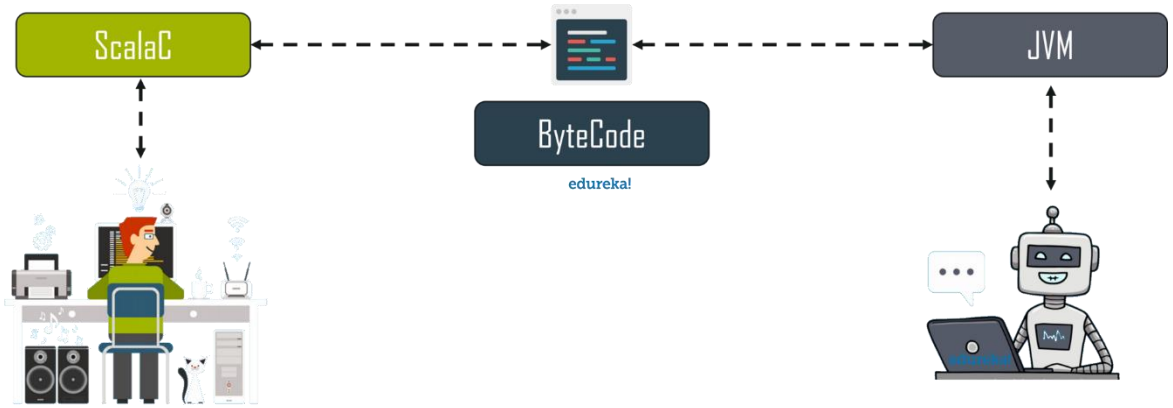
What is Scala? A Robust and High-Caliber programming language that changed the world of **big data.** Scala is capable enough to outrun the speed of the fastest existing programming languages. I'll walk you through this What is Scala article so that you can understand the true capabilities of Scala.

What is Scala?

Well, Scala is a programming language invented by **Mr. Martin Odersky** and his research team in the year **2003**.

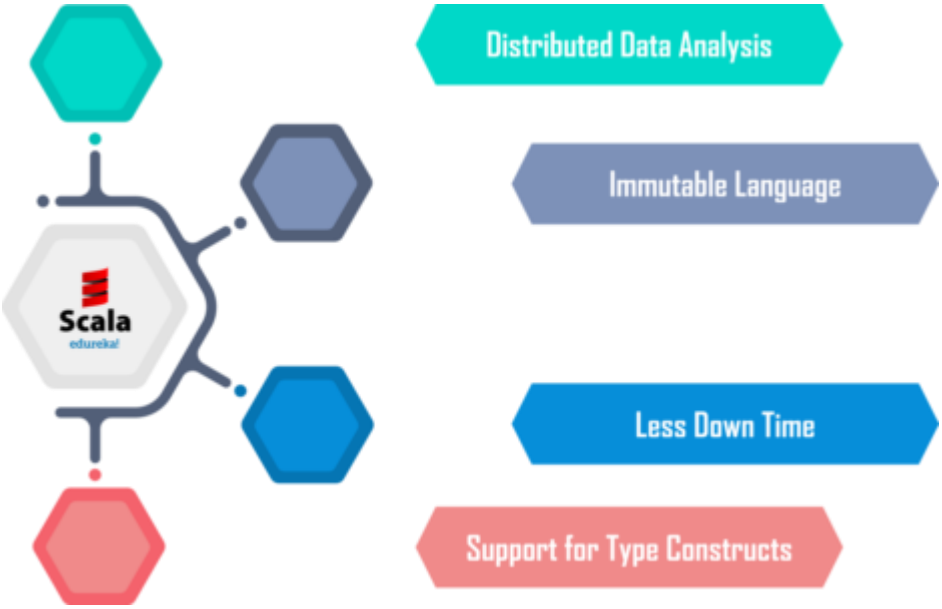
Scala is a compiler based and a multi-paradigm programming language which is **compact**, **fast** and **efficient**. The major advantage of Scala is the **JVM** (**Java Virtual Machine**). Scala code is first compiled by a **Scala compiler** and the **byte code** for the same is generated, which will be then transferred to the **Java Virtual Machine** to generate the output.



Thus, the **Scala** became the key to success for managing the huge amount of **big-data.**

Now that we know the importance of Scala, let us now understand why actually it is the most preferred language in the present trends.

Why we need Scala?



- Scala is capable to work with the data which is stored in a **Distributed** fashion. It accesses all the available resources and supports parallel data processing.
- Scala supports **Immutable** data and it has support to the higher order functions.
- Scala is an **upgraded version of Java** which was designed to eliminate unnecessary code. It supports multiple Libraries and APIs which will allow the programmer to achieve **Less Down Time**.
- Scala supports multiple type **Constructs** which enables the programmer to work with wrappers/container types with ease.

Now that we have understood the requirements for which we needed Scala. Let us move into the comparison between the other languages and find out why it gets an edge over the other similar programming languages.

Scala and Other Languages

The Name **Scala** portraits the **scalability** the language is capable of offering, now you might raise a question. Aren't the latest programming Languages like **Python**, **Ruby**, **Perl** and the Legendary **Java** not scalable?

The answer is **yes**, they are **scalable**, but with some **restrictions** like the boiler plated codes like **system.print.ln** in **Java**. Scala is invented to overcome these limitations and minimize the execution time and complexity of the code.

In the year **2006** Twitter was introduced in America and the developers used **ruby on rails** as their weapon of choice to develop this application, which later proved out to be a **wrong choice** when they had to manage the gigantic amount **Big-Data** which was dropping into the **Twitter**.

Then they switched their backend to **Java** and used **Scala** as their new programming language to handle the big data using **Hadoop** and **Spark** frameