don't worry if it confuses you with its syntax; we will get to explaining the basics pretty soon. Till then, share love, share happiness.

Scala Object Oriented Programming | Scala Tutorial

1. Objective

In our last Scala tutorial, we discussed **Scala Lists with example.** Now you have a good understanding of Scala. In this **Scala tutorial**, we will discuss Scala Object Oriented Programming. Moreover, we are going to learn Scala Class, Scala Objects, Scala examples. Lastly, we will discuss Anonymous Object.

So, let's begin Scala Object Oriented Programming.

Scala Object Oriented Programming

2. Scala Object Oriented Programming

As we know, Scala is purely object-oriented. This means we can create classes and then objects from those classes. Now, let's see what classes and objects are.

3. Scala Class

Think of a class like a blueprint for all objects of its kind. In other words, it is a collection of objects of the same kind.

A class may hold the following members:

- Data members
- Member methods
- Constructors
- Blocks
- Nested classes
- Information about the superclass

..and more.

Let's explore Scala operator

In Scala, we must initialize all instance variables. Also, the access scope is public by default. We define the main method in an object. This is where the execution of the program begins.

4. Scala Objects

Moving on to objects, if a class is a blueprint, objects are prototypes of that blueprint. An object is essentially a real-world entity with state and behaviour. An object's state is its data values at any point in time. Its behaviour is the functionality it performs.

So, Scala object is an instance of a class and is a runtime entity. For a class fruit, an Orange can be an object.

5. Scala Examples

Let's take an example of Scala Class.

```
1. scala> class car{
2. | var fuel:Float=0
3. | var color:String="Black"
4. | }
```

Defined class car

Now, let's declare an object for this.

```
1. scala> var Brio=new car()
2. Brio: car = car@7645b7d
```

6. Example- with a Constructor

Let’s take another example. This time we’ll use a primary constructor to initialize values.

```
scala> class person(SSN:Int,Name:String){
| def sayhi(){
| println("I'm "+Name+" and my Social Security Number is "+SSN)
| }
| }
```

Defined class person

Scala String Method with Syntax and Method

Now, we create an object for this.

```
scala> val Ayushi=new person(798798,"Ayushi Sharma")
Ayushi: person = person@73a116d
```

And finally, we call the method sayhi() on this.

```
scala> Ayushi.sayhi()
```

I’m Ayushi Sharma and my Social Security Number is 798798.

7. Anonymous Objects

When we want to work with an object that we won’t need to use again, we can declare it anonymously.

Let’s define a new class to say Hi.

```
scala> class A{
| def sayhi(){
| println("Hi")
| }
| }
```

Defined class A

Now, we create an anonymous object of this and call sayhi() on it.

```
scala> new A().sayhi()
Hi
```

So, this was all about Scala Object Oriented Programming. Hope you like our explanation.

8. Conclusion

Now that we’re done with the basics of Scala Object Oriented Programming. Hence, in this article, we have discussed Scala Object Oriented Programming, Scala Class, Scala examples. At last, we saw how to create an anonymous

object. Furthermore, if you have any query, feel free to ask in the comment section.

Learn Scala Environment Setup and Get Started with an IDE

1. Scala Environment Setup

This article on **Scala Environment Setup** will guide you through setting up Scala for your system. If you still haven't begun with the language, now's your chance.

2. Prerequisites to Setup Scala Environment

We know that Scala runs on the JVM. So for Scala Environment Setup, you'll need to have Java installed on your machine. To check, open the command prompt and type the following:

Prerequisite to set up Scala Environment

This tells us that we have Java 1.8 involved. If it isn't installed for you, you can download it here:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

3. Install Scala

Now, we'll need to Install Scala. For learning purposes, we can simply download the binaries from the official site:

https://www.scala-lang.org/download/

This is around 123.7MB heavy.

Once you hit download, you'll see the following:

Scala Setup Environment: Install Scala

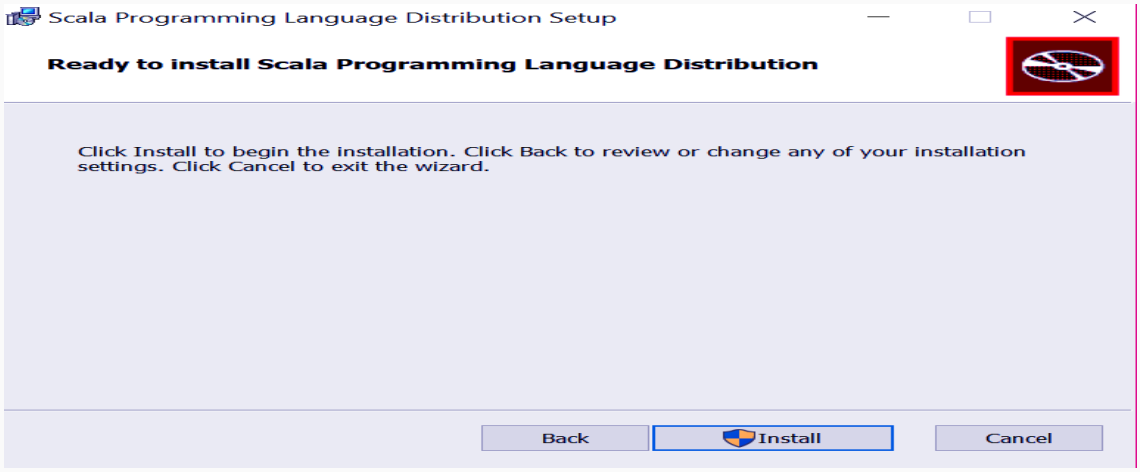
Click Next.

End Use License Agreement

Here, make sure the checkbox for the terms and conditions is checked. Then, click Next.

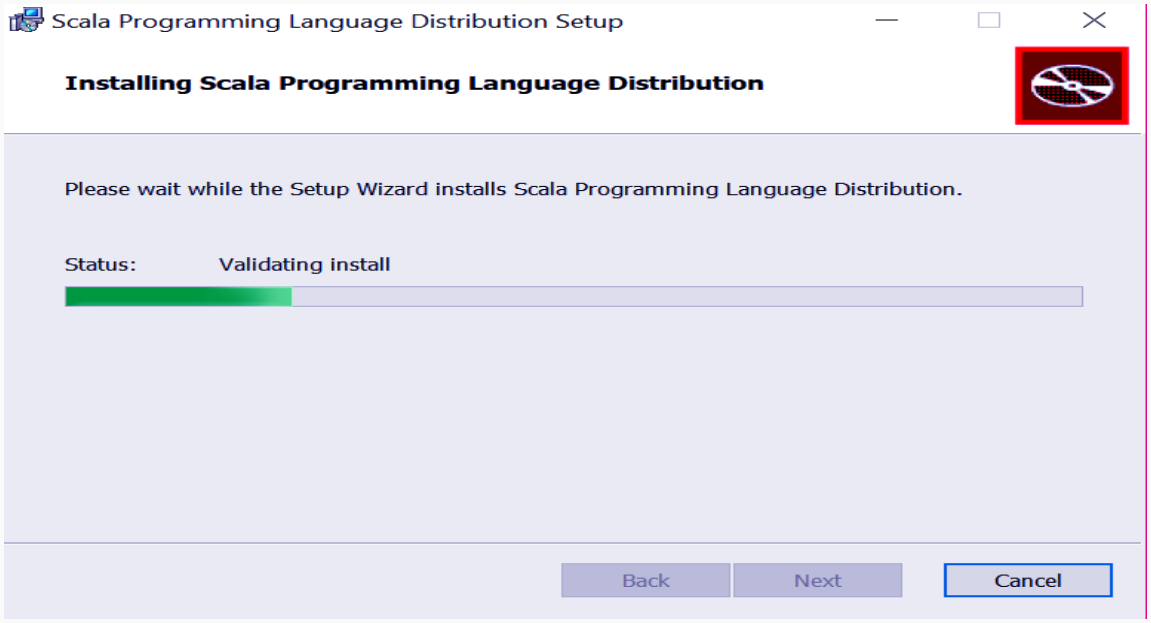
Environment Setup in Scala: Custom Setup

Now, this dialog lets you select the features you want to install.



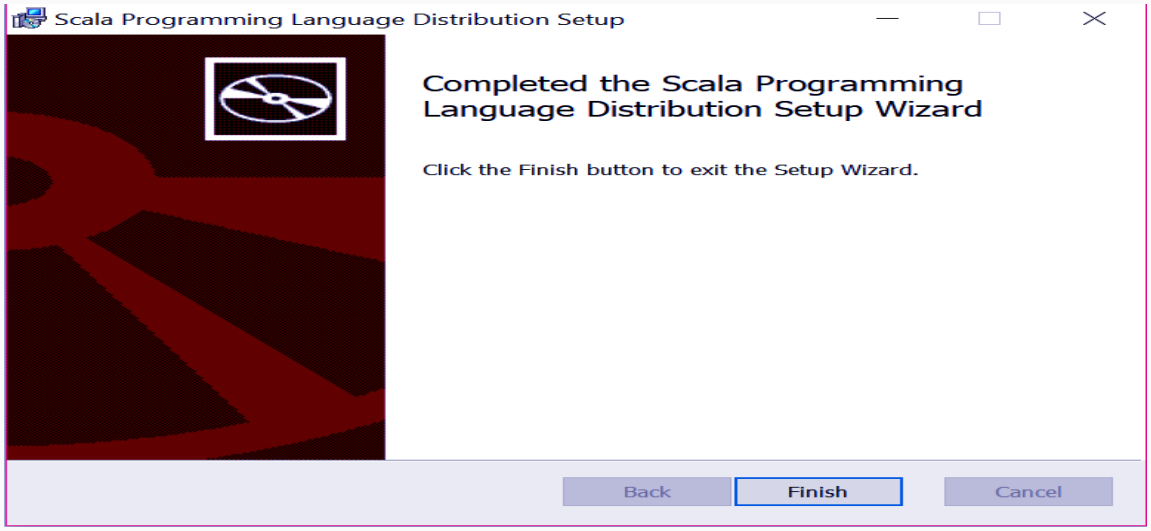
Setup Scala Environment: Ready to Install Scala Programming

We're ready to install it. Click on 'Install'.



Install Scala Programming Language Distribution

Wait as it installs.



Completed the Scala Programming Language Distribution Setup Wizard

We're done; click 'Finish'.

Now, when you get in the command prompt, type scala:

C:\Users\lifei>scala

Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_151).

Type in expressions for evaluation. Or try :help.

```
scala>
```

This will give you the Scala prompt. Here, you can type in expressions to evaluate:

```
scala> println("Hello")
Hello
scala> 2+3
res1: Int = 5
```

If, however, you want to run your program as a script, save it as a file with a .scala extension, and move to that location. Here's the file we use:

```
object hello extends App{
println("Hello")
}
```

First, let's quit Scala in the command prompt by pressing Ctrl+C and then pressing y, and then Enter:

```
scala> Terminate batch job (Y/N)? y
```

C:\Users\lifei>

Now, let's get to the Desktop, since that is where we saved the file:

```
C:\Users\lifei>cd Desktop
```

Then, to compile the script, run the following command:

```
C:\Users\lifei\Desktop>scalac hello.scala
```

Next, to run the script, do the following:

```
C:\Users\lifei\Desktop>scala hello
Hello
C:\Users\lifei\Desktop>
```

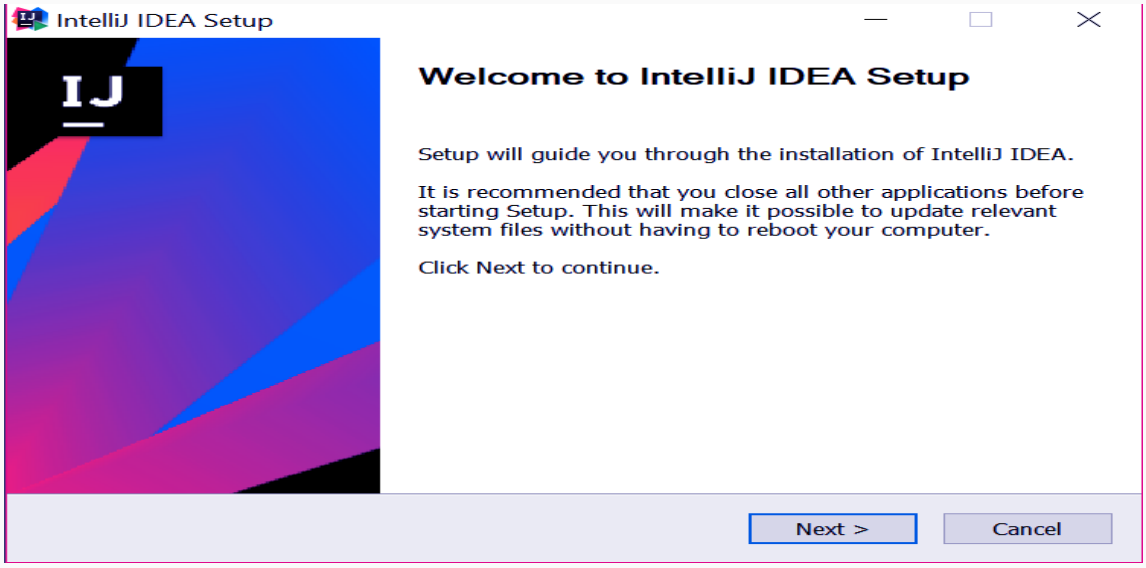
There you go! All set for your journey with Scala.

4. Getting Started with an IDE

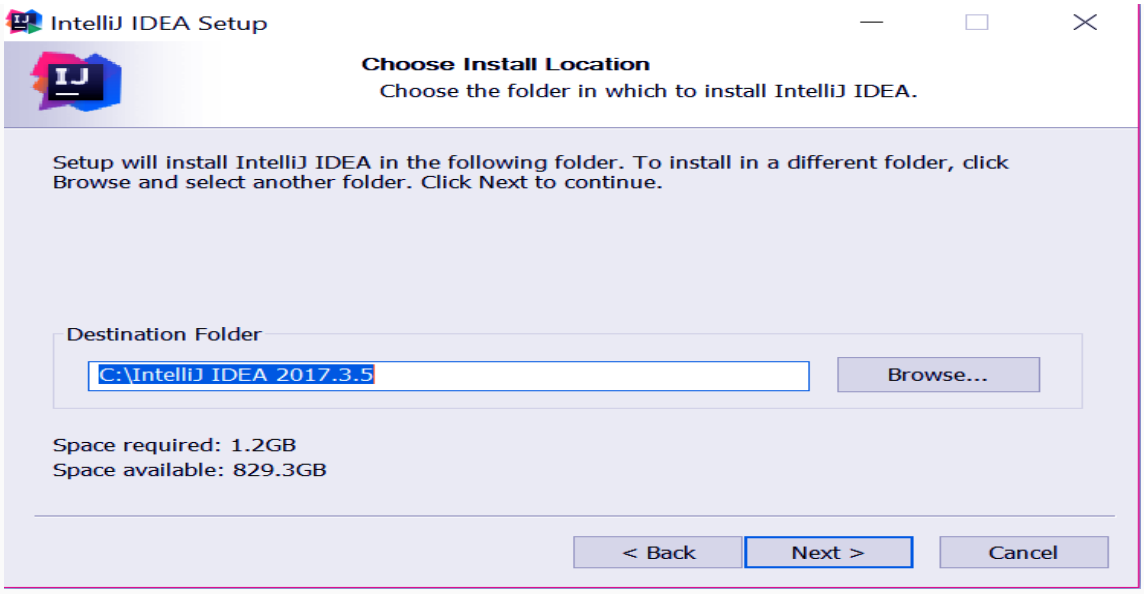
To build projects with Scala, you'll need to work with an IDE. In this tutorial, we'll use IntelliJ IDEA. First, download it from here:

https://www.scala-lang.org/download/

Then, you'll drive through the following screens:

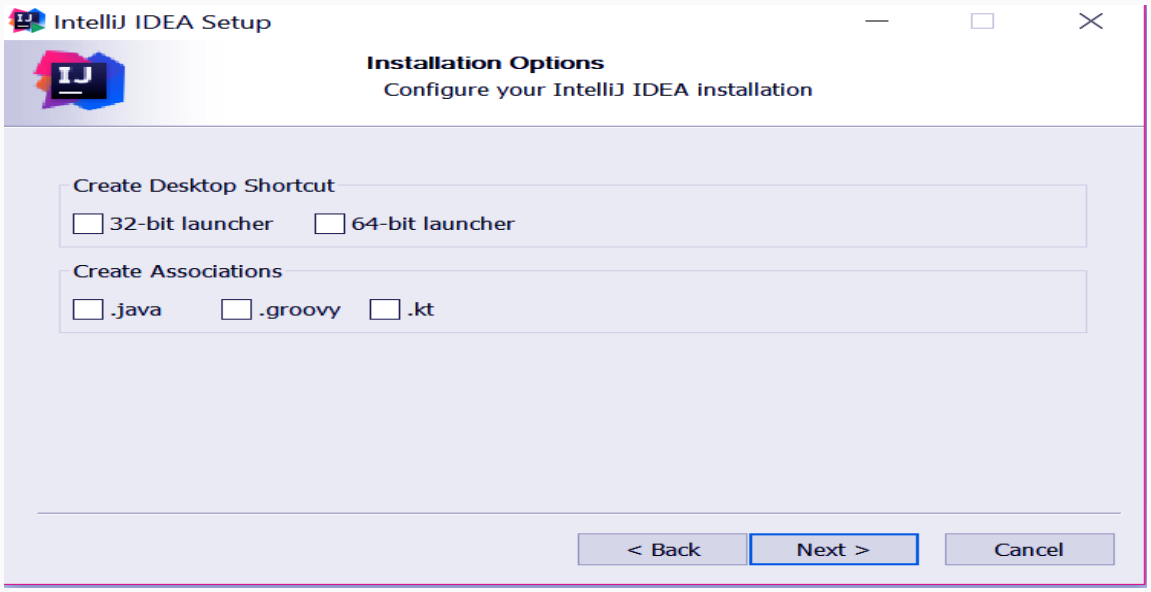


Scala Environment Setup: IDEA Setup



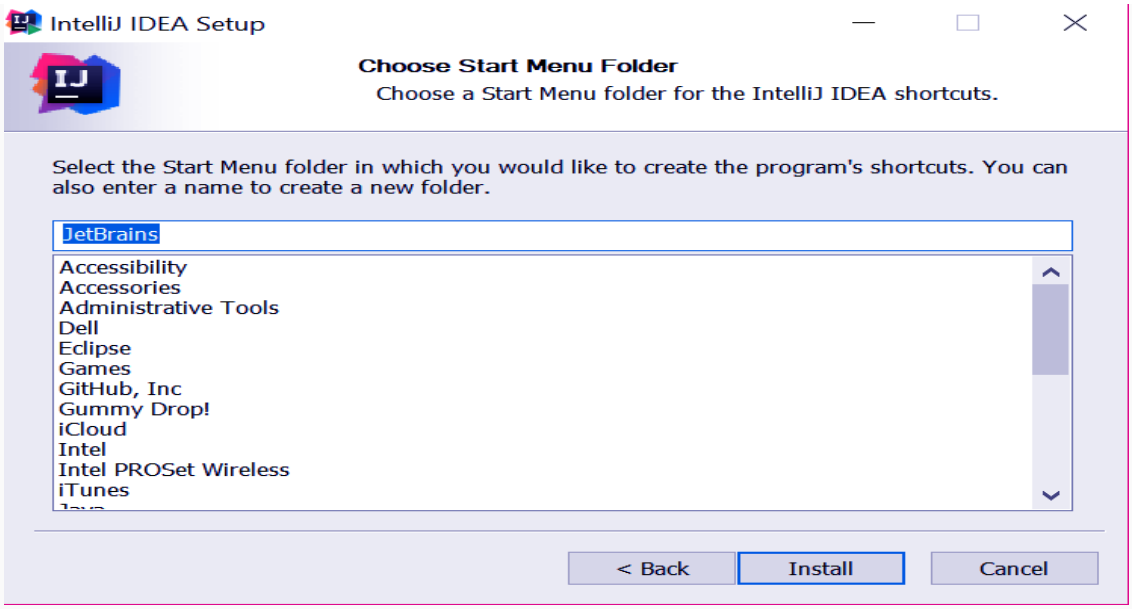
Scala Environment Setup: Choose Install Location

Here, you can select which folder to install Scala in.



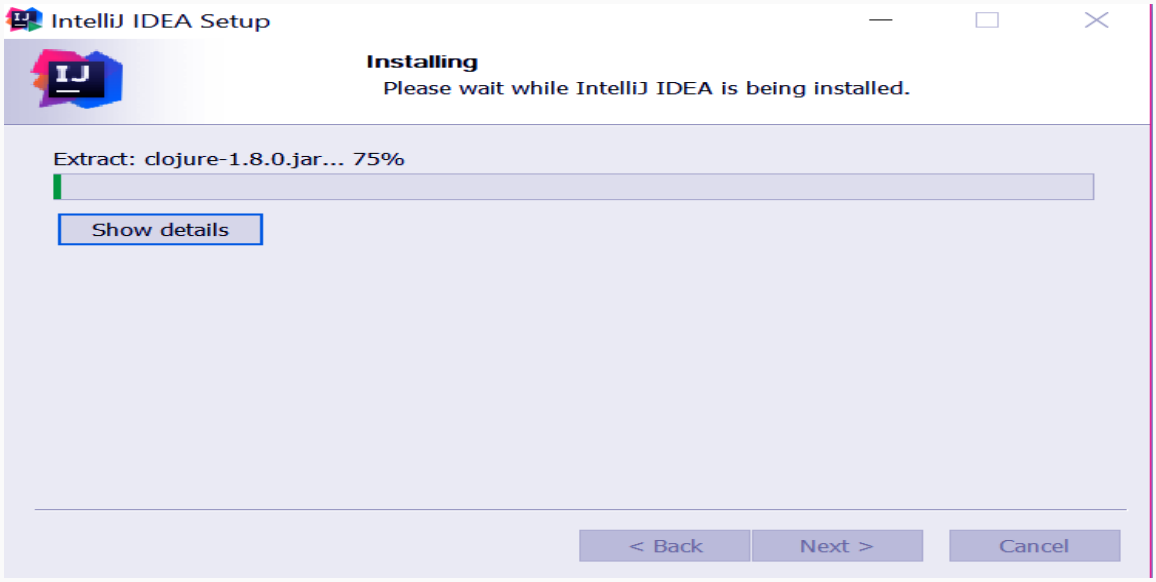
Setup Scala Environment: Installation Options

You may choose to create a Desktop shortcut, or even create associations with Java, Groovy, and kt.



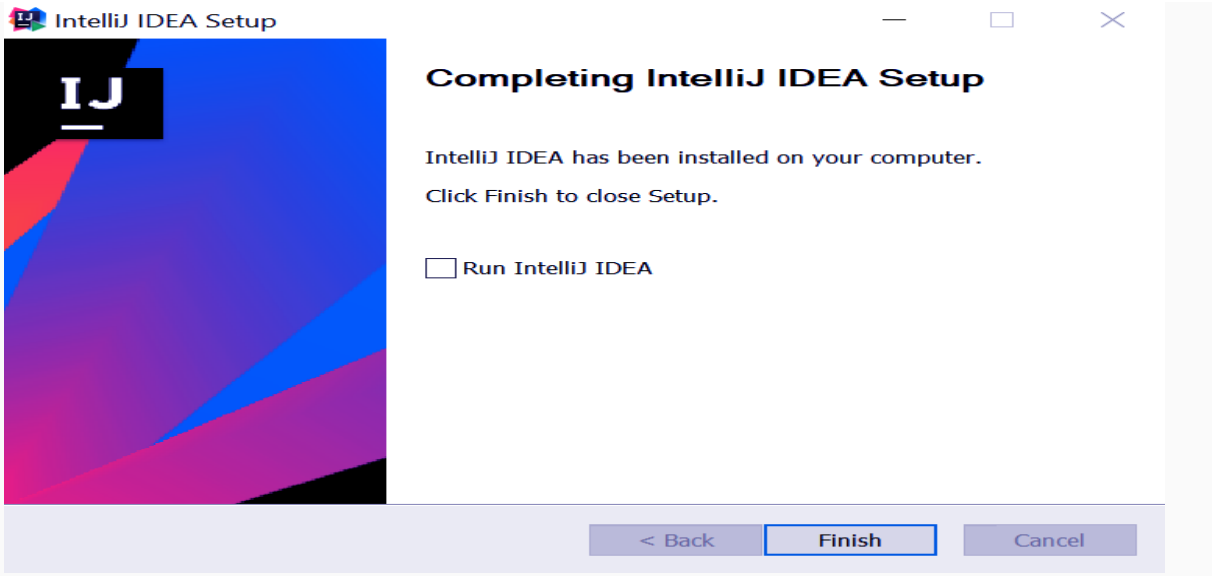
Choose Start Menu Folder

Where do you want to create shortcuts for the program in the Start Menu? Or would you like to create a new folder?



IDEA Installing

Let it install.



Completing IDEA Setup

Phew, we're done. Now, click on 'Finish'.

Import IDEA setting form

A couple final formalities.

License Agreement

Accept the user license agreement to continue.

Data Sharing Options

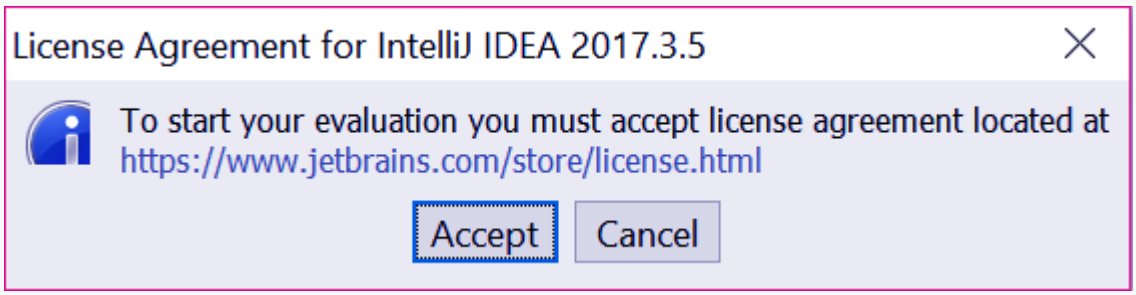
You can opt to anonymously send usage statistics to JetBrains, the brain of IntelliJ.

Scala Environment Setup

You can either activate your license or evaluate it for free. We downloaded the Ultimate edition.

Setup Scala Environment

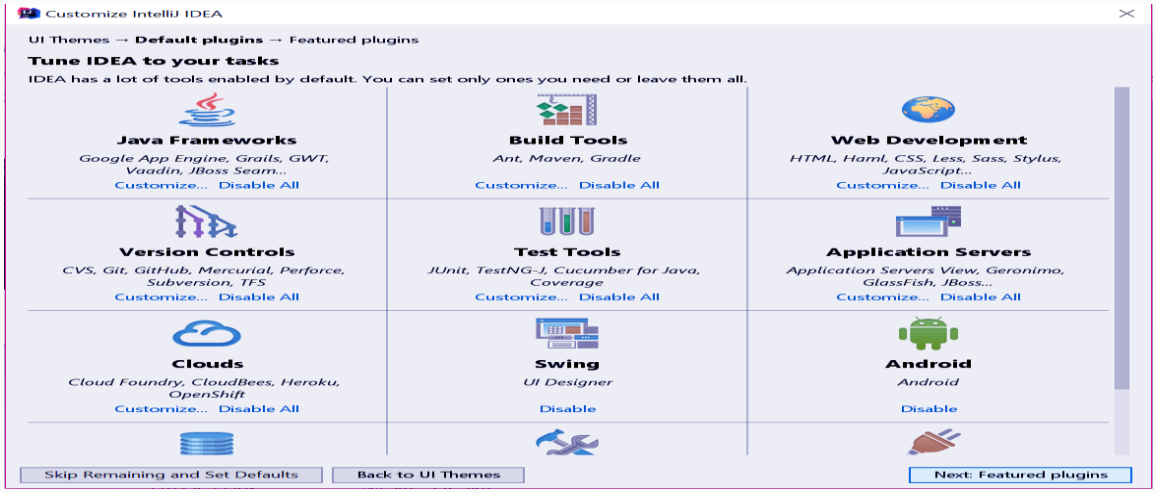
Let’s evaluate it for free.



Accept.

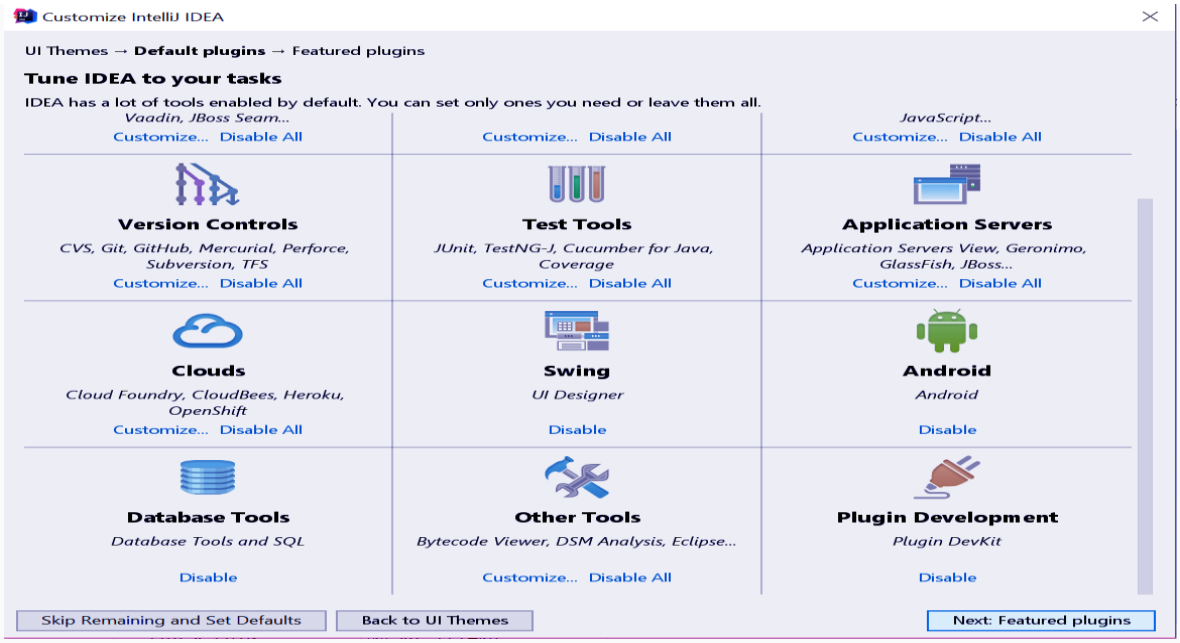
Set UI Theme

Do you prefer a lighter theme or the darker, more popular Darcula?

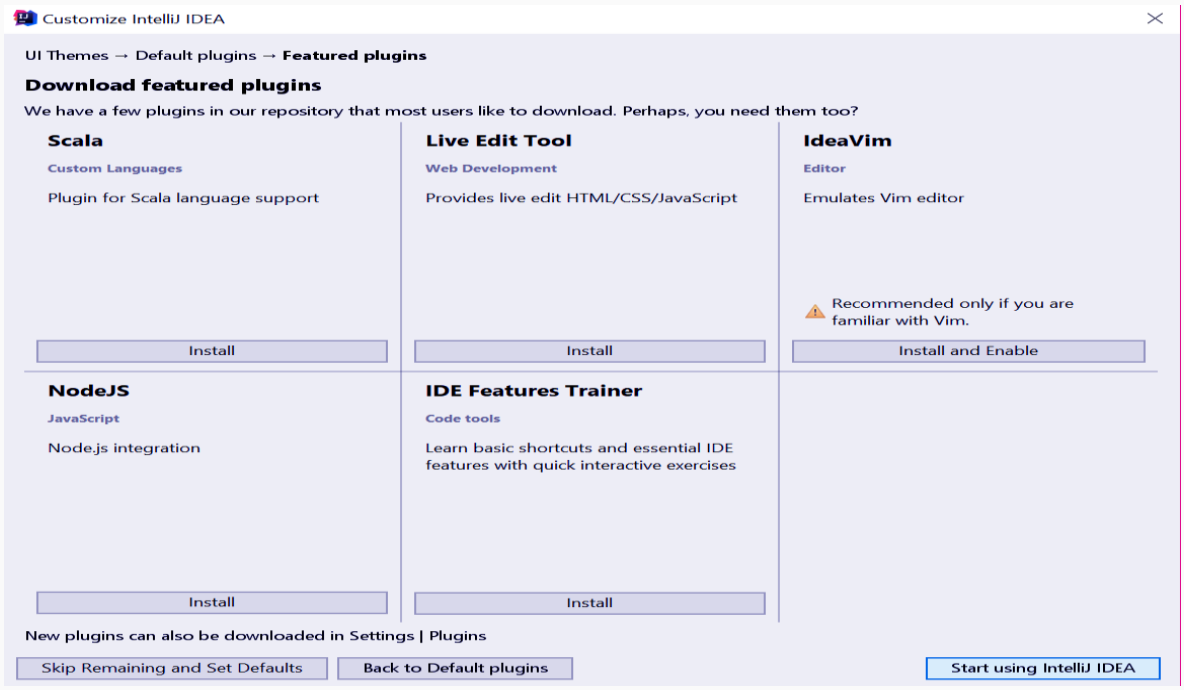


UI Themes Default Plugin

Choose the plugins you want.

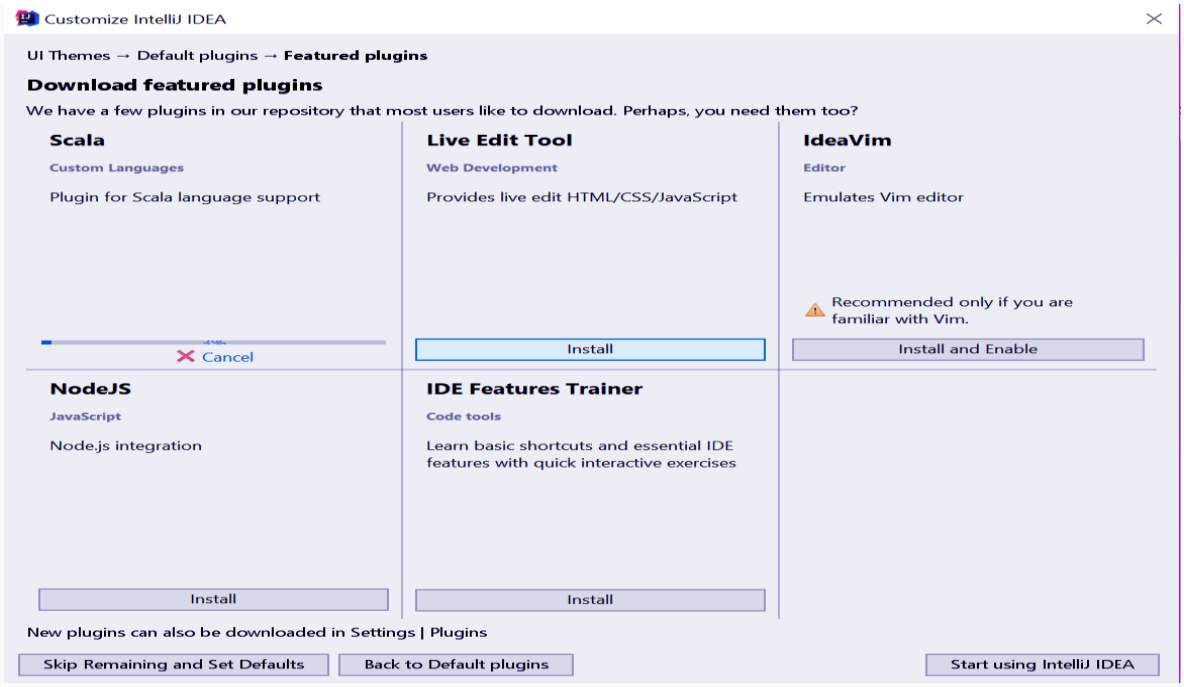


Customize IDEA



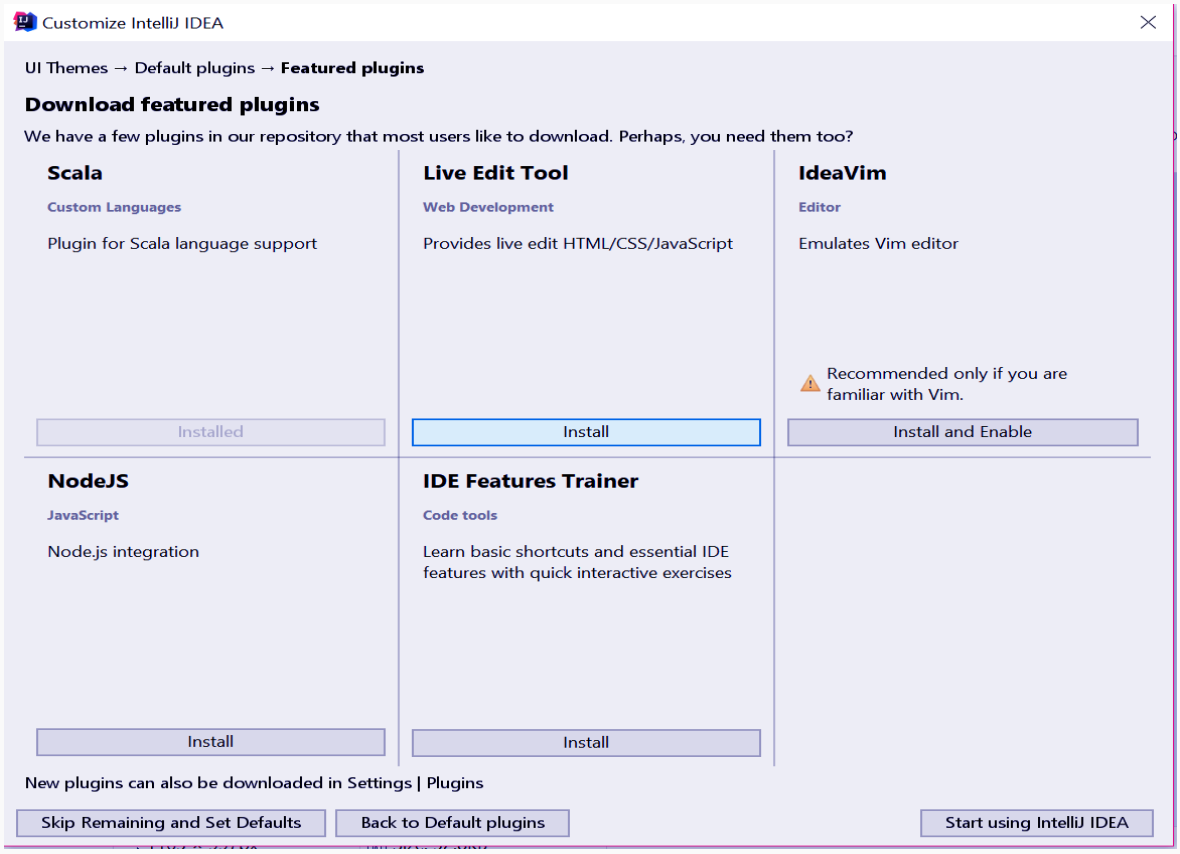
Download Featured Plugins

You can also download featured plugins.



Download Featured Plugins

Let’s download for Scala.

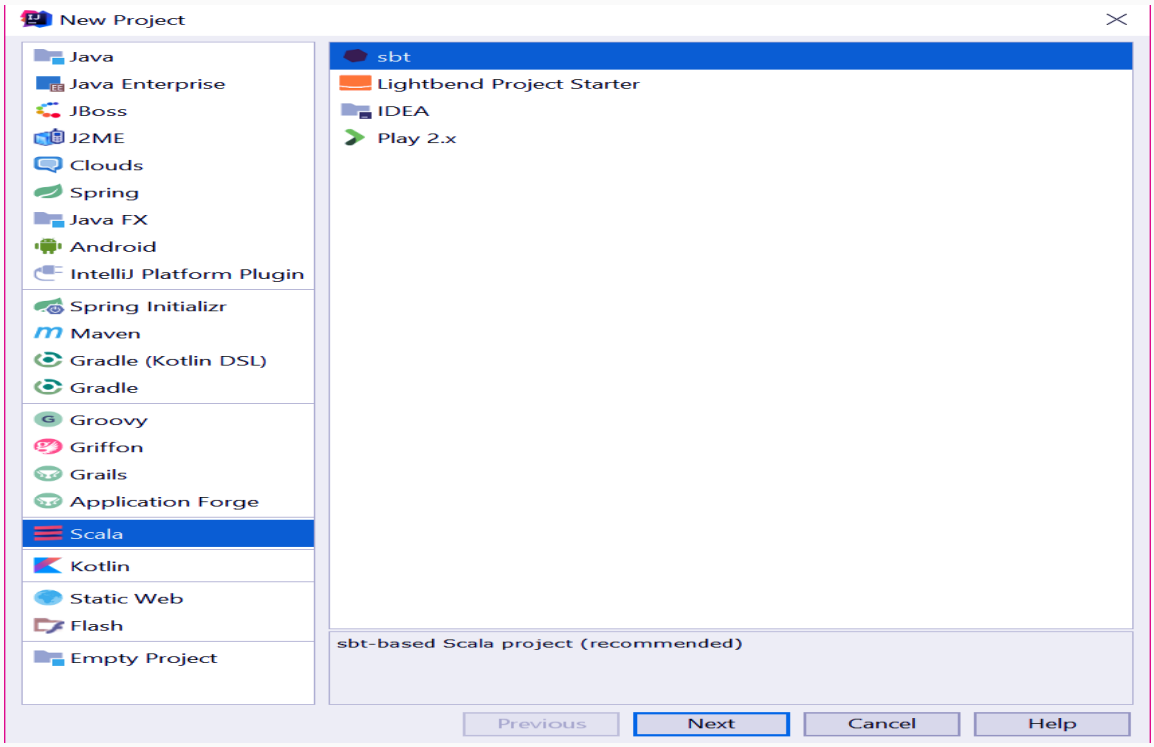


Download Featured Plugins

It’s installed.

We're all done! Open it.

Create a new project.



IDEA New Project

Choose Scala.

After this, you can choose your own configurations for your project, including the JDK.

5. Conclusion

Hope you had fun installing Scala on your machine. Tell us what you'll do next.

Scala Syntax: An Introductory Scala Tutorial

1. Scala Syntax

What does Scala look like? This article aims to explain to you the basic Scala Syntax. Though it is a bit different from Java or C++, we'll get used to it. Let's begin.

Scala Syntax

2. Introduction to Scala Syntax

We can look at a Scala program as a collection of objects that invoke each other's methods to communicate. One major difference between Scala and Java is that here, in Scala, we don't need to end each statement with a semicolon(;).

3. Basic Constructs

Let's start with the basic structures- expressions, blocks, classes, objects, functions, methods, traits, Main method, fields, and closures.

a. Expression

An expression is a computable statement. For example, the following is an expression:

```
1+2
```

We use println() to print the output of an expression:

```
println(1) //1
println(1+2) //3
println("Hi") //Hi
println("Hi"+" User") //Hi User
```

1. Values

An expression returns a result; we can name it using the 'val' keyword.

```
val name='Ayushi'
println(name)
```

Such an entity has only one value; we cannot reassign it. Also, when we reference it again, that does not recompute its value.

```
val name="Ayushi"
name="Megha" //This doesn't compile
```

Usually, Scala will infer the type of an expression- type inference, as we've discussed in Features of Scala. But we can also explicitly state the type:

```
val roll:Int = 30
```

2. Variables

A variable is a value that we can reassign. To declare a variable, we use the 'var' keyword. Scala Syntax for Variables are given below.

```
var x=2
x=3 //This changes the value of x from 2 to 3
println(x*x) //This prints 9
```

We can declare the type of the variable:

var roll:Int = 30

b. Block

A block is a group of expressions delimited by curly braces({}). A block returns whatever its last expression returns. The Scala Syntax for the Same is Below.

```
println
(
{
val x=1+1
x+1
}
) //3
```

c. Class

A class is a blueprint that we can use to create an object. It may hold values, variables, types, classes, functions, methods, objects, and traits. We collectively call them members. To declare a class, we use the keyword 'class' and an identifier.

object Main extends App

```
{
class Fruit{}
val orange=new Fruit
}
```

Here, to create an instance of this, we use the keyword 'new'. An instance of a class can have behaviors and states. For example, a car has states- brand, model, color, and behaviors- start, turn, stop. Here, orange is an instance of class Fruit.

We'll see more on classes and objects in a later tutorial.

d. Object

It is a single instance of its own definition(a singleton of its own class). To define an object, we use the keyword 'object'.

```
object Main extends App {
object MyObject{
def plusthree(){
val x=3*3
x+3}
}
println(MyObject.plusthree) //This prints 12
}
```

We'll cover this in a later tutorial.

e. Function

A function is an expression that takes parameters. First, let's observe an anonymous function.

```
(x:Int)=>x*x
```

This function takes an Integer argument x, and returns its square.

So, here we see that on the left of => is the list of parameters, and on the right is an expression it will return. Now, let's give it a name.

```
val squared=(x:Int)=>x*x
```

To get the output, we call the function:

```
println(squared(3)) //This prints 9
```

A function may also take multiple parameters or none.

```
1. val add=(x:Int,y:Int,z:Int)=>x+y+z
2. println(add(2,3,7)) //This prints 12
3. val sayhi=()=>(println("Hi")) //We could also have put the println in a block as {print("Hi")}
4. sayhi() //This prints Hi
```

f. Method

A method is similar to a function, but we define it using the keyword 'def'. What follows is an identifier, parameter lists, a return type, and a body. Let's take an example.

```
1. def squared(x:Int):Int=x*x
2. println(squared(3)) //This prints 9
```

We denote the return type after a colon after the parameter list. Also, here, we use = instead of =>.

A method can also take multiple parameter lists.

```
1. def mymethod(x:Int,y:Int)(z:Int):Int={(x+y)*z}
2. println(mymethod(2,3)(4)) //This prints 20 from ((2+3)*4)
3. Now, let's try something with no parameter lists.
4. def method1():String="Hello"
5. println(method1) //This would work too: println(method1())
```

A method can have multiline expressions:

```
1. def method2(x:Int,y:Int)(z:Int):Int={
2. val a=z+y+x
3. (x+y)*a
4. }
5. println(method2(2,3)(4))
```

The last expression is the return value. Scala does have a 'return' keyword, but we'll see that later.

g. Trait

A trait holds fields and methods, and we define it using the 'trait' keyword. We can also combine traits.

```
1. trait greeter
2. {
3.
4. def greet(name:String):Unit
5.
6. }
```

We'll see this in detail later.

h. Main Method

The main method is an entry point for the program. JVM needs a method with an argument which is an array of strings.

```
object Main
{

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

println("Hello, Scala developer!")

}
```

i. Fields

A unique set of instance variables belongs to each object. These are fields, and they define the object's state.

j. Closure

Any function whose return value depends on variables declared outside it.

Any Doubt yet in Scala Syntax? Please Comment.

4. An Example Program

Let's try the first "Hello, World!" program in Scala. Like many other languages, we may execute a Scala program in one of two modes- interactive mode and script mode.

a. Interactive Mode

Open the Command Prompt on Windows using 'cmd' in Run. Then, type 'scala'.

Microsoft Windows [Version 10.0.16299.309]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\lifei>scala

Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_151).

Type in expressions for evaluation. Or try :help.

scala>

Now, you can execute any statement here.

b. Script Mode

To execute a program in script mode, type your program in the Notepad.

```
object hello extends App
{
println("Hello")
}
```

Now, get back to the command prompt, and get to the location where your file is.

C:\Users\lifei>cd Desktop

C:\Users\lifei\Desktop>

Then, to compile your program, type the following:

C:\Users\lifei\Desktop>scalac hw.scala

This creates some class files in the current directory. One of these is hello.class. This is the bytecode, and finally, to run this bytecode, use the command scala:

C:\Users\lifei\Desktop>scala hello

Hello

5. Syntax Rules

There are some basic syntax rules we must take care of while coding with Scala. Let's discuss those.

a. Case-Sensitive

Scala is case-sensitive. This means that it treats the identifiers 'hello' and 'Hello' differently.

b. Class Names

You should name a class in Pascal case. This means that instead of naming it like 'hihelloworld', you will name it 'HiHelloWorld'.

c. Method Names

A method's name should be in Camel case. So, the correct way would be something like 'sayHelloToTheWorld()'.

6. Keywords

Scala reserves some words so it can function. These are:

Abstract case catch class

def do else extends

false final finally for

forSome if implicit import

lazy match new Null

object override package private

protected return sealed super

this throw trait Try

true type val Var

while with yield –

: = => <-

<: <% >: #

@

We can’t use any of these words as identifiers or constants.

7. Identifiers

What names can we use for entities like objects, classes, methods, and variables? We call them identifiers, and there are some rules to what names we can use for this purpose:

1. It cannot be a reserved keyword.
2. It is case-sensitive.

Scala supports four kinds of identifiers:

a. Alphanumeric Identifiers

These begin with a letter(A-Z/a-z) or an underscore. This can be followed by letters, digits, or underscores. You cannot use ‘$’, since it is a keyword.

Legally valid examples: age, _8bit, __magic__

Legally invalid examples: $99, 9lives, -23.47

b. Operator Identifiers

These may have one or more operator characters. So, what are operator characters? Printable ASCII characters like +, :, ?, ~, or #.

Legally valid examples:

+

++

:::

<?>

:>

The compiler internally mangles these to turn them into legal Java identifiers with embedded $ characters. For example, it would represent :-> as $colon$minus$greater.

c. Mixed Identifiers

These contain an alphanumeric identifier followed by an underscore and an operator identifier.

Legally valid examples:

unary_+

mine_=

d. Literal Identifiers

Such an identifier is an arbitrary string delimited by back ticks(`…..`).

Legally valid examples:

`x`

`yield`

`hello`

8. Comments

Just like Java, Scala supports single-line and multiline comments.

// This is a single-line comment

/* This is the first line of a multiline comment

This is the middle

And this is the last*/

The compiler ignores comments. Read up on Comments in Scala for a detailed explanation on comments.

9. Blank Lines and Whitespace

A line that contains only whitespace, and possibly comments, is a blank line, and Scala just ignores it. Also, we can separate tokens by whitespace characters and/or comments.

10. Newline Characters

Scala is line-oriented; we may terminate statements with semicolons(;) or with newlines. This means that semicolons are optional to end a statement. If only a single statement appears on a line, we don't have to use it necessarily. But if we must fit multiple statements into one line, we must separate them with semicolons:

```
1. val x=7; println(x)
```

11. Packages

A module of code is a package with a name. For example, take a look at the Lift utility package net.liftweb.util. In the source file, the package declaration is the first non-comment line:

package com.liftcode.stuff

To refer a package in the current compilation scope, we can import a package:

import scala.xml._

We can also import a single class and object from a package:

import scala.collection.mutable.HashMap

And to import more than one class or object:

import scala.collection.immutable.{TreeMap, TreeSet}

This was all about the Scala Syntax.

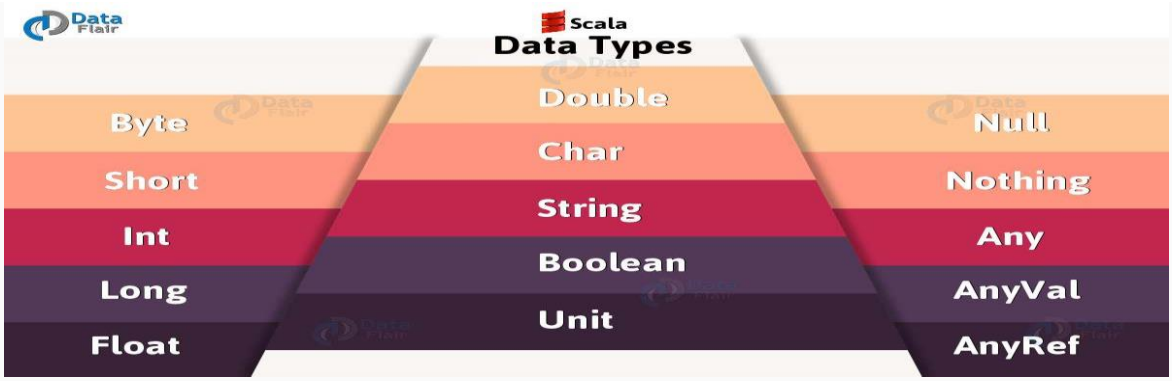
12. Conclusion

We can guarantee you'll be able to write basic Scala code now. Tell us how it works for you. Also leave your doubts and comments regarding the Scala Syntax in the comment section.

Scala Data Types with Examples | Escape Value & Typecasting

1. Scala Data Types

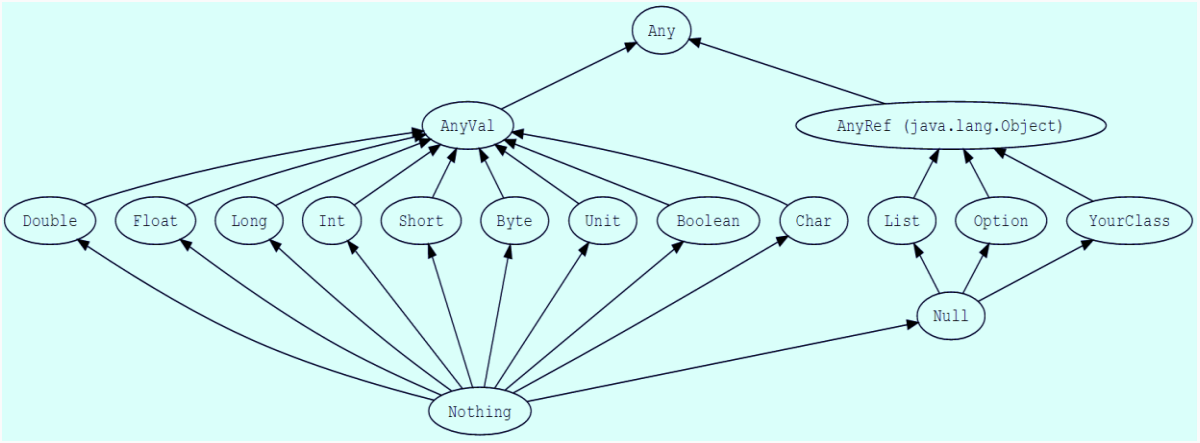
Hello, readers! Welcome back to learning Scala with DataFlair. Today, we will discuss Scala Data Types with basic literals, Escape value, typecasting and its examples. Hop on.



Scala Data Types

2. Introduction to Data Types in Scala

Like every other language, Scala has a type for every value. We saw this when we discussed variables and values. Even functions have a value. So, let's understand the Scala data types hierarchy for unified types.



Introduction to Scala Data Types

Here, the supertype for all types is Any. It has universal methods like equals, hashCode, and toString.

Any parents two subclasses: AnyVal and AnyRef.

AnyVal represents value types. The nine predefined and non-nullable value types are: Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean. We'll discuss these values in a short while. AnyRef represents reference types. A user-defined type is a subtype of this. And in the context of a JRE, AnyRef corresponds to java.lang.Object. Lets now directly jump to the Scala Data Types in Scala Language.

3. Data Types

Let’s discuss the basic in-built Scala data types in detail.

a. Scala Byte

Size: 8-bit

Signed value

Range: -128 to 127

b. Scala Short

Size: 16-bit

Signed value

Range: -32768 to 32767

c. Scala Int

Size: 32-bit

Signed value

Range: – 2147483648 to 2147483647

d. Scala Long

Size: 64-bit

Signed value

Range: -9223372036854775808 to 9223372036854775807

e. Scala Float

Size: 32-bit

It follows the IEEE 754 standard, and is a single-precision float.

f. Scala Double

Size: 32-bit

It follows the IEEE 754 standard, and is a double-precision float.

g. Scala Char

Size: 16-bit

It is an unsigned Unicode character

Range: U+0000 to U+FFFF

h. Scala String

A string is a sequence of Chars.

i. Scala Boolean

A Boolean value is either true or false.

j. Scala Unit

There is only one instance of unit, and that is (). It carries no meaningful information. And since all functions must return something, sometimes, we have them return Unit.

k. Scala Null

This refers to an empty or null reference. It is a subtype of all reference types. This makes it a subtype of any subtype of AnyRef. Null helps with interoperability with other JVM languages, and we almost never use it.

l. Scala Nothing

Nothing is a subtype to every other type. And trust us, it holds no value at all.

Since it is a subtype of all types, we also call it the bottom type. Actually, no value is of type Nothing. So where would we use it? We can use it to signal non-termination like a thrown exception, program exit, or an infinite loop.

m. Scala Any

This is the supertype for all other types. This means that any object is of the type Any.

n. Scala AnyVal

This represents value types.

o. Scala AnyRef

AnyRef represents reference types.

Since all these Scala Data types are objects, and not primitives, it is possible to call methods on these objects.

4. Basic Literals

In this section, we will see integral, floating point, Boolean, symbol, character, and string literals in Scala. We will also see multi-line strings and null values.

a. Integral Literals

These are usually Ints. When we use an 'l' or 'L' suffix, these are Longs.

Some valid integral literals:

07

0

7

111

0xFFFFFFFF

0798L

b. Floating-Point Literals

We've seen floating-point numbers like 7.0 and 7.7 rather than 7. When there's a suffix of 'f' or 'F', these are of type Float. Otherwise, they're of type Double.

Some valid floating-point literals:

0.0

1e70f

3.24179f

1.0e100

.1

c. Boolean Literals

We have two Boolean literals- true and false.

d. Symbol Literals

Scala has a Symbol case class:

```
1. package scala
2. final case class Symbol private (name: String) {
3. override def toString: String = "'" + name
4. }
```

So, a symbol ‘x’ is equivalent to scala.Symbol(“x”).

e. Character Literals

This is a single character delimited by quotes. A character is a printable Unicode character, and can be described by an escape sequence. We’ll discuss escape sequences next.

Some valid character literals:

‘a’

‘\n’

‘\u0042’

‘\t’

f. String Literals

Such a literal is a sequence of characters delimited by double quotes.

Some valid string literals:

“Hannah\nMontana”

“And then she said, \”Be here now is a dog’s purpose\””

g. Multi-line Strings

Like in Python, we can use three sets of double quotes to delimit a string to span it across multiple lines.

A valid multi-line string:

“””The first line

The second line\n

The fourth line”””

h. Null Values

A null value has the type scala.Null. This makes it compatible with each reference type. But what is really denotes is a reference value referring to a special ‘null’ object.

Any doubt yet in Data Types in Scala

5. Escape Values

An escape value is a backslash with a character that will escape that character to execute a certain functionality. We can use these in character and string literals. We have the following escape values in Scala:

Escape Sequences	Unicode	Description
\b	\u0008	Backspace BS
\t	\u0009	Horizontal Tab HT
\n	\u000a	Newline
\f	\u000c	Formfeed FF
\r	\u000d	Carriage Return CR
\”	\u0022	Double quote “
\’	\u0027	Single quote ‘
\\	\u005c	Backslash \

We can also represent characters with Unicodes between 0 and 255 with octal escapes (using backslashes). Upto three octal characters may follow. Take an example:

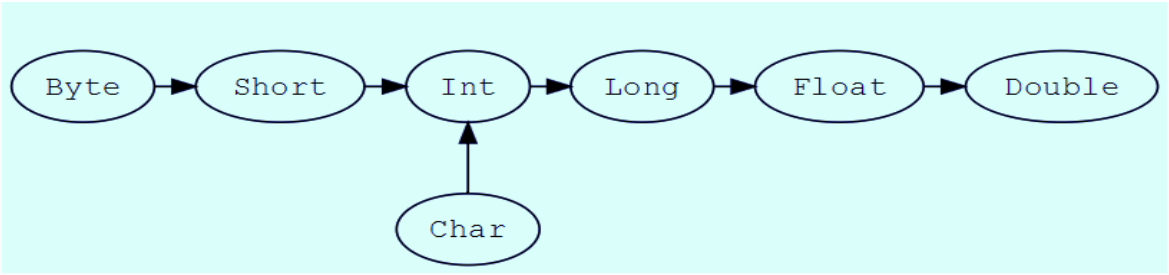
```
scala> println("Hey, how \b are you?\n\tI've been\r...\"waiting\"")
```

Hey, how are you?

…”waiting” been

6. Type Casting

In Scala, we can convert one type to another:



Scala Data Types: Type Casting

Let's take an example.

```
scala> val a:Int=7
a: Int = 7
scala> val b:Float=a
b: Float = 7.0
Hmm, let's take another example to make it clear.
scala> val c:Char='a'
c: Char = a
scala> val d:Int=c
d: Int = 97
scala> val e:Char='A'
e: Char = A
scala> val f:Float=e
f: Float = 65.0
```

This was all on Scala Data Types

7. Conclusion

This is all about Scala Data Types. Doesn't it feel just a bit different than Java and C++, yet so similar? Let us know in the comments if you have any doubts.

Scala Functions: Quick and Easy Tutorial

1. Functions in Scala

In this tutorial, we will discuss some Scala Functions. After this article, you'll be able to declare, define, and call your own functions in Scala.

Scala Functions

2. An Introduction to Scala Functions

When we want to execute a group of statements to perform a task a hundred times, we don't type them a hundred times. We group them into a function, put a call to that function inside a loop, and make that loop run a hundred times. Well, dividing our code into functions also makes for modularity – it makes it easier to debug and modify code. We can do this division according to the tasks we carry out in our code.

Scala is rich with built-in functions, and it also lets us create our own. Here, Scala functions are first-class values.

Scala also has methods, but these differ only slightly from Scala functions. A method belongs to a class; it has a name, a signature, [optionally,] some annotations, and some bytecode. A function is a complete object that we can store in a variable. Then, a method is a function that is a member of some object. You can also learn Scala Variable and Other Scala tutorials from our blog long with Scala Functions.

3. Declaring a Function in Scala

To create a function, we use the keyword 'def'.

a. Syntax

Here's the syntax you'll want to follow:

```
1. def functionName(parameters:typeofparameters):returntypeoffunction={
2. //statements to be executed
3. }
```

Now here, while you must mention the types of parameters, the type of function is optional. What else is optional is the '=' operator. Let's see examples for each kind of declaration.

b. Without = | Without Parameters

You can consider this to be the simplest kind of function declaration.

```
scala> def sayhello(){
| println("Hello")
| }
sayhello: ()Unit
```

Now, let's call it.

```
scala> sayhello()
```

Hello

As you can see, since we make this function return nothing, it returns Unit. All that this function does is take no parameters and simply print Hello.

c. Without = | With Parameters

We can declare a function that takes parameters to work on to produce a result. Here, we simply print that result instead of returning it.

```
scala> def sum(a:Int,b:Int)
| {
| println(a+b)
| }
sum: (a: Int, b: Int)Unit
scala> sum(2,5)
```

7

This code prints the sum of two integers. By default, this function returns Unit.

d. With = | Without Parameters

With the = operator, a function takes a return type, and also returns a value of that type.

```
scala> def func():Int={
| return 7
| }
func: ()Int
In Scala, if you type in the same commands again, it will detect that:
scala> scala> def func():Int={
```

// Detected repl transcript. Paste more, or ctrl-D to finish.

```
| return 7
| }
func: ()Int
```

To break out of this, press Ctrl+D.

```
// Replaying 1 commands from transcript.

scala> def func():Int={
return 7
}
func: ()Int
scala>
```

Note that this, however, wouldn't work:

```
scala> def func()={
| return 7
| }
<console>:15: error: method func has return statement; needs result type
return 7
^
```

e. With = | With Parameters

Let's try defining a function to work on some parameters to return a result.

```
scala> def sum(a:Int,b:Int):Int={
| return a+b
| }
sum: (a: Int, b: Int)Int
scala> sum(2,5)
res22: Int = 7
```

This would've worked too:

```
scala> def sum(a:Int,b:Int):Int={
| println("Adding")
| a+b
| }
sum: (a: Int, b: Int)Int
scala> sum(2,5)
Adding
res23: Int = 7
```

This means that a function will return the value of the expression right before the closing curly brace; the 'return' keyword isn't necessary.

Learn: Scala DataTypes with Syntax and Examples

4. Recursion in Scala Functions

A function involves in recursion when it makes a call to itself. Let's take an example:

```
scala> def factorial(n:Int):Int={
| if(n==1)
| {
| return 1
| }
```

```
| n*factorial(n-1)
| }
factorial: (n: Int)Int
scala> factorial(6)
res0: Int = 720
scala> factorial(1)
res1: Int = 1
scala> factorial(4)
res2: Int = 24
scala> factorial(10)
res4: Int = 3628800
```

This function correctly calculates the factorial of an integer one or greater.

Learn: Scala Comments with Syntax and Example

5. Default Arguments in Scala

If we elide an argument in a function call, Scala will take the default value we provided for it.

```
scala> def func(a:Int,b:Int=7){
| println(a*b)
| }
func: (a: Int, b: Int)Unit
scala> func(2,5)
10
scala> func(2)
14
```

But make sure that any default arguments must be after all non-default arguments. The following code raises an error:

```
scala> def func(a:Int=7,b:String){
| println(s"$a $b")
| }
func: (a: Int, b: String)Unit
scala> func("Ayushi")
<console>:13: error: not enough arguments for method func: (a: Int, b: String)Unit.
Unspecified value parameter b.
func("Ayushi")
^
```

Learn: Scala String: Creating String, Concatenation, String Length

6. Scala Named Arguments

When we want to pass arguments to a function in a different order, we can pass them with names:

```
scala> def diff(a:Int,b:Int):Int={
| return b-a
| }
diff: (a: Int, b: Int)Int
```

```
scala> diff(2,3)
res12: Int = 1
scala> diff(b=3,a=2)
res13: Int = 1
```

Learn: Scala Arrays and Multidimensional Arrays in Scala

7. Scala Functions with Variable Arguments

When we don't know how many arguments we'll want to pass, we can use a variable argument for this:

```
scala> def sum(args:Int*):Int={
| var result=0
| for(arg<-args){
| result+=arg
| }
| result}
sum: (args: Int*)Int
Let's try calling this with different number of arguments.
scala> sum(1)
res0: Int = 1
scala> sum(2,3)
res1: Int = 5
scala> sum(4,5,2,7)
res2: Int = 18
```

8. Higher-Order Functions

Since Scala is a highly functional language, it treats its functions as first-class citizens. This means that we can pass them around as parameters, or even return them from functions.

Let's first define a function sayhello:

```
scala> def sayhello(s:String){
| println(s"Hello, $s")
| }
sayhello: (s: String)Unit
Suppose we have a string 'name':
scala> val name:String="Ayushi"
name: String = Ayushi
```

Now, let's define a function func that calls 'sayhello' on 'name'.

```
scala> def func(f:String=>Unit,s:String){
| f(s)
| }
func: (f: String => Unit, s: String)Unit
scala> func(sayhello,name)
```

Hello, Ayushi

In this example, we used a Scala function as a parameter to another.

Learn: Tuples in Scala – A Quick Introduction

9. Nested Functions in Scala

With Scala, we can define a function inside another. This is what a local function looks like:

```
scala> def outer(){
| println("In outer")
| def inner(){
| println("In inner")
| }
| inner()
| }
outer: ()Unit
scala> outer()
```

In outer

In inner

Let's take another example.

```
scala> def outer(a:Int){
| println("In outer")
| def inner(){
| println(a*3)
| }
| inner()
| }
outer: (a: Int)Unit
Now, let's call it:
scala> outer(3)
```

In outer

9

Learn: Scala Closures with Examples | See What is Behind the Magic

10. Scala Anonymous Functions

Anonymous functions in Scala is lightweight syntax to create a function in one line of code. We've been using this in our articles so far. Anonymous functions are function literals, and at runtime, they instantiate into objects called function values.

```
scala> val sum=(a:Int,b:Int)=>a+b
sum: (Int, Int) => Int = $$Lambda$1116/1013657610@5bdb6ea8
scala> sum(2,3)
res3: Int = 5
```

Let's take another Example.

```
scala> val x=()=>println("Hello")
x: () => Unit = $$Lambda$1122/1244890076@72ce8a9b
scala> x()
```

Hello

11. Scala Currying Functions

If a Scala function takes multiple parameters, we can transform it into a chain of functions where each takes a single parameter. We use multiple parameter lists for curried functions.

```
scala> def sum(a:Int)(b:Int)={
| a+b
| }
sum: (a: Int)(b: Int)Int
scala> sum(2)(5)
```

res3: Int = 7

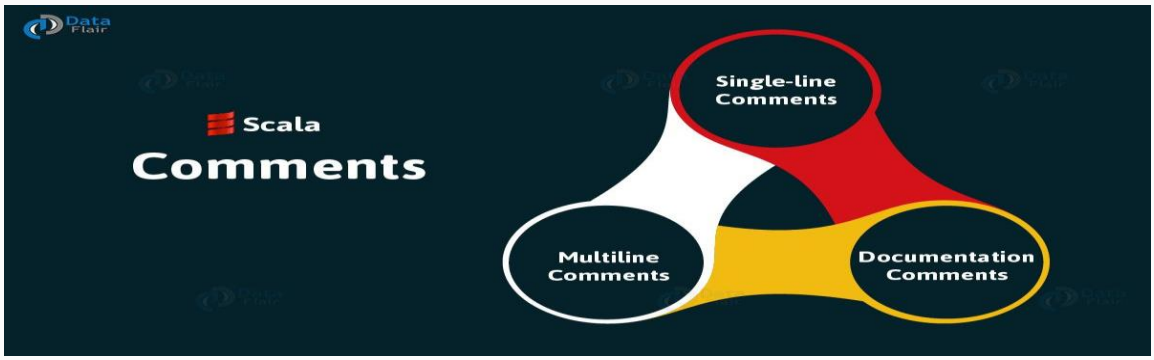
12. Conclusion

In this blog, we discussed Scala functions, recursion, and default, named, and variable arguments. We also learned about higher-order functions, nested functions, anonymous functions, and currying.

Scala Comments with Syntax and Examples

1. Scala Comments

The objective of this 'Scala Comments' article is to tell you about comments in Scala. Let's begin.



Scala Comments

2. Introduction to Scala Comments

Comments are entities, in your code, that the compiler/interpreter ignores. You can use them to explain code, and also to hide code details.

3. Scala Single-line Comments

When you want only one line of a comment, you can use the characters '//' preceding the comment.

object Main extends App

```
1. {
2. print("Hi!")
3. //This is a comment, and will not print
4. }
```

Here's the output:

Hi!

4. Scala Multiline Comments

When your comments will span more than one line, you can use a multiline comment. To declare it, we use the characters '/*' and '*/' around the comment.

object Main extends App

```
1. {
2. print("Hi!")
3. /*This is a multiline comment,
4. and will not print
5. kbye*/
6. }
```

The output:

Hi!

5. Documentation Comments

A documentation comment is available for quick documentation lookup, and are open for review on pressing Ctrl+Q in the IntelliJ. Your compiler uses these comments to document the source code.

We have the following syntax for creating a documentation comment:

object Main extends App

```
1. {
2. print("Hi!")
3. /**
```

```
4.  * This is a documentation comment
5.  * This is a demo
6.  */
7.  }
```

Output:

Hi!

To declare such a comment, type the characters '/**', and then type something, or press Enter. The IDE will put in a '*' every time you press enter. To end such a comment, type '/' after one of the carets(*).

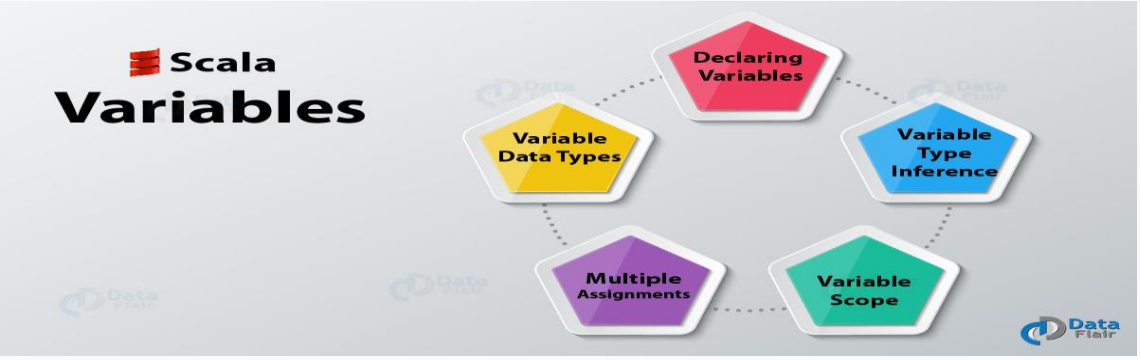
6. Conclusion

Concluding Scala Comments article, Scala has three kinds of comments- single-line, multiline, and documentation comments.

Scala Variables with Examples

1. Scala Variables

What are Scala variables, and how do we declare them? What is type inference? How do we perform multiple assignments? What is variable scope? These are the questions we'll be answering today. If you face any query regarding Scala Variables, please ask in comments.



Scala Variables

2. Introduction to Variables in Scala

A variable is a reserved memory location to store values. The compiler allocates memory and decides what goes in based on a variable's data type. We can assign different types like integers, characters, or decimals to variables. Let's begin.

3. Declaring Scala Variables

We can declare a Scala variable as a constant or as a variable:

a. Constant

To declare a value that we cannot mutate/change, we use the keyword 'val'. This is immutable.

```
1. val x:Int=7
2. val name:String="Ayushi"
```

b. Variable

A variable is mutable. We can change its value as many times as we like. Declare it using the keyword 'var'.

```
1. var name:String="Ayushi"
2. name="Megha"
```

4. Scala Variable Type Inference

When we assign an initial value to a variable, the compiler infers its type based on the types of the subexpressions and the literals. This means that we don't always have to declare the type of a variable.

```
1. val x=1+2
2. var name="Ayushi"
```

5. Variable Data Types

We can specify a variable's type after a colon after its name. Take an example:

```
1. val x:Int=7
2. val x:Int
```

6. Multiple Assignments

How do we fit multiple assignment statements into one statement? We assign a Tuple to a val variable.

```
1. val (x: Int, y: String) = Pair(7, "Ayushi")
2. val (x, y) = Pair(7, "Ayushi")
```

7. Scala Variable Scope

In Scala, we have three kinds of scopes for variables. Let’s see them one by one.

a. Fields

If an object owns a variable, the variable is a field in it. We can access fields from inside every method in the object. Well, we can also access them outside the object if we declared them with the right access modifiers. A field may be mutable or immutable, and we can define them using ‘var’ or ‘val’.

b. Method Parameters

A method parameter is a variable that we can use to pass a value inside a method when we call it. We can only access it inside the method, but if we have a reference to that object from outside the method, we can access the objects from outside. A method parameter is always immutable, and we define it using the ‘val’ keyword.

c. Local Variables

A local variable is one we declare inside a method. While we can only access it from inside a method, if we return the objects, that we create, from the method, they may escape it. A local variable may be mutable or immutable, and we may define it using the keywords ‘var’ or ‘val’.

8. Conclusion

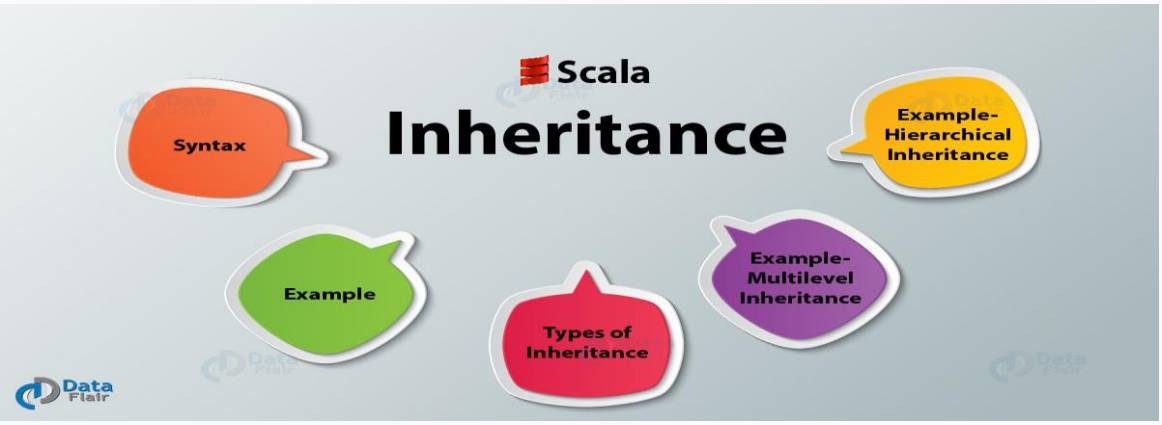
That is all about variables. Hope you like our Scala Variable tutorial.If you have any query regarding Scala Variables, please comment. See you again with another topic to learn with Scala. Have a good day.

Scala Inheritance – Syntax, Example & Types of Inheritance in Scala

1. Objective

In our last **Scala Tutorial**, we discuss **Scala Case Class**. Now, we will discuss Scala inheritance with syntax and examples. Along with this, we will cover different types of inheritance in Scala Programming. At last, we will see examples of multilevel inheritance and Hierarchical Inheritance in Scala.

So, let's start Scala Inheritance.



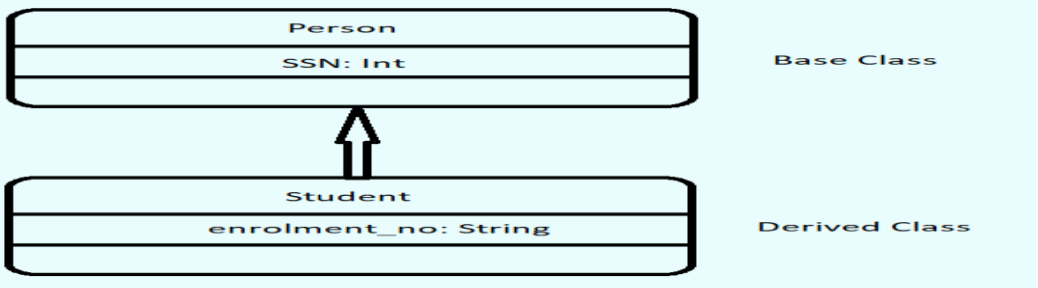
Scala Inheritance – Syntax, Example & Types of Inheritance in Scala

2. Scala Inheritance

When a class inherits from another, it means it extends another. We use the 'extends' keyword for this. This lets a class inherit members from the one it extends and lets us reuse code.

Let's Learn Scala Map with Examples Quickly & Effectively

The class that extends is the subclass, the child class, or the derived class. The other class is the superclass, the parent class, or the base class.



Scala Inheritance

Scala Inheritance is an IS-A relationship. You can also call it a generalization. A Student is a Person.

3. A syntax of Scala Inheritance

To carry out Scala inheritance, we use the keyword 'extends':

```
class Student extends Person(){
/*your code
*goes here
*/
```

Let's Discuss Scala String Interpolation – s String, f String and raw string Interpolator

4. Scala Inheritance Example

Let's take an example of Scala Inheritance.

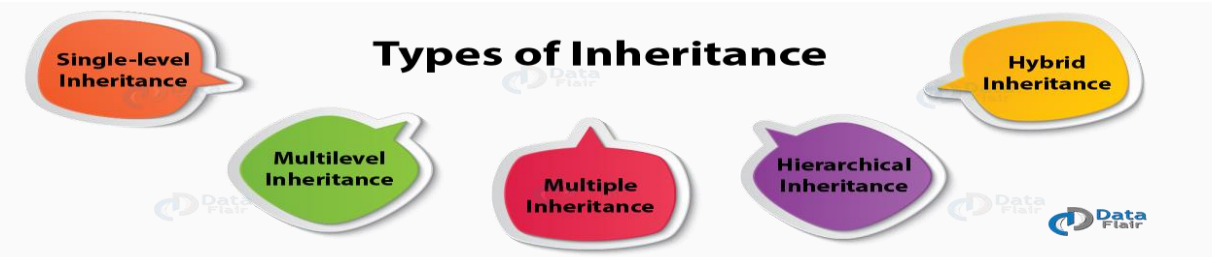
```
scala> class Person{
| var SSN:String="999-32-7869"
| }
defined class Person
scala> class Student extends Person{
| var enrolment_no:String="0812CS141028"
| println("SSN: "+SSN)
| println("Enrolment Number: "+enrolment_no)
| }
defined class Student
scala> new Student()
SSN: 999-32-7869
Enrolment Number: 0812CS141028
res0: Student = Student@42cfd794
```

In this example, we have two classes- Person and Student. We make Student extend Person. This means a Student is a Person.

Since Student extends Person, it inherits the attribute holding the social security number. In class Student, we print SSN and enrolment_no. Finally, we create an object of class Student.

5. Types of Inheritance in Scala

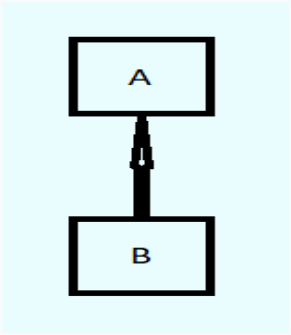
Scala supports five kinds of inheritance:



Types of Inheritance in Scala

a. Single-level Inheritance in Scala

Single-level inheritance is when one class inherits from a single other class.

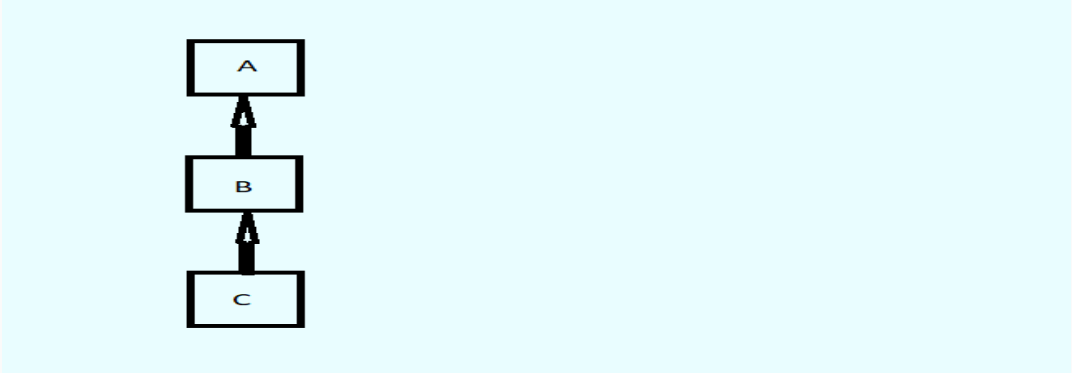


Single-level Inheritance

Read about Scala Access Modifiers: Public, Private and Protected Members

b. Multilevel Inheritance in Scala

When one class extends another, which in turn extends another, it is an instance of multilevel inheritance.

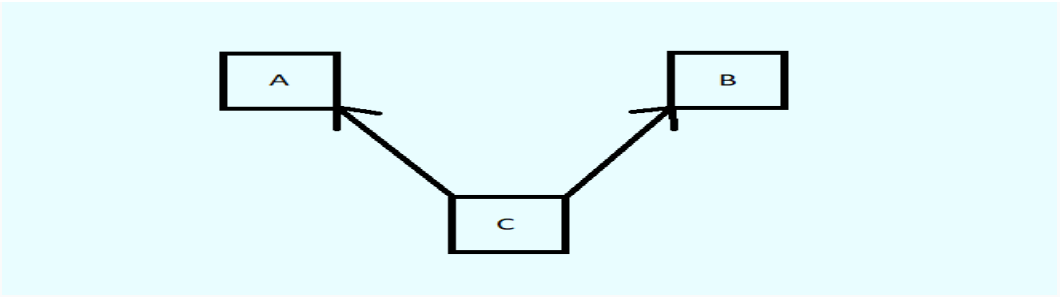


Multilevel Inheritance in Scala

Let's Scala if-else Statements with examples

c. Multiple Inheritance in Scala

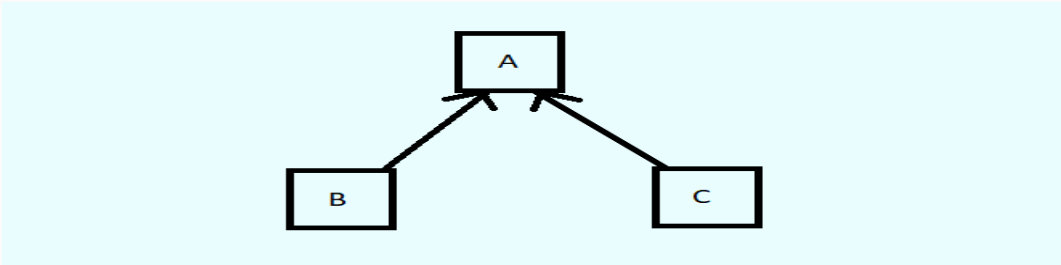
When one class inherits from multiple base classes, it is a case of multiple inheritances.



Multiple Inheritance in Scala

d. Hierarchical Inheritance in Scala

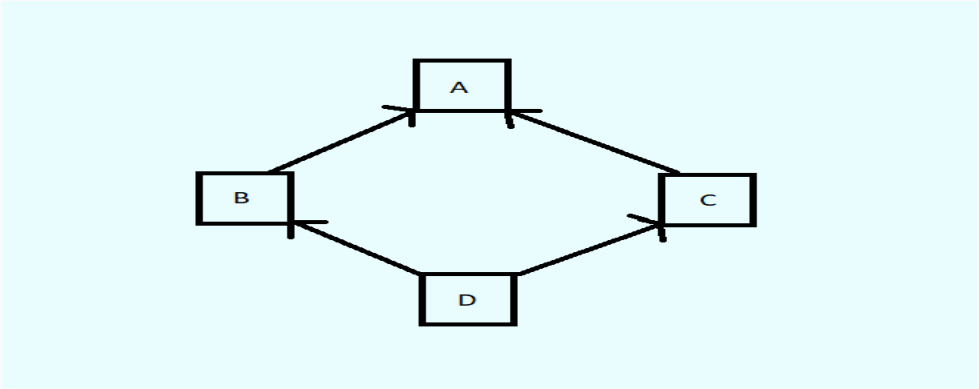
When more than one class inherits from one base class, it is said to be hierarchical inheritance.



Hierarchical Inheritance in Scala

e. Hybrid Inheritance in Scala

Hybrid inheritance is a combination of at least two kinds of inheritance.



Hybrid Inheritance in Scala

Let’s explore Scala Operator in detail

6. Example- Multilevel Inheritance

```
scala> class A{
| println("A")
| }
defined class A
scala> class B extends A{
```

```
| println("B")
| }
defined class B
scala> class C extends B{
| println("C")
| }
defined class C
scala> new C()
A
B
C
res1: C = C@347f8029
```

In this example, we observe that class C extends class B and class B extends A.

7. Example- Hierarchical Inheritance

```
scala> class A{
| println("A")
| }
defined class A
scala> class B extends A{
| println("B")
| }
defined class B
scala> class C extends A{
| println("C")
| }
defined class C
scala> new B()
A
B
res2: B = B@64c1a76e
scala> new C()
A
C
res3: C = C@193f5509
```

In this example, classes B and C inherit from class A.

Do you know Scala Job Opportunities: Profile, Salary & Top Organizations

So, this was all about Scala Inheritance, Hope you like our explanation.

8. Conclusion

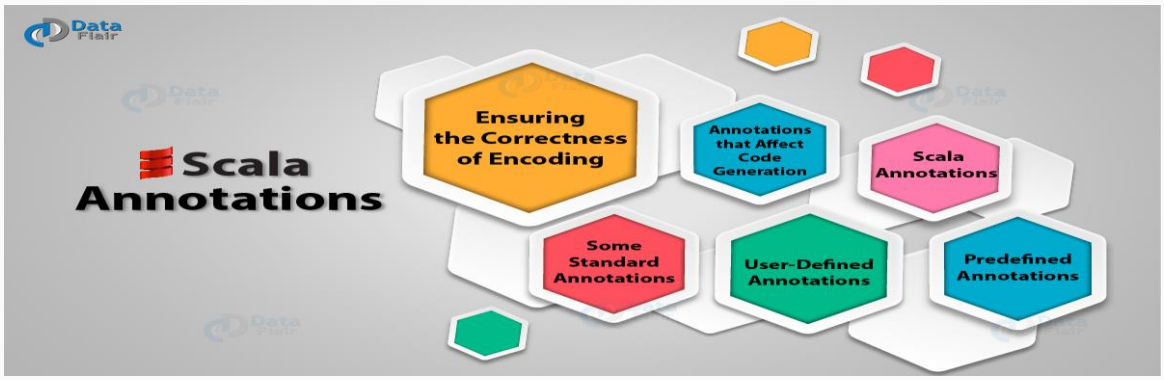
Hence, we studied what is Scala Inheritance with example and syntax. In addition, we saw different types of inheritance in Scala. If you have a doubt, feel free to ask in the comment section.

Scala Annotations – Predefined & User-Defined Annotations

1. Objective

In our last tutorial, we study **Scala Currying Function** and here, we will see Scala Annotations, we will learn about Predefined Annotations in **Scala Programming Langauge**. Along with this, we will cover Scala User defined annotation and define how annotations affect code generation. At last, we will discuss some Standard Scala Annotations.

So, let's begin with Scala Annotations.



2. Scala Annotations

Scala Annotations let us associate meta-information with definitions. We apply an annotation clause to the first definition or the declaration following it. We can use multiple annotations before a definition or declaration in any order.

Scala annotation is of the form @c or @c(a1,…,an), where c is a constructor for class C, which conforms to the scala.Annotation class.

One such annotation is @deprecated. When we put this before a method, the compiler issues a warning when we use this method. Let's take an example of Scala Annotations.

Let's Study Scala Method Overloading with Example

```
scala> @deprecated
| def sayhello()={"hello"}
<console>:11: warning: @deprecated now takes two arguments; see the scaladoc.
@deprecated
^
sayhello: ()String
scala> print(sayhello())
<console>:13: warning: method sayhello is deprecated
print(sayhello())
```

```
10. ^
```

Hello

This lets us use the method; it returns a warning, but not an error.

We can attach an annotation to a **variable**, an expression, a method, or any other element. This can be a class, an object, a trait, or anything else. When with a declaration or a definition, it appears in front; when with a type, it appears after. With an expression, it appears after and is separated by a colon. To an entity, we can apply more than one such annotation. Here's an example:

For classes: @deprecated("Use D", "1.0") class C { … }

For types: String @local

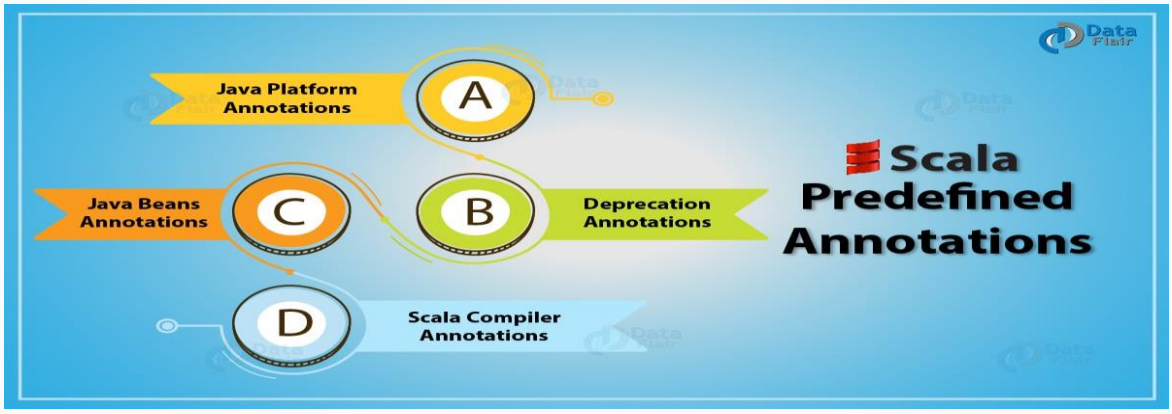
For variables: @transient @volatile var m: Int

For expressions: (e: @unchecked) match { … }

Let's Know Reasons to Learn Scala Programming Langauge

3. Predefined Annotations in Scala

In this Scala Annotations tutorial, we will talk about four kinds of Scala annotations-**Java** Platform, Java Beans, Deprecation, and Scala Compiler annotations.



Predefined Annotations in Scala

a. Java Platform Annotations

We have the following annotations on the Java platform:

- **@transient-** This means a field is non-persistent.
- **@volatile-** A volatile field is one that can change its value outside the program's control.
- **@SerialVersionUID(<longlit>)-** A serial version identifier is a long constant. This annotation attaches this **identifier** to a class.

- **@throws(<classlit>)-** This mentions the class or one of the superclasses of the class for a certain **checked exception**.

b. Java Beans Annotations

Under this, we have two annotations:

- **@scala.beans.BeanProperty-** To the class containing the **variable** to which we apply this, it adds getter and setter methods getX and setX. It does so in the Java bean style. Notice the uppercase for the first letter of the variable after 'get'/'set'. It only generates a getter for immutable values.
- **@scala.beans.BooleanBeanProperty-** Unlike scala.reflect.BeanProperty, it names the getter method as isX instead of getX.

Read about Scala if-else Statements with Statements

c. Deprecation Annotations

- **@deprecated(message: <stringlit>, since: <stringlit>)-** Like we've seen in the previous section, this marks a definition as deprecated. Accessing such a definition only issues a warning. It holds an attribute 'since' which denotes from when it is to be considered deprecated.
- **@deprecatedName(name: <symbollit>)-** This lets us mark a formal parameter as deprecated.

d. Scala Compiler Annotations

- **@unchecked-** When we apply this annotation to a match expression selector, it suppresses warnings about non-exhaustive pattern matches. For an example, it produces no warnings for the following definition:

```
1. def f(x: Option[Int])=(x: @unchecked) match{
2. case Some(y)=>y
3. }
```

- **@uncheckedStable-** This definition lets a value appear in a path- this is even if it is volatile. Take a look at the following legal definitions:

```
1. type A{type T}
2. type B
3. @uncheckedStable val x: A with B
```

```
4. val y: x.T
```

Let's Discuss Scala do while Loop with Examples

In this, the type A with B is volatile. So, without Scala annotation, the designator x isn't a path. Then, x.T is malformed. However, the annotation has no effect on definitions with non-volatile types.

- **@specialized-** This tells the compiler to generate specialized definitions for primitive types. Optionally, we can provide a list of primitive types. Check the following code:

```
1. trait Function0[@specialized(Unit, Int, Double)T]{
2. def apply:T
3. }
```

The compiler uses the specialized version when the static type for an expression matches a specialized variant of a definition.

4. Scala User-Defined Annotations

We also have some platform-dependent or application-dependent tools. We use two sub-traits from a Scala. Annotation to indicate how we retain these annotations. Class files hold an annotation class instances for a class that inherits from the trait Scala.ClassfileAnnotation. For one that inherits from the trait scala.StaticAnnotation, the instances are visible to the Scala type-checker for every compilation unit where we access the annotated symbol. An annotation class may inherit from both of these classes, but if it inherits from neither, its instances are visible only locally within the compilation run analyzing them.

5. Ensuring the Correctness of Encoding

Some Scala annotations like @tailrec check whether the condition is met. If it isn't, compilation fails. @tailrec makes sure that a method is tail-recursive. Tail recursion is when a function calls itself at its end/tail. This helps keep memory requirements constant. Let's take an example.

```
1. scala> import scala.annotation.tailrec
2. import scala.annotation.tailrec
3. scala> def facto(n:Int):Int={
4. | @tailrec
5. | def fact(n:Int,result:Int):Int={
6. | if(n==1) result else fact(n-1,result*n)
7. | }
8. | fact(n,1)
9. | }
10. facto: (n: Int)Int
11. Scala> facto(4)
```

```
res1: Int = 24
```

Here, @tailrec makes sure that facto() is tail-recursive. The following format would raise an error:

```
scala> def facto(n:Int):Int={
| @tailrec
| def fact(n:Int):Int={
| if(n==1) 1 else n*fact(n-1)
| }
| fact(n)
| }
<console>:15: error: could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position
if(n==1) 1 else n*fact(n-1)
^
```

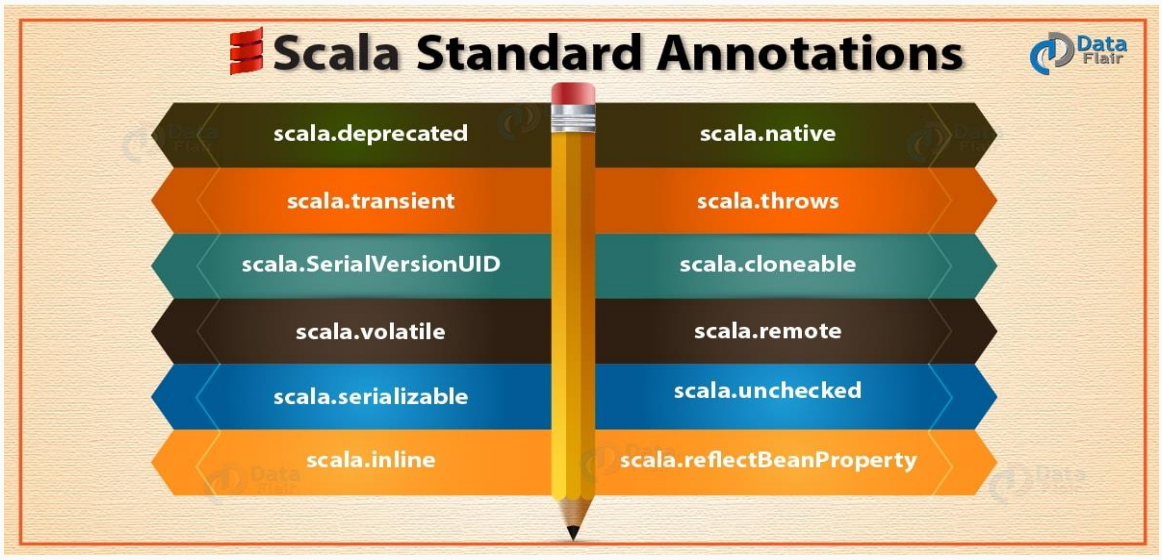
Let's discuss Scala Access Modifiers: Public, Private and Protected Members

6. Scala Annotations Affects Code Generation

Remember inline methods from Java? They insert the method body code at the place where a call is made to it. Although this results in a longer bytecode, such code runs faster. @inline makes the compiler inline a method if a heuristic about the size of the generated code is met. But it doesn't guarantee inlining. Scala Annotations like these affect the generated code.

7. Some Standard Annotations

Scala has various standard Scala annotations. Here, we discuss some of those.



Standard Annotations in Scala

a. scala.deprecated

When a method is deprecated, it means it is removed.

b. scala.transient

A transient field is non-persistent.

c. scala.SerialVersionUID

This denotes the static field SerialVersionUID of a serializable class.

d. scala.volatile

Using this, we can use a mutable state in concurrent programs.

e. scala.serializable

This marks a class as serializable.

Read about Scala Data Types with Examples | Escape Value & Typecasting

f. scala.inline

This means the compiler should try to make the annotated method inline.

g. scala.native

This marks native methods. Such a method is one whose implementation is in another language.

h. scala.throws

This denotes the **exceptions** a method throws.

i. scala.cloneable

This marks a class as cloneable.

j. scala.remote

This marks a class as remotable.

k. scala.unchecked

This applies to a selector in a match expression.

Let's discuss Scala Functions in detail

l. scala.reflectBeanProperty

This adds setter and getter methods. When attached to a field, this process follows the JavaBean convention.

So, this was all about Scala Annotations Tutorial. Hope you like our explanation.

8. Conclusion

Hence, we studied about Scala Annotations, associates meta-information definitions. This means they add extra information to the code. In addition, we discussed predefined and user-defined annotations in Scala. At last, we covered, how to ensure Correctness of Encoding and define the effect of Scala annotations in code generation. Hope you understood what we had to say. Drop your queries in the comments.

Scala if-else Statements with Example

1. Scala if else

Decision making is an important part of any programming language. For this purpose, we have if-else statements. Let's see how Scala if else works. Before that, you can refer our previous article on **Scala Control Statements**. We will start with Scala if Statements, Scala if else Statements, Scala if else – if – else statements and nested if else statements.

Scala if else

2. Scala if Statement

We begin with the if statement. This statement evaluates an expression. If true, it executes the block of statements under it.

a. Syntax

Take a look at the syntax:

```
1. if(Boolean_expression)
2. {
3. //Code to execute if expression is true
4. }
```

If the Boolean expression evaluates to true, the block of code under the if-statement executes. Otherwise, the code right after the code block under the if-statement executes.

b. Scala if Example

Okay, let's do an example.

object Main extends App

```
{
val x=7
if(x<10)
{
println("Woohoo!")
}
}
```

And here's the output:

Woohoo!

Read: Tuples in Scala

3. Scala if-else Statement

What to do if the expression turns out to be false? Execute the code under the else statement.

a. Syntax

For an if-else statement, follow this syntax:

```
if(Boolean_expression)
{
//Code to execute if expression is true
}
else
{
//Code to execute if expression is false
}
```

b. Scala if else example

Let's take an example.

object Main extends App

```
{
val x=17
if(x<10)
{
println("Woohoo!")
}
else
{
println("Oh no!")
}
}
```

And the output is:

Oh no!

Read: Scala Partial Functions

4. Scala if-else if-else Statement

When we want to check another condition when one fails, and yet another when that one fails, and so on, we can use this. One 'if' statement can have any number of 'else if' statements. But remember to end these with a final 'else' statement.

a. Syntax

Follow this syntax:

```
if(Boolean_expression 1)
{
//Code to execute if expression 1 is true
}
else if(Boolean_expression 2)
{
//Code to execute if expression 2 is true; checked only when expression 1 is false
}
else if(Boolean_expression 3)
{
//Code to execute if expression 3 is true; checked only when expression 2 is false
}
else
{
//Code to execute if expression 3 is false
```

b. Scala if else if else example

Time for an example.

object Main extends App

```
{
val x=17
if(x<10)
{
println("Woohoo!")
}
else if(x<20)
{
println("Okay")
}
else
{
println("Oh no!")
}
}
```

And who's guessing the output?

Okay

Read: Pros and Cons of Scala

5. Scala Nested if-else Statement

You can put an ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement. Full flexibility!

a. Syntax

Follow this syntax:

```
if(Boolean_expression 1)
{
//Code to execute if expression 1 is true
if(Boolean_expression 2)
{
//Code to execute if expression 2 is true
}
else
{
//Code to execute if expression 2 is false
}
}
else
{
//Code to execute if expression 1 is false
```

b. Scala Nested If Else Example

Let’s take an example for this.

object Main extends App

```
{
val x=7
if(x<10)
{
println("Woohoo!")
if(x<5)
{
println("Hi")
}
else
{
println("Bye")
}
}
else
{
println("Oh no!")
}

}
```

And the output is:

Woohoo!

Bye

6. Conclusion

In this lesson, we learned about Scala if, Scala if else, Scala if-else if-else, and Scala nested if statements. Next, we'll discuss loops in Scala.

Scala String: Creating String, Concatenation, String Length

1. Scala String

In this chapter, we will discuss Scala String in Detail. After this, you will be able to create a string, find its length, create a format string, and concatenate strings. We can't wait to begin.

Introduction to Scala String

2. Introduction to Strings in Scala

An immutable object, a string in Scala is a sequence of characters. Once we declare it, we cannot modify it. The following is a string:

"Ayushi"

3. Creating a Scala String

To define a Scala string, we can use a 'var' or a 'val'. When we use 'var', we can reassign to it. This isn't the same with 'val':

```
scala> var word="rest"
word: String = rest
scala> word="reset"
word: String = reset
```

By the way, reassigning 'word' to an Int will cause an error:

```
scala> word=7
<console>:12: error: type mismatch;
found : Int(7)
required: String
word=7
^
```

Now with 'val':

```
scala> val word="rest"
word: String = rest
scala> word="reset"
<console>:12: error: reassignment to val
word="reset"
```

We can also mention the data type if we don't want to rely on type inference.

```
scala> var word:String="rest"
word: String = rest
scala> word="reset"
word: String = reset
```

To make a lot of changes in your Scala string, you can use the StringBuilder class instead.

4. Finding String Length

An accessor method is one that'll tell us about an object. One such method for strings is length(). It returns the number of characters a string holds. In the previous example,

```
scala> word.length()
res0: Int = 5
```

Here are more than a couple more examples:

```
scala> "Ayushi".length()
res1: Int = 6
scala> "".length()
res2: Int = 0
scala> " ".length()
res3: Int = 1
```

5. Concatenating Strings

Concatenating two strings in Scala means joining the second to the end of the first. For this, we have the method concat() in the String class:

```
scala> "ab".concat("cde")
res5: String = abcde
```

Let's try using more than one:

```
scala> "Ayushi".concat(" ".concat("Sharma"))
res6: String = Ayushi Sharma
```

We could also have done this using the + operator:

```
scala> "Ayushi"+" "+"Sharma"
res7: String = Ayushi Sharma
```

6. Creating Format Strings

For the times we want to format numbers/values into our string, we can make use of one of the methods printf() and format(). Other than this, the class String has the method format() to return a String object instead of a PrintStream object.

```
scala> var (a:Int,b:Int,c:Int)=(7,8,9)
a: Int = 7
b: Int = 8
c: Int = 9
scala> printf("a=%d, b=%d, c=%d",a,b,c)
```

a=7, b=8, c=9

For Integer data, we used %d. For Float and String data, we use %f and %s, respectively.

7. Conclusion

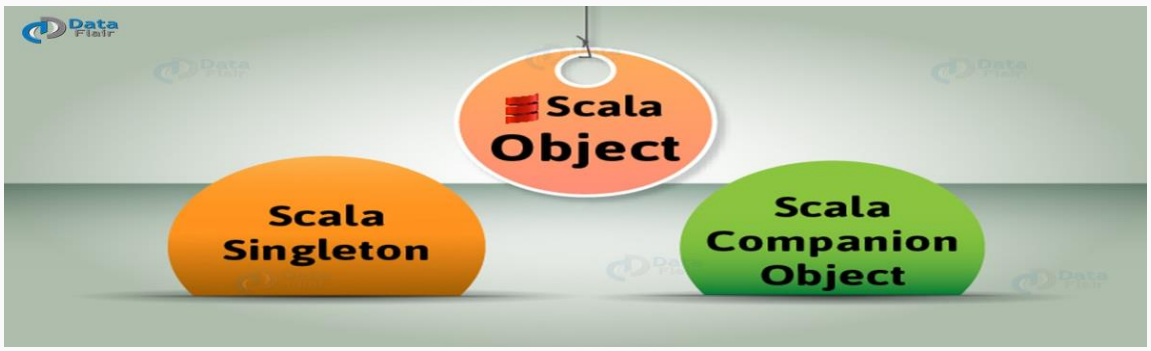
With a little practice, you'll go a long way with Scala. Next, we will learn string interpolation and methods on strings. See you again.

Scala Object – Scala Singleton & Scala Companion Object

1. Objective

In our last **Scala Programming Tutorial**, we talked about **Scala Annotations**. Today, we will discuss Scala Object and different types of Objects in Scala. Moreover, we sill see Scala singleton object with example and Scala companion objects with example.

So, let's start Scala Singleton & Scala Companion Object.



Scala Object – Scala Singleton & Scala Companion Object

2. Scala Singleton Objects

In Scala, an object is a class with exactly one instance. Like a lazy val, it creates lazily when we reference it. It is a value, and as a top-level value, it is a Scala singleton. To define an object, we use the keyword 'object':

```
scala> object Box
defined object Box
```

The methods we declare inside Scala singleton object are globally accessible, we don't need an object for this. We can import them from anywhere in the program. And since there is no idea of 'static' in Scala, we must provide a point of entry for the program to execute. Scala singleton object makes for this. Without such an object, the code compiles but produces no output.

Let's Learn Scala Case Class and How to Create Scala Object

Consider a package:

```
package logging
object Logger{
def info(message:String):Unit=println("Info: "+message)
}
Now consider another package:
import logging.Logger.info
class Project(name:String,days:Int)
class Test{
val project1=new Project("Book writing",365)
val project2=new Project("Revision",30)
info("Planned")
}
```

With the import statement, we can use the method info() without a problem.

Scala singleton object can also extend traits and classes.

3. Scala Singleton Example

```
scala> object SingletonObject{
| def greet(){
| println("Hello")
| }
| }
defined object SingletonObject
scala> SingletonObject.greet()
```

Hello

As you can see, we don't need an object to call the method greet() of this singleton object.

Do You Know Why Scala Called Object Oriented Programming

4. Scala Companion Object

Coming from Scala singleton objects, we now discuss companion objects. A Scala companion object is an object with the same name as a class. We can call it the object's companion class. The companion class-object pair is to be in a single source file. Either member of the pair can access its companion's private members. Let's take an example.

```
scala> class CompanionClass{
| def greet(){
| println("Hello")
| }
| }
defined class CompanionClass
scala> object CompanionObject{
| def main(args:Array[String]){
| new CompanionClass().greet()
| println("Companion object")
| }
| }
defined object CompanionObject
```

So, this was all about Scala Object Tutorial. Hope you like our explanation.

5. Conclusion

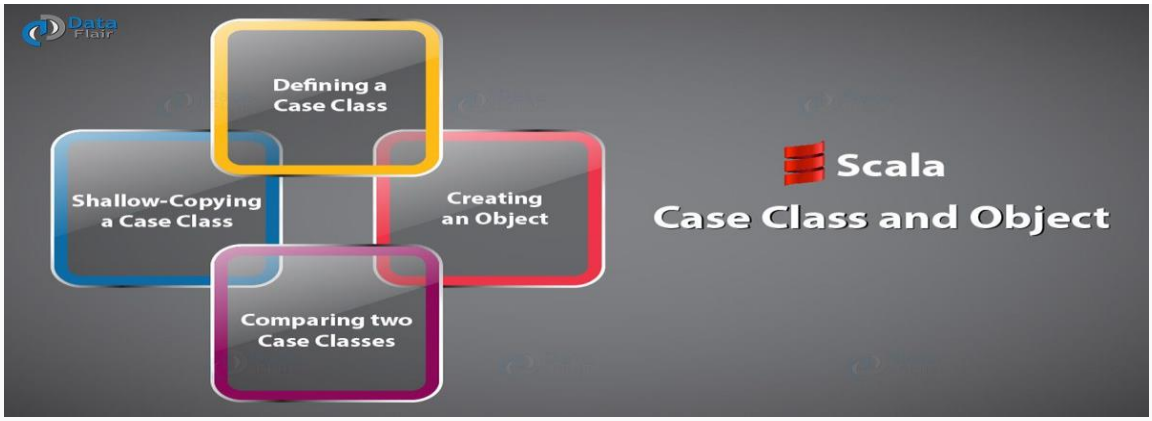
Hence, after reading this Scala Object Tutorial, we got to know what is singleton object and how it provides an entry for program execution. In addition, we talked about Scala companion object, an object for a class with the same name. Furthermore, if have any doubt, feel free to ask in the comment section.

Scala Case Class | Create Scala Object

1. Objective

In this **Scala tutorial**, we will see how to define Scala case class. Moreover, we will also take a look at how to compare case classes and create shallow copies. Along with this, we will also learn how to create a Scala object.

So, let's explore Scala Case Class and Scala Object.



Scala Case Class and Scala Objects

Read Scala Syntax: An Introductory Scala Tutorial

2. What is Scala Case Class?

A Scala Case Class is like a regular class, except it is good for modeling immutable data. It also serves useful in pattern matching, such a class has a default apply() method which handles object construction. A scala case class also has all *vals*, which means they are immutable.

Let's Revise Scala Syntax with example.

3. Defining a Case Class

To define a minimal Scala Case Class, we need the keywords 'case class', an identifier, and a parameter list. We can keep the parameter list empty.

So, let's define a class 'Song'.

```
scala> case class Song(title:String,artist:String,track:Int)
```

4. Creating a Scala Object

And now, it's time to create a Scala Object for this Scala class.

```
scala> val stay=Song("Stay","Inna",4)
stay: Song = Song(Stay,Inna,4)
```

To create a Scala Object of a case class, we don't use the keyword 'new'. This is because its default apply() method handles the creation of objects.

Let's try accessing the title of this object:

```
scala> stay.title
res1: String = Stay
And now, let's try modifying it.
scala> stay.title="Me Gusta"
<console>:12: error: reassignment to val
stay.title="Me Gusta"
```

```
^
```

Failed.

Explore Scala Closures with Examples | See What is Behind the Magic

So, this tells us that Scala case classes hold all *val*s, and this makes them immutable. We cannot reassign them. And while it is discouraged to do so, we can use *var*s in a case class.

5. Comparing two Case Classes

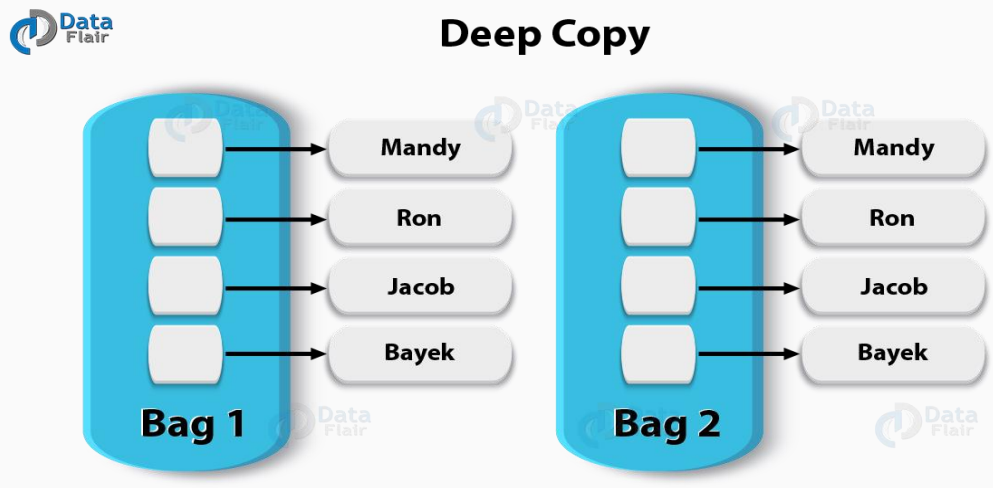
Now, let's take a look at how to compare two Scala case classes.

```
scala> val crybaby=Song("Cry Baby","Melanie Martinez",7)
crybaby: Song = Song(Cry Baby,Melanie Martinez,7)
scala> val cry_baby=Song("Cry Baby","Melanie Martinez",7)
cry_baby: Song = Song(Cry Baby,Melanie Martinez,7)
They hold the same contents, but are they the same? Let's ask Scala.
scala> crybaby==cry_baby
res2: Boolean = true
Yes. Yes, they are.
```

6. Shallow-Copying a Scala Case Class

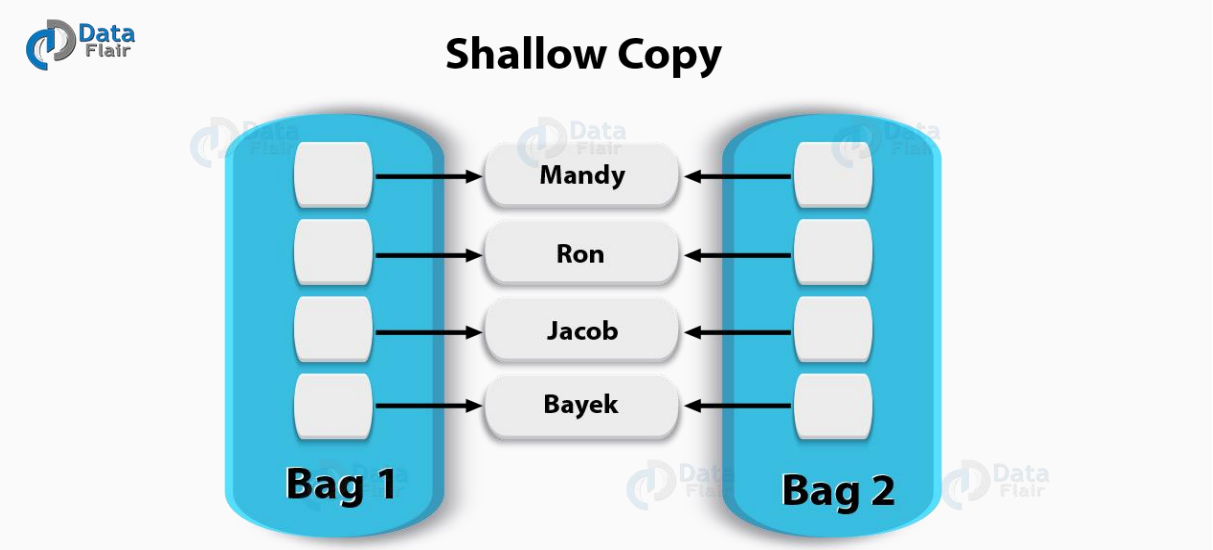
Since a case class is immutable, we might sometimes need a copy to make changes in without changing the original. So, we now see how to create a shallow copy of it. But before that, let's see a little about shallow and deep copies.

A **deep copy** is a copy to another object where any changes we make to it don't reflect in the original object.



Scala Case Class- Deep Copy

A **shallow copy**, however, is one where changes to the copy do reflect in the original.



Scala Case Class- Shallow copy

So, Scala uses the method copy() to carry out a shallow copy.

```
scala> val chandelier1=chandelier.copy()
chandelier1: Song = Song(Chandelier,Sia Furler,3)
```

It is also possible to change the constructor arguments.

```
scala> val chandelier2=chandelier.copy(title=chandelier.artist,artist="Sia")
chandelier2: Song = Song(Sia Furler,Sia,3)
```

So, this was all about Scala Case Class and Scala Object. Hope you like our explanation.

7. Conclusion: Scala Case Class and Scala object

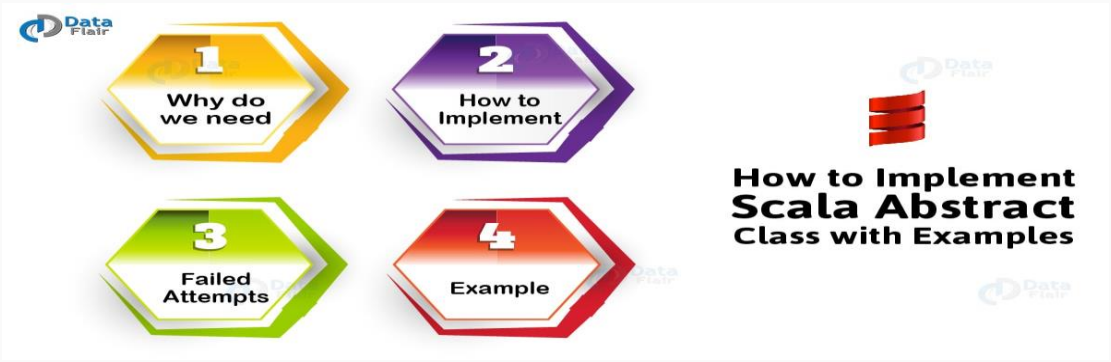
So, this is how we define a case class and process it. Hence, we saw Scala Case Class. Moreover, we discussed Creating a Scala object. Also, we compared two Scala Case Classes. Finally, we looked at shallow copying a Scala Case Class. Furthermore, feel free to leave your doubts and suggestions in the comment section.

How to Implement Scala Abstract Class with Examples

1. Objective

In this Scala tutorial, we will discuss Scala abstract class. Moreover, we will study different examples of Scala Abstract Classes. In addition, we will look at the importance of Abstract Class in **Scala Programming Language**.

So, let's begin Scala Abstract Class.



How to Implement Scala Abstract Class with Examples

2. Why do we need Scala Abstract Class?

Abstraction is often the answer to TMI (Too Much Information). It applies to almost everything in life and keeps us from insanity. And in case you were wondering, abstraction is hiding unnecessary details and putting forward only what interests the subject. So, how does this relate to Scala? Let's find out.

Scala abstract class is one that can have both abstract and non-abstract methods. An abstract method is one without a definition right away. So that means, Scala abstract class lets us declare a method for a class without a definition. We can tell the compiler that the class is supposed to have a particular method, but we may not be sure what to put in that right away. Or we may also want to use this for **inheritance**.

When more than one class inherits from a superclass, the superclass may not want to provide a definition for a certain method, as we may not want to create an instance of it to call the method on. We can then declare the method abstract. Also, you cannot instantiate Scala abstract class. In simple words, you cannot create an object of it. A class can only inherit from one Scala abstract class. Let's take an example of Scala abstract class.

Let's Explore Scala Iterator in Detail

3. How to Implement Scala Abstract Class?

To implement Scala abstract class, we use the keyword 'abstract' against its declaration. It is also mandatory for a child to implement all abstract methods of the parent class.

```
scala> abstract class Person{
| def greet()
| }
defined class Person
scala> class Student extends Person{
| def greet(){
| println("Hi")
| }
| }
defined class Student
scala> var s=new Student()
s: Student = Student@3ceeb6ba
scala> s.greet()
```

Hi

We can also use traits to implement abstraction; we will see that later.

4. Failed Attempts

The following codes will not work, and hence, they tell us about the nature of Scala abstract classes.

a. Instantiating

```
scala> abstract class Person{
| def greet()
| }
defined class Person
scala> var p=new Person()
<console>:12: error: class Person is abstract; cannot be instantiated
var p=new Person()
^
```

Let's Discuss Scala String Interpolation – s String, f String and raw string Interpolator

b. Implementing less than all abstract methods

```
scala> abstract class Person{
| def greet()
| def greethello()
```

```
4. | }
5. defined class Person
6. scala> class Student extends Person{
7. | def greethello(){
8. | println("Hello")
9. | }
10. | }
11. <console>:12: error: class Student needs to be abstract, since method greet in class Person of type ()Unit is not defined
12. class Student extends Person{
```

c. Two levels deep

Although not exactly a failed attempt, but if a subclass must skip the implementation of an abstract method from its parent, it must itself be marked abstract.

```
1. scala> abstract class Person{
2. | def greet()
3. | def greethello()
4. | }
5. defined class Person
6. scala> abstract class Student extends Person{
7. | def greet(){
8. | println("Hi")
9. | }
10. | def greethello()
11. | }
12. defined class Student
```

In this example, we cannot instantiate either of these classes since both are abstract.

Let's Discus Scala Access Modifiers: Public, Private and Protected Members

5. Scala Abstract Class Example

In this example, we also include a constructor and variables.

```
1. scala> abstract class Person(age:Int){
2. | var SSN:String=""
3. | def greet()
4. | def greethello(){
5. | println("Hello")
6. | }
7. | }
8. defined class Person
9. scala> class Student(age:Int) extends Person(age){
10. | SSN="323-44-798"
11. | def greet(){
12. | println("Hi, I'm a student")
13. | println("I am "+age+" years old")
14. | println("My social security number is "+SSN)
15. | }
```

```
16. |}
17. defined class Student
18. scala> var s=new Student(18)
19. s: Student = Student@13de62ef
20. scala> s.greet()
```

Hi, I'm a student

I am 18 years old

My social security number is 323-44-798

```
1.  scala> s.greethello()
```

Hello

Let's Look at Scala do while Loop – A Quick and Easy Tutorial

So, this was all about Scala Abstract Class Tutorial. Hope you like our explanation.

6. Conclusion

Hence, in this Scala abstract class tutorial, we study how to use the 'abstract' keyword to declare a class abstract. In addition, we talked about Scala abstract class example and how can we implement Scala abstract classes. Furthermore, if have any query, tell us in the comments.

Scala String Method with Syntax and Method

1. Scala String Method

In this tutorial on Scala String Method, we will discuss the methods defined by the class java.lang.String. Before you proceed with this, you should check out a brief introduction to Strings in Scala.

Scala String Method

2. List of String Method in Scala with Example

1. char charAt(int index)

This method returns the character at the index we pass to it. Isn't it so much like Java?

```
scala> "Ayushi".charAt(1)
res2: Char = y
```

2. int compareTo(Object o)

This Scala String Method compares a string to another object.

```
scala> "Ayushi".compareTo("Ayush")
res10: Int = 1
```

3. int compareTo(String anotherString)

This is like the previous one, except that it compares two strings lexicographically. If they match, it returns 0. Otherwise, it returns the difference between the two(the number of characters less in the shorter string, or the maximum ASCII difference between the two).

```
scala> "Ayushi".compareTo("Ayushi ")
res3: Int = -1
scala> "Ayushi".compareTo("Ayushi")
res4: Int = 0
scala> "Ayushi".compareTo("ayushi")
res5: Int = -32
```

Comparing to a different type raises a type-mismatch error.

```
scala> "Ayushi".compareTo(7)
<console>:12: error: type mismatch;
found : Int(7)
required: String
"Ayushi".compareTo(7)
^
```

4. int compareToIgnoreCase(String str)

int compareToIgnoreCase Scala String Method compares two strings lexicographically, while ignoring the case differences.

```
scala> "Ayushi".compareToIgnoreCase("ayushi")
res21: Int = 0
scala> "Ayushi".compareToIgnoreCase("ayushi ")
res22: Int = -1
scala> "Ayushi".compareToIgnoreCase("aYushi")
res41: Int = 0
```

5. String concat(String str)

This will concatenate the string in the parameter to the end of the string on which we call it. We saw this when we talked about strings briefly.

```
scala> "Ayushi".concat("Sharma")
res23: String = AyushiSharma
```

6. Boolean contentEquals(StringBuffer sb)

contentEquals compares a string to a StringBuffer's contents. If equal, it returns true; otherwise, false.

```
scala> val a=new StringBuffer("Ayushi")
a: StringBuffer = Ayushi
scala> "Ayushi".contentEquals(a)
res24: Boolean = true
scala> "ayushi".contentEquals(a)
res25: Boolean = false
```

7. Boolean endsWith(String suffix)

This Scala String Method returns true if the string ends with the suffix specified; otherwise, false.

```
scala> "Ayushi".endsWith("i")
res32: Boolean = true
scala> "Ayushi".endsWith("sha")
res33: Boolean = false
```

8. Boolean equals(Object anObject)

This Scala String Method returns true if the string and the object are equal; otherwise, false.

```
scala> val b=Array('A','y','u','s','h','i')
b: Array[Char] = Array(A, y, u, s, h, i)
scala> "Ayushi".equals(b)
res34: Boolean = false
scala> "Ayushi".equals("ayushi")
res35: Boolean = false
scala> "Ayushi".equals("Ayushi")
res36: Boolean = true
scala> val b=Array("Ayushi")
b: Array[String] = Array(Ayushi)
scala> "Ayushi".equals(b)
res37: Boolean = false
scala> "Ayushi".equals(7)
res39: Boolean = false
scala> "7".equals(7)
res40: Boolean = false
scala> "Ayushi".equals("aYushi")
res43: Boolean = false
```

9. Boolean equalsIgnoreCase(String anotherString)

This is like equals(), except that it ignores case differences.

```
scala> "Ayushi".equalsIgnoreCase("aYushi")
```

```
res42: Boolean = true
```

10. byte getBytes()

getBytes Scala String Method encodes a string into a sequence of bytes and stores it into a new byte array. It uses the platform's default charset for this.

```
scala> "Ayushi".getBytes()
res44: Array[Byte] = Array(65, 121, 117, 115, 104, 105)
scala> "ABCcba".getBytes()
res45: Array[Byte] = Array(65, 66, 67, 99, 98, 97)
scala> " ".getBytes()
res46: Array[Byte] = Array(32)
```

11. byte[] getBytes(String charsetName)

With a named charset as a parameter, getBytes will use that charset to encode the string.

```
scala> "Ayushi".getBytes("Unicode")
res47: Array[Byte] = Array(-2, -1, 0, 65, 0, 121, 0, 117, 0, 115, 0, 104, 0, 105)
scala> "Ayushi".getBytes("UTF-8")
res48: Array[Byte] = Array(65, 121, 117, 115, 104, 105)
```

However, providing an invalid charset name will result in an error:

```
scala> "Ayushi".getBytes("Union")
java.io.UnsupportedEncodingException: Union
at java.lang.StringCoding.encode(Unknown Source)
at java.lang.String.getBytes(Unknown Source)
... 28 elided
```

12. void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

This Scala String Method copies characters from the string into the destination character array. scrBegin denotes the index to begin at in the source string, and srcEnd denotes the index to end at in the source string. dstBegin denotes the index to begin at in the destination character array. dst is the destination character array.

```
scala> var c:Array[Char]=Array('A','y','u','s','h','i')
c: Array[Char] = Array(A, y, u, s, h, i)
scala> var d="01234"
d: String = 01234
scala> d.getChars(2,4,c,2)
scala> c
res57: Array[Char] = Array(A, y, 2, 3, h, i)
```

13. int hashCode()

This int hashCode method returns a hash code for the string.

```
scala> "Ayushi".hashCode()
res58: Int = 1976240247
scala> val name="Ayushi"
name: String = Ayushi
scala> name.hashCode()
```

```
res60: Int = 1976240247
```

14. int indexOf(int ch)

This Scala Sring returns the index of the first occurrence of the character ch in the string.

```
scala> "Banana".indexOf('a')
res61: Int = 1
scala> "Banana".indexOf('n')
res62: Int = 2
```

15. int indexOf(int ch, int fromIndex)

This is like indexOf, except that it begins searching at the index we specify.

```
scala> "Banana".indexOf('a',2)
res63: Int = 3
```

16. int indexOf(String str)

This Scala String Method returns the index of the first occurrence of the substring we specify, in the string.

```
scala> "Banana".indexOf("na")
res64: Int = 2
```

17. int indexOf(String str, int fromIndex)

Like the other version of indexOf for a single character, this begins searching at the index we specify.

```
scala> "Banana".indexOf("na",3)
res66: Int = 4
```

18. String intern()

intern returns the canonical representation for the string object.

```
scala> "Hello,\n\tWorld".intern()
res69: String =
Hello,
World
```

19. int lastIndexOf(int ch)

Unlike indexOf, this Scala String Method returns the index of the last occurrence of the character we specify.

```
scala> "Banana".lastIndexOf('n')
res70: Int = 4
```

20. int lastIndexOf(int ch, int fromIndex)

This is like lastIndexOf, except that it begins searching backwards(right to left), starting at the index we specify.

```
scala> "Banana".lastIndexOf('n',1)
res71: Int = -1
```

```
Here, it returns -1 because it couldn't find 'n'.
scala> "Banana".lastIndexOf('n',3)
res72: Int = 2
```

21. int lastIndexOf(String str)

This String Method in Scala returns the index of the last occurrence of the substring we specify, in the string.

```
scala> "Banana".lastIndexOf("na")
res73: Int = 4
```

22. int lastIndexOf(String str, int fromIndex)

This String Method is like the previous method, except that it begins searches at the index we specify, and searches right to left.

```
scala> "Banana".lastIndexOf("na",3)
res74: Int = 2
scala> "Banana".lastIndexOf("na",2)
res75: Int = 2
scala> "Banana".lastIndexOf("na",1)
res76: Int = -1
```

23. int length()

This Scala String Method method simply returns the length of a string.

```
scala> "Ayushi!".length()
res77: Int = 7
scala> "".length()
res78: Int = 0
```

24. Boolean matches(String regex)

If the string matches the regular expression we specify, this returns true; otherwise, false.

```
scala> "Ayushi".matches(".i.*")
res80: Boolean = false
scala> "Ayushi".matches(".*i")
res79: Boolean = true
```

25. Boolean regionMatches(boolean ignoreCase, int toffset, String other, int offset, int len)

If two string regions are equal, this returns true; otherwise, false. ignoreCase, when set to true, will ignore the case differences.

```
scala> "Ayushi".regionMatches(true,0,"Ayushi",0,4)
res81: Boolean = true
scala> "Ayushi".regionMatches(true,0,"Ayush",0,4)
res82: Boolean = true
scala> "Ayushi".regionMatches(true,0,"Ayush",0,3)
res83: Boolean = true
scala> "Ayushi".regionMatches(true,0,"Ay",0,3)
res84: Boolean = false
```

26. Boolean regionMatches(int toffset, String other, int offset, int len)

This is like the previous method, except that it doesn't have ignoreCase.

```
scala> "Ayushi".regionMatches(0,"Ayu",0,2)
res85: Boolean = true
scala> "Ayushi".regionMatches(0,"Ayu",0,4)
res86: Boolean = false
```

27. String replace(char oldChar, char newChar)

replace will replace all occurrences of oldChar in the string with newChar, and return the resultant string.

```
scala> "Ayushi sharma".replace('s','$')
res87: String = Ayu$hi $harma
scala> "Ayushi Sharma".replace('s','$')
res88: String = Ayu$hi Sharma
```

28. String replaceAll(String regex, String replacement)

This will replace each substring matching the regular expression. It will replace it with the replacement string we provide.

```
scala> "potdotnothotokayslot".replaceAll(".ot","**")
res89: String = ********okays**
```

29. String replaceFirst(String regex, String replacement)

If in the above example, we want to replace only the first such occurrence:

```
scala> "potdotnothotokayslot".replaceFirst(".ot","**")
res90: String = **dotnothotokayslot
```

30. String[] split(String regex)

This Scala String Method splits the string around matches of the regular expression we specify. It returns a String array.

```
scala> "xpotxdotynotzhotokayslot".split(".ot")
res93: Array[String] = Array(x, x, y, z, okays)
```

31. String[] split(String regex, int limit)

This String Method in Scala is like split, except that we can limit the number of members for the array. The last member, then, is whatever part of the string that's left.

```
scala> "xpotxdotynotzhotokayslot".split(".ot",2)
res94: Array[String] = Array(x, xdotynotzhotokayslot)
scala> "xpotxdotynotzhotokayslot".split(".ot",5)
res95: Array[String] = Array(x, x, y, z, okayslot)
```

32. Boolean startsWith(String prefix)

If the string starts with the prefix we specify, this returns true; otherwise, false. This is like endsWith.

```
scala> "Ayushi".startsWith("Ay")
res97: Boolean = true
scala> "Ayushi".startsWith(" A")
res99: Boolean = false
scala> "Ayushi".startsWith("ayu")
res100: Boolean = false
```

33. Boolean startsWith(String prefix, int toffset)

If the string starts with the specified prefix at the index we specify, This string method returns true; otherwise, false.

```
scala> "Ayushi".startsWith("yu",1)
res101: Boolean = true
```

34. CharSequence subSequence(int beginIndex, int endIndex)

This returns a subsequence from the string, beginning at *beginIndex* and ending at *endIndex*.

```
scala> "Ayushi".subSequence(1,4)
res102: CharSequence = yus
```

35. String substring(int beginIndex)

This Scala String Method method returns the contents of the string beginning at *beginIndex*.

```
scala> "Ayushi".substring(3)
res103: String = shi
```

36. String substring(int beginIndex, int endIndex)

This returns the part of the string beginning at *beginIndex* and ending at *endIndex*. Wait, isn't that the same as subsequence? Check the types of the results:

```
scala> var a="Ayushi".subSequence(1,4)
a: CharSequence = yus
scala> var b="Ayushi".substring(1,4)
b: String = yus
```

While one returns a CharSequence, the other returns a String.

37. char[] toCharArray()

This converts the string into a CharArray, and then returns it.

```
scala> "Ayushi".toCharArray()
res104: Array[Char] = Array(A, y, u, s, h, i)
```

38. String toLowerCase()

This String Method in Scala converts all characters in the string to lower case, and then returns the resultant string.

```
scala> "AyU$#I!".toLowerCase()
res105: String = ayu$#i!
```

39. String toLowerCase(Locale locale)

This is like toLowerCase, except that we can specify the locale to follow the rules of.

40. String toString()

This returns a String object itself.

```
scala> "7".toString()
res108: String = 7
```

41. String toUpperCase()

This Scala String Method is like toLowerCase, and converts all characters in the string to upper case.

```
scala> "Ayushi".toUpperCase()
res109: String = AYUSHI
```

42. String toUpperCase(Locale locale)

This is like the previous method, except that it will follow the rules of the given locale.

43. String trim()

trim will elide the leading and trailing whitespaces from the string, and then return it.

```
scala> " Ayushi ".trim()
res110: String = Ayushi
```

3. Conclusion

These are the 43 methods we can directly call on a String in Scala. Hope we've done our job in explaining them efficiently and clearly. See you tomorrow.

Scala Method Overloading with Example

scala> var h=new sayHello() h: sayHello = sayHello@406ad6d51.

Objective

In our previous **Scala tutorial**, we had discussed **Scala Case Class**, now we are going to study Scala Method Overloading. In addition, we will learn example of Overloading Methods in Scala.

Let's begin Scala Method Overloading.

Scala Method Overloading with Example

2. What is Scala Method Overloading?

Scala Method overloading is when one class has more than one method with the same name but different signature. This means that they may differ in the number of parameters, **data types**, or both. This makes for optimized code.

Let's discuss Scala String Method with Syntax and Method

3. Example of Overloading Method in Scala

Here, we are going to understand Scala Method Overloading with examples.

Example – 1

Let's take an example with a different number of parameters.

```
scala> class sayHello{
| def hello(){
| println("Hello, user")
| }
| def hello(admin:String){
| println("Hello, "+admin)
| }
| def hello(admin:String,guest:String){
| println("Hello, "+admin+", Hello "+guest)
| }
| }
defined class sayHello
scala> var h=new sayHello()
h: sayHello = sayHello@406ad6d5
```

We defined a new object 'h' for this class. Now, what this prints depends on which version of the method it calls. This depends on how many parameters we pass it (which version of the method definition satisfies the call format). While one version addresses the admin, another address both- the admin and the guest. The default version just says hello to user.

Read about Scala Access Modifiers: Public, Private and Protected Members

```
scala> h.hello()
```

Hello, user

```
scala> h.hello("Ayushi")
```

Hello, Ayushi

```
scala> h.hello("Ayushi","Megha")
```

Hello, Ayushi, Hello Megha

Example – 2

Now, let's try Scala overloading method with versions owning different types. This way, we can carry out the same functionality on different types depending on what the user asks for.

```
scala> class Calculator{
| def sum(a:Int,b:Int){
| var sum=a+b
| println(sum)
| }
| def sum(a:Double,b:Double){
| var sum=a+b
| println(sum)
| }
| }
```

```
defined class Calculator
scala> var c=new Calculator()
c: Calculator = Calculator@1d3e5a05
scala> c.sum(1,2)
```

3

```
scala> c.sum(1.1,2.2)
```

3.3000000000000003

In this example, we have two versions of the method 'sum'- one that adds two integers, and the other that adds two floats. It then prints out the sum.

Let's Learn Scala Environment Setup and Get Started with an IDE

So, this was all about Scala Method Overloading tutorial. Hope you like our explanation.

4. Conclusion

Hence, we come to the end of Scala Method Overloading tutorial, we finally knew method overloading in Scala with example. Do let us know what you think in the comments.

Scala Method Overriding & Field Overriding with Example

1. Objective

In our last tutorial, we saw **How to Implement Scala Abstract Class**.
In this **Scala tutorial**, we commit to discussing Scala Method overriding and fields overriding in Scala. Moreover, we will learn examples of Scala Method Overriding and Field Overriding.

So, let's begin with Scala Method Overriding and Field Overriding.

Scala Method Overriding & Field Overriding with Real – Time Example

2. What is Scala Method Overriding?

Scala overriding method provides your own implementation of it. When a class inherits from another, it may want to modify the definition for a method of the superclass or provide a new version of it. This is the concept of Scala method overriding and we use the 'override' modifier to implement this.

Let's see Scala String Method with Syntax and Method

Technically, Scala method overriding is when a member M of class C matches a non-private member M' of a class from which C inherits. There are some restrictions:

- M is not privatecla
- M' is not final

Let's take Scala method overriding example.

3. Real-Time Examples of Method Overriding in Scala

These are the following Scala method overriding examples, let's see them one by one:

a. Example 1

In this example, class Student extends class Person. It overrides the method greet() from class Person to its own definition using the Scala 'override' keyword.

```
scala> class Person{
| def greet(){
| println("I'm a person")
| }
| }
defined class Person
scala> class Student extends Person{
| override def greet(){
| println("I'm a student")
| }
| }
defined class Student
scala> var s=new Student()
s: Student = Student@6f1d799
scala> s.greet()
```

I'm a student

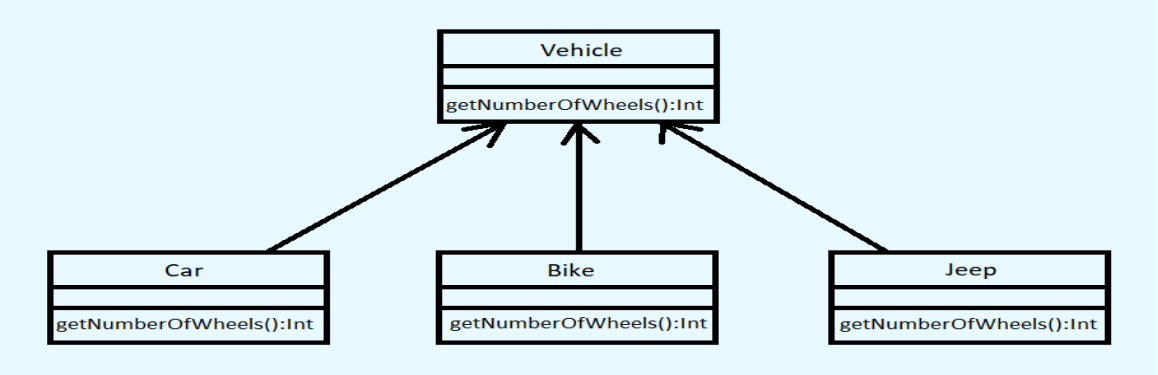
Read About Scala Method Overloading with Example

b. Example 2

Let's take another Scala method Overriding example.

```
scala> class Vehicle{
| def getNumberOfWheels()={
| 0
| }
| }
defined class Vehicle
scala> class Car extends Vehicle{
| override def getNumberOfWheels()={
| 4
| }
| }
defined class Car
scala> class Bike extends Vehicle{
| override def getNumberOfWheels()={
| 2
| }
| }
defined class Bike
scala> class Auto extends Vehicle{
| override def getNumberOfWheels()={
| 3
| }
| }
defined class Auto
scala> var c=new Car();
c: Car = Car@4f72078d
scala> c.getNumberOfWheels()
res1: Int = 4
cala> var b=new Bike();
b: Bike = Bike@2d6e09f0
scala> b.getNumberOfWheels()
res2: Int = 2
scala> var a=new Auto()
a: Auto = Auto@bedebe9
scala> a.getNumberOfWheels()
res3: Int = 3
```

In this example, we define a class Vehicle that defines a method getNumberOfWheels(). Three classes- Car, Bike, and Auto- that inherit from this class. They define their own versions of getNumberOfWheels().



Real-Time Example of Scala Method Overriding

c. Example 3

Without the Scala override keyword or **annotation**, we get the following error:

```
scala> class A{
| def greet(){
| print("A")
| }
| }
defined class A
scala> class B extends A{
| def greet(){
| print("B")
| }}
<console>:13: error: overriding method greet in class A of type ()Unit;
method greet needs `override' modifier
def greet(){
^
```

Let's Explore Scala Option – 13 Simple Methods to Call Option in Scala

4. What is Scala Field Overriding?

In Scala, we can also override fields.

a. Example of Field Overriding in Scala

In the following example, we have a superclass Vehicle and a subclass Car. A vehicle has a variable 'wheels' with the value 0. Car overrides this and gives it a value of 4. When we call the method show() on an object of class Car, it gives us the value 4.

```
scala> class Vehicle{
| val wheels=0
| }
defined class Vehicle
scala> class Car extends Vehicle{
```

```
| override val wheels=4
| def show(){
| println("The car has "+wheels+" wheels")
| }
| }
defined class Car
scala> var c=new Car()
c: Car = Car@6d512521
scala> c.show()
```

The car has 4 wheels

Let's Learn Scala Final – Variable, Method & Class | Scala This

b. What Won't Work

You can only override val fields; this won't work with vars. This is because we can only read vals, but we can both read and write vars. Let's take an example.

```
scala> class Vehicle{
| var wheels=0
| }
defined class Vehicle
scala> class Car extends Vehicle{
| override var wheels=4
| def show(){
| println("The car has "+wheels+" wheels")
| }
| }
<console>:13: error: overriding variable wheels in class Vehicle of type Int;
variable wheels cannot override a mutable variable
override var wheels=4
^
```

And then take a look at this:

```
scala> class Vehicle{
| val wheels=0
| }
defined class Vehicle
scala> class Car extends Vehicle{
| override var wheels=4
| def show(){
| println("The car has "+wheels+" wheels")
| }
| }
<console>:13: error: overriding value wheels in class Vehicle of type Int;
variable wheels needs to be a stable, immutable value
override var wheels=4
^
```

Let's Look at Scala Operator in Detail

So, this was all about Scala Method Overriding Tutorial. Hope you like our explanation.

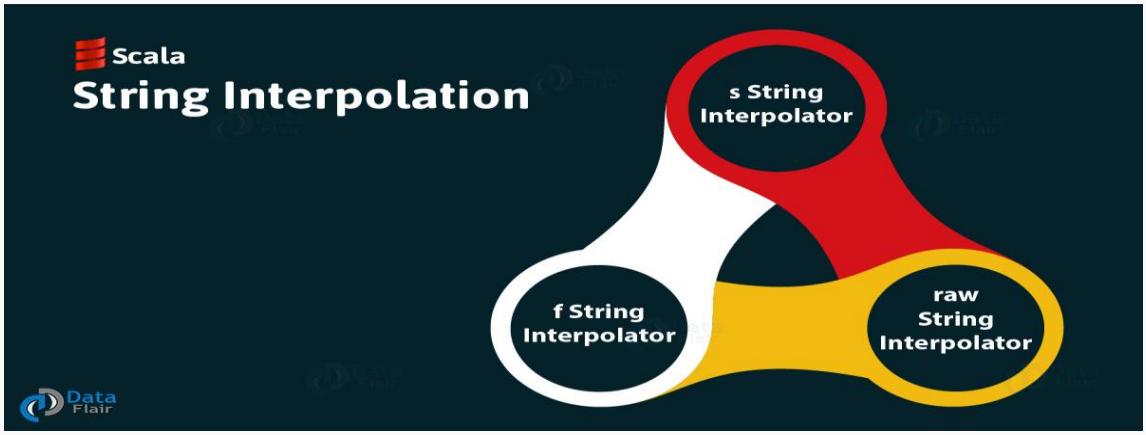
5. Conclusion

Hence in this tutorial, we have seen that what is Scala Method Overriding with real-time examples. In addition, we learned Scala Field Overriding with example. Furthermore, if you have a query, feel free to ask in the comment tab.

Scala String Interpolation | s String, f String and raw string Interpolator

1. Scala String Interpolation

Today in Scala String Interpolation tutorial, we will discuss the string interpolators ‘s’, ‘f’, and ‘raw’. But before we begin, let’s find out a little about string interpolation.



Scala String Interpolation

2. Introduction to Interpolation in String

Interpolation of string is a new way to create them. But we didn’t always have this feature at hand. This was introduced in Scala 2.10, and is available in all newer versions as well.

By interpolating a string, we can embed variable references directly in a processed string literal. We have three kinds of interpolators. Let’s see them one by one.

3. s String Interpolator

When we prepend ‘s’ to any string, we can directly use variables in it with the use of the ‘$’ character. But that variable should be in scope in that string; it should be visible. Well, ‘s’ is a method.

```
scala> val name="Ayushi"
name: String = Ayushi
scala> println(s"Hello, $name, how are you?")
Hello, Ayushi, how are you?
See how simple that was? We can also process arbitrary expressions using ${}.
scala> println(s"Hello, 2+3 is ${2+3}")
```

Hello, 2+3 is 5

```
scala> println(s"Is 2+3 equal to 5? That is ${2+3==5}")
```

Is 2+3 equal to 5? That is true

We’ll also see this with case classes later.

4. f String Interpolator

Let’s first see the problem with the s interpolator.

```
scala> val a=77.000
```

a: Double = 77.0

```
scala> println(s"The number is $a")
```

The number is 77.0

This prints 77.0, not 77.000.

Now, see this with the f interpolator.

```
scala> println(f"The number is $a%.3f")
```

The number is 77.000

This is quite like the printf style format specifiers in C. These are %d, %i, %f, and so on.

```
scala> println(f"$name%s says the number is $a%.3f")
```

Ayushi says the number is 77.000

This is type-safe. If the variable reference and the format specifier don’t match, it raises an error. By default, the specifier is %s. Next in Scala String Interpolation is raw String Interpolator.

5. raw String Interpolator

raw is like s, except that it doesn't escape literals within a string.

```
scala> println(raw"$name says the number is $a\nOkay, bye")
```

Ayushi says the number is 77.0\nOkay, bye

The same with s will give:

```
scala> println(s"$name says the number is $a\nOkay, bye")
```

Ayushi says the number is 77.0

Okay, bye

This tells us that the raw interpolator interpolates the string in a very raw sense.

6. Conclusion

These are the three string interpolators we have in Scala. But apart from these, you can also create your own.

Scala Arrays and Multidimensional Arrays in Scala

1. Scala Arrays and Multidimensional Arrays

Today, we will learn about Scala arrays, how to declare and process them, and multidimensional arrays. We will also see how to create them with Range and concatenating them. So, let's begin our tutorial on Arrays in Scala.

2. Introduction to Arrays in Scala

When we want to hold a number of elements of the same kind in a collection, we use an array. An array is sequential and is of a fixed size.

3. Declaring a Scala Array

In Scala, we can declare arrays in two ways.

a. Adding Elements Later

We can create an array in Scala with default initial elements according to data type, and then fill in values later.

```
scala> var a=new Array[Int](3) //This can hold three elements
a: Array[Int] = Array(0, 0, 0)
scala> a(1)
res6: Int = 0
scala> a(1)=2 //Assigning second element
scala> a
res8: Array[Int] = Array(0, 2, 0)
```

We can also mention the type of array when declaring it:

```
scala> var a:Array[Int]=new Array[Int](3)
a: Array[Int] = Array(0, 0, 0)
scala> a(4/2)=3
scala> a
res10: Array[Int] = Array(0, 0, 3)
```

b. Defining an Array with Values

We can also define a Scala array specifying its values in place.

```
scala> var a=Array(1,2,3)
a: Array[Int] = Array(1, 2, 3)
scala> a(4)
java.lang.ArrayIndexOutOfBoundsException: 4
... 28 elided
scala> a(0)
res12: Int = 1
```

4. Processing an Array

Since we know the type of elements and the size of the array, we can use loop control structures to process an array. Let's take an example of processing Scala Array.

To iterate over the array:

```
scala> var a=Array(1,2,3)
a: Array[Int] = Array(1, 2, 3)
scala> for(i<-a){
| println(i)
| }
```

```
1

2

3
```

To calculate the sum of all the elements:

```
scala> var sum=0.0
sum: Double = 0.0
scala> for(i<-a){
| sum+=i}
scala> sum
res2: Double = 6.0
Finding the highest value from the array:
scala> var max=a(0)
max: Int = 1
scala> for(i<-a){
| if(i>max){
| max=i}
| }
scala> max
res4: Int = 3
```

5. Concatenating Arrays

We can append a Scala array to another using the concat() method. This takes the arrays as parameters- in order.

```
scala> var a=Array(1,2,3)
a: Array[Int] = Array(1, 2, 3)
scala> var b=Array(4,5,6)
b: Array[Int] = Array(4, 5, 6)
```

We'll need to import the Array._ package

```
scala> import Array._
import Array._
```

Now, let's call concat().

```
scala> var c=concat(a,b)
c: Array[Int] = Array(1, 2, 3, 4, 5, 6)
```

Any Doubt yet in Scala Arrays? Please Comment.

6. Creating an Array with Range

The range() method will give us integers from the start to the end. We can also specify a step to denote the interval. Check Range() in Python for reference.

We can use this to create an array to iterate on.

```
scala> var a=range(2,15)
a: Array[Int] = Array(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
scala> var b=range(15,2,-2)
b: Array[Int] = Array(15, 13, 11, 9, 7, 5, 3)
```

7. Scala Multidimensional Arrays

Why settle at two dimensions when we can have multiple? Sometimes, we may need more than two.

```
scala> var a=ofDim[Int](3,3)
a: Array[Array[Int]] = Array(Array(0, 0, 0), Array(0, 0, 0), Array(0, 0, 0))
Now, let's fill in some values.
scala> for(i<-0 to 2){
| for(j<-0 to 2){
| a(i)(j)={i+j}
| }
| }
scala> a
res10: Array[Array[Int]] = Array(Array(0, 1, 2), Array(1, 2, 3), Array(2, 3, 4))
```

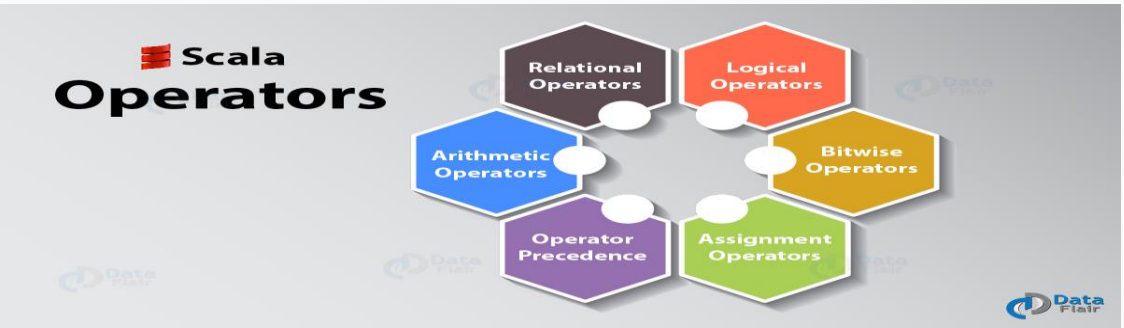
8. Conclusion

With arrays in Scala, we can deal with a set of values of the same kind. Feel free to ask us any doubts in the comment section.

Scala Operator

1. Scala Operator

What is an operator? What are the different kinds of Scala operators? After this tutorial on operators in Scala, you will be able to answer that. Dive right in.



Scala Operators

2. Introduction to Operators in Scala

A symbol that instructs the compiler to perform certain mathematical or logical manipulations, is an operator. We have the following types of operators in Scala:

Arithmetic Operators

Relational Operators

Logical Operators

Bitwise Operators

Assignment Operators

Let’s discuss all Scala operator one by one.

3. Arithmetic Operators

This Scala operator tells Scala to perform arithmetic operations on two values. We have five of these. Let’s take two values for exemplar purposes. (We did this in the Command Prompt).

```
scala> val a=2
a: Int = 2
scala> val b=3
b: Int = 3
```

a. Addition (+)

This adds two operands. See an example.

```
scala> a+b
res0: Int = 5
```

b. Subtraction (-)

This subtracts the second operand from the first.

```
scala> a-b
res1: Int = -1
```

c. Multiplication (*)

This multiplies the two operands.

```
scala> a*b
res2: Int = 6
```

d. Division (/)

This divides the first value by the second.

```
scala> a/b
res3: Int = 0
scala> b/a
res4: Int = 1
```

e. Modulus (%)

This returns the remainder after dividing the first number by the second.

```
scala> a%b
res6: Int = 2
scala> b%a
res5: Int = 1
```

4. Relational Operators

Now, let's discuss the Scala operators that let us compare values. They are rational Scala Operators.

a. Equal to (==)

This returns true if the two values are equal; otherwise, false.

```
scala> a==b
res7: Boolean = false
scala> println(a==b)
false
```

b. Not Equal to (!=)

This returns true if the two values are unequal; otherwise, false.

```
scala> a!=b
res9: Boolean = true
```

c. Greater Than (>)

This returns true if the first operand is greater than the second; otherwise, false.

```
scala> a>b
res10: Boolean = false
```

d. Less Than (<)

This returns true if the first operand is lesser than the second; otherwise, false.

```
scala> a<b
res11: Boolean = true
```

e. Greater Than or Equal to (>=)

This returns true if the first operand is greater than or equal to the second; otherwise, false.

```
scala> a>=b
res12: Boolean = false
```

f. Less Than or Equal to (<=)

This returns true if the first operand is less than or equal to the second; otherwise, false.

```
scala> a<=b
res13: Boolean = true
```

5. Logical Operators

Moving on to logical Scala operators, we have three. Let's take Boolean values for this.

a. Logical AND (&&)

This returns true if both operands are true; otherwise, false.

```
scala> true&&true
res22: Boolean = true
scala> true&&false
res23: Boolean = false
```

b. Logical OR (||)

This returns true if either or both operands are true; otherwise, false.

```
scala> true||false
res25: Boolean = true
scala> false||false
res26: Boolean = false
```

c. Logical NOT (!)

This reverses the operand's logical state. It turns true to false, and false to true.

```
scala> !true
res27: Boolean = false
scala> !false
res28: Boolean = true
scala> !(true&&false)
res29: Boolean = true
```

6. Bitwise Operators

Coming to Bitwise Scala operators, we have seven in Scala. But first, let's see the truth table for these:

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Let's take two values:

```
scala> var a=60 //Its binary is 00111100
a: Int = 60
scala> var b=13 //Its binary is 00001101
b: Int = 13
```

a. Binary AND Operator (&)

This performs an AND operation on each pair of bits in the binary equivalents for the two operands.

```
scala> a&b
res43: Int = 12
```

This gives us 12 because of 00001100.

b. Binary OR Operator (|)

This performs an OR operation on each pair of bits in the binary equivalents for the two operands.

```
scala> a|b
res44: Int = 61
```

The result is 61 because of 00111101.

c. Binary XOR Operator (^)

This Scala Operator performs an XOR operation on each pair of bits in the binary equivalents for the two operands.

```
scala> a^b
res45: Int = 49
```

49 is decimal for 00110001.

d. Binary One's Complement (~)

This flips the bits in the operand's binary equivalent.

```
scala> ~a //This will be 11000011
res46: Int = -61
scala> ~b
res47: Int = -14
```

e. Binary Left-Shift Operator (<<)

This operator in Scala moves the bits of the left operand by (the right operand) number of bits to the left.

```
scala> a<<2
res48: Int = 240
```

This is because it turns 00111100 to 11110000.

f. Binary Right-Shift Operator (>>)

This Scala Operator moves the bits of the left operand by (the right operand) number of bits to the right.

```
scala> a>>2
res49: Int = 15
```

This is because it turns 00111100 to 1111.

g. Shift Right Zero Fill Operator (>>>)

Shift Right Zero Fill Scala Operator moves the bits of the left operand by (the right operand) number of bits to the right, and fills these places with zeroes.

```
scala> a>>>2
res50: Int = 15
```

This turns 00111100 to 00001111.

7. Assignment Operators

Scala supports 11 different assignment operators:

a. Simple Assignment Operator (=)

This assigns the value on the right to the operand on the left.

```
scala> var a=1+2
a: Int = 3
```

b. Add and Assignment Operator (+=)

This Scala Operators adds the value of the right operand to the one on the left, and then assigns the result to the left operand.

```
scala> a+=2 //This is the same as a=a+2
scala> a
res31: Int = 5
```

c. Subtract and Assignment Operator (-=)

This operator in Scala subtracts the second operand from the first, then assigns the result to the left operand.

```
scala> a-=3
scala> a
res33: Int = 2
```

d. Multiply and Assignment Operator (*=)

This Scala Operator multiplies the two operands, and then assigns the result to the left operand.

```
scala> a*=7
scala> a
res38: Int = 14
```

e. Divide and Assignment Operator (/=)

Divide And Assignment Scala Operator divides the first operand by the second, and then assigns the result to the left operand.

```
scala> a/=2
scala> a
```

```
res40: Int = 7
```

f. Modulus and Assignment Operator (%=)

Modulus And Assignment Scala Operator calculates the modulus remaining after dividing the first operand by the second. Then, it assigns this result to the first operand.

```
scala> a%=4
scala> a
res42: Int = 3
```

g. Left-Shift and Assignment Operator (<<=)

This operator in Scala shifts the left operand by (the right operand) number of bits to the left. Then, it assigns the result to the left operand.

```
scala> a<<=2
scala> a
res57: Int = 240
```

We'll reset a to 60 every time so we can properly demonstrate the operators.

```
scala> a=60
a: Int = 60
```

h. Right-Shift and Assignment Operator (>>=)

This Scala Operator shifts the left operand by (the right operand) number of bits to the right. Then, it assigns the result to the left operand.

```
scala> a>>=2
scala> a
res59: Int = 15
scala> a=60
a: Int = 60
```

i. Bitwise AND Assignment Operator (&=)

This applies the bitwise AND operator on the two operands, and then assigns the result to the left operand.

```
scala> a&=b
scala> a
res61: Int = 12
scala> a=60
a: Int = 60
```

j. Bitwise Exclusive-OR and Assignment Operator (^=)

This performs a bitwise XOR operation on the two operands, and then assigns this result to the left operand.

```
scala> a^=b
scala> a
res63: Int = 49
```

```
scala> a=60
a: Int = 60
```

k. Bitwise Inclusive-OR and Assignment Operator (|=)

This performs a bitwise inclusive-OR operation on the two operands, and then assigns this result to the left operand.

```
scala> a|=b
scala> a
res65: Int = 61
```

8. Operator Precedence

When we use multiple terms in an expression, how does Scala group them? This is decided by operator precedence; this is how an expression evaluates. Some operators have a higher precedence than others:

Category	Operator	Associativity
Postfix	() []	Left to Right
Unary	! ~	Right to Left
Multiplicative	* / %	Left to Right
Additive	+ –	Left to Right
Shift	>> >>> <<	Left to Right
Relational	> >= < <=	Left to Right
Equality	== !=	Left to Right
Bitwise AND	&	Left to Right

Bitwise XOR	^	Left to Right
Bitwise OR	|	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to Left
Comma	,	Left to Right

In the table, the precedence decreases from top to bottom. Hence, the topmost operators have the highest precedencies. In an expression, the operators with a higher precedence evaluate first.

For example, in val x=7+7*7, what evaluates first is 7*7, because * has a higher precedence than +. Then, 7+49 gives us 56.

this is all on Scala Operators.

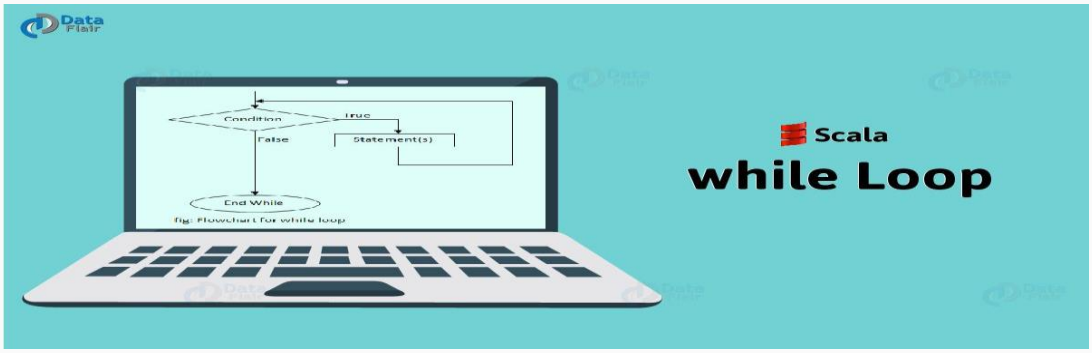
9. Conclusion

With this, we conclude this blog on Scala Operators. Tell us what you think in the comments section.

Scala While Loop with Syntax and Example

1. Scala While Loop

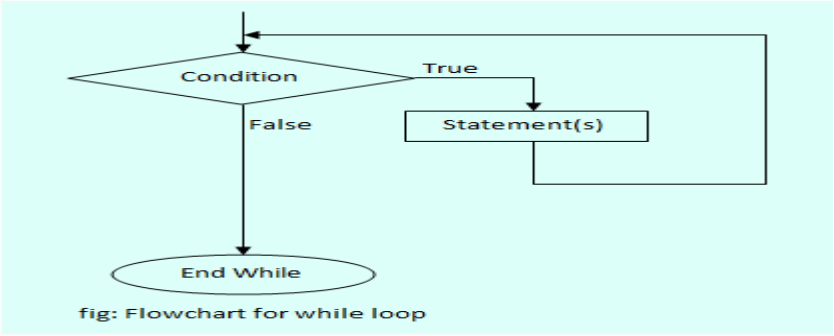
Now, after the for-loop, we come to the Scala while loop. Dive right in.



Scala While Loop

2. Introduction

A while loop that will execute a set of statements as long as a condition is true.



Scala While Loop

3. Working of Scala While Loop

A Scala while loop first checks the condition. This may be any expression. Any non-zero value is true; zero is false.

So, if the condition is true, it executes the code in the block under it. The block of code may be one statement or more. Then, it checks the condition again. If still true, it executes the block of code again. Otherwise, it skips to the first statement outside the loop.

If the condition is false the first time, the loop skips to the first statement outside the loop, instead.

4. Syntax

We have the following syntax for a Scala while-loop:

while(condition)

```
{
//Code to execute if condition is true
}
```

5. Example: While Loop in Scala

Example time! Let's build a loop that counts down form 7 to 1.

object Main extends App

```
{
var x=7
while(x>0)
{
println(x)
x=x-1
}
```

Here's the output:

7

6

5

4

3

2

1

Initially, x is 7. The condition is initially true, because 7>0. The loop reduces it to 6. And so on, this loop runs as long as x reaches 1. Then, inside the loop's body, it becomes 0. Now, when the condition is checked, it evaluates to false. So, it comes out of the loop.

6. One Infinite Loop

It is easy to forget putting in the statement that modifies the variable involved in the condition. This creates an infinite loop- one that never ends, because the condition is always true. Allow us to demonstrate using a while-loop.

import scala.util.control._

object Main extends App

```
{
var a=0
while(a<7)
{
print(a)
}
}
```

This keeps printing 0 infinitely:

000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
0

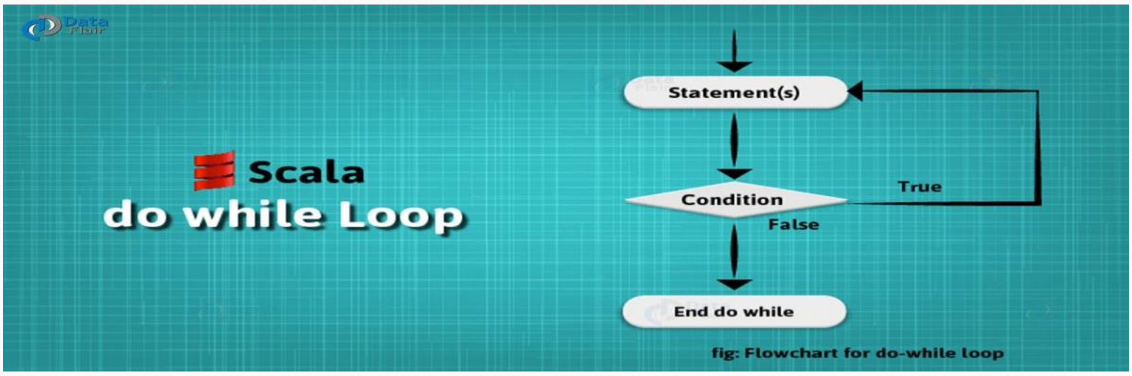
7. Conclusion

That’s all for the while-loop with Scala. See you again with another basic to learn about. Good day.

Scala do while Loop | A Quick and Easy Tutorial

1. Scala do while loop

So, we did the while-loop yesterday. But there’s yet one more- a Scala do while loop. Let’s find out what it is.

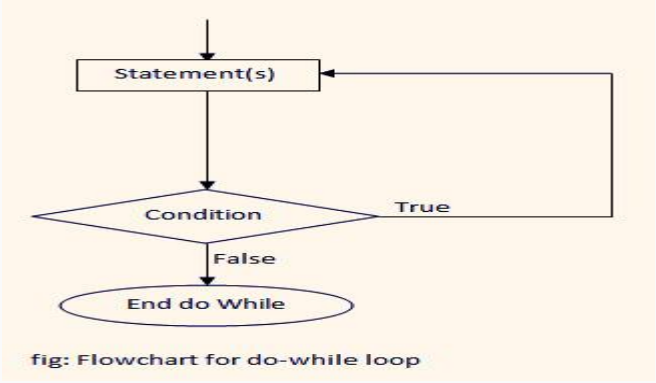


Scala do while Loop

2. Introduction

A Scala do while loop will execute a set of statements as long as a condition is true. This is like a while-loop. However, there are two differences:

1. A while-loop checks the condition before entering the loop. A do-while loop checks it after executing the loop.
2. This means that a do-while loop will execute at least once. But a while loop may never run.



Scala do – while loop flow chart

3. Working

A Scala do while loop first executes the loop once. Then, it checks the condition, which may be any expression. While zero means false, any non-zero value is true.

The rest of the working is like a while-loop. If the condition is true, it executes the code block under itself. It can be one or more statements. Next, it checks the condition again. If true, it executes the block again. Otherwise, it skips to the first statement outside the do-while loop.

Even if the condition is false for the first time, the loop still runs at least once.

4. Syntax

This is the syntax for a do-while loop:

```
1. do
2. {
3. statement(s);
4. }
5. while(condition);
```

5. Examples

Let’s count down from 7 to 1.

object Main extends App

```
{

var x=7
do
{
println(x)
x=x-1
}while(x>0);
```

Here’s the output:

7

6

5

4

3

2

1

Initially, x is 7. The loop executes the loop once. This makes x equal to 6. Then, it checks the condition. Now since 6>0, the condition is true. So, it executes the body yet again. This goes on until x becomes 1. The loop’s body makes it 0. Then, it checks the condition, which is now false. Hence, it comes out of the loop.

Note that this code works even if you skip the semicolon after the while.

Now, let’s take the case when the condition is never true:

object Main extends App

```
{
var x=0
do
{
println(x)
x=x-1
}while(x>0)
}
```

See? It runs at least once.

This was all on Scala do while loop.

6. Conclusion

So, that's all we have for do while loops. Next, we'll see a loop control statement. Stay tuned.

Scala for loop with Syntax and Examples

1. Scala for loop

Sometimes, in your code, you may want to execute a set of statements for, say, 100 times. Would you type them a hundred times? Obviously not. This is where loops come in. A loop lets us execute a group of statement a set number of times, or until an expression becomes false. In this lesson, we will see the Scala for loop. Other kinds include 'for', 'while', and 'do while'.

Scala for loop

Learn: Loop Control statement in Scala

2. Introduction to for loops in Scala

Scala for loop lets us execute specific code a certain number of times. It is a control structure in Scala, and in this article, we'll talk about the forms of for-loop we have here.

3. Types of Scala for-loop

Let's first check out the syntax for Scala for Loop.

```
1. for(var x <- Range){
2. statement(s);
3. }
```

The arrow pointing leftward is a generator; it generates individual values from a range. Range is a range of numbers; we may use a list for this. We can also represent this as *i to j*, or as *i until j*.

a. i to j

Let's take an example to demonstrate the syntax for i to j.

```
object Main extends App {
var a=7
for(a<-1 to 10)
{
println(a)
}
}
```

Here's the output:

1

2

3

4

5

6

7

8

9

10

Learn: While loop in Scala

b. i until j

The other syntax we talked about is the until-syntax. Here's an example:

```
object Main extends App{
var a=0
for(a<-1 until 10)
{
println(a)
}
}
```

c. Using Multiple Ranges in Scala for loop

We can use multiple ranges in a Scala for loop. For this, we just separate the ranges with semicolons(;).

```
object Main extends App{
var a=0
var b=0
var c=0
for(a<-1 until 3;b<-7 to 9;c<-12 until 15)
{
println()
```

```
8.  println(a)
9.  println(b)
10. println(c)
11. }
12. }
```

What would the output be? Combinations of all values:

1

7

12

1

7

13

1

7

14

1

8

12

1

8

13

1

8

14

1

9

12

```
1
9
13

1
9
14

2
7
12

2
7
13

2
7
14

2
8
12

2
8
13

2
8
```

```
14


2
9
12


2
9
13


2
9
14
```

Learn: do-while loop in Scala

d. Using Collections

We can also use a collections object to provide values to the for-loop to iterate on. Let's take an example with a list.

```
1. object Main extends App{
2. var a=0
3. val odds=List(1,3,5,7,9)
4. for(a<-odds)
5. {
6. println(a)
7. }
8. }
```

The output:

```
1
3
5
7
9
```

e. Using Filters

Adding extra if-conditions can let us filter out values from a collection.

```
object Main extends App{
var a=0
val odds=List(1,3,5,7,9,11,13,15,17,19)
for(a<-odds
if a>3; if a<15)
{
println(a)
}
}
```

And the output:

5

7

9

11

13

Learn: Scala if-else statement

f. Scala for loop Yield

The last option is to return a value from a for-loop, and store it in a variable. We can also return it from a function. Then, we follow this by a yield statement. Note the curly braces.

```
object Main extends App{
var a=0
val odds=List(1,3,5,7,9,11,13,15,17,19)
var ret=for{a<-odds
if a>3; if a<15} yield a
for(a<-ret)
{
println(a)
}
}
```

And the output is:

5

7

9

11

13

This was all in Scala for loop.

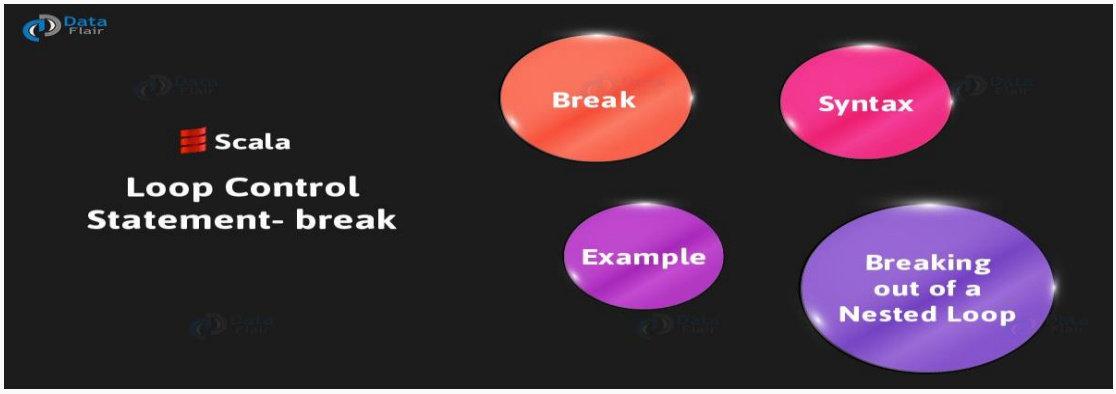
4. Conclusion

These are the ways we can implement a for-loop. Try your own questions, and ask us your doubts in the comments section.

Loop Control Statement – Scala Break

1. Scala Break Statement

So, we discussed about three kinds of loops. But what about controlling these loops? What if we wanted to break out of one midway? Read ahead this Scala Break Tutorial..



Loop Control Statement – Scala Break

2. Introduction to Scala Break

A loop control statement lets us exercise a little more control over a loop. It prevents normal execution.

When we leave a scope, Scala destroys all automatic objects created in that scope. Actually, Scala did not support this functionality until version 2.8. Technically, there are no 'break' or 'continue' statements in Scala, unlike Java. But we must have something, right? Let's first look at what this means.

3. break

When we're inside a loop, and we encounter a break, the loop exhausts(terminates), and we skip to the first statement outside the loop.

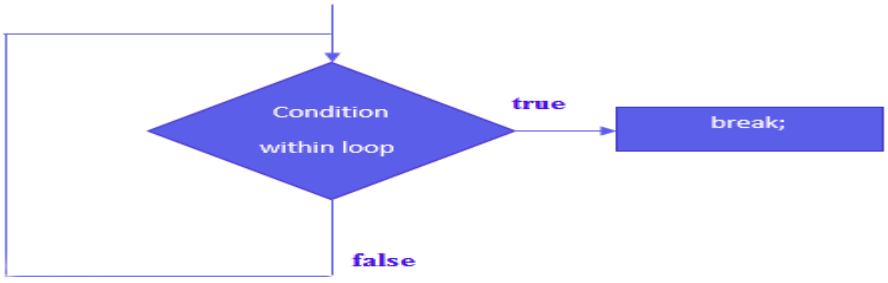


Figure: Flowchart of break statement

Scala Break

4. Syntax

This is the syntax for Scala break statement:

import scala.util.control._

val loop=new Breaks;

loop.breakable

```
{
for(...)
{
//Code
loop.break;
}
}
```

5. Scala 'break' Example

So, what is all this about? let's take an example to clear the air.

import scala.util.control._

object Main extends App

```
{
val loop=new Breaks
val nums=List(1,2,3,4,5)
var a=0
loop.breakable
{
for(a<-nums)
{
{
val loop=new Breaks
val nums=List(1,2,3,4,5)
var a=0
loop.breakable
{
```

```
for(a<-nums)
{
println(a)
if(a==3)
{
loop.break
}
}
}
println(a)
if(a==3)
{
loop.break
}
}
}
}
```

The output is:

1

2

3

Any doubt yet in break statement in Scala? Please Comment

6. Breaking out of a Nested Loop

For this to work on nested loops, we do this:

import scala.util.control._

object Main extends App

```
{

val inner=new Breaks
val outer=new Breaks
val nums1=List(1,2,3,4)
val nums2=List(5,6,7,8)
var a=0
var b=0
outer.breakable
{
for(a<-nums1)
{
println(a)
inner.breakable
{
for(b<-nums2)
{
println(b)
if(b==7)
{
inner.break
```

```
22. }
23. }
24. }
25. if(a==3)
26. {
27. outer.break
28. }
29. }
30. }
31.
32. }
```

And the output is:

1

5

6

7

2

5

6

7

3

5

6

7

Whoa, that was confusing, what happened? Let's see this step by step.

Initially, a and b are 0. The outer for-loop prints 1 from nums1. Then, the inner for-loop prints 5,6, and 7. It stops at 7 because of the break statement. Then, it checks if a==3. It isn't. So, it gets back to the outer for-loop. Now, a takes the value 2 from nums1. This goes on until a takes on the value 3. It prints 3, and then prints 5,6, and 7 from the inner for-loop. Now, it checks if a==3. This is true this time. So, it gets out of the outer loop. Hence, the output.

This was all on Scala break statements.

7. Conclusion

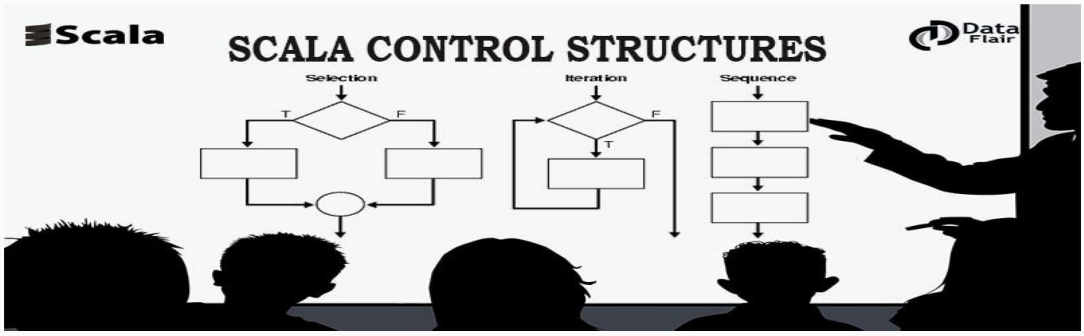
Not as convenient as Java, of course, but the 'break' in Scala still works. What's your excuse?

Scala Control Structures – A Comprehensive Guide

1. Objective

This Scala tutorial will help you in learning Scala programming language using Scala Control structures with syntax. You will understand how to write Scala programs using Scala if else function, for loop in Scala, Match expressions in scala and while or do while loop in scala along with scala examples.

Before starting with control structures in Scala, let us understand Scala basics for beginners and features of Scala that make it programmers choice.



2. Scala control structures

Do you know what is control structure?

A control structure is a block of programming that analyzes variables and selects how to proceed based on given parameters. It is the basic decision-making process in computing that determines the program flow based on certain conditions and parameters.

So let us see various types of control structures used in Scala.

a. if...Else Control Structure

The If..Else conditional expression is a classic programming construct for choosing a branch of code based on whether an expression resolves to true or false. In many languages it starts with an "if," continues with zero to many "else if " sections, and ends with a final "else" catch-all statement.

Syntax: Using an If Expression

if (<Boolean expression>) <expression>

The term Boolean expression here indicates an expression that will return either true or false. Below is if block that prints a notice if the Boolean expression is true:

scala> if (47 % 3 > 0) println(“Not a multiple of 3”)

Not a multiple of 3

Of course 47 isn’t a multiple of 3, so the Boolean expression was true and the println was triggered.

Although if block can act as an expression, it is better suited for statement like this. The problem with using if block as expression is that they only conditionally return a value. If the Boolean expression returns false, what do you expect if block to return?

Syntax: If .. Else Expressions

```
1. if (<Boolean expression>) <expression>
2. else <expression>
```

Example:

```
1. scala > val x = 10; val y = 20
2. x: Int = 10
3. y: Int = 20
4. scala > val max = if (x > y) x else y
5. max: Int = 20
```

Here x and y values make up the entirety of if and else expressions. The resulting value is assigned to max, which we and the Scala compiler know will be an Int because both expressions have return values of type Int. You can use it like a Java ternary operator:

```
1. val absValue = if (a < 0) -a else a // ternary
2.
3. println(if (i == 0) "a" else "b") // in println
4.
5. hash = hash * prime + (if (name == null) 0 else name.hashCode) // in equation
6.
7. def abs(x: Int) = if (x >= 0) x else -x // as a method body
```

b. Match Expression

Match expressions are similar to “switch” statements of C and Java, in which a single input item is checked and the first pattern that is “matched” is executed and its value returned. Like “switch” statement of C and Java, match expressions in Scala support a default “catch-all” pattern. Unlike them, in match expressions only zero or one patterns can match. There is no break statement or “fall-through” from one pattern to the next one in line that would prevent this fall-through.

Syntax:

```
1. <expression> match {
```

```
case <pattern match> => <expression>
[case...]
}
```

Example:

```
val month = i match {
case 1 => "January"
case 2 => "February"
case 3 => "March"
case 4 => "April"
case 5 => "May"
case 6 => "June"
case 7 => "July"
case 8 => "August"
case 9 => "September"
case 10 => "October"
case 11 => "November"
case 12 => "December"
case _ => "Invalid month" // the default, catch-all
}
```

Match expression in function body for different type value:

```
def getClassAsString(x: Any):String = x match {
case s: String => s + " is a String"
case i: Int => "Int"
case f: Float => "Float"
case l: List[_] => "List"
case p: Person => "Person"
case _ => "Unknown"
}
```

use 'if' expressions in case statements

```
i match {
case a if 0 to 9 contains a => println("0-9 range: " + a)
case b if 10 to 19 contains b => println("10-19 range: " + a)
case c if 20 to 29 contains c => println("20-29 range: " + a)
case _ => println("Hmmm...")
}
```

reference class fields in your 'if' statements:

```
stock match {
case x if (x.symbol == "XYZ" && x.price < 20) => buy(x)
case x if (x.symbol == "XYZ" && x.price > 50) => sell(x)
case x => doNothing(x)
}
```

c. For Loop

A loop is a term for exercising a task repeatedly and may include iterating through a range of data or repeating until a Boolean expression returns false.

The most important looping structure in Scala is the for-loop which is also called “for comprehension”. For loops in scala can iterate over a range of data executing an expression every time and optionally return values that is a collection of all the expression’s return values. These loops are highly supporting nested iterating, filtering, value binding and are customizable.

Examples:

// simple for loops

```
for (arg <- args) println(arg)
for (i <- 0 to 5) println(i)
for (i <- 0 to 10 by 2) println(i)
```

// for loop with multiple counters:

```
scala > for (i <- 1 to 2; j <- 1 to 2) printf("i = %d, j = %d\n", i, j)
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
```

// print all even numbers by adding a ‘guard’

```
scala > for (i <- 1 to 10 if i % 2 == 0) println(i)
2
4
6
8
10
```

// multiple guards

```
for {
file <- files
if passesFilter1(file) // guard
if passesFilter2(file) // guard
} doSomething(file)
```

// for loop with guard and yield

```
for {
file <- files
if hasSoundFileExtension(file)
if !soundFileIsLong(file)
} yield file
```

// more for loops with ‘yield’

```
val a = Array("apple", "banana", "orange") // Array(apple, banana, orange)
val newArray = for (e <- a) yield e.toUpperCase // Array(APPLE, BANANA, ORANGE)
val a = Array(1, 2, 3, 4, 5) // Array(1, 2, 3, 4, 5)
for (e <- a) yield e // Array(1, 2, 3, 4, 5)
(e <- a) yield e * 2 // Array(2, 4, 6, 8, 10)
for (e <- a) yield e % 2 // Array(1, 0, 1, 0, 1)
for (e <- a if e > 2) yield e // Array(3, 4, 5)
```

d. While loop / Do..while loop

In addition to for-loop, "while" and "Do..While" loops are also being supported by Scala. They repeat a statement until false is returned by Boolean expression. These are not as commonly used as for-loops in scala, however, because they are not expressions and cannot be used to yield values.

Syntax of While Loop:

while (<Boolean expression>) statement

Example:

```
1. var i = 0
2. while (i < array.length) {
3. println(array(i))
4. i += 1
5. }
```

The do..While loop is similar but the statement is executed before the Boolean expression is first evaluated. In this example we have a boolean expression that will return false, but is only checked after the statement has had a chance to run –

Syntax of Do While loop in Scala:

do {// expression…} while (condition)

Example:

```
1. val x = 0
2. do println("Here I am, x = $x") while (x > 0) // Here I am, x = 0
```

Source:

https://www.scala-lang.org/

Tuples in Scala – A Quick Introduction

1. Objective

This tutorial will help you in understanding the concept of tuples in Scala programming language. You will learn creating scala tuples with examples and syntax. You will also learn iteration over the tuples in scala, swapping of tuple elements and way to convert tuple into string in scala function. Before starting with Scala tuples, let us brush our Scala concepts and features of Scala that make it so popular and create comparison between Java and Scala.

2. Introduction to Tuples in Scala

Tuple is a Scala collection which can hold multiple values of same or different type together so they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types but they are also immutable.

Let us understand the same with an example of tuple holding an integer, a string, and a double:

```
val t = new Tuple3(1, "hello",20.2356)
```

A short cut for the above code is :-

```
val t = (1, "hello",20.2356)
```

You can also create a tuple using the below syntax:

```
1 -> “a”
```

This syntax would be seen a lot when creating maps.

The actual type of a tuple depends upon the number of elements it contains and the type of those elements. Thus the type of (1,”hello”,20.2356) is Tuple3[Int , String , Double].Tuple are of type Tuple1 , Tuple2 ,Tuple3 and so on. There is an upper limit of 22 for the element in the tuple in the scala, if you need more elements, then you can use a collection, not a tuple.

For each tuple type, Scala defines a number of element access methods.

Examples:

```
val t = (4,3,2,1)
val sum = t._1 + t._2 + t._3 + t._4
println( "Sum of elements: " + sum )
```

// Sum of elements : 10

3. Iteration over the tuple

You can use Tuple.productIterator() method to iterate over all of the elements of a tuple.

Example:

```
val t = (4,3,2,1)
t.productIterator.foreach{ i => println("Value = " + i )}
```

Output

```
Value = 4
Value = 3
Value = 2
Value = 1
```

4. Swapping of tuple elements

We can perform swapping of tuple elements by using tuple.swap method.

Example

```
val t = new Tuple2("tuple", "test")
println("Swapped Tuple: " + t.swap )
```

Output

Swapped tuple: (test,tuple)

5. Converting tuple to String:

We can use tuple.toString() method to concatenate elements of tuple to string.

Example:

```
val t = (1, "hello",20.2356)
println("Concatenated String: " + t.toString() )
```

Output

Concatenated String:(1, hello,20.2356)

For further Scala knowledge, refer Best Scala books for beginners

Learn Scala Map with Examples Quickly & Effectively

1. Scala Map

We Have Already Studies Scala sets in our last tutorial. In this tutorial on Scala Map, we will see how to define and process maps, and what methods to call on them. We will learn to declare a Scala Map, Operations on a Map in Scala, Concatenating Maps, Printing Keys and Values from a Scala Map, Searching for a Key in a Map, Methods to Call on a Map etc. So lets begin…

2. An Introduction to Maps in Scala

A Map in Scala is a collection of key-value pairs, and is also called a hash table. We can use a key to access a value. These keys are unique; however, the values may be common. The default Scala Map is immutable. To use a mutable Map, we use the scala.collection.mutable.Map class. To use both in the same place, refer to the immutable Map as Map, and to the mutable Map as mutable.Map. I also recommend you refer our latest blog on **Scala Closures** which is explained in detail with Examples. For now lets jump to learn declaring a Scala Map.

3. Declaring a Scala Map

We can either declare an empty map or one with values.

a. Declaring an Empty Scala Map

```
scala> var m:Map[String,Int]=Map()
m: Map[String,Int] = Map()
```

Here, we must include the type annotation so it can assign proper types to the variables.

b. Declaring a Map with Values

When we provide values, Scala will use those to infer the type of variables. So, we don't need to include the type annotations.

```
scala> var m=Map("Ayushi"->0,"Megha"->1)
m: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1)
```

Now, to add a key-value pair to this, we do the following:

```
scala> m+=("Ruchi"->2)
scala> m
```

```
3. res1: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1, Ruchi -> 2)
```

Learn: Scala Arrays and Multidimensional Arrays in Scala

4. Operations on a Map in Scala

These are the basic operations we can carry out on a Map:

a. keys

This returns an iterable with each key in the Map.

```
1. scala> m.keys
2. res2: Iterable[String] = Set(Ayushi, Megha, Ruchi)
3. scala> m("Megha")
4. res56: Int = 1
```

b. values

This returns an iterable with each value in the Scala Map.

```
1. scala> m.values
2. res3: Iterable[Int] = MapLike.DefaultValuesIterable(0, 1, 2)
```

c. isEmpty

If the Map is empty, this returns true; otherwise, false.

```
1. scala> m.isEmpty
2. res4: Boolean = false
3. scala> Map().isEmpty
4. res5: Boolean = true
```

5. Concatenating Maps in Scala

We can concatenate/joing two Maps in more than one way. Let's take another Map for this.

```
1. scala> var m1=Map("Megha"->3,"Ruchi"->2,"Becky"->4)
2. m1: scala.collection.immutable.Map[String,Int] = Map(Megha -> 3, Ruchi -> 2, Becky -> 4)
```

The ++ Operator

```
1. scala> m++m1
2. res6: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 3, Ruchi -> 2, Becky -> 4)
3. scala> m1++m
4. res7: scala.collection.immutable.Map[String,Int] = Map(Megha -> 1, Ruchi -> 2, Becky -> 4, Ayushi -> 0)
```

See the difference in the values for "Megha" in both cases?

Learn: Scala Operator

6. Printing Keys and Values from a Map

We can use a foreach loop to walk through the keys and values of a Map:

```
scala> m.keys.foreach{i=>println(i+" "+m(i))}
```

Ayushi 0

Megha 1

Ruchi 2

7. Searching for a Key in a Map

The Map.contains() method will tell us if a certain key exists in the Map.

```
scala> m.contains("Ayushi")
res10: Boolean = true
scala> m.contains("Fluffy")
res11: Boolean = false
```

Any Doubt yet in Scala Map? Please Comment.

8. Methods to Call on a Map

We can call the following methods on a Map. (Note that they don't modify the original Map)

a. def ++(xs: Map[(A, B)]): Map[A, B]

This concatenates two Maps.

```
scala> m.++(m1)
res15: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 3, Ruchi -> 2, Becky -> 4)
scala> m1.++(m)
res16: scala.collection.immutable.Map[String,Int] = Map(Megha -> 1, Ruchi -> 2, Becky -> 4, Ayushi -> 0)
```

b. def -(elem1: A, elem2: A, elems: A*): Map[A, B]

This returns a new Map eliding the pairs for the keys mentioned in the arguments.

```
scala> m.-("Ayushi","Ruchi")
res21: scala.collection.immutable.Map[String,Int] = Map(Megha -> 1)
```

c. def get(key: A): Option[B]

This returns the value associated with the key; it returns this as an Option.

```
scala> m.get("Megha")
res27: Option[Int] = Some(1)
```

```
scala> m.get("Fluffy")
res28: Option[Int] = None
```

d. def iterator: Iterator[(A, B)]

This returns an iterator over the Map.

```
scala> m.iterator
res29: Iterator[(String, Int)] = non-empty iterator
```

Learn: Scala Arrays and Multidimensional Arrays in Scala

e. def addString(b: StringBuilder): StringBuilder

This appends all elements of the Map to the String Builder.

```
scala> m.addString(new StringBuilder())
res30: StringBuilder = Ayushi -> 0Megha -> 1Ruchi -> 2
```

f. def addString(b: StringBuilder, sep: String): StringBuilder

This does what the above method does, except it introduces a separator between the pairs.

```
scala> m.addString(new StringBuilder(),"*")
res31: StringBuilder = Ayushi -> 0*Megha -> 1*Ruchi -> 2
```

g. def apply(key: A): B

This searches for a key in the Scala Map.

```
scala> m.apply("Fluffy")
java.util.NoSuchElementException: key not found: Fluffy
at scala.collection.immutable.Map$Map3.apply(Map.scala:167)
... 28 elided
scala> m.apply("Ayushi")
res33: Int = 0
```

h. def clear(): Unit

This actually removes all bindings from the Map in Scala.

```
scala> import scala.collection.mutable.Map
import scala.collection.mutable.Map
scala> var m2=scala.collection.mutable.Map("One"->1,"Two"->2,"Three"->3)
m2: scala.collection.mutable.Map[String,Int] = Map(One -> 1, Two -> 2, Three -> 3)
scala> m2.clear()
scala> m2
res36: scala.collection.mutable.Map[String,Int] = Map()
```

Doing this to m1 will result in the following error:

```
scala> m1.clear()
<console>:13: error: value clear is not a member of scala.collection.immutable.Map[String,Int]
m1.clear()
^
```

i. def clone(): Map[A, B]

This creates a clone/copy of the receiver object.

We can’t clone an immutable Map. So, let’s revive m2.

```
scala> var m2=scala.collection.mutable.Map("One"->1,"Two"->2,"Three"->3)
m2: scala.collection.mutable.Map[String,Int] = Map(One -> 1, Two -> 2, Three -> 3)
scala> m2.clone()
res38: scala.collection.mutable.Map[String,Int] = Map(One -> 1, Two -> 2, Three -> 3)
```

j. def contains(key: A): Boolean

If the Map contains this key, this returns true; otherwise, false.

```
scala> m.contains("Megha")
res40: Boolean = true
scala> m.contains("Fluffy")
res41: Boolean = false
```

k. def copyToArray(xs: Array[(A, B)]): Unit

This fills key-value pairs from the Map into an Array.

```
scala> var arr:Array[Any]=Array(0,0,0,0,0,0,0,0)
arr: Array[Any] = Array(0, 0, 0, 0, 0, 0, 0, 0)
scala> m.copyToArray(arr)
scala> arr
res47: Array[Any] = Array((Ayushi,0), (Megha,1), (Ruchi,2), 0, 0, 0, 0, 0)
```

l. def count(p: ((A, B)) => Boolean): Int

This returns the number of key-value pairs in the Scala Map that satisfy the given predicate.

```
scala> m.count(x=>true)
res49: Int = 3
```

Learn: Tuples in Scala – A Quick and Easy Tutorial

m. def drop(n: Int): Map[A, B]

This returns all elements except the first n.

```
scala> m.drop(2)
res59: scala.collection.immutable.Map[String,Int] = Map(Ruchi -> 2)
```

n. def dropRight(n: Int): Map[A, B]

This returns all elements except the last n.

```
scala> m.dropRight(2)
res60: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0)
```

o. def dropWhile(p: ((A, B)) => Boolean): Map[A, B]

This drops pairs until the predicate becomes false for a pair.

```
scala> m.dropWhile(x=>false)
res61: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1, Ruchi -> 2)
```

p. def empty: Map[A, B]

This returns an empty Map of the same kind.

```
scala> m.empty
res68: scala.collection.immutable.Map[String,Int] = Map()
```

q. def equals(that: Any): Boolean

This returns true if both Maps contain the same key-value pairs; otherwise, false.

```
scala> m.equals(m1)
res69: Boolean = false
```

r. def exists(p: ((A, B)) => Boolean): Boolean

If the predicate holds true for some elements of the Map, this returns true; otherwise, false.

s. def filter(p: ((A, B))=> Boolean): Map[A, B]

This returns all such elements (see above).

t. def filterKeys(p: (A) => Boolean): Map[A, B]

This returns all pairs where the key satisfies the predicate.

u. def find(p: ((A, B)) => Boolean): Option[(A, B)]

This returns the first element that satisfies the predicate.

v. def foreach(f: ((A, B)) => Unit): Unit

This applies the function to all elements of the Scala Map.

w. def init: Map[A, B]

This returns all elements except the last.

```
scala> m.init
res82: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1)
```

x. def isEmpty: Boolean

If the Map is empty, this returns true; otherwise, false.

```
scala> m.isEmpty
res83: Boolean = false
```

y. def keys: Iterable[A]

This returns an iterator over all keys in the Map.

```
scala> m.keys
res84: Iterable[String] = Set(Ayushi, Megha, Ruchi)
```

z. def last: (A, B)

This returns the last element from the Map in Scala.

```
scala> m.last
res85: (String, Int) = (Ruchi,2)
```

aa. def max: (A, B)

This returns the largest element.

```
scala> m.max
res86: (String, Int) = (Ruchi,2)
```

ab. def min: (A, B)

This returns the smallest element.

```
scala> m.min
res87: (String, Int) = (Ayushi,0)
```

ac. def mkString: String

This represents the elements of the Map as a String.

```
scala> m.mkString
res88: String = Ayushi -> 0Megha -> 1Ruchi -> 2
```

ad. def product: (A, B)

This returns the product of all elements of the Scala Map.

ae. def remove(key: A): Option[B]

This removes a key from the Map and returns the value. Let's take a new Map for this.

```
scala> var d=Map(1->2,3->4,5->6)
d: scala.collection.mutable.Map[Int,Int] = Map(5 -> 6, 1 -> 2, 3 -> 4)
scala> d.remove(3)
res93: Option[Int] = Some(4)
```

af. def retain(p: (A, B) => Boolean): Map.this.type

This retains, in the Map, only the pairs that satisfy the predicate.

Learn: Scala Functions: Quick and Easy Tutorial

ag. def size: Int

This returns the number of elements in the Map.

```
scala> m.size
res94: Int = 3
```

ah. def sum: (A, B)

This returns the sum of all elements in the Map in Scala.

ai. def tail: Map[A, B]

This returns all elements except the last.

```
scala> m.tail
res97: scala.collection.immutable.Map[String,Int] = Map(Megha -> 1, Ruchi -> 2)
```

aj. def take(n: Int): Map[A, B]

This returns the first n elements from the Map.

```
scala> m.take(2)
res98: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1)
```

ak. def takeRight(n: Int): Map[A, B]

This returns the last n elements.

```
scala> m.takeRight(2)
res99: scala.collection.immutable.Map[String,Int] = Map(Megha -> 1, Ruchi -> 2)
```

al. def takeWhile(p: ((A, B)) => Boolean): Map[A, B]

This returns elements from the Map as long as the predicate is satisfied.

am. def toArray: Array[(A, B)]

This returns an Array from the Map.

```
scala> m.toArray
res100: Array[(String, Int)] = Array((Ayushi,0), (Megha,1), (Ruchi,2))
```

an. def toList: List[A]

This returns a List from the Map.

```
scala> m.toList
res101: List[(String, Int)] = List((Ayushi,0), (Megha,1), (Ruchi,2))
```

ao. def toSeq: Seq[A]

This returns a Sequence from the Scala Map.

```
scala> m.toSeq
res102: Seq[(String, Int)] = Vector((Ayushi,0), (Megha,1), (Ruchi,2))
```

ap. def toSet: Set[A]

This returns a Set from the Map.

```
scala> m.toSet
res103: scala.collection.immutable.Set[(String, Int)] = Set((Ayushi,0), (Megha,1), (Ruchi,2))
```

aq. def toString(): String

This returns a String from the Map.

```
scala> m.toString
res104: String = Map(Ayushi -> 0, Megha -> 1, Ruchi -> 2)
```

This was all on Scala Map.

9. Conclusion

A Map in Scala is a collection holding key-value pairs. In this tutorial, we saw how to create, process, and call methods on them.

Scala Sets | Learn About Sets in Scala Collections

1. Scala Sets

In this Scala Sets tutorial, we will learn about sets, how to define Scala Sets, how to process them, and what methods to call on sets in Scala. We will also learn Declaring a set, Operations on sets in Scala collections, Concatenating Sets, Max and Min in a Scala set, and Finding Values common to two sets in Scala.

So, let's start the discussion on Scala Sets.

Scala Sets

2. An Introduction to Sets in Scala

A Scala Set is a collection that won't take duplicates. By default, Scala uses immutable sets. But if you want, you can import the scala.collection.mutable.Set class. To be able to refer to both of these in the same collection, we can refer to the immutable set as Set and the mutable set as mutable.Set. You can also learn Scala Array from our tutorial, once you are done with Scala sets.

3. Declaring a Set

You can either declare an empty set in Scala or one with values.

a. Declaring an Empty Set

We must include the type annotation when declaring an empty set, so it can be decided what concrete type to assign to a variable.

```
scala> var s:Set[Int]=Set()
s: Set[Int] = Set()
```

b. Declaring a Set with Values

We may choose to elide the type annotation. When we do so, Scala will infer that from the type of values inside the set.

```
scala> var s=Set(1,4,4,3)
s: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
scala> var s:Set[Int]=Set(1,4,4,3)
```

```
s: Set[Int] = Set(1, 4, 3)
```

Note that this does not rearrange values as {1,3,4} as Python would do.

4. Operations on Scala Sets

Scala has the following three operations on a set:

a. head

This will return the first element of a Scala set.

```
scala> s.head
res0: Int = 1
```

b. tail

This will return all elements of a set except the first.

```
scala> s.tail
res1: scala.collection.immutable.Set[Int] = Set(4, 3)
```

c. isEmpty

If the set is empty, this will return a Boolean true; otherwise, false.

```
scala> s.isEmpty
res2: Boolean = false
```

5. Concatenating Scala Sets

There are two ways to concatenate sets. Let's take another Scala set for this:

```
scala> var s1=Set(7,9,8,9)
s1: scala.collection.immutable.Set[Int] = Set(7, 9, 8)
```

a. The ++ Operator

```
scala> s++s1
res3: scala.collection.immutable.Set[Int] = Set(1, 9, 7, 3, 8, 4)
scala> s++s1
res4: scala.collection.immutable.Set[Int] = Set(1, 9, 7, 3, 8, 4)
scala> s1++s
res5: scala.collection.immutable.Set[Int] = Set(1, 9, 7, 3, 8, 4)
```

b. The Set.++() Method

We can call the ++() method to either set and pass the other to it.

```
scala> s.++(s1)
res6: scala.collection.immutable.Set[Int] = Set(1, 9, 7, 3, 8, 4)
scala> s1.++(s)
res7: scala.collection.immutable.Set[Int] = Set(1, 9, 7, 3, 8, 4)
```

Read: Scala Operators with Syntax and Examples

6. Max and Min in a Set

We can find the maximum and minimum values in a set.

a. max

This returns the maximum value from a set.

```
scala> s.max
res8: Int = 4
```

b. min

This returns the minimum value from a set.

```
scala> s.min
res9: Int = 1
```

7. Finding Values Common to Two Sets

To determine the values common to two certain sets, we can use one of two methods. Let's take two new sets for this:

```
scala> var a=Set(1,4,3,4,2)
a: scala.collection.immutable.Set[Int] = Set(1, 4, 3, 2)
scala> var b=Set(7,9,8,2,8)
b: scala.collection.immutable.Set[Int] = Set(7, 9, 8, 2)
```

a. The Set.&() Method

We can call &() on one set, and pass another to it.

```
scala> a.&(b)
res10: scala.collection.immutable.Set[Int] = Set(2)
```

b. Set.intersect() Method

We can use the intersect() method.

```
scala> a.intersect(b)
```

```
res11: scala.collection.immutable.Set[Int] = Set(2)
scala> b.intersect(a)
res12: scala.collection.immutable.Set[Int] = Set(2)
```

Read: Scala Functions with Syntax and Examples

8. Methods to Call on a Set

Playing around with sets in Scala, you can call these methods on them. (Note that they do not modify the set itself)

a. def +(elem: A): Set[A]

This adds an element to the set and returns it. (But doesn't modify the original set)

```
scala> a.+("6")
res15: String = Set(1, 4, 3, 2)6
```

b. def -(elem: A): Set[A]

This removes the element from the set and then returns it. Note that this takes an Int instead of a string for our set of Ints.

```
scala> a.-(2)
res19: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
scala> a.-(6)
res21: scala.collection.immutable.Set[Int] = Set(1, 4, 3, 2)
```

c. def contains(elem: A): Boolean

If the set contains that element, this returns true; otherwise, false.

```
scala> a.contains(2)
res22: Boolean = true
scala> a.contains(6)
res23: Boolean = false
```

d. def &(that: Set[A]): Set[A]

This returns an intersection of two sets, as we just saw.

```
scala> a.&(b)
res24: scala.collection.immutable.Set[Int] = Set(2)
```

e. def &~(that: Set[A]): Set[A]

&~ stands for set difference.

```
scala> a.&~(b) #All elements that are in a, but not in b
res25: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
scala> b.&~(a) #All elements that are in b, but not in a
res26: scala.collection.immutable.Set[Int] = Set(7, 9, 8)
```

f. def +(elem1: A, elem2: A, elems: A*): Set[A]

This adds multiple elements to a Scala set and returns it.

```
scala> a.+(0,6,7)
res27: scala.collection.immutable.Set[Int] = Set(0, 1, 6, 2, 7, 3, 4)
```

g. def ++(elems: A): Set[A]

This concatenates a set with another collection.

```
scala> a.++(List(7,9,8))
res29: scala.collection.immutable.Set[Int] = Set(1, 9, 2, 7, 3, 8, 4)
scala> a.++(b)
res30: scala.collection.immutable.Set[Int] = Set(1, 9, 2, 7, 3, 8, 4)
```

h. def -(elem1: A, elem2: A, elems: A*): Set[A]

This removes, each element mentioned from the set.

```
scala> a.-(2,3,6)
res31: scala.collection.immutable.Set[Int] = Set(1, 4)
```

i. def addString(b: StringBuilder): StringBuilder

This adds all elements of the set to the String Builder.

```
scala> a.addString(new StringBuilder())
res32: StringBuilder = 1432
```

j. def addString(b: StringBuilder, sep: String): StringBuilder

This uses a separator to the above functionality.

```
scala> a.addString(new StringBuilder(),"*")
res33: StringBuilder = 1*4*3*2
```

k. def apply(elem: A)

This checks whether the element is part of the set.

```
scala> a.apply(1)
res34: Boolean = true
scala> a.apply(7)
res35: Boolean = false
```

Read: Scala DataTypes

l. def count(p: (A) => Boolean): Int

This returns the count of elements that satisfy the predicate.

```
scala> a.count(x=>{x%2!=0})
res36: Int = 2
```

m. def copyToArray(xs: Array[A], start: Int, len: Int): Unit

These copies len elements from the set to the Array *xs*, starting at position *start*.

```
scala> var c=Array(0,0,0,0,0,0,0,0)
c: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0)
scala> a.copyToArray(c,3,2)
```

```
scala> c
res39: Array[Int] = Array(0, 0, 0, 1, 4, 0, 0, 0)
```

n. def diff(that: Set[A]): Set[A]

This returns the set difference(elements existing in one set, but not in another)

```
scala> a.diff(b)
res40: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
scala> b.diff(a)
res42: scala.collection.immutable.Set[Int] = Set(7, 9, 8)
```

o. def drop(n: Int): Set[A]]

This returns all elements except the first n.

```
scala> a.drop(2)
res45: scala.collection.immutable.Set[Int] = Set(3, 2)
```

p. def dropRight(n: Int): Set[A]

This Scala set returns all elements except the last n.

```
scala> a.dropRight(2)
res46: scala.collection.immutable.Set[Int] = Set(1, 4)
```

q. def dropWhile(p: (A) => Boolean): Set[A]

This drops elements until the first element that doesn't satisfy the predicate.

```
scala> a.dropWhile(x=>{x%2!=0})
res48: scala.collection.immutable.Set[Int] = Set(4, 3, 2)
```

r. def equals(that: Any): Boolean

This set in Scala compares the set to another sequence.

```
scala> a.equals(List(1,4,3,2))
res50: Boolean = false
```

s. def exists(p: (A) => Boolean): Boolean

If the predicate holds true for some elements in the set, this returns true; otherwise, false.

```
scala> a.exists(x=>{x%4==0})
res52: Boolean = true
scala> a.exists(x=>{x%5==0})
res53: Boolean = false
```

t. def filter(p: (A) => Boolean): Set[A]

This filters such elements (see the previous method)

```
scala> a.filter(x=>{x%4==0})
res54: scala.collection.immutable.Set[Int] = Set(4)
```

u. def find(p: (A) => Boolean): Option[A]

This Scala set returns the first element that satisfies the predicate.

```
scala> a.find(x=>{x%2==0})
res55: Option[Int] = Some(4)
```

v. def forall(p: (A) => Boolean): Boolean

This returns true if all elements of the set satisfy the predicate; otherwise, false.

```
scala> a.forall(x=>{x%2==0})
res56: Boolean = false
```

w. def head: A

This Scala Set returns the first element from the set.

```
scala> a.head
res59: Int = 1
```

x. def init: Set[A]

This returns all elements from the set, except the last.

```
scala> a.init
res62: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
```

y. def intersect(that: Set[A]): Set[A]

This returns the intersection of two sets(elements common to both).

```
scala> a.intersect(b)
res64: scala.collection.immutable.Set[Int] = Set(2)
scala> b.intersect(a)
res65: scala.collection.immutable.Set[Int] = Set(2)
```

z. def isEmpty: Boolean

This set in Scala returns true if the set is empty; otherwise, false.

```
scala> a.isEmpty
res66: Boolean = false
```

Read: Scala String: Creating String, Concatenation, String Length

aa. def iterator: Iterator[A]

This creates a new iterator over the set.

```
scala> a.iterator
res67: Iterator[Int] = non-empty iterator
```

ab. def last: A

This returns the last element from a set.

```
scala> a.last
res68: Int = 2
```

ac. def map[B](f: (A) => B): immutable.Set[B]

This Scala Set applies the function to all elements of the set and returns it.

```
scala> a.map(x=>x*2)
res69: scala.collection.immutable.Set[Int] = Set(2, 8, 6, 4)
```

ad. def max: A

This returns the highest value from the set.

```
scala> a.max
res70: Int = 4
```

ae. def min: A

This returns the lowest element.

```
scala> a.min
res71: Int = 1
```

af. def mkString: String

This Scala set represents all elements of the set as a String.

```
scala> a.mkString
res72: String = 1432
```

ag. def mkString(sep: String): String

This lets us define a separator for the above method's functionality.

```
scala> a.mkString("*")
res73: String = 1*4*3*2
```

ah. def product: A

This returns the algebraic product of all elements in the set.

```
scala> a.product
res74: Int = 24
```

ai. def size: Int

This returns the size of the set.

```
scala> a.size
res75: Int = 4
```

aj. def splitAt(n: Int): (Set[A], Set[A])

This splits the set at the given index and returns the two resulting subsets.

```
scala> a.splitAt(2)
res76: (scala.collection.immutable.Set[Int], scala.collection.immutable.Set[Int]) = (Set(1, 4),Set(3, 2))
scala> a.splitAt(3)
res77: (scala.collection.immutable.Set[Int], scala.collection.immutable.Set[Int]) = (Set(1, 4, 3),Set(2))
```

ak. def subsetOf(that: Set[A]): Boolean

If the set passed as argument is a subset of this set, this returns true; else, false.

```
scala> Set(3,2).subsetOf(a)
res79: Boolean = true
scala> Set(2,3).subsetOf(a)
res80: Boolean = true
scala> Set(1,3,4).subsetOf(a)
res82: Boolean = true
```

al. def sum: A

This returns the sum of all elements of the set.

```
scala> a.sum
res83: Int = 10
```

am. def tail: Set[A]

This returns all elements of the set except the first.

```
scala> a.tail
res84: scala.collection.immutable.Set[Int] = Set(4, 3, 2)
```

an. def take(n: Int): Set[A]

This Scala set returns the first n elements from the set.

```
scala> a.take(3)
res85: scala.collection.immutable.Set[Int] = Set(1, 4, 3)
```

ao. def takeRight(n: Int):Set[A]

This returns the last n elements.

```
scala> a.takeRight(3)
res86: scala.collection.immutable.Set[Int] = Set(4, 3, 2)
```

ap. def toArray: Array[A]

This returns an Array holding elements from the set.

```
scala> a.toArray
res87: Array[Int] = Array(1, 4, 3, 2)
```

aq. def toList: List[A]

This returns a List from elements of the set.

```
scala> a.toList
res88: List[Int] = List(1, 4, 3, 2)
```

ar. def toSeq: Seq[A]

This returns a sequence from the set.

```
scala> a.toSeq
res90: Seq[Int] = Vector(1, 4, 3, 2)
```

as. def toString(): String

This represents the elements of the set as a String.

```
scala> a.toString
res91: String = Set(1, 4, 3, 2)
```

This was all about Scala Sets.

9. Conclusion

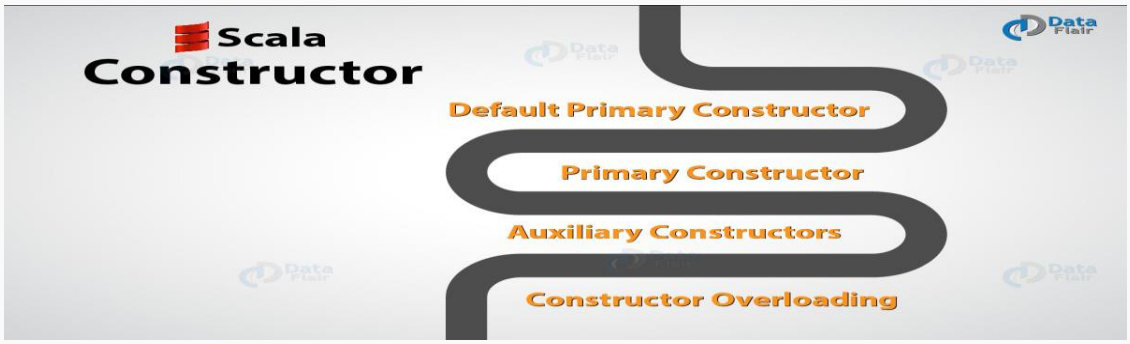
This is all about how to declare and process Scala sets. Next, we will discuss Scala Maps. Furthermore, if you have any query, feel free to ask in the comment section.

Scala Constructor – 2 Popular Types of Constructors in Scala

1. Objective

In our previous tutorial, we studied **Scala trait Mixins** and now we are going to discuss Scala constructor and types of constructor in **Scala Programming Language**: Primary Constructor and Auxiliary Constructors in Scala. At last, we will cover Constructor Overloading in Scala.

So, let's begin Scala Constructor.



Scala Constructor – 2 Popular Types of Constructors in Scala

2. Scala Constructor

Scala constructor is used for creating an instance of a class. There are two types of constructor in Scala – Primary and Auxiliary. Not a special method, a constructor is different in Scala than in **Java constructors**. The class' body is the primary constructor and the parameter list follows the class name. The following, then, is the default primary constructor.

Let's see Scala Currying Function – Example & Partially Applied Function

3. Default Primary Constructor in Scala

```
scala>class Vehicle{
| println("I am in the default constructor")
| }
defined class Vehicle
scala> val v=new Vehicle()
```

I am in the default constructor

v: Vehicle = Vehicle@78288f83

Here, the class body has one print statement. This is the code in the default constructor. We could also have defined it without parentheses:

```
scala> val v=new Vehicle
```

I am in the default constructor

v: Vehicle = Vehicle@7c751692

4. A Primary Constructor in Scala

A class in Scala may have only one primary constructor. Such a constructor may have zero or more parameters.

```
scala> class Vehicle(model:String){
| println("I am a "+model+".")
| }
defined class Vehicle
scala> val v=new Vehicle("Brio")
```

I am a Brio.

v: Vehicle = Vehicle@42cfd794

Read about Scala Method Overloading with Example

Here, a model is a val; we cannot reassign it. Take a look:

```
scala> class Vehicle(model:String){
| println("I am a "+model+".")
| model="Verna"
| println("I am now a "+model+".")
| }
<console>:13: error: reassignment to val
model="Verna"
^
```

But we can declare it to be a var:

```
scala> class Vehicle(var model:String){
| println("I am a "+model+".")
```

```
| model="Verna"
| println("I am now a "+model+".")
| }
defined class Vehicle
scala> val v=new Vehicle("Brio")
```

I am a Brio.

I am now a Verna.

v: Vehicle = Vehicle@11174bf

In this case, the compiler generates setter and getter methods for the var.

Take another example:

```
object Main extends App{
class Vehicle(model:String,color:String){
println("I am a "+color+" "+model+".")
}
var v=new Vehicle("Verna","red")
}
```

In the command prompt, using the Java Class File Disassembler:

C:\Users\lifei\Desktop>javap Main.class

Compiled from "prim.scala"

public final class Main {

public static void main(java.lang.String[]);

public static void delayedInit(scala.Function0<scala.runtime.BoxedUnit>);

public static void delayedEndpoint$Main$1();

public static long executionStart**();**

public static void v_$eq(Main$Vehicle);

public static Main$Vehicle v();

}

Let's discuss Scala Object – Scala Singleton & Scala Companion Object

5. Auxiliary Constructors in Scala

A class may have any number of auxiliary constructors. You must make sure that it must make a call to another auxiliary constructor or to the primary constructor in the first line of its body. Hence, each auxiliary constructor directly or indirectly invokes the primary constructor.

Let's take an example.

```
scala> class Vehicle(model:String,color:String){
| var year:Int=2000
| def show(){
| println("I am a "+color+" "+model+" from "+year)
| }
| def this(model:String,color:String,year:Int){
| this(model,color) //Call to the primary constructor
| this.year=year
| }
| }
defined class Vehicle
scala> var v=new Vehicle("Verna","red",2015)
```

v: Vehicle = Vehicle@18dd7767

```
scala> v.show()
```

I am a red Verna from 2015

```
scala> var v1=new Vehicle("Verna","black")
```

v1: Vehicle = Vehicle@b2d8dcd

```
scala> v1.show()
```

I am a black Verna from 2000

Do you know about Scala File i/o: Open, Read and Write a File in Scala

6. Scala Constructor Overloading

Finally, let us take a look at how to overload constructors.

```
scala> class Vehicle(model:String){
| def this(model:String,color:String)={
| this(model)
| println("I am a "+color+" "+model)
| }
| println(model)
| }
defined class Vehicle
scala> new Vehicle("Prius")
```

Prius

```
res2: Vehicle = Vehicle@421d7900
scala> new Vehicle("Prius","silver")
```

Prius

I am a silver Prius

res3: Vehicle = Vehicle@42ea14b8

This lets us make a call to the constructor that suits the number of arguments we provide.

Have a Look at Scala String Method with Syntax and Method

So, this was all about Scala Constructor. Hope you like our explanation of types of Constructor in Scala.

7. Conclusion

Hence, in this Scala constructor tutorial, we studied types of constructor: Primary Constructor and Auxiliary Constructors in Scala. In Addition, we will cover constructor overloading in Scala Programming Language.

Scala Extractors | Extractors vs Case Classes

1. Objective

In this **Scala tutorial**, we learned what are Extractors in Scala. Moreover, we will discuss Scala Extractors example. Also, we will discuss unapplySeq and apply the method in Scala extractors. Along with this, we will see how can we use Extractors in pattern matching in Scala. At last, we will learn variable argument Scala Extractors.

So let's start Scala extractors.

Scala Extractors | Extractors vs Case Classes

2. What are Scala Extractors?

A Scala extractor is an object in Scala that defines a method unapply() and optionally a method apply(). One use-case of the apply() method is that Scala invokes it on an instance when we instantiate a class with parentheses with a list of parameters zero or more. While an extractor has other members too, these are what we will discuss here.

An apply() method lets us build values and an unapply() method helps us take them apart. Let's take an example.

Read Scala Method Overriding

3. An Example of Extractors in Scala

In the following example, we define a method apply() to turn ("Ayushi","Sharma") into "Ayushi Sharma". unapply() turns this class into an extractor and works to return two strings here from "Ayushi Sharma"- the fname and the lname. Where it observes a string other than a name(because there's no space), it returns None. That is why we make it return an Option.

```
scala> //Optional injection
scala> def apply(fname:String,lname:String)={
| fname+" "+lname
| }
apply: (fname: String, lname: String)String
scala> //Mandatory extraction
scala> def unapply(str:String):Option[(String,String)]={
| val parts=str split " "
| if(parts.length==2){
| Some(parts(0),parts(1))
| }else{
| None
| }
| }
unapply: (str: String)Option[(String, String)]
scala> apply("Ayushi","Sharma")
res4: String = Ayushi Sharma
scala> unapply("Ayushi Sharma")
res5: Option[(String, String)] = Some((Ayushi,Sharma))
scala> unapply("AyushiSharma")
res6: Option[(String, String)] = None
```

4. Using Extractors for Pattern Matching

Where can we make use of this? One such application is pattern-matching. Before you continue, you should read up on Pattern Matching in Scala.

Let's explore 14 Types Scala Pattern Matching

We can compare Scala extractor objects using a match statement. Here, Scala executes the unapply() method. Let's take an example.

```
object Demo{
def main(args: Array[String]){
val y=Demo(3)
//Apply invoked
println("y: "+y)
y match{
case Demo(num)=>println(y+" is twice bigger than "+num)
//Unapply invoked
case _=>println("Sorry")
}
}
def apply(y:Int)=y*2
def unapply(y:Int):Option[Int]=if(y%2==0) Some(y/2) else None
}
```

This gives us the following **output**:

y: 6

6 is twice bigger than 3

5. Variable Argument Extractors in Scala

What happens when we want a Scala extractor that returns a variable number of element values? Let's try taking a domain apart. A domain may be made of more than two sub-patterns. We can then express such patterns:

```
domain match{
case Domain("ro","versuri")=>println("versuri.ro")
case Domain("com", "sun", "java") => println("java.sun.com")
case Domain("net", _*) => println("a .net domain")
```

Let's explore Scala Regex | Scala Regular Expressions

Here, we arrange the domains such that we expand them top-down- from the top-level domain to the subdomains. _* is a wildcard pattern that let us match any remaining elements in a sequence.

To carry out extraction with vararg matching, we can use the method unapplySeq(). This is the Scala extractor object for the same:

```
scala> object Domain{
| //Optional injection
| def apply(parts:String*):String=
| parts.reverse.mkString(".")
|
| //Mandatory extraction
| def unapplySeq(whole:String):Option[Seq[String]]=
| Some(whole.split("\\.").reverse)
| }
defined object Domain
```

In this object, the unapplySeq() method splits the string using periods to get an array of substrings. This is an array with all elements reversed and wrapped in a Some.

6. Extractors vs Case Classes

We have seen about case classes in **Case Classes and Objects in Scala**. They let us model immutable values. But they represent concrete representation of data. So, the class name in a constructor pattern pertains to the concrete representation type for the selector object.

a. Representation Independence

One property of extractors is representation independence. They enable patterns without any relation to the data type for the selected object. This is important in open systems of large size as they let us change implementation type without changing clients. This is where extractors win over case classes.

However, in some places, case classes have their own advantages over extractors. It is easier to define them and set them up. They also need less code. Other than that, the compiler patterns over case classes better than it

does so over extractors. This makes for more efficient pattern matches for case classes. Finally, there are no exhaustiveness checks for extractors.

Concluding, use a case class when writing for a closed application for conciseness, speed, and static checking. Use extractors when you want to expose a type to unknown clients.

So, this was all about Scala Extractors. Hope you like our explanation.

7. Conclusion

Hence, in this Scala Extractors tutorial, we discussed Extractors in Scala. Where it helps us break an object. Moreover, we saw Scala Extractors examples and extractors vs case class. Also, we learned how to use extractors for pattern matching. Tell us your thoughts in the comments.

Scala Iterator: A Cheat Sheet to Iterators in Scala

1. Scala Iterator

We believe you've come here after all other collections. Before proceeding, you should read up on Scala Tuples, another kind of collection. In this tutorial on Scala Iterator, we will discuss iterators, how to create iterator in Scala and process them, and what methods to call on them. So, let's begin.

Scala Iterator

2. An Introduction to Iterators

After collections like Lists, Sets, and Maps, we will discuss a way to iterate on them(how we can access their elements one by one).

Two important methods in Scala Iterator are next() and hasNext. hasNext will tell if there is another element to return, while next() will return that element.

3. Declaring a Scala Iterator

Basically, to declare an iterator in Scala over a collection, we pass values to Iterator().

```
scala> val it=Iterator(7,8,9,2,3)
it: Iterator[Int] = non-empty iterator
```

4. Accessing values with a Scala Iterator

We take this iterator:

```
scala> val it=Iterator(2,4,3,7,9)
it: Iterator[Int] = non-empty iterator
```

Hence, let's take a simple while loop to iterate:

```
scala> while(it.hasNext){
| println(it.next())}
```

2

4

3

7

9

If we hadn't provided the println() to it.next(), it would print nothing. Also, once it reaches the end of the iterator, the iterator is now empty.

5. Determining Largest and Smallest Elements

We can use it.min and it.max to determine the largest and smallest elements in a Scala iterator.

a. min

This will return the smallest value in the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.min
res1: Int = 2
```

b. max

This Scala Iterator will return the largest value in the iterator.

```
scala> it.max
java.lang.UnsupportedOperationException: empty.max
at scala.collection.TraversableOnce.max(TraversableOnce.scala:229)
at scala.collection.TraversableOnce.max$(TraversableOnce.scala:227)
at scala.collection.AbstractIterator.max(Iterator.scala:1432)
```

```
... 28 elided
```

Wow, okay, we must declare the iterator again since it's empty now.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
```

And now:

```
scala> it.max
res3: Int = 9
```

So, let's check if the iterator has exhausted now:

```
scala> it.hasNext
res4: Boolean = false
```

Yes. Yes, it has.

6. An Iterator's Length

Further, we can use either of two methods- size and length.

a. size

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.size
res5: Int = 5
```

b. length

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.length
res6: Int = 5
```

7. Methods to Call on an Iterator

Next, here are some common methods we can call on an iterator:

a. def hasNext: Boolean

If the iterator has another element to return, this returns true; otherwise, false.

```
scala> it.hasNext
res7: Boolean = true
scala> while(it.hasNext){it.next()}
scala> it.hasNext
res9: Boolean = false
```

b. def next(): A

Now, this returns the next element from the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.next()
res10: Int = 3
scala> it.next()
res11: Int = 2
scala> it.next()
res12: Int = 4
scala> it.next()
res13: Int = 9
scala> it.next()
res14: Int = 7
scala> it.next()
```

```
java.util.NoSuchElementException: next on empty iterator
at scala.collection.Iterator$$anon$2.next(Iterator.scala:38)
at scala.collection.Iterator$$anon$2.next(Iterator.scala:36)
at scala.collection.IndexedSeqLike$Elements.next(IndexedSeqLike.scala:60)
... 28 elided
```

c. def ++(that: => Iterator[A]): Iterator[A]

This concatenates two Scala iterators.

```
scala> val a=Iterator(1,2,3).++(Iterator(5,6,7))
a: Iterator[Int] = non-empty iterator
scala> while(a.hasNext){println(a.next())}
```

1

2

3

5

6

7

d. def addString(b: StringBuilder): StringBuilder

This appends the elements of the Scala iterator to a String Builder and returns it.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.addString(new StringBuilder())
res18: StringBuilder = 32497
```

e. def addString(b: StringBuilder, sep: String): StringBuilder

This lets us include a separator for the above functionality.

```
scala> it.addString(new StringBuilder(),"*")
res19: StringBuilder = 3*2*4*9*7
```

f. def buffered: BufferedIterator[A]

This returns a buffered iterator from the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.buffered
res20: scala.collection.BufferedIterator[Int] = non-empty iterator
```

g. def contains(elem: Any): Boolean

If the Scala iterator contains element *elem*, this returns true; otherwise, false.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.contains(9)
res21: Boolean = true
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.contains(8)
res22: Boolean = false
```

h. def copyToArray(xs: Array[A], start: Int, len: Int): Unit

This copies the elements of the array, beginning at index *start*, and for length *len*, into an Array.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> var arr=Array(0,0,0,0,0,0,0,0,0,0)
arr: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
scala> it.copyToArray(arr,2,3)
scala> arr
res28: Array[Int] = Array(0, 0, 3, 2, 4, 0, 0, 0, 0, 0)
```

i. def count(p: (A) => Boolean): Int

This counts the number of elements that satisfy a predicate and returns this count.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.count(x=>{x%2!=0})
res29: Int = 3
```

j. def drop(n: Int): Iterator[A]

This moves the iterator n places ahead. If n is greater than the iterator's length, this simply exhausts it.

```
scala> var it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it=it.drop(2)
it: Iterator[Int] = non-empty iterator
scala> it.next()
res35: Int = 4
```

```
scala> it=it.drop(4)
it: Iterator[Int] = empty iterator
scala> it.next()

java.util.NoSuchElementException: next on empty iterator
at scala.collection.Iterator$$anon$2.next(Iterator.scala:38)
at scala.collection.Iterator$$anon$2.next(Iterator.scala:36)
at scala.collection.IndexedSeqLike$Elements.next(IndexedSeqLike.scala:60)
... 28 elided
```

k. def dropWhile(p: (A) => Boolean): Iterator[A]

This keeps advancing the iterator as long as the predicate is satisfied.

```
scala> var it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it=it.dropWhile(x=>{x<4})
it: Iterator[Int] = non-empty iterator
scala> it.next()
res37: Int = 4
scala> it.next()
res38: Int = 9
scala> it.next()
res39: Int = 7
```

l. def duplicate: (Iterator[A], Iterator[A])

This creates a duplicate of the iterator that will iterate over the same sequence of values.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.duplicate
it1: (Iterator[Int], Iterator[Int]) = (non-empty iterator,non-empty iterator)
```

m. def exists(p: (A) => Boolean): Boolean

If the predicate holds true for some values in the iterator, this returns true; otherwise, false.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.exists(x=>{x%3==0})
res41: Boolean = true
scala> it.exists(x=>{x%5==0})
res42: Boolean = false
```

n. def filter(p: (A) => Boolean): Iterator[A]

This filters out such elements (see above)

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.filter(x=>{x%3==0})
it1: Iterator[Int] = non-empty iterator
scala> it1.next()
res44: Int = 3
scala> it1.next()
```

```
res45: Int = 9
scala> it1.next()
java.util.NoSuchElementException: next on empty iterator
at scala.collection.Iterator$$anon$2.next(Iterator.scala:38)
at scala.collection.Iterator$$anon$2.next(Iterator.scala:36)
at scala.collection.Iterator$$anon$12.next(Iterator.scala:516)
... 28 elided
```

o. def filterNot(p: (A) => Boolean): Iterator[A]

Parallelly, this will create a new iterator with only those elements that do not satisfy the predicate.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.filterNot(x=>{x%3==0})
it1: Iterator[Int] = non-empty iterator
scala> it1.next()
res47: Int = 2
scala> it1.next()
res48: Int = 4
scala> it1.next()
res49: Int = 7
scala> it1.next()
```

p. def find(p: (A) => Boolean): Option[A]

This returns the first value, if any, that satisfies the predicate.

```
scala> val it1=it.find(x=>{x%3==0})
it1: Option[Int] = Some(3)
```

This creates an Option.

q. def forall(p: (A) => Boolean): Boolean

If the predicate holds true for all values in the Scala iterator, this returns true; otherwise, false.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.forall(x=>{x%3==0})
res51: Boolean = false
```

r. def hasDefiniteSize: Boolean

If the iterator is empty, this returns true; otherwise, false.

```
scala> it.hasDefiniteSize
res56: Boolean = true
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.hasDefiniteSize
res57: Boolean = false
```

s. def indexOf(elem: B): Int

This returns the index of the first occurrence of the element *elem* in the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.indexOf(2)
res58: Int = 1
```

t. def indexWhere(p: (A) => Boolean): Int

This returns the index of the first value that satisfies the predicate. If there's none, it returns -1.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.indexWhere(x=>{x%7==0})
res60: Int = 4
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.indexWhere(x=>{x%8==0})
res61: Int = -1
```

u. def isEmpty: Boolean

If hasNext returns false, then the iterator is empty.

```
scala> it.isEmpty
res62: Boolean = true
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.next()
res63: Int = 3
scala> it.hasNext
res64: Boolean = true
scala> it.isEmpty
res65: Boolean = false
```

v. def isTraversableAgain: Boolean

If we can repeatedly traverse the iterator, this returns true; otherwise, false.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.isTraversableAgain
res66: Boolean = false
```

w. def length: Int

Here, this returns the number of elements in the iterator. Once we've called this method, the iterator exhausts.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.length
res69: Int = 5
scala> it
res70: Iterator[Int] = empty iterator
```

x. def map[B](f: (A) => B): Iterator[B]

This applies the function to every value in the iterator and then returns a new iterator from this.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.map(x=>{x*x})
it1: Iterator[Int] = non-empty iterator
scala> it1.next()
res72: Int = 9
scala> it1.next()
res73: Int = 4
scala> it1.next()
res74: Int = 16
scala> it1.next()
res75: Int = 81
scala> it1.next()
res76: Int = 49
```

y. def max: A

This returns the largest element from the Scala iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.max
res77: Int = 9
```

z. def min: A

This returns the smallest element from the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.min
res78: Int = 2
```

aa. def mkString: String

This represents all the elements of the iterator as a String.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.mkString
res79: String = 32497
```

ab. def mkString(sep: String): String

This allows us to declare a separator for the same (see above)

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.mkString("*")
res80: String = 3*2*4*9*7
```

ac. def nonEmpty: Boolean

If the iterator in Scala is empty, this returns false; otherwise, true.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.nonEmpty
res81: Boolean = true
scala> while(it.hasNext){it.next()}
scala> it.nonEmpty
res83: Boolean = false
```

ad. def product: A

This multiplies all elements from the iterator and returns the result.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.product
res92: Int = 1512
```

ae. def sameElements(that: Iterator[_]): Boolean

If both iterators produce the same elements in the same order, this returns true; otherwise, false.

```
val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=Iterator(3,4,2,9,7)
it1: Iterator[Int] = non-empty iterator
scala> it.sameElements(it1)
res93: Boolean = false
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it2=Iterator(3,2,4,9,7)
it2: Iterator[Int] = non-empty iterator
scala> it.sameElements(it2)
res99: Boolean = true
```

af. def seq: Iterator[A]

This returns a sequential view of the iterator in Scala.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.seq
res100: Iterator[Int] = non-empty iterator
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> for(i<-it.seq){println(i)}
```

```
3
2
4
9
7
```

ag. def size: Int

This returns the size of the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.size
res102: Int = 5
```

ah. def slice(from: Int, until: Int): Iterator[A]

This creates a new iterator with elements from *from* until *until*.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.slice(1,4)
it1: Iterator[Int] = non-empty iterator
scala> while(it1.hasNext){println(it1.next())}
```

2

4

9

ai. def sum: A

This returns the sum of all elements in the Scala iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.sum
res1: Int = 25
```

aj. def take(n: Int): Iterator[A]

This returns the first n values from the iterator, or the entire iterator, whichever is shorter.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> val it1=it.take(3)
it1: Iterator[Int] = non-empty iterator
scala> while(it1.hasNext){println(it1.next())}
```

3

2

4

ak. def toArray: Array[A]

This returns an Array from the elements of the iterator.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.toArray
```

```
4. res4: Array[Int] = Array(3, 2, 4, 9, 7)
```

al. def toBuffer: Buffer[B]

This returns a Buffer from the iterator's elements.

```
1. scala> val it=Iterator(3,2,4,9,7)
2. it: Iterator[Int] = non-empty iterator
3. scala> it.toBuffer
4. res5: scala.collection.mutable.Buffer[Int] = ArrayBuffer(3, 2, 4, 9, 7)
```

am. def toIterable: Iterable[A]

This returns a Scala iterable holding all elements of the iterator in Scala. This doesn't terminate for infinite iterators. Let's take a Scala Iterable Example.

```
1. scala> val it=Iterator(3,2,4,9,7)
2. it: Iterator[Int] = non-empty iterator
3. scala> it.toIterable
4. res6: Iterable[Int] = Stream(3, ?)
5. an. def toIterator: Iterator[A]
```

Basically, this returns a new iterator from the iterator.

```
1. scala> val it=Iterator(3,2,4,9,7)
2. it: Iterator[Int] = non-empty iterator
3. scala> val it1=it.toIterator
4. it1: Iterator[Int] = non-empty iterator
5. scala> while(it1.hasNext){println(it1.next())}
```

3

2

4

9

7

an. def toList: List[A]

This Scala iterator to list returns a List from the iterator.

```
1. scala> val it=Iterator(3,2,4,9,7)
2. it: Iterator[Int] = non-empty iterator
3. scala> it.toList
4. res9: List[Int] = List(3, 2, 4, 9, 7)
5. ap. def toSeq: Seq[A]
```

This returns a Sequence from the iterator.

```
1. scala> val it=Iterator(3,2,4,9,7)
2. it: Iterator[Int] = non-empty iterator
3. scala> it.toSeq
4. res11: Seq[Int] = Stream(3, ?)
```

ao. def toString(): String

This represents the iterator as a String.

```
scala> val it=Iterator(3,2,4,9,7)
it: Iterator[Int] = non-empty iterator
scala> it.toString
res12: String = non-empty iterator
```

ap. def zip[B](that: Iterator[B]): Iterator[(A, B)

This returns a new Scala iterator holding pairs of corresponding elements in the iterator. How many elements does this return? Whatever's minimum of the sizes of both iterators.

```
scala> val it1=Iterator(0,0,0,0,0,0,0,0,0,0)
it1: Iterator[Int] = non-empty iterator
scala> it.zip(it1)
res14: Iterator[(Int, Int)] = non-empty iterator
```

8. Conclusion

Finally, we have seen Scala iterators.Moreover, we also discuss the ways to access collection elements one by one. Furthermore, if you have any query, feel free to ask in the comment section.

14 Types Scala Pattern Matching with Example – Latest 2018

1. Objective

In our previous Scala Tutorial, we studied **Scala Trait Mixins** and today we will see what is Scala Pattern Matching with examples. Moreover, we will discuss the types of pattern matching in **Scala Programming Langauge**. At last, we will discuss pattern matching with example & syntax and Scala Pattern Matching Case Class.

So, let's start Scala Pattern Matching.

14 Types Scala Pattern Matching with Example – Latest 2018

2. Scala Pattern Matching

Scala lets us check a value against a pattern with pattern-matching. This is like a**switch in Java**. Not only we can use it to replace if-else ladders also we can use it to deconstruct a value into its constituent parts.

Scala pattern matching is made of constants, **variables**, constructors, and type tests. Using such a pattern, we can test if a value follows a pattern. We can only bind a variable name once in a pattern.

3. Examples of Scala Pattern Matching

Scala pattern may look something like this:

- **1|2|3-** This will match integers between 1 and 3.
- **Some(x)-** This will match values of the format Some(v), where it binds x to the argument v for the constructor.
- **ex: IOException-** This will match all instances of class IOException.
- **x::y::xs-** This will match lists of length two or more. Here, it binds x to the first element in the list, y to the second, and xs to the remainder.
- **(x,_)-** This will match pairs of values. x binds to the pair's first component; the other is a wildcard.

Read about Scala Exceptions and Exception Handling

Let's discuss the types of Scala Pattern Matching.

4. Types of Patterns Matching in Scala

Following are the different types of Scala Pattern Matching, let's discuss them one by one:

Types of Patterns Matching in Scala

a. Variable Patterns

An identifier for a variable pattern begins with a lowercase letter. It matches a value and binds to it the variable name. The wildcard pattern '_' is a special case of this.

SimplePattern ::= `_'

| varid

b. Typed Patterns

Typed patterns are of the format x:T, where x is a pattern variable and T is a type pattern. Here, x is of type T.

PatternOne ::= varid `:' TypePat

| `_' `:' TypePat

c. Pattern Binders

Pattern binders look like x@p. Here, x is a pattern variable and p is the pattern.

PatternTwo ::= varid `@' PatternThree

d. Literal Patterns

Literal patterns match the value that is equal to the literal we specify.

SimplePattern ::= Literal

e. Stable Identifier Patterns

Such a pattern is a stable identifier r. It will match value v that is equal to r.

SimplePattern ::= StableId

Let's review Scala Iterator – Cheat Sheet to Iterators in Scala

f. Constructor Patterns

A constructor pattern takes the format $c(p_1,..,p_n)$, with n>=0. Here, c is a stable identifier and $p_1,..,p_n$ are element patterns. C is a constructor that denotes a case class.

SimplePattern ::= StableId `(' [Patterns] `)

g. Tuple Patterns

A tuple pattern is of the format $(p_1,..,p_n)$. It is an alias for scala.Tuple $n(p_1,..p_n)$, which is a constructor pattern with n>=2.

SimplePattern ::= `(' [Patterns] `)'

h. Extractor Patterns

An extractor pattern is of the format $x(p_1,..p_n)$, where n>=0. While this is the form of a constructor pattern, x does not denote a case class. It denotes an object which allows pattern-matching through a member method unapply or unapplySeq.

SimplePattern ::= StableId `(' [Patterns] `)'

i. Pattern Sequences

Pattern sequences are of the format $p_1,..,p_n$. It may be as a constructor pattern or an extractor pattern. The constructor pattern is of the form $c(q_1,..,q_m,p_1,..p_n)$ and the extractor pattern is of the form $x(q_1,..,q_m,p_1,..p_n)$.

SimplePattern ::= StableId `(' [Patterns `,'] [varid `@'] `_' `*' `)'

Let's discuss Scala File i/o: Open, Read and Write a File in Scala

j. Infix Operation Patterns

This is of the form p; op; q and is a shorthand for op(p,q) which is a constructor or extractor pattern.

Pattern3 ::= SimplePattern {id [nl] SimplePattern}

k. Pattern Alternatives

We can also specify a number of alternative patterns as $p_1|\ldots|p_n$.

Pattern ::= Pattern1 { `|' Pattern1 }

l. XML Patterns

For an XML pattern, the opening bracket '<' for an element pattern should be able to start the lexical XML mode. It should be a single element pattern.

XmlPattern ::= ElementPattern

m. Regular Expression Patterns

Since version 2.0, these patterns have been discontinued in Scala.

n. Irrefutable Patterns

For a pattern p to be irrefutable for type T, one of the following conditions must be met:

- It is a variable pattern
- Typed pattern x:T' and T<:T'
- Constructor pattern $c(p_1,..,p_n)$

Let's explore Scala Functions: Quick and Easy Tutorial

5. Type Patterns in Scala

These are made of types, type variables, and wildcards. For type patterns, we have the following forms:

- A reference to one of these classes- C, p.C, or T#C. This will match any non-null instance of class.
- Singleton type p. type. This will match the value denoted by path p.
- A compound type pattern T_1 with… with T_n. It will match all values matched by each type pattern.

- A parameterized type pattern $T[a_1,\ldots,a_n]$. Here, a_i can type variable patterns or wildcards. This will match all values matching T for arbitrary instantiations of type variables and wildcards.
- Parameterized type pattern $scala.Array[T_1]$. Here, T_1 is a type pattern. This will match non-null instances of type $scala.Array[U_1]$. Here, U_1 matches T_1.

6. A Syntax for Pattern Matching

Here, we use the 'match' keyword and at least one 'case' clause. With the match keyword, we can apply a function to an object.

```
scala> import scala.util.Random
import scala.util.Random
scala> val x:Int=Random.nextInt(7)
x: Int = 5
scala> x match{
| case 1=>"One"
| case 2=>"Two"
| case _=>"One too many"
| }
```

res3: String = One too many

This gives us a random integer between 0 and 7. x is the left operand of the 'match' operator and the right operand is an expression with three cases/alternatives. In the end, we provide a "catch all" case for any value that doesn't match any of the given case values. When a value matches the value of x, the value of the corresponding expression is evaluated.

Do you know about Scala Data Types with Examples

7. Example of Pattern Matching

In the following example, it tests the integer x. If it is 1, it returns the string "One"; if it is 2, it returns "Two". If it is anything else, it returns "One too many". You can say that this maps integers to strings.

```
scala> def test(x:Int):String=x match{
| case 1=>"One"
| case 2=>"Two"
| case _=>"One too many"
| }
test: (x: Int)String
scala> test(4)
res0: String = One too many
scala> test(1)
res1: String = One
scala> test(0)
res2: String = Many
```

8. Pattern Matching Case Class

As we've previously discussed in our article on **Case Classes**, a case class is good for modeling immutable data. It has all vals; this makes it immutable. Let's see how it helps with Scala pattern matching.

```
scala> case class Dog(breed:String,name:String,age:Int)
defined class Dog
scala> val leo=new Dog("Dogue de Bordeaux","Leo",2)
leo: Dog = Dog(Dogue de Bordeaux,Leo,2)
scala> val fluffy=new Dog("Lesser Indian Spitz","Fluffy",8)
fluffy: Dog = Dog(Lesser Indian Spitz,Fluffy,8)
scala> val candy=new Dog("Pug","Candy",6)
candy: Dog = Dog(Pug,Candy,6)
scala> for(dog<-List(leo,fluffy,candy)){
| dog match{
| case Dog("Dogue de Bordeaux","Leo",2)=>println("Hi, Leo")
| case Dog("Lesser Indian Spitz","Fluffy",8)=>println("Hi, Fluffy")
| case Dog(breed,name,age)=>println("Hi, "+name)
| }
| }
```

Hi, Leo

Hi, Fluffy

Hi, Candy

For the case class, the compiler assumes the parameters to be vals. However, we can use the 'var' keyword to attain mutable fields. The compiler implements methods equals(), hashCode(), and toString() for the class. These use the fields as constructor arguments. This means we do not need our own toString() methods.

9. Scala Pattern Guards

We can also add conditions to the code. Simply add an if statement after the case.

```
scala> val x=7
x: Int = 7
scala> x match{
| case 7 if x%2==0=>println("Odd")
| case 7 if x%2!=0=>print("Even")
| }
```

Even

Do You How to Implement Scala Abstract Class with Examples

So, this was all about Scala Pattern Matching Tutorial. Hope you like our explanation.

10. Conclusion

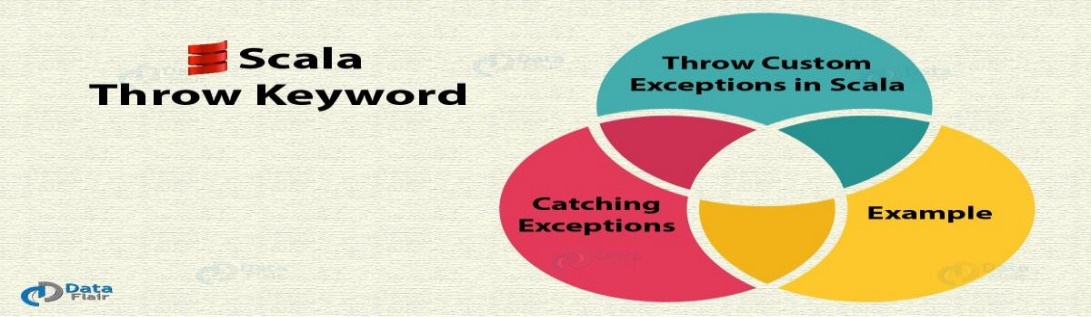
Hence, in this tutorial, we learned introduction to Scala Pattern Matching with examples and syntax. In addition, we will discuss the types of Scala pattern matching. At last, we discussed pattern matching with example & syntax and Scala Pattern Matching Case Class. Furthermore, if you have any query regarding Scala Pattern Matching, feel free to ask in the comment section.

How Can We Use Scala Throw Keyword – Scala Exception

1. Objective

In this tutorial, we will talk about the Scala Throw Keyword. So, in this Scala Throw keyword Tutorial, we are going to see how can we Throw Custom Exception in **Scala Programming Language**. Moreover, we will discuss Catching Exceptions in Scala.

Let's begin Sacla Throw Keyword.



How Can We Use Scala Throw Keyword – Scala Exception

2. How Can We Throw Custom Exceptions in Scala?

Scala lets you create your own custom exceptions using the Scala Throw Keyword. With this, you can explicitly throw an exception in your code.

Let's Get a Clear Information About Scala Exceptions and Exception Handling

Let's take a simple example.

```
scala> def person(age:Int){
| if(age!=15)
| throw new Exception("Wait a little")
| else println("Enjoy your quinceanera")
```

```
5.  | }
6.  person: (age: Int)Unit
7.  scala> person(15)
```

Enjoy your quinceanera

```
1.  scala> person(14)
```

java.lang.Exception: Wait a little

```
1.  at .person(<console>:13)
2.  ... 28 elided
```

3. Scala Throw Keyword

Now, when we know a method can throw an exception, we may want to tell Scala. We use the Scala throws keyword for this, but we can also place the @throws annotation right before a method.

```
1.  @throws(classOf[Exception])
2.  override def play{
3.  //exception-throwing code
4.  }
```

But a method may throw more than one code. To mention all of these, we use multiple **annotations in Scala**.

```
1.  @throws(classOf[IOException])
2.  @throws(classOf[LineUnavailableException])
3.  @throws(classOf[UnsupportedAudioFileException])
4.  def playSoundFileWithJavaAudio{
5.  //exception-throwing code
6.  }
```

@throws is also a way to provide the method signature for 'throws' to those who work with **Java Programming Language**. Java would have the following code for this:

```
1.  public void play() throws FooException{
2.  //code
3.  }
```

Do you Know 2 Popular Types of Constructors in Scala

4. More On Throw Keyword in Scala

Talking about checked exceptions, Scala doesn't need methods to declare that they can throw exceptions. In Scala, all exceptions are *RuntimeExceptions*, and it is the developer who must decide when to handle them. It doesn't force us to handle these exceptions.

```
1.  scala> def demo{
2.  | throw new Exception
3.  | }
```

```
demo: Unit
```

We don’t even need to call these methods to catch the exceptions.

```
scala> demo
java.lang.Exception
at .demo(<console>:12)
... 28 elided
```

But we must test for them if we want our code to work fine:

```
scala> object Demo extends App{
| def demo{
| throw new Exception}
| println("After demo")
| demo
| //Code never executed
| println("After demo")
| }
defined object Demo
```

Read about Scala Regular Expressions – Replacing Matches

5. Catching Exceptions in Scala

In the following example, we throw an exception and then have a wildcard pattern case catch it. This pattern catches any exception of the kind throwable.

```
scala> try{
| throw new Exception("Failed")
| } catch{
| //catching all Throwable exceptions
| case _:Throwable=>println("An exception occurred")
| }
An exception occurred
```

6. Example- Exceptions With Pattern Matching

Let’s take another example. Here, we combine **pattern-matching** with **exception handling**. This is a bit like the previous example.

```
scala> import java.net.URL
import java.net.URL
scala> import java.net.MalformedURLException
import java.net.MalformedURLException
scala> import java.io.IOException
import java.io.IOException
scala> try{
| val url=new URL("http://data-flair.training")
| }catch{
```

```
10. | case e:MalformedURLException=>println("Bad URL "+e)
11. | case e:IOException=>println("An IO issue "+e)
12. | case _:Throwable=>println("Something else")
13. | }finally{
14. | //cleanup
15. | }
```

Let's Explore Scala Singleton & Scala Companion Object

So, this was all about Scala Throw Keyword. Hope you like our explanation.

7. Conclusion

Hence, in this Scala Throw tutorial, we have seen how we can throw custom exceptions in Scala. Drop your queries in the comments.

What is Scala Thread & Multithreading | File Handling in Scala

1. Objective

In our last Scala tutorial, we discussed **Scala Pattern Matching**. Here, we divide this article into two sections- File Handling in Scala Programming Language and Scala Thread. Moreover, how can we write & read files in Scala and multithreading in Scala? Along with this, we will study what is Scala thread and how can we create a thread in Scala. At last, we learn Scala thread methods.

So, let's start Scala Thread.

2. Scala File Handing

Scala lets us read from and write to files. Here, we will learn how to open files, read from them, and write to them. Let's take a quick look.

3. How to Write Files in Scala

Since Scala doesn't have a class that will help us with writing files, we use java.io._ from Java.

C:\Users\lifei>cd Desktop

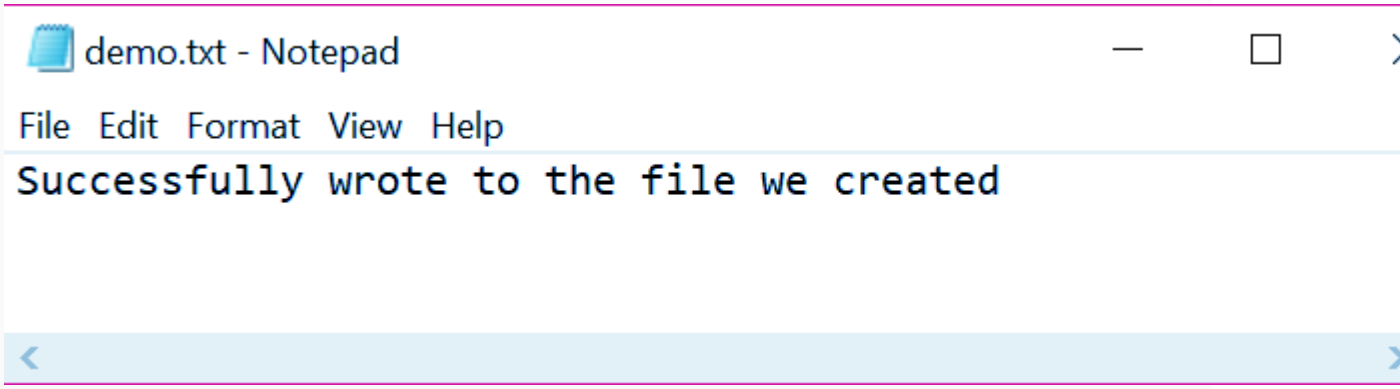
C:\Users\lifei\Desktop>scala

Refer For More Detail Scala File i/o: Open, Read and Write a File in Scala

Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try:help.

```
scala> import java.io._
import java.io._
scala> val writer=new PrintWriter(new File("demo.txt"))
writer: java.io.PrintWriter = java.io.PrintWriter@3c3e363
scala> writer.write("Successfully wrote to the file we created")
scala> writer.close()
```

This gives us the following file on the desktop:



Scala Thread & Multithreading | File Handling in Scala

4. How to Read Files in Scala

To read from a file, we use scala.io.Source. Take a look.

```
scala> import scala.io.Source
import scala.io.Source
scala> Source.fromFile("demo.txt").mkString
res2: String = Successfully wrote to the file we created
```

But if the file had more than one line:

```
scala> Source.fromFile("demo.txt").getLines.foreach{x=>println(x)}
```

Successfully wrote to the file we created
Line 1
Line 2
Using an **iterator**:

```
scala> val it=Source.fromFile("demo.txt").getLines()
it: Iterator[String] = non-empty iterator
scala> it.next()
res4: String = Successfully wrote to the file we created
scala> it.next()
res5: String = Line 1
scala> it.next()
res6: String = Line 2
```

For more on file handling, read up on Scala File IO.

Let's Learn Scala Map with Examples Quickly & Effectively

5. What is Scala Multithreading

Scala supports multithreading, which means we can execute multiple threads at once. We can perform multiple operations independently and achieve multitasking. This lets us develop concurrent applications.

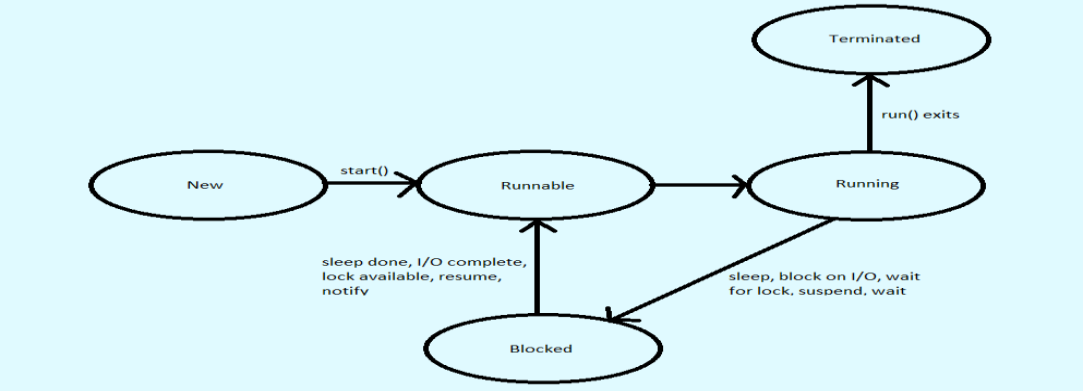
6. What is Scala Threads

A thread is a lightweight sub-process occupying less memory. To create it, we can either extend the Thread class or the Runnable interface. We can use the run() method then. A thread holds less memory than a full-blown process. Let's take a look at its life-cycle.

Read about Scala Closures with Examples

a. Scala Thread Life-Cycle

A Scala thread life-cycle is an account of when it starts to when it terminates. It has the following five states:
1. New- Scala thread is in this state when it is created. This is effectively the first state.
2. Runnable- When the thread has been created but the thread scheduler hasn't selected it for running, it is in the runnable state.
3. Running- When the thread scheduler has selected a thread, it is running.
4. Blocked- In the blocked state, the thread is alive, but waiting for input or resources.
5. Terminated- When the run() method exits, the Scala thread is dead.



Scala Thread & Multithreading | File Handling in Scala

7. How to Create Thread in Scala

Scala Thread is created by two steps:

How to Create Thread in Scala

a. Extending the Thread Class

Like discussed above, we can extend the Scala thread class or the runnable interface. Let's try the first option.

```
scala> class Demo extends Thread{
| override def run(){
| println("Running")
| }
| }
defined class Demo
scala> var d=new Demo()
d: Demo = Thread[Thread-2,5,main]
scala> d.start()
scala> Running
```

Let's Learn Scala String Interpolation – s String, f String

b. Extending the Runnable Interface

Now, let's try extending the interface runnable.

```
scala> class Demo extends Runnable{
| override def run(){
| println("Running")
| }
| }
defined class Demo
scala> var d=new Demo()
d: Demo = Demo@7d28cdcd
scala> var t=new Thread(d)
t: Thread = Thread[Thread-2,5,main]
scala> t.start()
```

```
scala> Running
```

8. Scala Thread Methods

We discussed the states in a Scala thread life-cycle. We can control thread flow to deal with these states using these methods. Let's take a few examples.

Scala Thread Methods

a. sleep()

We can use this method to put a Scala thread to sleep for a number of milliseconds. Let's take an example.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-1 to 7){
| println(i)
| Thread.sleep(400)
| }
| }
| }
defined class Demo
scala> var d1=new Demo()
d1: Demo = Thread[Thread-3,5,main]
scala> var d2=new Demo()
d2: Demo = Thread[Thread-4,5,main]
scala> d1.start(); d2.start()
scala> 1
1
2
2
3
3
4
4
5
5
6
6
7
7
```

This code prints numbers 1 to 7 in pairs. You can say it prints two 1s, then waits for 400 milliseconds, then prints two 2s, then waits 400 milliseconds, and so on.

Let's read about Scala Variables with Examples

b. join()

join() lets us hold execution of the Scala thread currently running until it finishes executing. It waits for a thread to die before it can start more.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-1 to 7){
| println(i)
| Thread.sleep(400)
| }
| }
| }
defined class Demo
scala> var d1=new Demo(); var d2=new Demo(); var d3=new Demo()
d1: Demo = Thread[Thread-14,5,main]
d2: Demo = Thread[Thread-15,5,main]
d3: Demo = Thread[Thread-16,5,main]
scala> d1.start(); d1.join(); d2.start(); d3.start()
1
2
3
4
5
6
7
1
1
scala> 2
2
3
3
4
4
5
5
6
6
7
7
```

In this code, we call start() on d1, then join() on the same object. Then, we call start() on objects d2 and d3. This calls start() on d1; then join() executes it until it dies. And then, it calls start() on d2 and d3, and they run. To understand this, let's take another example.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-0 to 5){
| println("1")
| Thread.sleep(400)
| }
| }
| }
defined class Demoscala> class Demo1 extends Thread{
| override def run(){
| println("Demo1")
| Thread.sleep(400)
| }
| }
defined class Demo1
scala> class Demo2 extends Thread{
```

```
| override def run(){
| println("Demo2")
| Thread.sleep(400)
| }
| }
defined class Demo2
scala> var d=new Demo(); var d1=new Demo1(); var d2=new Demo2()
d: Demo = Thread[Thread-29,5,main]
d1: Demo1 = Thread[Thread-30,5,main]
d2: Demo2 = Thread[Thread-31,5,main]
scala> d.start(); d.join(); d1.start(); d2.start()
1
1
1
1
1
1
Demo1
Demo2
```

In this example, we see in the REPL that 1 prints six times with an interval of 400 milliseconds. And then, Demo1 and Demo2 print simultaneously. This clears it. Let's move to another method.

Let's See Scala if-else Statements with Statements

c. setName() and getName()

The methods setName() and getName() let us set and get names of threads. Let's see how.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-1 to 7){
| println(this.getName()+"- "+i)
| Thread.sleep(400)
| }
| }
| }
defined class Demo
scala> var d1=new Demo(); var d2=new Demo(); var d3=new Demo()
d1: Demo = Thread[Thread-35,5,main]
d2: Demo = Thread[Thread-36,5,main]
d3: Demo = Thread[Thread-37,5,main]
scala> d1.setName("One"); d2.setName("Two"); d3.setName("Three")
scala> d1.start(); d2.start(); d3.start()
Two- 1
One- 1
Three- 1
scala> Two- 2
One- 2
Three- 2
Two- 3
One- 3
Three- 3
Two- 4
One- 4
```

```
Three- 4
Two- 5
One- 5
Three- 5
Two- 6
One- 6
Three- 6
Two- 7
One- 7
Three- 7
```

This code runs the loop seven times for each Scala thread. This time, the order is Two, One, Three. The next time you try it, it may be One, Three, Two.

d. getPriority() and setPriority()

Using these methods, we can get and set the priority for a Scala thread. Let's take an example.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-1 to 7){
| println(this.getName())
| println(this.getPriority())
| Thread.sleep(400)
| }
| }
| }
defined class Demo
scala> var d1=new Demo(); var d2=new Demo()
d1: Demo = Thread[Thread-47,5,main]
d2: Demo = Thread[Thread-48,5,main]
scala> d1.setName("One"); d2.setName("Two")
scala> d1.setPriority(Thread.MIN_PRIORITY); d2.setPriority(Thread.MAX_PRIORITY)
scala> d1.start(); d2.start()
scala> Two
One
10
1
Two
10
One
1
Two
10
One
1
Two
10
One
1
Two
10
One
1
Two
10
One
1
```

```
Two
10
One
1
Two
10
One
1
```

This example tells us that the maximum priority is 10 and the minimum is 1.

Have a look – Scala Access Modifiers: Public, Private and Protected Members

e. task()

In this example, we create multiple Scala threads and use them to run multiple tasks. This lets us implement multitasking in Scala.

```
scala> class Demo extends Thread{
| override def run(){
| for(i<-1 to 4){
| println(i)
| Thread.sleep(400)
| }
| }
| def task(){
| for(i<-5 to 8){
| println(i)
| Thread.sleep(200)
| }
| }
| }
defined class Demo
scala> var d1=new Demo()
d1: Demo = Thread[Thread-51,5,main]
scala> d1.start(); d1.task()
1
5
6
2
7
8
3
scala> 4
```

f. More Methods

Other than and including what we just saw, we have the following methods:

1. Public static void sleep(long millis) throws InterruptedException- This puts the thread to sleep for a certain number of milliseconds.
2. Public final String getName()- This method gets the name of the thread.

3. Public final void setName(String name)- This method lets us set the name of the thread.
4. Public final int getPriority()- This method gives us the priority of the thread.
5. Public final void setPriority(int newPriority)- This method lets us set priority for a thread.
6. Public final boolean isAlive()- This tells us if a thread is alive (somewhere between states New and Terminated.
7. Public final void join() throws InterruptedException- Like we've seen, this method waits for the thread to die before starting another.
8. Public static void yield()- Yield compels the current running thread to give up control to other threads. It pauses the current thread.
9. Public Thread.State getState()- This returns the state of the thread. It lets us monitor system state.

So, this was all about Scala Thread & Multithreading. Hope you like our explanation.

9. Conclusion

Hence, in this Scala thread tutorial, we studied what is threading in Scala Programming. Along with this, we learned about File handling in Scala. Furthermore, if you have any query regarding Scala Threading, Feel free to ask in the comment section.

Learn Scala List with Different Methods and Examples

1. Scala List

In Scala, we have 6 kinds of collections. These are lists, sets, maps, **tuples**, options, and iterators. In this tutorial on Scala list, we will focus on Lists. We will learn to declare a Scala list, Creating Uniform List, Methods on a Scala list, Operations on a Scala list, Reversing a List in Scala Collections, Defining List using:: Nil, Tabulating a Function, Concatenating Scala list.

So, let's start learning Scala Lists.

Scala List

2. Introduction of List in Scala Collections

A collection holds things. This can be strict or lazy(don't consume memory until accessed). This can also be mutable(reassignable) or immutable. In this tutorial, we will discuss Lists. Next, we'll look at Sets. Let's begin.

A list of Scala Collections is much like a **Scala array**. Except that, it is immutable and represents a linked list. An Array, on the other hand, is flat and mutable. Here's an Array:

```
scala> var a=Array(1,2,'a')
a: Array[Int] = Array(1, 2, 97)
scala> a(2)
res3: Int = 97
scala> a(2)=98
scala> a
res5: Array[Int] = Array(1, 2, 98)
```

3. Declaring a Scala List

Now, here's a list to compare that to:

```
scala> var a=List(1,2,'a')
a: List[Int] = List(1, 2, 97)
scala> a(2)
res8: Int = 97
scala> a(3)=98
<console>:13: error: value update is not a member of List[Int]
a(3)=98
^
scala> a=List(1,2,"Ayushi",true)
a: List[Any] = List(1, 2, Ayushi, true)
```

While declaring a Scala list, we can also denote the type of elements it will hold. If it'll hold more than one kind, we use 'Any'. Here are few examples.

```
scala> val a:List[Int]=List(1,3,2)
a: List[Int] = List(1, 3, 2)
scala> val b:List[Char]=List('a','c','b')
b: List[Char] = List(a, c, b)
scala> val c:List[String]=List("Ayushi","Megha")
c: List[String] = List(Ayushi, Megha)
scala> val d:List[Nothing]=List()
d: List[Nothing] = List()
scala> val e:List[List[Int]]=List(List(1,2,3),List(4,5,6),List(7,8,9))
e: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
```

4. Defining Lists using :: and Nil

Think of Nil as an empty list, and :: as cons. If we had to define the above lists this way, we'd:

```
scala> val a=1::(2::("Ayushi"::(true::Nil)))
a: List[Any] = List(1, 2, Ayushi, true)
scala> val a=1::2::3::Nil
a: List[Int] = List(1, 2, 3)
scala> val b='a'::('c'::('b'::Nil))
b: List[Char] = List(a, c, b)
scala> val c="Ayushi"::("Megha"::Nil)
c: List[String] = List(Ayushi, Megha)
scala> val d=Nil
d: scala.collection.immutable.Nil.type = List()
scala> val e=(1::(2::(3::Nil)))::(4::(5::(6::Nil)))::(7::(8::(9::Nil)))::Nil
e: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
```

5. Operations on Lists

We can carry out three operations on a Scala list to get a result:

a. head

This returns the first element of a list.

```
scala> e.head
res12: List[Int] = List(1, 2, 3)
scala> b.head
res13: Char = a
```

b. tail

This returns the last element of a list.

```
scala> c.tail
res14: List[String] = List(Megha)
```

c. isEmpty

If the list is empty, this returns a Boolean true; otherwise, false.

```
scala> d.isEmpty
res15: Boolean = true
scala> a.isEmpty
res16: Boolean = false
```

6. Concatenating Lists

We can join or concatenate two Scala lists in one of three ways. Let's take two lists for this:

```
scala> val a=List(1,2,3)
a: List[Int] = List(1, 2, 3)
scala> val b=List(4,5,6)
b: List[Int] = List(4, 5, 6)
```

a. The ::: Operator

```
scala> a:::b
res0: List[Int] = List(1, 2, 3, 4, 5, 6)
```

b. The List.:::() Method

We can call the :::() method on the first list.

```
scala> a.:::(b)
res2: List[Int] = List(4, 5, 6, 1, 2, 3)
scala> a
res3: List[Int] = List(1, 2, 3)
```

c. The List.concat() Method

We can also call the concat() method on List in Scala Collections. We pass both the lists as arguments.

```
scala> List.concat(a,b)
res5: List[Int] = List(1, 2, 3, 4, 5, 6)
```

Read: Scala Operators with Syntax and Examples

7. Creating Uniform Lists

The method List.fill() creates a list and fills it with zero or more copies of an element.

```
scala> val f=List.fill(7)(1)
f: List[Int] = List(1, 1, 1, 1, 1, 1, 1)
```

This fills the list with seven instances of the integer 1.

8. Tabulating a Function

Using the List.tabulate() method with a function, we can tabulate a list in Scala. The first argument specifies the dimensions; the second describes the elements(computed from a function).

```
scala> val g=List.tabulate(7)(n=>n*2)
g: List[Int] = List(0, 2, 4, 6, 8, 10, 12)
```

We can also pass more than one size argument:

```
scala> val h=List.tabulate(3,7)(_*_)
h: List[List[Int]] = List(List(0, 0, 0, 0, 0, 0, 0), List(0, 1, 2, 3, 4, 5, 6), List(0, 2, 4, 6, 8, 10, 12))
```

This made a List holding three Lists of lengths 7 each.

```
scala> val i=List.tabulate(7,5)(_*_)
i: List[List[Int]] = List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12), List(0, 4, 8, 12, 16), List(0, 5, 10, 15, 20), List(0, 6, 12, 18, 24))
```

You can see that this creates a List holding 7 Lists each holding 5 elements. These are tables of integers from 0 up.

Read: Scala Functions with Syntax and Examples

9. Reversing a List

This reverses the order of elements in a list using the List.reverse method.

```
scala> a
res6: List[Int] = List(1, 2, 3)
scala> a.reverse
res7: List[Int] = List(3, 2, 1)
scala> a
```

```
res8: List[Int] = List(1, 2, 3)
```

10. Methods on a List

We can call the following methods on a Scala List: (Note that these don't modify the Lists)

a. def +(elem: A): List[A]

This postpends an element to the list.

```
scala> a.+("2")
res86: String = List(1, 2, 3)2
```

b. def :::(prefix: List[A]): List[A]

This joins the List in the argument to the other List.

```
scala> a.:::(List(7,8,9))
res16: List[Int] = List(7, 8, 9, 1, 2, 3)
```

c. def ::(x: A): List[A]

This adds an element to a List's beginning.

```
scala> a.::("2")
res17: List[Any] = List(2, 1, 2, 3)
```

d. def addString(b: StringBuilder): StringBuilder

This appends all elements of a list to a String Builder.

```
scala> var b=new StringBuilder()
b: StringBuilder =
scala> a.addString(b)
res19: StringBuilder = 123
scala> a
res20: List[Int] = List(1, 2, 3)
scala> b
res21: StringBuilder = 123
```

e. def addString(b: StringBuilder, sep: String): StringBuilder

This does the same thing, except with a separator between the elements.

Let's first reset b to an empty string.

```
scala> b=new StringBuilder()
b: StringBuilder =
scala> a.addString(b,"*")
res23: StringBuilder = 1*2*3
```

Read: Scala Data Types

f. def apply(n: Int): A

This selects an element in the Scala List by its index.

```
scala> a.apply(2)
res24: Int = 3
```

g. def contains(elem: Any): Boolean

If the list contains a certain element, this returns true; otherwise, false.

```
scala> a.contains(2)
res25: Boolean = true
scala> a.contains(4)
res26: Boolean = false
```

h. def copyToArray(xs: Array[A], start: Int, len: Int): Unit

This copies the elements of a List to an Array. Start decides where to write, and len decides the length of elements to copy.

```
scala> var arr=Array(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
arr: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
scala> a.copyToArray(arr,2,3)
scala> arr
res42: Array[Int] = Array(0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Let’s try changing the length to 2.

```
scala> var arr=Array(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
arr: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
scala> a.copyToArray(arr,2,2)
scala> arr
res44: Array[Int] = Array(0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

i. def distinct: List[A]

Let’s take a new Scala List for this.

```
scala> var j=List(1,1,4,1,3,2,1)
j: List[Int] = List(1, 1, 4, 1, 3, 2, 1)
```

Distinct returns a new List without duplicates.

```
scala> j.distinct
res46: List[Int] = List(1, 4, 3, 2)
```

j. def drop(n: Int): List[A]

This returns all elements except the first n.

```
scala> j.drop(2)
res47: List[Int] = List(4, 1, 3, 2, 1)
```

k. def dropRight(n: Int): List[A]

This returns all elements except the last n.

```
scala> j.dropRight(2)
res48: List[Int] = List(1, 1, 4, 1, 3)
```

Read Scala String: Creating String, Concatenation, String Length

l. def dropWhile(p: (A) => Boolean): List[A]

This drops the longest prefix of elements that satisfy the predicate.

```
scala> j.dropWhile(x=>{x%2!=0})
res52: List[Int] = List(4, 1, 3, 2, 1)
```

This dropped the elements 1 and 1. Then it stopped at 4 because it doesn't satisfy the predicate.

m. def equals(that: Any): Boolean

This compares two sequences.

```
scala> a.equals(b)
res53: Boolean = false
scala> a.equals(List(1,2,3))
res55: Boolean = true
n. def exists(p: (A) => Boolean): Boolean
```

This returns true if a predicate holds true for any value in the List.

```
scala> a.exists(x=>{x%2!=0})
res56: Boolean = true
o. def filter(p: (A) => Boolean): List[A]
```

This returns all such values.

```
scala> a.filter(x=>{x%2!=0})
res57: List[Int] = List(1, 3)
```

p. def forall (p: (a) => Boolean) : Boolean

This returns true if the predicate holds true for all values in the List.

```
scala> a.forall(x=>{x%2!=0})
res58: Boolean = false
```

q. def indexOf(elem: A, from: Int): Int

This returns the index of the first occurrence of the element in the List.

```
scala> a.indexOf(2)
res61: Int = 1
```

r. def init: List[A]

This returns all elements except the last.

```
scala> a.init
res62: List[Int] = List(1, 2)
```

s. def iterator: Iterator[A]

This creates an iterator over the List.

```
scala> a.iterator
res63: Iterator[Int] = non-empty iterator
```

t. def lastIndexOf(elem: A, end: Int): Int

This returns the index of the last occurrence of an element, before or at given index.

```
scala> List(1,2,1,2,1).lastIndexOf(2)
res64: Int = 3
```

u. def length: Int

This returns a List's length.

```
scala> a.length
res65: Int = 3
```

v. def map[B](f: (A) => B): List[B]

This applies the function to every element in the list.

```
scala> a.map(x=>x*x)
res66: List[Int] = List(1, 4, 9)
```

w. def max: A

This returns the highest element.

```
scala> a.max
res67: Int = 3
```

x. def min: A

This returns the lowest element.

```
scala> a.min
res68: Int = 1
```

y. def mkString: String

This displays all elements of a list in a String.

```
scala> a.mkString
res69: String = 123
```

z. def mkString(sep: String): String

This lets us define a separator for mkString.

```
scala> a.mkString("*")
res71: String = 1*2*3
```

aa. def reverse: List[A]

This reverses a List of Scala Collections.

```
scala> a.reverse
res72: List[Int] = List(3, 2, 1)
```

ab. def sum: A

This returns the sum of all elements.

```
scala> a.sum
```

```
2. res74: Int = 6
```

ac. def take(n: Int): List[A]

This returns the first n elements.

```
1. scala> a.take(2)
2. res75: List[Int] = List(1, 2)
```

ad. def takeRight(n: Int): List[A]

This returns the last n elements.

```
1. scala> a.takeRight(2)
2. res76: List[Int] = List(2, 3)
```

ae. def toArray: Array[A]

This returns an Array from a List.

```
1. scala> a.toArray
2. res78: Array[Int] = Array(1, 2, 3)
```

af. def toBuffer[B >: A]: Buffer[B]

This returns a Buffer from a List.

```
1. scala> a.toBuffer
2. res79: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3)
```

ag. def toSeq: Seq[A]

This returns a Seq from a List.

```
1. scala> a.toSeq
2. res81: scala.collection.immutable.Seq[Int] = List(1, 2, 3)
```

ah. def toSet[B >: A]: Set[B]

This returns a Set from a List.

```
1. scala> a.toSet
2. res82: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

ai. def toString(): String

This returns a String from a List.

```
1. scala> a.toString
2. res83: String = List(1, 2, 3)
```

This was all about Scala List. Hope you like our explanation.

11. Conclusion

This is all about Scala List. Stay tuned for tutorials on other kinds of collections like Sets and Maps. Furthermore, if you have any query, feel free to ask in the comment section.

Scala Closures with Examples | See What is Behind the Magic

1. Scala Closures

In this chapter, we find out about closures in Scala. We will learn about the problem that Scala Closures solve, Examples of Closures in Scala, see what is behind the magic and working of Datatypes.

Scala Closures

2. The Problem

What do you do when you want to pass a function around like a variable? How do you let it refer to one or more fields in the same scope? This piece is to address this issue.

3. Scala Closures Introduction and Example

A function whose return value depends on variable(s) declared outside it, is a closure. This is much like that in Python.

We've seen how to declare an anonymous function in Scala:

```
scala> val sum=(a:Int,b:Float)=>a+b
sum: (Int, Float) => Float = $$Lambda$1101/539731466@12ebcdf6
```

Let's make a call to it:

```
scala> sum(2,3)
res2: Float = 5.0
```

This added the numbers 2 and 3, and returned 5.

Now, consider this version:

```
scala> var c=7
```

```
c: Int = 7
scala> val sum1=(a:Int,b:Int)=>(a+b)*c
sum1: (Int, Int) => Int = $$Lambda$1103/1497170291@4a0a93ce
```

Here, the function ‘sum1’ refers to the variable/value ‘c’, which isn’t a formal parameter to it. Let’s try calling the function.

```
scala> sum1(2,3)
res3: Int = 35
```

So, while ‘sum’ is trivially closed over itself, ‘sum1’ refers to ‘c’ every time we call it, and reads its current value. Let’s try changing the value of c:

```
scala> c=8
c: Int = 8
```

We changed ‘c’ from 7 to 8. Let’s try calling sum1 again:

```
scala> sum1(2,3)
res5: Int = 40
```

As you can see, it reads the current value of c.

4. Behind the Magic

Coming from a Java or C++ background, this must be nonplussing. Not only did it use the correct value of ‘c’ the first time, it picked up the change in it when we called it a second time. Let’s find out how this works.

In our example, ‘a’ and ‘b’ are formal parameters to the function sum1. ‘c’, however, is a reference to a variable in the enclosing scope. Scala closes over ‘c’ here.

Paul Cantrell, in his article on Closures in Ruby, mentions three criteria for a closure:

1. We can pass around the code block as a value
2. At any time, anyone with the value can execute the code block
3. It can refer to variables from the context we created it in

It is like two lovers separated in distance, yet united by the heart. Not only do they remember each other; they can sense what happens to the other.

5. Another Example

Let’s work up another example.

```
scala> var age=22
age: Int = 22
scala> val sayhello=(name:String)=>println(s"I am $name and I am $age")
sayhello: String => Unit = $$Lambda$1065/467925240@78b9155e
scala> sayhello("Ayushi")
```

I am Ayushi and I am 22

Now, let's take another function func that takes two arguments: a function and a string, and calls that function on that string. Did we mention that Scala supports higher-order functions?

```
scala> def func(f:String=>Unit,s:String){
| f(s)
| }
func: (f: String => Unit, s: String)Unit
scala> func(sayhello,"Ayushi")
```

I am Ayushi and I am 22

With this example, we hope it gets clearer to you.

6. Working with Other Data Types

Just to be clearer, let's try this on a data type different that an Integer.

a. Integer Arrays

First, we'll do this on an Integer array. Let's declare an array with three numbers:

```
scala> val nums=Array(1,2,4)
nums: Array[Int] = Array(1, 2, 4)
```

Next, we define a function to work with this:

```
scala> val saynum=(a:Int)=>println(s"The number is $a")
saynum: Int => Unit = $$Lambda$1194/2122460177@5b810b54
```

Now, let's define a function func:

```
scala> for(i<-0 to 2){
| saynum(nums(i))
| }
```

The number is 1

The number is 2

The number is 4

b. ArrayBuffer

Now let's try this with an ArrayBuffer. First, we'll need to import ArrayBuffer.

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer
```

Now, we declare an ArrayBuffer with three strings:

```
scala> val colors=ArrayBuffer("Red","Green","Blue")
colors: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer(Red, Green, Blue)
```

Finally, the function func to call sayhello on colors:

```
scala> val func=(f:String=>Unit,s:String)=>f(s)
func: (String => Unit, String) => Unit = $$Lambda$1270/151863667@5d16f27b
```

Now, let's use a for-loop to call this on each value in the ArrayBuffer.

```
scala> for(i<-0 to 2){
| func(sayhello,colors(i))
| }
```

I am Red and I am 22

I am Green and I am 22

I am Blue and I am 22

This was all on Scala Closures. Hope you like the article on closures in Scala.

7. Conclusion

Does this give Scala a plus over other languages? What do you think? Let us know in the comments.

Scala Option – 13 Simple Methods to Call Option in Scala

1. Objective

In this **Scala tutorial**, we are going to learn about what is Scala Option. Moreover, we will see Scala Option getOrElse() Method, and Scala isEmpty() Method. Along with this we will discuss several Methods to Call on an Option in Scala: def get: A,
def isEmpty: Boolean, def productArity: Int, def productElement(n: Int): Any, def exists(p: (A) => Boolean): Boolean, def orNull etc.

So, let's explore Scala Option.

Scala Option – 13 Simple Methods to Call Option in Scala

2. Scala Option

A Scala Option holds zero or one element of a type. This means that it is either a Some[T] or a none object. One place we get an Option value is through the get() method for a Map.

Let's Learn Scala Map with Examples Quickly & Effectively

We have a Map m here:

```
scala> m
res109: scala.collection.immutable.Map[String,Int] = Map(Ayushi -> 0, Megha -> 1, Ruchi -> 2)
```

```
scala> m.get("Megha")
res110: Option[Int] = Some(1)
scala> m.get("Fluffy")
res111: Option[Int] = None
```

Here, it returns Some(1) when it finds the key "Megha" in the Map (where 1 is the value for that key). And when it doesn't find the key "Fluffy" in there, it returns None, stating that it couldn't find the key. This is like java.util.HashMap in **Java**.

We can also implement a pattern match here.

```
scala> def show(x:Option[Int])=x match{
| case Some(i)=>i
| case None=>"?"
| }
show: (x: Option[Int])Any
scala> show(m.get("Ayushi"))
res113: Any = 0
scala> show(m.get("Fluffy"))
res114: Any = ?
```

Learn: Scala Closures with Examples | See What is Behind the Magic

3. getOrElse() Method

This is like get(), except it will give us the default when no value exists. Let's take two values 'a' and 'b'.

```
scala> val a:Option[Int]=Some(5)
a: Option[Int] = Some(5)
scala> val b:Option[Int]=None
b: Option[Int] = None
And now, the paydirt.
scala> a.getOrElse(1)
res0: Int = 5
```

This checks if 'a' has a value. Since it does, this returns that value, which is 5.

```
scala> b.getOrElse(7)
res1: Int = 7
```

Now here, since 'b' has no value, this returns the default, which is 7, as we state.

4. Scala isEmpty() Method

If the Scala Option is None, this returns true; otherwise, it returns false.

```
scala> a.isEmpty
res2: Boolean = false
scala> b.isEmpty
res3: Boolean = true
```

Learn: Scala Arrays and Multidimensional Arrays in Scala

5. Methods to Call on an Option in Scala

These are some of the methods you will commonly used in Scala.

a. def get: A

This will return the Option's value.

```
scala> a.get
res5: Int = 5
scala> b.get
java.util.NoSuchElementException: None.get
at scala.None$.get(Option.scala:349)
at scala.None$.get(Option.scala:347)
... 28 elided
```

b. def isEmpty: Boolean

If the Option is None, this returns true. Otherwise, it returns true.

```
scala> a.isEmpty
res7: Boolean = false
scala> b.isEmpty
res8: Boolean = true
```

c. def productArity: Int

This Scala Option returns the size of the product.

```
scala> a.productArity
res9: Int = 1
scala> b.productArity
res10: Int = 0
```

A product A(x_1, …, x_k) has size k

Read Scala String: Creating String, Concatenation, String Length in detail

d. def productElement(n: Int): Any

This returns the n-th element of product, where indexing begins at 0.

```
scala> a.productElement(0)
res11: Any = 5
scala> a.productElement(1)
java.lang.IndexOutOfBoundsException: 1
```

```
at scala.Some.productElement(Option.scala:333)
... 28 elided
scala> b.productElement(0)
java.lang.IndexOutOfBoundsException: 0
at scala.None$.productElement(Option.scala:347)
... 28 elided
```

e. def exists(p: (A) => Boolean): Boolean

If the Option has a value, and this value satisfies the predicate, this returns true. Otherwise, this returns false.

```
scala> a.exists(x=>{x%2!=0})
res14: Boolean = true
```

f. def filter(p: (A) => Boolean): Option[A]

If the Scala Option's value satisfies the predicate, this returns that value.

```
scala> a.filter(x=>{x%2!=0})
res15: Option[Int] = Some(5)
```

g. def filterNot(p: (A) => Boolean): Option[A]

If the Option's value does not satisfy the predicate, this returns that value.

```
scala> a.filterNot(x=>{x%2!=0})
res16: Option[Int] = None
scala> a.filterNot(x=>{x%2==0})
res17: Option[Int] = Some(5)
```

h. def flatMap[B](f: (A) => Option[B]): Option[B]

If the Option has a value, this applies the function to that value, and then returns it.

Do you know about Scala Arrays and Multidimensional Arrays

i. def foreach[U](f: (A) => U): Unit

It applies the procedure to the Option's value and returns it. If the Option is None, this does nothing.

j. def getOrElse[B >: A](default: => B): B

If the Option has a value, this returns it; otherwise, returns the default value.

```
scala> a.getOrElse(7)
res3: Int = 5
```

```
scala> b.getOrElse(7)
res4: Int = 7
```

Learn: Scala Operator

k. def isDefined: Boolean

If the Scala Option is an instance of Some, this returns true; otherwise, false.

```
scala> a.isDefined
res5: Boolean = true
scala> b.isDefined
res6: Boolean = false
```

l. def iterator: Iterator[A]

This returns an iterator on the Option.

```
scala> a.iterator
res8: Iterator[Int] = non-empty iterator
scala> for(i<-a.iterator){println(i)}
```

5

m. def map[B](f: (A) => B): Option[B]

If the Scala Option has a value, it applies this function to it, and then returns it.

```
scala> a.map(x=>{x*x})
res10: Option[Int] = Some(25)
```

n. def orElse[B >: A](alternative: => Option[B]): Option[B]

If the Option has a value, this returns that. Otherwise, this evaluates the alternative and returns it.

Must Read about Scala Iterator in Detail

o. def orNull

If the Option has a value, this returns it. Otherwise, this returns Null.

so, this was all on Scala Option. Hope you like our explanation.

6. Conclusion

Hence, we studied Scala Option. In addition, we discussed Scala Option getOrElse() Method and Scala isEmpty() Method. At last, we saw several Methods to Call on an Option in Scala.

Scala Final – Variable, Method & Class | Scala This

1. Objective

In this **Scala Programming tutorial**, we will talk about two keywords called Scala This and Scala Final. Along with this, we will also discuss an example of Scala Final & Scala This. In Scala Final, we will discuss Scala Final Variables, Methods, and Class with their examples.

Scala Final – Variable, Method & Class | Scala This

2. What is Scala This & Scala Final

In Scala, the 'this' keyword lets us refer to the current object. 'final' prevents a class from deriving members from its superclass. These can be variables, methods, and classes. Let's first learn about 'this'.

Let's discuss Scala Variables with Examples

a. Scala This

When we want to refer to the current object for a class, we use the 'this' keyword. Then using the dot operator, we can refer to instance variables, and the class' methods and constructors. Let's take an example.

Example. 1 Scala This-

Let's see how ' Scala This' works. In the following example, we use 'Scala This' to call the primary constructor and the instance variables.

```
scala> class Marks{
| var midterm=0
| var finals=0
| def this(midterm:Int,finals:Int){
| this()
| this.midterm=midterm
| this.finals=finals | }
| def show(){
| println("You've got "+midterm+" in midterms and "+finals+" in finals")
| }
```

```
| }
defined class Marks
scala> var chemistry=new Marks(96,99)
chemistry: Marks = Marks@7e3ee128
scala> chemistry.show()
```

You've got 96 in midterms and 99 in finals.

Must read about Scala Case Class | Create Scala Object

Example. 2 Scala This-

```
scala> class Calculator(a:Int){
| def this(a:Int,b:Int){
| this(a)
| println(a+"+"+b+"="+{+b})
| }
| }
defined class Calculator
scala> var c=new Calculator(3,4)
3+4=7
c: Calculator = Calculator@3723de42
```

Here, we call one constructor from another. This call statement has to be the first statement in this body.

Example. 3 Scala This-

```
scala> class Calculator(){
| def this(a:Int){
| this()
| println(a)
| }
| }
defined class Calculator
scala> var c=new Calculator(3)
3
c: Calculator = Calculator@4e8036bb
scala> var d=new Calculator()
d: Calculator = Calculator@45c3cf0c
```

Let's explore Scala String Method with Syntax and Method

b. Scala Final

When we don't want a class to be able to inherit a member from its superclass, we declare that member final. This member may be a variable, a method, or even a class.

i. Scala Final Variables

When a variable is 'final', we cannot override it in the subclass.

```
scala> class Super{
| final var age=18
| }
defined class Super
```

```
scala> class Sub extends Super{
| override var age=21
| def show(){
| println("age="+age)
| }
| }
<console>:13: error: overriding variable age in class Super of type Int;
variable age cannot override final member
override var age=21
^
```

As you can see, Scala threw an error when we tried to override the final variable.

Let's Discuss Scala Access Modifiers: Public, Private and Protected Members

ii. Scala Final Methods

Like the Scala final variable, a final method won't let you override it in the subclass. Let's watch.

```
scala> class Super{
| final def show(){
| println("Hello")
| }
| }
defined class Super
scala> class Sub extends Super{
| override def show(){
| println("Hi")
| }
| }
<console>:13: error: overriding method show in class Super of type ()Unit;
method show cannot override final member
override def show(){
^
```

Learn Scala Closures with Examples

iii. Scala Final Classes

Finally, when we don't want a class to be inheritable, we declare it 'final'. This means another class cannot extend it.

```
scala> final class Super{
| def show(){
| println("Hello")
| }
| }
defined class Super
scala> class Sub extends Super{
| def sayhi(){
| println("Hi")
| }
| }
<console>:12: error: illegal inheritance from final class Super
```

```
13. class Sub extends Super{
14. ^
```

Let's explore Scala Functions: Quick and Easy Tutorial

So, this was all about Scala Final tutorial. Hope you like our Explanation on Scala This.

3. Conclusion

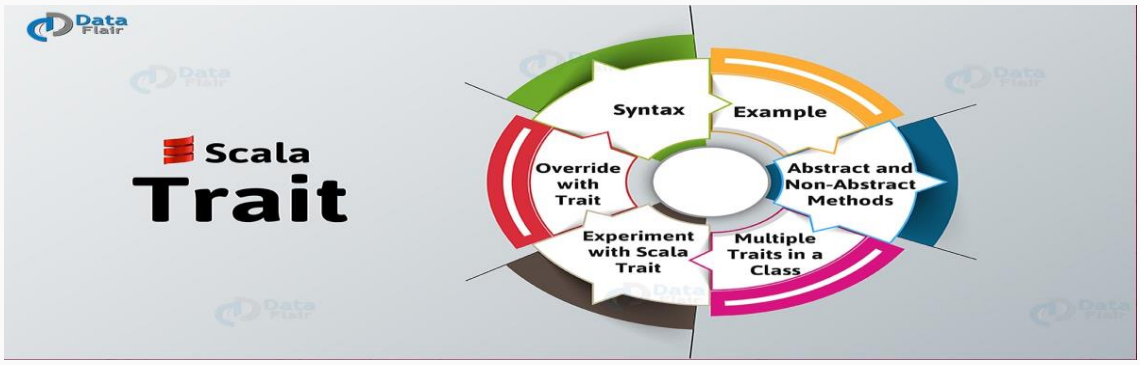
Hence, in this tutorial, we studied "Scala This" lets us refer to the current object and "Scala Final" stops us from overriding a variable or a method and inheriting a class. In addition, we talked about Scala Final Variables, Methods & Classes with their examples. Drop your queries in the comments.

Scala Trait – Multiple Class and Examples with New Experiments

1. Objective

In our last Scala Tutorial, we studied **Scala Annotations** and in this Scala Trait tutorial, we will learn about what is a trait in **Scala Programming Language**. Moreover, we will discuss syntax and examples of Scala traits. Along with this, we will look at Scala traits with abstract and non-abstract methods. At last, we will cover override with a trait and multiple traits in a Class.

So, let's start Scala Trait.



Scala Trait – Multiple Class and Examples with New Experiments

2. Introduction to Scala Trait

Traits in Scala are like partially implemented interfaces. It may contain abstract and non-abstract methods. It may be that all methods are abstract, but it should have at least one abstract method. Not only are they similar to **Java interfaces**, Scala compiles them into those with corresponding implementation classes holding any methods implemented in the traits.

You can say that using Scala trait, we can share interfaces and fields between classes. Within Scala trait, we can declare **variables** and values. When we do not initialize them, they are abstract. In other cases, the implementing class for the trait internally implements them.

Finally, we can never instantiate Scala trait, and it has no parameters, but we can have classes and objects extend it.

Let's Revise Scala Case Class and Create Scala Object in Detail

3. A Syntax of Scala Trait

To define Scala trait, we use the 'trait' keyword:

```
scala> trait Footwear
```

```
defined trait Footwear
```

We can also use Scala traits with abstract methods and as generic types:

```
scala> trait Iterator[A]{
| def hasNext:Boolean
| def next():A
|}
```

```
defined trait Iterator
```

Now, let's take an example of a simple trait.

4. Scala Trait Example

```
scala> trait Greeting{
| def greet()
|}
defined trait Greeting
scala> class Rendezvous extends Greeting{
| def greet(){
| println("Hello")
|}
|}
defined class Rendezvous
scala> var r=new Rendezvous()
r: Rendezvous = Rendezvous@4cf5d999
scala> r.greet()
```

```
Hello
```

This was a simple trait with one abstract method. Now, let's take another example.

Read About Scala Final – Variable, Method & Class | Scala This

5. With Abstract and Non-Abstract Methods

Here, we will see the example of Scala trait with the abstract and non-abstract method.

```
scala> trait Greeting{
| def greet()
| def show(){ //non-abstract
| println("This is the out method")
| }
| }
defined trait Greeting
scala> class Rendezvous extends Greeting{
| def greet(){
| println("Hello")
| }
| }
defined class Rendezvous
scala> var r=new Rendezvous()
r: Rendezvous = Rendezvous@4c6fe482
scala> r.greet()
```

Hello

```
scala> r.show()
```

This is the out method

While we cannot extend multiple abstract classes, we can extend multiple traits.

6. Multiple Traits in a Class

For a class extending multiple traits in Scala, we use 'extends' for one trait and 'with' for the rest. Hence, using traits, we can achieve multiple **inheritances**.

```
scala> trait A{
| def showA()
| }
defined trait A
scala> trait B{
| def showB()
| }
defined trait B
scala> class MyClass extends A with B{
| def showA(){
| print("A")
| }
| def showB(){
| print("B")
| }
```

```
| }
defined class MyClass
scala> var m=new MyClass()
m: MyClass = MyClass@3a4181ba
scala> m.showA()
A
Scala> m.showB()
B
```

Let's Explore Scala Exceptions and Exception Handling

7. An Experiment with Scala Trait

What happens when we do not implement all members of Scala trait in a class that extends it? Let's find out.

```
scala> trait Greeting{Scala Trait Mixins
Scala Trait Mixins
| def greet()
| }
defined trait Greeting
scala> class Rendezvous extends Greeting{
| def show(){
| println("This is a rendezvous")
| }
| }
<console>:12: error: class Rendezvous needs to be abstract, since method greet in trait Greeting of type
()Unit is not defined
class Rendezvous extends Greeting{
^
```

Since we did not implement greet() from the trait Greeting, we must declare it abstract. This works:

```
scala> trait Greeting{
| def greet()
| }
defined trait Greeting
scala> abstract class Rendezvous extends Greeting{
| def show(){
| println("This is a rendezvous")
| }
| }
defined class Rendezvous
```

Do You Know About Scala List with Examples

8. Override with Scala Trait

To implement abstract members of a trait, we can use the 'override' keyword.

```
scala> trait Iterator[A]{
| def hasNext:Boolean
| def next():A
```

```
4.  | }
5.  defined trait Iterator
6.  scala> class IntIterator(to:Int) extends Iterator[Int]{
7.  | private var current=0
8.  | override def hasNext:Boolean=current<to
9.  | override def next():Int={
10. | if(hasNext){
11. | val t=current
12. | current+=1
13. | t
14. | } else 0
15. | }
16. | }
17. defined class IntIterator
18. scala> val iterator=new IntIterator(10)
19. iterator: IntIterator = IntIterator@5b2728db
20. scala> iterator.next()
21. res6: Int = 0
22. scala> iterator.next()
23. res7: Int = 1
```

So, this was all about Scala Trait Tutorial. Hope you like our explanation.

Read About Scala Method Overloading with Example

9. Conclusion

Hence, in this article, we can say that Scala trait is a partially implemented interface. We can use it to share fields and interfaces between classes. In addition, we got know about example and syntax of Scala traits and experiment with Scala trait. In conclusion, we cover the multiple Scala traits in a class and use abstract and non-abstract methods with scala trait. In our next tutorial, we will study **Scala Trait Mixins** in detail. Furthermore, if you have any query, feel free to ask in the comment section.

What is Scala Trait Mixins – Top Examples of Scala Mixins

1. Objective

Before we discuss Scala Trait Mixins, we revise **Scala Trait**. Here, we will discuss what is Scala Mixins Trait and examples of Scala Mixins.

So, let's start Scala Trait Mixins.

What is Scala Trait Mixins – Top Examples of Scala Mixins

2. Scala Trait Mixins

It is possible to extend any number of Scala traits with a class or with an abstract class. We can extend traits, combinations of traits and classes, or combinations of traits and abstract classes. But to avoid an error from the compiler, we must maintain an order of the mixins.

Let's discuss Scala Inheritance – Syntax, Example & Types of Inheritance in Scala

3. Example of Scala Mixins

Here, we are going to study 3 different examples of Scala Trait Mixins, cover all kind of conditions, so let's discuss them one by one:

a. Example – 1

Let's try extending a trait and an abstract class together. As seen before, we use the 'with' keyword with this.

```
scala> trait Greeting{
| def greet()
| }
defined trait Greeting
scala> abstract class Welcome{
| def welcome()
| }
defined class Welcome
```

```
scala> class MyClass extends Greeting with Welcome{
| def greet(){
| println("Hello")
| }
| def welcome(){
| println("Welcome")
| }
| }
<console>:14: error: class Welcome needs to be a trait to be mixed in
class MyClass extends Greeting with Welcome{
^
```

Since we do not maintain the order of the Scala mixins, the compiler throws us an error.

Do You Know About Scala Exceptions and Exception Handling

b. Example – 2

You should first extend any class or abstract class, and then extend any existing Scala traits.

```
scala> trait Greeting{
| def greet()
| }
defined trait Greeting
scala> abstract class Welcome{
| def welcome()
| }
defined class Welcome
scala> class MyClass extends Welcome with Greeting{
| def greet(){
| println("Hello")
| }
| def welcome(){
| println("Welcome")
| }
| }
defined class MyClass
scala> var m=new MyClass()
m: MyClass = MyClass@47f71f50
scala> m.greet()
Hello
scala> m.welcome()
Welcome
```

Let's learn Scala Case Class and to Create Scala Object

c. Example – 3

Let's try extending only Scala trait.

```
scala> trait Greeting{
| def greet()
| }
defined trait Greeting
```

```
scala> abstract class Welcome{
| def welcome()
| }
defined class Welcome
scala> class MyClass extends Welcome{
| def greet(){
| println("Hello")
| }
| def welcome(){
| println("Welcome")
| }
| }
defined class MyClass
scala> var m=new MyClass() with Greeting
m: MyClass with Greeting = $anon$1@43cb7e55
scala> m.greet()
Hello
scala> m.welcome()
Welcome
```

As you can see, here, we use the 'with' keyword while creating the object.

So, this was all about Scala Trait Mixins. Hope you like our explanation.

4. Conclusion

Hence, in this Scala Trait Mixins Tutorial, we study what is Scala Mixins Traits with examples. Furthermore, if have any query regarding Scala Trait Mixins, feel free to ask in the comment section.

Scala Regex | Scala Regular Expressions – Replacing Matches

1. Objective

In our last tutorial, we studied **Scala Trait_Mixins**. Today, we are going to discuss Scala Regular Expressions or in general terms, we call it Scala Regex. In this Scala Regex cheatsheet, we will learn syntax and example of Scala Regex, also how to Replace Matches and Search for Groups of Scala Regex.

So, let's begin Scala Regular Expression (Regex).

Scala Regex Tutorial – What is Scala Regular Expression

2. What is Scala Regex?

We now move on to regular expressions. Using certain strings, we can find patterns and lack of patterns in data.' To convert a string into a regular

expression, use the .r method. We import the Regex class from the package scala.util.matching.Regex.

Let's Discuss Scala Closures with Examples

```
scala> val word="portmanteau".r
word: scala.util.matching.Regex = portmanteau
scala> val str=" Scala is a portmanteau of scalable and language"
str: String = Scala is a portmanteau of scalable and language
scala> word findFirstIn str
res9: Option[String] = Some(portmanteau)
```

Here, Scala converts the String to a RichString and invokes r() for an instance of Regex. We define a word and a string to search in. Then, we call the method findFirstIn() to find the first match. For a string with multiple occurrences, we can use findAllIn() to find all occurrences.

3. Scala Regular Expressions Example

Let's take another example of Scala Regex.

```
scala> import scala.util.matching.Regex
import scala.util.matching.Regex
scala> val word=new Regex("(i|I)ntern")
word: scala.util.matching.Regex = (i|I)ntern
scala> val str="An intern on the Internet"
str: String = An intern on the Internet
scala> (word findAllIn str).mkString(", ")
res10: String = intern, Intern
```

Here, we use the Regex() constructor to create the pattern, the method findAllIn() to find all occurrences of the word, and a pipe(|) to allow any case of a certain letter.

4. How to Replace Matches in Scala Regex

To replace a first match, we use the method replaceFirstIn(). To replace all matches, we use replaceAllIn().

```
scala> val word="Python".r
word: scala.util.matching.Regex = Python
scala> val str="I'm doing Python"
str: String = I'm doing Python
scala> word replaceFirstIn(str,"Scala")
res11: String = I'm doing Scala
```

5. Searching for Groups of Scala Regex

We can use parentheses to search for groups of regular expressions in Scala.

```
scala> val pattern="([0-9a-zA-Z-#() ]+): ([0-9a-zA-Z-#() ]+)".r
pattern: scala.util.matching.Regex = ([0-9a-zA-Z-#() ]+): ([0-9a-zA-Z-#() ]+)
scala> val str=
| """background-color: #A03300;
| background-image: url(img/header100.png);
| background-position: top center;
| background-repeat: repeat-x;
| background-size: 2160px 108px;
| margin: 0;
| height: 108px;
| width: 100%;""".stripMargin
str: String =
background-color: #A03300;
background-image: url(img/header100.png);
background-position: top center;
background-repeat: repeat-x;
background-size: 2160px 108px;
margin: 0;
height: 108px;
width: 100%;
scala> for(patternMatch<-pattern.findAllMatchIn(str))
| println(s"key:${patternMatch.group(1)} value:${patternMatch.group(2)}")
key:background-color value:#A03300
key:background-image value:url(img
key:background-position value:top center
key:background-repeat value:repeat-x
key:background-size value:2160px 108px
key:margin value:0
key:height value:108px
key:width value:100
```

Let's study Inheritance in Java Programming Language

6. The syntax of Scala Regex

Where Java inherits many features from Perl, Scala inherits the syntax of Scala regular expressions from Java. Here's the list of metacharacter syntax:

Subexpression	Matches
^	Beginning of line
$	End of line

.	Single character; with option m, matches newline too
[…]	Single character in square brackets
[^…]	Single character not in square brackets
\\A	Beginning of string
\\z	End of string
\\Z	End of string except allowable final line terminator
re*	Zero or more occurrences of preceding expression
re+	One or more occurrences of preceding expression
re?	Zero or one occurrences of preceding expression
re{ n}	n occurrences of preceding expression
re{ n,}	n or more occurrences of preceding expression

re{ n, m}	At least n, at most m occurrences of preceding expression
a|b	Either a or b
(re)	Groups regular expressions; remembers matched text
(?: re)Scala Regex	Groups regular expressions; doesn’t remember matched text
(?> re)	Independent pattern; doesn’t backtrack
\\w	Word characters
\\W	Nonword characters
\\s	Whitespace; equivalent to [\t\n\r\f]
\\S	Nonwhitespace
\\d	Digits; equivalent to [0-9]
\\D	Nondigits
\\A	Beginning of string
\\Z	End of string; matches till before newline, if any

\\z	End of string
\\G	Point where last match ended
\\n	Beck-reference to capture group number ‘n’
\\b	Word boundaries when outside brackets; backspace (0x08) when inside brackets
\\B	Nonword boundaries
\\n, \\t,etc.	Newlines, carriage returns, tabs, and so
\\Q	Escape (quote) all characters up to \\E
\\E	Ends quoting begun with \\Q

Let’s Learn Scala Inheritance – Syntax, Example & Types in Detail

So, this was all about Scala Regex Tutorial. Hope you like our explanation of Scala Regular expression.

7. Conclusion

Hence, in this Scala regex cheat sheet, we studied what is Scala Regular expression. In addition, we saw example & syntax of Scala Regex and how to Replace Matches and Search for Groups of Scala Regex. Furthermore, if you have a query regarding Scala Regex, feel free to ask in the comment box.

Partial Functions in Scala – A Comprehensive Guide

1. Objective

This tutorial on Partial functions in Scala will help in learning the different types of functions in Scala, basics of scala partial functions and different ways to define it while doing Scala programming. These Scala functions would help programmers to do programming in Scala in more effective manner.

So are you all ready to learn Scala functions?

Let us brush our Scala concepts and Scala features before going ahead.

2. Introduction to Partial Functions in Scala

Most of the functions that we study fall under the category of **Total function** which means the

function properly supports every possible value that meets the type of the input parameters. A simple function like def double(x: Int) = x*2 can be considered a total function as there is no input x that the double() function could not process.

However there are some functions that do not support every possible value that meets the input type. For example, a function that returns the square root of the input number will not work if the input number is negative. Similarly a division function with 0 in denominator isn't applicable. Such functions are called partial functions because they can only partially apply to their input data.

Although this particular situation can be handled by catching and throwing an exception, Scala programming lets you define the divide function as a Partial Function. When doing so, you can explicitly state that the function is defined when the input parameter is not zero.

Example:

```
1. val divide = new PartialFunction[Int, Int] {
2. def apply(x: Int) = 42 / x
3. def isDefinedAt(x: Int) = x != 0
4. }
```

This approach provides you with many Scala function flexibilities. One thing you can do is test the function before you attempt to use it:

```
scala > divide.isDefinedAt(1)
res0: Boolean = true
scala > if (divide.isDefinedAt(1)) divide(1)
res1: AnyVal = 42
scala > divide.isDefinedAt(0)
res2: Boolean = false
```

3. Ways to define Scala Partial Functions

Scala differs with java in many ways. There are many features that creates difference between Scala and Java. Partial Functions in Scala can be defined in different ways as below using Scala control structures:

a. Using case statements

You can define partial function as case statements in Scala. Let us understand this with example.

Example:

```
val divide2: PartialFunction[Int, Int] = {
case d: Int if d != 0 => 42 / d
}
```

b. Using Else, orElse

Do you know what is the best part of partial function in Scala programming?

A terrific feature of partial functions is that you can chain them together. For Example – one method may only work with odd numbers and another method may only work with even numbers. Together they can be used to solve all integer problems.

In below mentioned example, two functions are defined that can each handle a small number of Int inputs and convert them to String results:

// converts 1 to “one”, etc., up to 5

```
val convert1to5 = new PartialFunction[Int, String] {
val nums = Array("one", "two", "three", "four", "five")
def apply(i: Int) = nums(i-1)
def isDefinedAt(i: Int) = i > 0 && i < 6
}
```

// converts 6 to “six”, etc., up to 10

```
val convert6to10 = new PartialFunction[Int, String] {
val nums = Array("six", "seven", "eight", "nine", "ten")
```

```
3.  def apply(i: Int) = nums(i-6)
4.  def isDefinedAt(i: Int) = i > 5 && i < 11
5.  }
```

Taken separately, they can each handle only five numbers. But combined with orElse, they can handle ten numbers as shown below:

```
1.  scala > val handle1to10 = convert1to5 orElse convert6to10
2.  handle1to10: PartialFunction[Int,String] = < function1 >
```

It can be used at shown below:

```
1.  scala > handle1to10(3)
2.  res0: String = three
```

```
1.  scala > handle1to10(8)
2.  res1: String = eight
```

c. Using Collect method

You can run partial functions with the collect method on collections' classes. The collect method takes a partial function as input, and builds a new collection by applying a partial function to all elements of this list on which the function is defined.

Example:

The divide function shown earlier is a partial function that is not defined at the Int value zero. Here's that function again:

```
1.  val divide: PartialFunction[Int, Int] = {
2.  case d: Int if d != 0 = > 42 / d
3.  }
```

However, if you use the same function with the collect method, it works fine:

```
1.  scala > List(0,1,2) collect { divide }
2.  res0: List[Int] = List(42, 21)
```

This is because the collect method is written to test the isDefinedAt method for the elements given to it. So it doesn't run the divide algorithm when the input value is 0 (but does run it for every other element).

Example:

The first example shows how to create a list of even numbers by defining a PartialFunction named isEven and using that function with the collect method:

```
1.  scala> val sample = 1 to 5
2.  sample: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
3.  scala> val isEven: PartialFunction[Int, String] = {
4.  case x if x % 2 == 0 => x + " is even"
5.  }
6.  isEven: PartialFunction[Int,String] = <function1>
7.  scala> val evenNumbers = sample collect isEven
8.  evenNumbers: scala.collection.immutable.IndexedSeq[String] =
9.  Vector(2 is even, 4 is even)
```

Similarly, an isOdd function can be defined and orElse can be used to join these 2 functions to work with the map method:

```
scala> val isOdd: PartialFunction[Int, String] = {
case x if x % 2 == 1 => x + " is odd"
}
isOdd: PartialFunction[Int,String] = <function1>
scala> val numbers = sample map (isEven orElse isOdd)
numbers: scala.collection.immutable.IndexedSeq[String] =
Vector(1 is odd, 2 is even, 3 is odd, 4 is even, 5 is odd)
```

Scala Currying Function – Example & Partially Applied Function

1. Objective

Today, we will learn about Scala currying functions. Moreover, we will discuss advantages of currying in **Scala Programming Language** and how to call a scala currying function. Along with this, we will study Scala Currying vs partially applied functions. In addition, we will look at an example of Scala Currying.

So, let's see Currying in Scala.

Scala Currying: Calling Function & Example

2. What is Scala Currying?

Through Scala curry function, we can split a list with multiple parameters into a chain of functions-each with one parameter. This means we define them with more than one parameter list.

Follow this link to know about Scala String Method with Syntax and Method

Scala Currying Syntax:

We use the following syntax to carry out currying:

```
def multiply(a:Int)(b:Int)=a*b
```

Another way we can do this is:

```
def multiply(a:Int)=(b:Int)=>a*b
```

3. Calling Scala Function

To make a call to Scala function, then, we call it passing parameters in multiple lists:

multiply(3)(4)

4. Partially Applied Functions

An important concept to discuss here is partially applied functions. When we apply a function to some of its arguments, we have applied it partially. This returns a partially-applied function that we can use later. Let's take an example.

```
scala> def multiply(a:Int)(b:Int)(c:Int)=a*b*c
multiply: (a: Int)(b: Int)(c: Int)Int
scala> var mul=multiply(2)(3)(_)
mul: Int => Int = $$Lambda$1122/1609544540@4f92ded0
```

Here, the underscore is a placeholder for a missing value.

```
scala> mul(4)
res11: Int = 24
```

Let's Explore Scala Functions in detail

Well, this works too:

```
scala> var mul=multiply(2)(3)_
mul: Int => Int = $$Lambda$1173/256443308@207ceea4
scala> mul(4)
res12: Int = 24
```

5. Example of Scala Currying

Example – 1

Let's begin with Scala Currying example.

```
scala> class Add{
| def sum(a:Int)(b:Int)={
| a+b}
| }
defined class Add
scala> var a=new Add()
a: Add = Add@53cba89f
scala> a.sum(3)(4)
res4: Int = 7
```

Example – 2

Remember the other piece of syntax we looked at ?Let's try defining a function that way, but with three arguments.

```
scala> class Concatenate{
| def strcat(s1:String)(s2:String)=(s3:String)=>s1+s2+s3
| }
defined class Concatenate
scala> var c=new Concatenate()
c: Concatenate = Concatenate@5d55eb7a
scala> c.strcat("Hello")("World")("How are you?")
res7: String = HelloWorldHow are you?
```

Do you know about Scala Exceptions and Exception Handling

7. Advantages of Currying in Scala

Here, we will discuss advantages of Currying in Scala, let's discuss them:

1. One benefit is that Scala currying makes creating anonymous functions easier.

2. Scala Currying also makes it easier to pass around a function as a first-class object. You can keep applying parameters when you find them.

So, this was all about Currying Function in Scala. Hope you like our explanation.

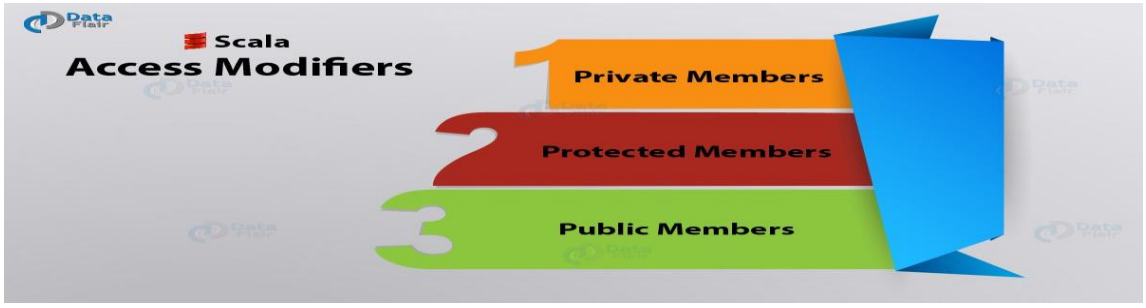
8. Conclusion

Hence, using the concept of partial functions, we use curry functions in Scala. This lets us split a list of multiple parameters into a chain of functions. Drop your queries in the comments.

Scala Access Modifiers: Public, Private and Protected Members

1. Access Modifiers in Scala

This article deals with private, protected, and public members of a class in Scala. Let's begin with the Scala Access Modifiers.



Scala Access Modifiers

2. Introduction to Scala Access Modifiers

Constructs like classes, objects, or packages can hold members like variables and methods/functions. We can declare these members to be private or protected. If we don't, Scala assumes them to be public. But what are these? These are Scala access modifiers; using these, we can restrict the members from being accessible to certain areas of the code. A private modifier to a member means that only the containing class and its objects will be able to access that member. Let's begin with this one.

3. Private Members

When we declare a member as private, we can only use it inside its defining class or through one of its objects. To declare a member privately, we use the modifier 'private':

```
1.  class Example {
2.  private var a:Int=7
3.  def show(){
4.  a=8
5.  println(a)
6.  }
7.  }
8.  object access extends App{
9.  var e=new Example()
10. e.show()
11. //e.a=8
12. //println(e.a)
13. }
```

Output: 8

In this code, we declare the variable 'a' to be private. This means that only the class Example can access it. To do this, we use the function show() to modify and access 'a'. The two lines that we've commented try to access 'a' outside of example. If we didn't comment them, we'd get the following two errors:

C:\Users\lifei\Desktop>scalac access.scala

access.scala:10: error: variable a in class Example cannot be accessed in Example

```
1.  e.a=8
2.  ^
3.  access.scala:11: error: variable a in class Example cannot be accessed in Example
4.  println(e.a)
5.  ^
```

two errors found

4. Protected Members

We can only access protected members from within a class, from within its immediate subclasses, and from within companion objects. We use the modifier 'protected' for this.

```
Class Example{
protected var a:Int=7
def show(){
a=8
println(a)
}
}
```

```
class Example1 extends Example{
def show1(){
a=9
println(a)
}
}
```

```
object access extends App{
var e=new Example()
e.show()
var e1=new Example1()
e1.show1()
//e.a=10
//println(e.a)
}
```

Output:

8

9

Here, because the class Example1 inherits from Example, it can access the variable ‘a’, and also modify it. The purpose for the comments is the same as in the previous case. Let’s see what happens when we modify the code just a bit:

```
class Example{
protected var a:Int=7
def show(){
println(a)
}
}
```

```
class Example1 extends Example {
def show1(){
a=9
println(a)
}
}
```

object access extends App

```
{
var e=new Example()
e.show()
var e1=new Example1()
e1.show1()
//e.a=10
//println(e.a)
```

```
8.  e1.show()
9.  }
```

Output:

7

9

9

Here, when we call show() on e1, it calls the inherited show() from Example. This simply prints out the value of ‘a’.

5. Public Members

All members are default by public. If we do not accompany them with the modifiers ‘private’ or ‘protected’, they’re public. We can access these anywhere.

```
1.  class Example {
2.  var a:Int=7
3.  }
4.  object access extends App{
5.  var e=new Example()
6.  e.a=8
7.  println(e.a)
8.  }
```

This prints 8.

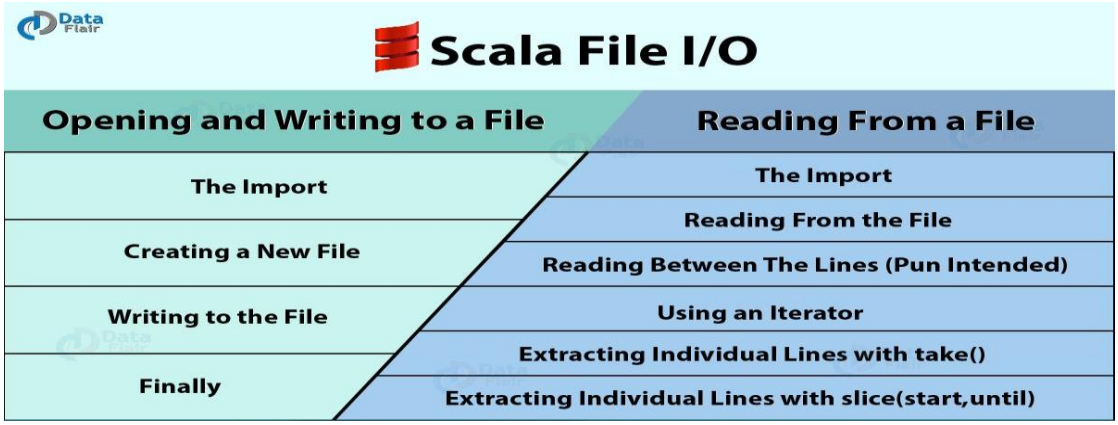
6. Conclusion

A member of a class may be public, private, or protected. For the latter two, we use the modifiers ‘private’ and ‘protected’. Keep practicing.

Scala File i/o: Open, Read and Write a File in Scala

1. Scala File i/o

In this tutorial Scala File io, we will learn how to Open, read and write files in Scala. I also recommend you to go through the **Scala Syntax** and **Scala Functions** Articles to clear your basics on Scala.



Scala File IO

2. Opening and Writing to a File

We don't have a class to write a file, in the Scala standard library, so we borrow java.io._ from Java. Or you could import java.io.File and java.io.PrintWriter.

a. The Import

```
C:\Users\lifei>cd Desktop
C:\Users\lifei\Desktop>scala
```

Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).

Type in expressions for evaluation. Or try :help.

```
scala> import java.io._
import java.io._
```

Learn: Scala Data Types with Examples

b. Creating a New File

To create a new file to write to, we create a new instance of PrintWriter, and pass a new File object to it.

```
scala> val writer=new PrintWriter(new File("demo1.txt"))
writer: java.io.PrintWriter = java.io.PrintWriter@31c7c281
```

c. Writing to the File

Now, to write to the file, we call the method write() on the object we created.

```
scala> writer.write("This is a demo")
```

d. Finally

At this point, nothing is really visible in the file. To see these changes reflect in the file demo1.txt, we need to close it with Scala.

```
scala> writer.close()
```

Now, when we check the file demo1.txt again, we find:

This is a demo

Together, the code looks something like this:

```
C:\Users\lifei>cd Desktop
C:\Users\lifei\Desktop>scala
```

Welcome to Scala 2.12.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).

Type in expressions for evaluation. Or try :help.

```
scala> import java.io._
import java.io._
scala> val writer=new PrintWriter(new File("demo1.txt"))
writer: java.io.PrintWriter = java.io.PrintWriter@31c7c281
scala> writer.write("This is a demo")
scala> writer.close()
```

Learn: **Scala Variables with Examples**

3. Reading From a File

Now Scala does provide a class to read files. This is the class Source. We use its companion object to read files. For this demonstration, we're going to read what we put in the file demo1.txt. Let's begin.

a. The Import

The class we need to import here is scala.io.Source.

```
scala> import scala.io.Source
import scala.io.Source
```

b. Reading From the File

To read the contents of this file, we call the fromFile() method of class Source- with the filename as argument. On this, we call the method mkString, like we've seen in different collections so far.

```
scala> Source.fromFile("demo1.txt").mkString
res4: String = This is a demo
```

c. Reading Between The Lines (Pun Intended)

To read individual lines instead of the whole file at once, we can use the getLines() method. For this, we change the contents of our file to this:

This is a demo

This is line 2

And this is line 3

The code we use is:

```
scala> Source.fromFile("demo1.txt").getLines.foreach{x=>println(x)}
```

This is a demo

This is line 2

And this is line 3

Learn: Scala String Interpolation

d. Using an Iterator

We can also make use of an iterator to get one line at a time:

```
scala> val it=Source.fromFile("demo1.txt").getLines()
it: Iterator[String] = non-empty iterator
scala> it.next()
res9: String = This is a demo
scala> it.next()
res10: String = This is line 2
scala> it.next()
res11: String = And this is line 3
scala> it.next()
```

```
java.util.NoSuchElementException: next on empty iterator
at scala.collection.Iterator$$anon$2.next(Iterator.scala:38)
at scala.collection.Iterator$$anon$2.next(Iterator.scala:36)
at scala.io.BufferedSource$BufferedLineIterator.next(BufferedSource.scala:79)
at scala.io.BufferedSource$BufferedLineIterator.next(BufferedSource.scala:64)
... 28 elided
```

e. Extracting Individual Lines with take()

When we talked iterators, we saw the use of the method take(n) to return the first n values from the iterator. This is what we're going to use here.

```
scala> val it=Source.fromFile("demo1.txt").getLines.take(2)
it: Iterator[String] = non-empty iterator
scala> while(it.hasNext){print(it.next())}
```

This is a demoThis is line 2

If we passed 1 to take(), it would only print “This is a demo”.

f. Extracting Individual Lines with slice(start,until)

The method slice(start,until) returns an iterator over lines *start* to *until*-1.

```
scala> val it=Source.fromFile("demo1.txt").getLines.slice(1,3)
it: Iterator[String] = non-empty iterator
scala> while(it.hasNext){print(it.next())}
```

This is line 2And this is line 3

This was all on Scala File io

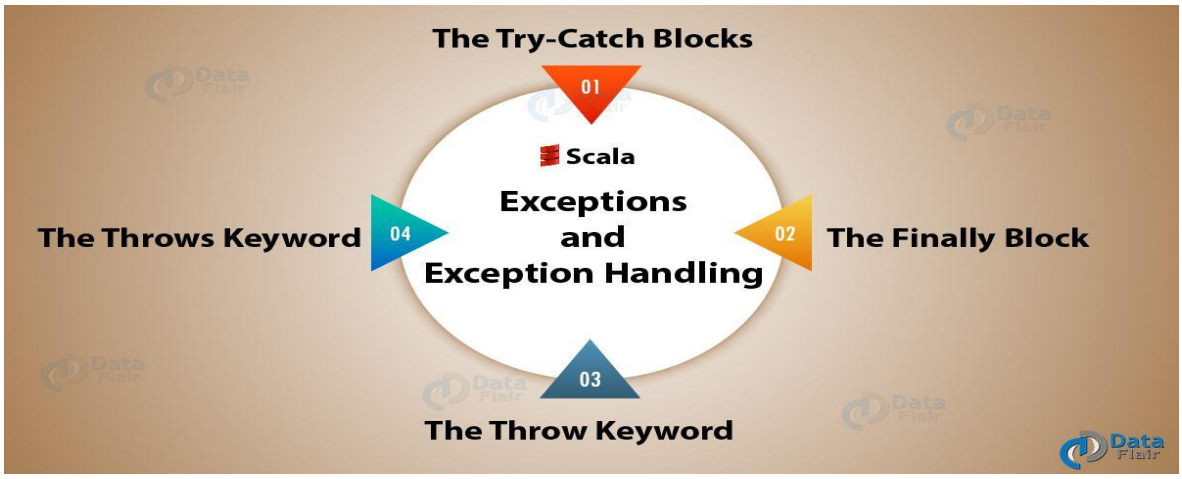
4. Conclusion

So this is how you read and write a file. In this article, we saw use of methods write(), close(), fromFile(), getLines(), take(), and slice().

Scala Exceptions and Exception Handling

1. Scala Exceptions and Scala Exception Handling

There may be situations your code may misfunction when you run it. These abnormal conditions may cause your program to terminate abruptly. Such runtime errors are called exceptions. We will discuss these errors in this tutorial on Scala Exception Handling and Scala Exceptions. We will also learn about the Try-Catch Blocks, The Throws Keyword, The Throw Keyword, and the Finally Block



Scala Exception Handling | Scala Exceptions

2. Introduction to Scala Exceptions

All Scala Exceptions are unchecked. This is unlike Java, where exceptions are either checked or unchecked. Here, even SQLException and IOException are unchecked. So, let's begin with the Exceptions in Scala. I also recommend you to go though our article on **Scala Job Opportunities** after reading this article.

3. Scala Exception Example

Take a look at the following Scala Exception example. We declare a function to divide two integers and return the result.

```
scala> def div(a:Int,b:Int):Float={
| a/b}
div: (a: Int, b: Int)Float
```

Now, let's call it.

```
scala> div(1,0)
java.lang.ArithmeticException: / by zero
at .div(<console>:12)
... 28 elided
```

As you can see, this raises an Arithmetic Exception.

Learn: Scala Environment Setup and get Started with IDE

4. The Try-Catch Blocks

When we suspect that a line or a block of code may raise an exception in Scala, we put it inside a try block. **What follows is a catch block.** We can make use of any number of Try-Catch Blocks in a program.

Here's how we would deal with the above situation:

```
scala> def div(a:Int,b:Int):Float={
| try{
| a/b
| }catch{
| case e:ArithmeticException=>println(e)
| }
| 0
| }
div: (a: Int, b: Int)Float
scala> div(1,0)
java.lang.ArithmeticException: / by zero
res1: Float = 0.0
```

a. **Another Example**

Let's take another example of Scala Exception Handling:

```
scala> def func(n:Int){
| try{
```

```
| print(1/n)
| var arr=Array(1,4)
| arr(17)
| }catch{
| case e:ArithmeticException=>println(e)
| case anon:Throwable=>println("Unknown exception: "+anon)
| }
| }
func: (n: Int)Unit
scala> func(1)
```

Unknown exception: java.lang.ArrayIndexOutOfBoundsException: 17

Throwable is a super class in the exception hierarchy. So, if we want our code to be able to handle any kind of exception, we use Throwable.

Learn: Scala Control Structures

5. The Finally Block

Imagine, if you're working with a resource and an exception occurs in the middle. You still haven't released the resource. This could be a file, a network connection, or even a database connection. To deal with such a situation, we have the Finally Block. We can release all resources in this block. Well, whether an exception happens in your code or not, the code under 'finally' will run, no matter what.

Let's take the example of our function for division here:

```
scala> def func(a:Int,b:Int):Float={
| try{
| a/b}
| catch{
| case e:ArithmeticException=>println(e)
| }
| finally{
| println("This will print no matter what")
| }
| 0
| }
func: (a: Int, b: Int)Float
scala> func(1,0)
java.lang.ArithmeticException: / by zero
```

This will print no matter what

res4: Float = 0.0

6. The Throw Keyword

We can also explicitly throw a Scala exception in our code. We use the Throw Keyword for this. Let's create a custom exception.

In Jewish culture, Bat Mitzvah is coming with age ceremony with age 12 for young boys.

```
scala> def batmitzvah(age:Int){
| if(age<12){
```

```
| throw new Exception("Sorry, you're ineligible yet")
| }
| else{ println("Eligible")}
| }
batmitzvah: (age: Int)Unit
scala> batmitzvah(10)
java.lang.Exception: Sorry, you're ineligible yet
at .batmitzvah(<console>:13)
... 28 elided
scala> batmitzvah(13)
```

Eligible

7. The Throws Keyword

When we know that certain code throws an exception in Scala, we can declare that to Scala. This helps the caller function handle and enclose this code in Try – Catch Blocks to deal with the situation. We can either use the throws keyword or the throws annotation.

```
@throws(classOf[NumberFormatException])
def validateit()={
"abcd".toInt
}
Here's another example:
@throws[IOException]("if the file doesn't exist")
def read() = in.read()
```

So, this was all about Scala Exceptions. Hope you like our explanation of Scala Exceptions.

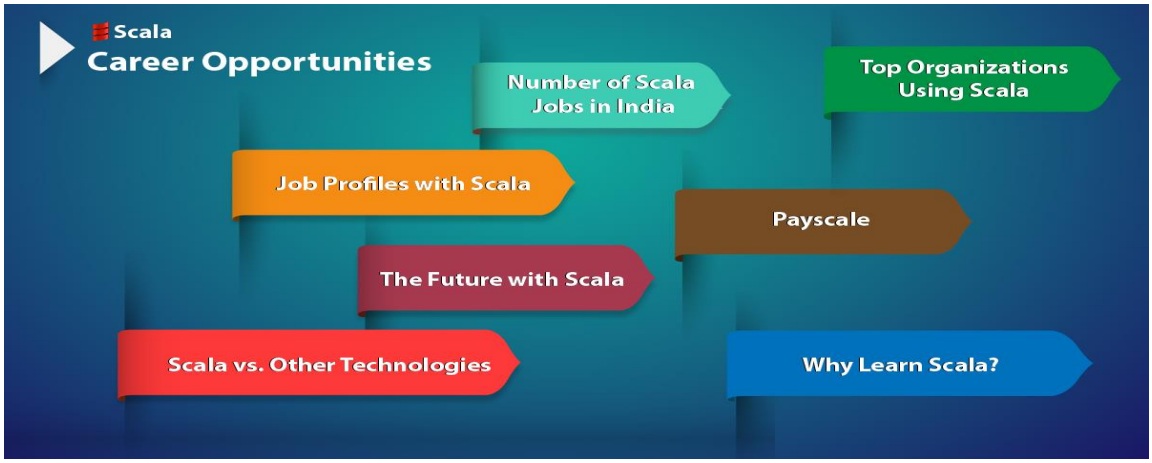
8. Conclusion

Now, after going through this Scala Exception Tutorial, you have the understanding of how to deal with your code and what it does when you run it. Stay tuned for more with Scala. Furthermore, if you have any query, feel free to ask in the comment Section.

Scala Job Opportunities 2018: Profile, Salary & Top Organizations

1. Scala Job Opportunities

If you put in time and effort to learn something, it should repay you, right? And Scala is a language that's a bit difficult to learn. So, why give in all the effort? There must be a very good reason. Let's see what Scala can do for us, what are the **Scala Job Opportunities** for you in future, what is the Scala programmer salary, work profile, top organizations using Scala, etc.



Scala Job Opportunities

2. Number of Scala Jobs in India

With one of the largest user bases, Scala is very popular. If you haven't learned it, chances are that you've at least heard of it. And since it runs on the JVM, it can run anywhere Java runs. This makes it highly flexible and functional. It also finds use in Machine Learning(ML) at large-scale, and helps build high-level algorithms.

While Scala is amply powerful, it is slightly difficult to learn. As a result, it is harder to find good talent in Scala compared to more popular technologies like Java. The demand is high, but the supply is low. You hear that? That's the scenario of Scala job opportunities in India .

Where indeed.com lays out about 888 listings for the keyword 'scala', Naukri.com shows 1729 listings all over India. LinkedIn displays 1314 results.

3. Job Profiles with Scala

What postings or profiles can you land when you're good with Scala?

a. Software Engineer

A software engineer deals with:

- Basic System Administration
- Executing full life-cycle software development
- Producing specifications and determining operational feasibility
- Design and implementation of news features for applications
- Testing system for bugs
- Fixing errors in existing features
- Maintenance support after deployment of application

b. Senior Software Engineer

A senior software engineer has the same roles and responsibilities as does a software engineer. The major difference is that a senior engineer has more experience and judgment. Consequently, they can build more complex systems, a broader range of systems, and, make architectural decisions. They may mentor and assist junior engineers.

c. Software Developer

A software developer must fulfil the following roles:

- Review current systems
- Present ideas for system improvements; propose costs
- Produce detailed specifications, write program codes
- Work closely with analysts and designers
- Prepare training manuals for users
- Testing the product in controlled, real situations before going live
- Maintain systems once up and running

d. Application Developer

An application developer has the following duties:

- Creating, maintaining, and implementing source code for an application
- Designing prototype application
- Indicate program unit structure
- Coordinate application plans with development team/client

e. IT Consultant

An IT consultant will:

- Meet clients; determine requirements
- Work with clients to define a project's scope
- Plan timescales and needed resources
- Clarify a client's system specifications, understand their work practices and their business' nature
- Travel to customer sites
- Liaise with staff at all levels of a client organization
- Define hardware, software, and network requirements
- Analyze IT requirements within companies
- Develop agreed solutions; implement new systems
- Present solutions as written or oral reports
- Help clients with change-management activities

- Purchase systems
- Design, test, install, and monitor new systems
- Prepare documentation; present progress reports to customers
- Organize training for users and other consultants
- Involve in sales and support; maintain contact with client organizations
- Identify potential clients; build and maintain contacts.

With so many profiles relating a technology, we cannot doubt the Scala job opportunities and future with it.

4. Scala vs. Other Technologies

For this, we'll look into an informal study performed by Tobias Hermann, aka Dobiasd. This considers more than 20 programming languages, and judges them of four criteria:

- Mutual mentions
- Cursing
- Happiness
- Word usage – abstract, category, pure, theory, and hardware

a. Mutual Mentions

This compares to the TIOBE Index's programming language value. So, what's a mutual mention? It is a mention of Scala that includes other programming languages.

Scala Job Opportunities: Comparison over other languages

As you can see in the chart, the most mutual mentions go to Java. Haskell, Python, and Clojure follow closely. Now, let's take a look at the TIOBE language ranking index, which uses Google/Bing index rankings.

Scala Career Opportunities

According to this, we believe that developers talk about Scala about 100x more than the relative TIOBE index ranking. Also, Haskell seems to be at over 140x.

b. Cursing

The four curse words we'll be using are- 'crap', 'fuck', 'hate', and 'shit'.

Scala Job and Careers

As you can see, Scala steps on the spectrum's lower half. This is about 1.15% cursing for around 98.85% of non-cursing. PHP, JavaScript, and Java top this list. This is almost equal to Mathematica, Visual Basic, Haskell, Rust, Clojure, and C combined.

c. Happiness

Coming to happiness, the words we'll consider are- 'awesome', 'cool', 'fun', 'happy', 'helpful', and 'interesting'.

Scala Job Opportunities

Here, functional languages like Scala, Lisp, Clojure, and Haskell make developers the happiest. Ruby and JavaScript rank high on both- happiness and curse. This negates it out.

d. Word Usage

Finally, we arrive at word usage. Here, we consider the words 'abstract', 'category', 'pure', and 'theory'.

Scala Careers

Scala sees one of the highest usages of these words, only falling second to Haskell. Of these, words 'abstract' and 'pure' fall higher than 'category' and 'theory'. And with usage of the word 'hardware', Scala lies as low as #6 on the list.

5. The Future with Scala

Before we can begin with what the future holds for Scala, let's take a look at its history.

Martin Odersky began working on Scala in 2001 at the Ecole Polytechnique Federale de Lausanne (EPFL). He released it to the public on Jan 20, 2004. The word Scala is a portmanteau of words 'scalable' and 'language'. Scalable, here, means that it can grow with user-demand. And while it isn't an extension of Java, it is interoperable with it as it runs on the JVM(Java Virtual Machine). Scala is both- object-oriented and functional; every value is an object and every function is a value. The latest version for Scala is the 2.13.x milestone, which released on Jan 31, 2018.

As we've often said, Java has adopted a few features from Scala, and it still does. Scala is still the first to put it on the plate. That said, with the number of companies using it, Scala has a bright future in our opinion. Other than that, nothing specific has been said on this.

6. Top Organizations Using Scala

If Scala is indeed a success, then it must mean that a lot of tech giants use it in their products, right? The following companies use Scala:

- LinkedIn
- Twitter
- Netflix
- Tumblr
- Sony
- Apple
- Foursquare

- The Guardian
- AirBnB
- Klout
- Precog
- Meetup.com
- AT&T
- eBay

In all, about 45 companies use Scala for their products and services. So we can say that there are Scala job opportunities in many top organization across the globe.

7. Scala Salary

Looking at payscale.com, we conclude that Scala developers indeed take home a hefty amount. These are the salaries of the following profiles:

a. Senior Software Engineer:

Minimum Salary: Rs. 452K

Maximum Salary: Rs. 2.9M

Average Salary: Rs. 1,218,943

b. Software Engineer

Minimum Salary: Rs. 296K

Maximum Salary: Rs. 1.8M

Average Salary: Rs. 919,253

c. Software Developer

Minimum Salary: Rs. 296K

Maximum Salary: Rs. 1M

Average Salary: Rs. 550,000

d. Senior Software Engineer/ Developer/ Programmer

Minimum Salary: Rs. 528K

Maximum Salary: Rs. 2.3M

Average Salary: Rs. 1,200,786

e. Software Engineer/ Developer/ Programmer

Minimum Salary: Rs. 240K

Maximum Salary: Rs. 550K

Average Salary: Rs. 400,333

f. Application Developer

Minimum Salary: Rs. 708K

Maximum Salary: Rs. 1.1M

Average Salary: Rs. 915,000

g. IT Consultant

Minimum Salary: Rs. 670K

Maximum Salary: Rs. 2.3M

Average Salary: Rs. 850,000

Good Scala Programmer Salaries another proof of good Scala job opportunities in India and Across the Globe.

8. Why Learn Scala?

What can you build with Scala? This is an important question as it determines your will to start with it. Let's see. With Scala, you can build:

- Android Applications
- Desktop Applications
- Concurrency and distributed data processing, for instance, Spark
- Front and back ends of web applications with scala.js
- Highly concurrent things, like messaging apps, with Akka
- Distributed computing; because of its concurrency capabilities
- Scala is used with Hadoop; Map/Reduce programs
- Big Data and data analysis with Apache Spark
- Data streaming with Akka
- Parallel batch processing
- AWS lambda expression
- Ad hoc scripting in REPL

9. Conclusion

Scala job opportunities definitely opens many doors for you in the IT industry. Also crediting to the lack of good, efficient Scala developers, we suggest you go ahead with Scala and build yourself a career worth living up to.

Scala Advantages & Disadvantages | Pros and Cons of Scala

1. Objective

In this **Scala Tutorial**, we will learn Scala Advantages & Disadvantages. A great power comes with knowledge of what you can or can't do with what you have. So, let's discuss the places where Scala wins and fails. But remember, overall, a language always wins.

Here, we start with the Scala benefits and limitations.

Scala Advantages & Disadvantages | Pros and Cons of Scala

2. Scala Advantages & Disadvantages

In this Scala Tutorial, we will explore benefits and limitations of Scala Programming Langauge.

A. Scala Advantages

What makes Scala special? What sets it apart from other languages? In Scala Advantages, we will explore the answers to these questions. So, let's discuss them one by one:

Scala Advantages

i. Easy to Pick Up

Coming from an **object-oriented background with Java** or another such language, **Scala's syntax** appears familiar. This makes it easier to pick up than languages like Haskell. Also, it is much more concise than Java. Where Java would need nine lines of code to perform an operation, Scala would only take about three, in most cases. This aids productivity.

ii. Pretty Good IDE Support

It is one of the most popular Scala Advantages. Many functional language communities usually favor editors like Emac or Vim. But contrary to what many developers think, Scala has a pretty good lineup of IDEs:

a. IntelliJ IDEA

Most developers consider this to be the best IDE for Scala. It has great UI, and the editor is pretty good. However, the Scala presentation compiler may unusually be slow to load an auto-complete list working with a relatively large project. This is because they rewrote the compiler to integrate with their platform. You will also need to purchase the ultimate edition if you want to be able to handle all your works with IntelliJ. It isn't completely free and open-source. The Ultimate version also provides support for Less/Sass.

To see how to set it up on your machine, **refer to Scala Environment Setup**.

Scala Advantages – IntelliJ IDEA

b. Scala IDE

The Scala IDE is based on Eclipse, as it says in the image. It provides more efficient support for Scala than does IntelliJ. It is able to handle big projects with accuracy and efficiency compared to other IDEs. Sometimes, you may have to deal with false positives, hiccups, and rare bugs. Other limitations include medieval block selection, and that changing the workspace loses the configuration.

Scala Advantages – Scala IDE

c. Emacs (ENSIME)

This provides the richest support for Scala in Emacs and is more modern than Vi. It supports a wide range of languages and frameworks, and it is possible to customize it and build our own version.

d. Atom (ENSIME)

While not as geeky as Emacs/Vi, it is a good editor built on top of web technologies. However, you may have to face more bugs than in Emacs.

Others include Vi, Sublime, and Visual Studio Code.

Let's Explore Scala vs Java – Feature wise Comparison Guide

iii. Scalability

'Scala' is a portmanteau of 'scalable' and 'language', as we've discussed. So, scalability is definitely one of biggest Scala advantages. This means we can use it to build highly

concurrent, fault-tolerant systems. For this, we can use the multiagent concurrency model like in Language Erlang in Akka.

iv. Mixins, Open Classes, and Monkey Patching

Some other facilities you will get with Scala are mixins(traits), open classes, and monkey patching. Using mixins, you can carry out multiple inheritances.

v. Great for Data Analytics

Scala can be a great choice for data analytics with support from tools like **Apache Spark**, among others. A lot of huge companies make use of Scala for their products and services.

Refer to Scala Career Opportunities for a brief list.

vi. Highly Functional

Scala is highly functional in paradigm, is one of the unique Scala advantages. It is a language that treats its functions as first-class citizens. This means we can say that it has first-class functions. In other words, it lets us pass functions as arguments to other functions, and to return them as values from other functions. It also lets them assign them to variables or store them in data structures. Hmm, this seems pretty much like everything is an **object in Python**. Ring a bell?

And if you face problems trying to solve one functionally, you can always switch to object-oriented. Like we've seen in collections like **Sets** and **Maps**, there is a mutable collections library to complement immutable collections in Scala.

vii. Inherently Immutable Objects

If you've worked with **Java**, you have faced quite some thread-safety concerns with your applications. Scala reduces these risks.

viii. Scala is Fun!

What is life without a challenge to beat, a purpose to pursue? Scala puts engineers face-to-face with challenges in an entertaining, yet teachable way.

And if you must encode documents in your products, Scala will let you do so use XML (Extensible Markup Language).

ix. Better Developers

Good Scala developers also do better with Java. They pick up programming practices like functional programming and can apply those to other languages like Java Programming Language.

Top 16+ Reasons to Learn Scala Programming Language

x. Better Quality Code

With functional programming, you are more likely to end up with fewer lines of code and fewer bugs. This means higher productivity and higher quality.

We complete all important Scala Advantages, let's move to Disadvantages of Scala Programming Language.

B. Scala Disadvantages

What keeps Scala from reaching the top? What issues could you face if you choose it as the language for your next project? The answers are a bit subtler:

1. Since with Scala, you can always switch back to an object-oriented paradigm. So, it doesn't force you to think functionally. You can think of this as a double-edged sword.

2. Being a hybrid of functional and object-oriented can sometimes make type-information a bit harder to understand.

3. Since it runs on the JVM, it has no true tail-recursive optimization. As a workaround, you can use the @tailrec annotation for partial benefits.

4. Scala has a limited developer pool. But while it is easier to find Java developers in numbers, not every Java developer has what it takes to code efficiently in Scala.

Whether to consider some of these as limitations is up to you.

Do You Know Why Scala is Object-Oriented Programming Langauge?

So, this was all about Scala Advantages and Disadvantages Tutorial. Hope you like our explanation.

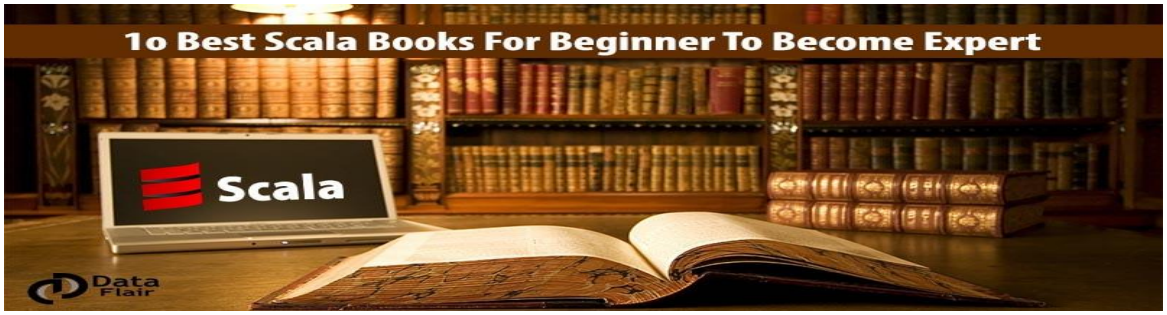
3. Conclusion

Hence, we studied Scala Benefits and Limitations and we have tried to cover a broad view of it in our Scala Advantages and Disadvantages Tutorial. Which language doesn't? That said, each language is beautiful. And if Scala makes for what it takes to run your project, you must definitely explore it, give it a try. If you have any queries regarding Scala advantages and limitations, please comment.

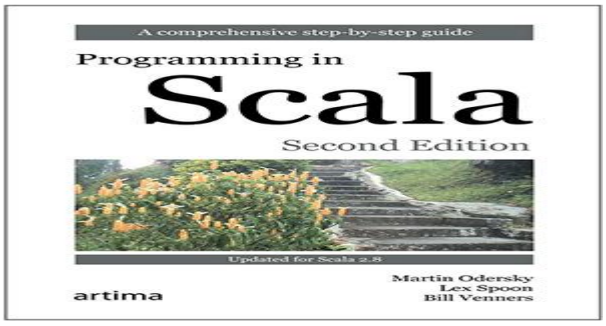
10 Best Scala Books For Beginner To Become Expert 2

1. Objective of Best Scala Books

Scala is now the language of Big Data and has been the most popular language that is supposed to be the only one that could replace Java. It has several new features along with Java features that make it so popular currently to start learning for. To start learning Scala, in this scala tutorial we will list the best books on Scala that would help you to learn Scala from basics to advanced level. Some of these are best Scala books for beginners and some would help you in learning advanced Scala programs to become Scala expert. This book is like Scala wiki contains each and every concepts of scala programming.

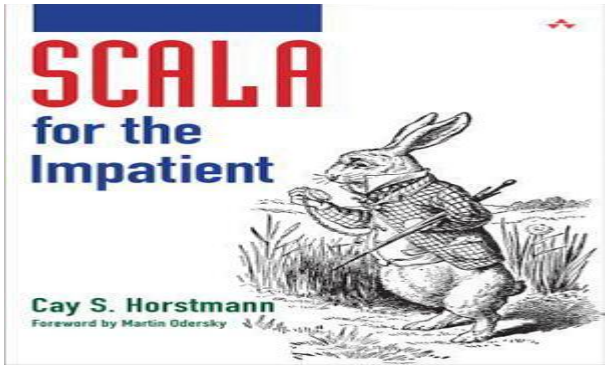


2. Programming in Scala: A comprehensive Step-by-Step Scala Programming Guide by Martin Odersky, Lex Spoon, Bill Venners



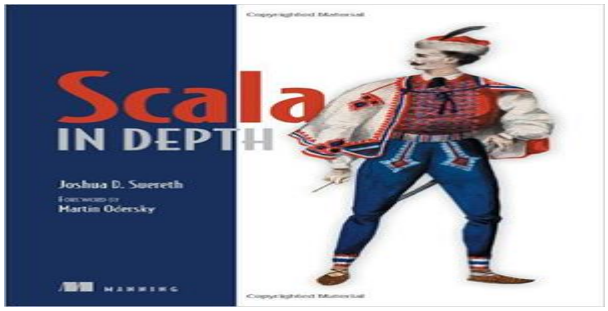
This book covers every concept of Scala starting from fundamentals and builds to advanced scala programming techniques. This book is a complete book to learn and master Scala programming.

3. Scala for the Impatient by Cay Hortsmann



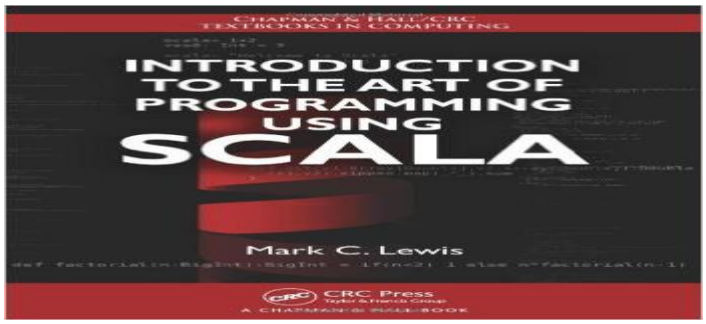
This Scala book provides a code-based introduction to the Scala language and is intended for experienced programmers explaining what Scala can do and how coding can be done effectively in Scala with Scala programs.

4. Scala in Depth by Joshua D Suereth



This Scala book is designed for Java programmers who help them understand how they can integrate Scala language into their existing projects. This programming book will help you in learning best practices for creating Scala applications.

5. Introduction to the Art of Programming Using Scala by Mark Lewis

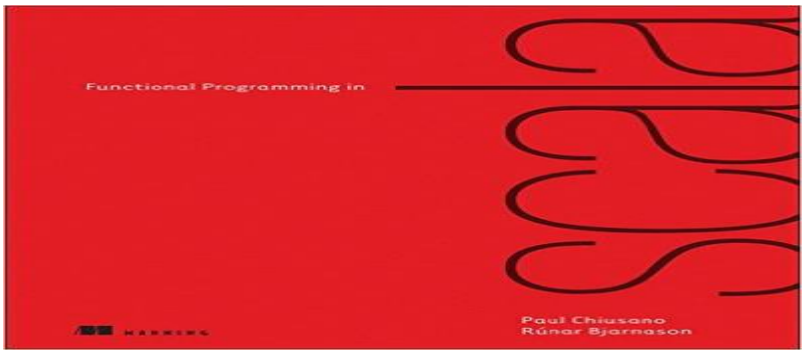


This is one of the few Scala books out there for Scala beginner programmers; this title was written for introductory computer science classes.

6. Atomic Scala by Eckel and Marsh

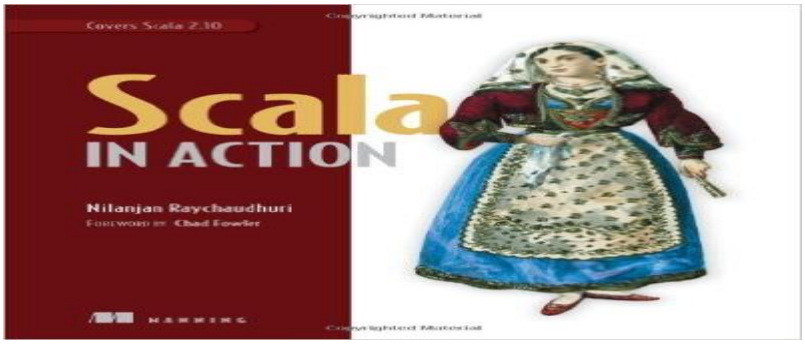
This Scala book is for beginners, new programmers and is specifically designed for professionals who are not from java background but want to learn Scala.

7. Functional Programming in Scala by Paul Chiusano and Rúnar Bjarnason



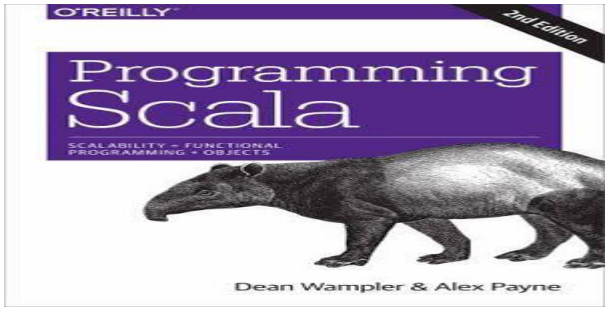
This book guides readers from basic concepts to advanced topics in a logical, concise, and clear progressive manner. Scala Concepts are being explained with examples and exercises to make you Scala expert. It focusses more on Functional programming concepts.

8. Scala in Action: Covers Scala 2.10 by Nilanjan Raychaudhuri and Chad Fowler



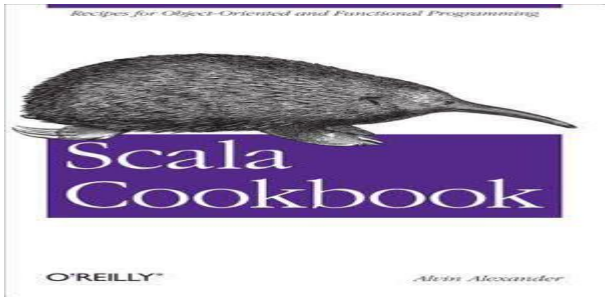
It explains the Scala language through numerous hands-on practical examples. It handles concurrent programming in Akka, explains how to work with Scala and Spring, and shows how to build DSLs and other productivity tools. It covers features of Scala 2.10 as well.

9. Programming Scala by Dean Wampler, Alex Payne



This book explains Scala concepts as JVM language and shows how Scala turns out to be the best development option for programmers. Scala can be learnt from basics to hands on level through this book, what is scala programming to Scala in depth.

10. Scala Cookbook by Alvin Alexander



In this book, author highlights Scala features in an efficient manner. It covers Scala features like Flatmap and provides answers to questions that a new Scala learner would have.

11. Scala Programming by Matthew Gimson

This book begins from Scala origin, uses and benefits and then guides the reader through setting up Scala environment for programming in different operating systems including

Windows, Linux, and Mac OS X. Scala programs syntax is explored to help you understand the various parts which make up a Scala program.

Scala vs Java – Feature wise Comparison Guide

1. Scala vs Java

While **Java** and **Scala** are both names common to the house, what separates them? Each has its own pros and cons. Let's find out what are they in this Scala vs Java tutorial. We will see the Scala vs Java performance, advantages of Scala over java and visa versa so that you will and why Scala or Why Java when you are to choose any one language.

Scala vs Java

2. Similarities in Scala and Java

First, let's see how they're similar.

a. Object-Oriented

Both languages are object-oriented. They can let us model the real-world.

b. JVM-Based

Both Java and Scala run on the JVM (Java Virtual Machine). While Java source code compiles into byte code that runs on the JVM, so does Scala's.

In fact, languages like Scala, Groovy, and JRuby, all hold similarity to Java. This is crediting to the fact that they all use the same memory space, type system, and run inside the same JVM.

c. Typing Discipline

Scala is a statically-typed language like Java. It is also strongly-typed. In Python, the type for an object is decided at runtime. This is duck-typing. So, Python is dynamically-typed, but Scala and Java are statically typed.

d. No NullPointerException

Java has no pointers; Scala has no Null. Consequently, neither of these languages will give you a NullPointerException like C++ will.

e. Programming Paradigm

Both Scala and Java are multi-paradigm, and are imperative(uses statements that change a program's state) and concurrent.

If you face any difficulty in the Scala vs Java Tutorial, Please comment.

Read: **Scala Features – A Comprehensive Guide**

3. Differences between Scala vs Java

Now, time for the differences. What makes Scala different from Java? Let's list it out.

a. Verbosity

We've lost count on how often we've vented over how verbose Java really is. It takes you four lines to execute what a language like Python will do in just one. With Scala, you'll only need to write about one-third the amount of code you'll write for Java for the same thing. To put our money where our mouth is, we'll just go with the simple "Hello, World!" program.

This is it with Java:

```
public class HelloWorld
{
public static void main(String[] args)
{
System.out.println("Hello, World!")
}
}
```

And then, this is it with Scala:

```
object HelloWorld extends App
{
println("Hello, World!")
}
```

Indeed, Scala is very concise.

Winner: Scala

Read: **Features of Java to Learn Why Java Is Important**

b. Readability

Although Java is too verbose, it is more readable than Scala. Why? Because Scala can get too nested at times. Imagine a function inside a function inside another, inside an object inside a class. Our brains aren't calibrated to that less a level of abstraction. Java is nested too, but lesser than Scala.

Winner: Java

c. Compile Time

Scala compiles source code to byte code slower than Java does. Sometimes, it may even take up to an hour if you don't work on a super-fast processor with a bundle of cores. However, a faster compiler is under development.

For Java, the javac compiler turns the source into byte code. For Scala, it is the Scala compiler.

Winner: Java

d. IDE Support

Java has a myriad of IDEs (Integrated Development Environment) available to aid you with development. Major names include Eclipse, NetBeans, and IntelliJ. Actually, IntelliJ is the one we'll be using for Scala in this whole tutorial series. Other IDEs like NetBeans and Eclipse do support Scala, but not that effectively.

Winner: Java

e. Programming Paradigm

While Java is a multi-paradigm, object-oriented language, Scala is multi-paradigm and functional. It supports functional-programming features like currying, type inference, immutability, lazy evaluation, and pattern-matching. However, it also supports object-oriented programming.

f. Multi-Core CPU Architecture

Java believes in adding more CPU cores instead of increasing the CPU cycle. Scala, however, makes use of a multi-core architecture to support a functional programming paradigm.

g. Read-Evaluate-Print-Loop (REPL)

Like Python, Scala believes in REPL, unlike Java. This lets developers explore and access their datasets. It also allows them to prototype their applications easily, without a full-on development cycle.

Winner: Scala

h. Operator Overloading

Java does not support operator overloading. But Scala will even let you create your own operators.

Winner: Scala

i. Simplicity and Learning

Java is simply easier to learn. Since Scala supports features like operator overloading, and uses a lot of nesting, it makes it much messier to learn Scala. Users may confuse between operators and their meanings when one operator can have multiple. Scala code looks repulsive on first look.

Winner: Java

Read: Scala Control Structures – A Comprehensive Guide

j. Lazy Evaluation

Lazy evaluation is something Scala supports, but Java doesn't. What this means is that it delays complex computation until absolutely necessary. For this, it uses the keyword 'lazy'. Let's take an example.

Load an image only if you must, since it is a slow process. We can apply lazy evaluation on this:

```
lazy val images=getImages()

if(viewProfile)

{

showImages(image)

}

else(editProfile)

{

showImages(images)

showEditor()

}

else

{

//Do whatever, without loading the image

}
```

Winner: Scala

Any doubt yet in Scala vs Java Tutorial? Please Comment.

k. Backward Compatibility

With a language that is backward-compatible, code written in a newer version runs without a problem on all older versions. Java is this way; Java 8 code will run on all older versions. This isn't the same with Scala.

Winner: Java

l. Chances of Errors

Because Scala supports operator overloading, it has a higher risk of programming errors than does Java.

Winner: Java

Read: How to Install Java in Windows & Linux

m. Debugging

Java is much easier on the programmers with debugging. It lists out all errors with full traceability, and double-clicking an error will take you to its exact location in the source. In Scala, tracking errors from the stack trace is a pain in the head.

Winner: Java

This was all on Scala vs Java Tutorial.

4. Conclusion: Scala vs Java Tutorial

Now that you know what sets both languages apart from this article on Scala vs Java, have you made your mind to begin with Scala? See you next. And remember, every language is beautiful.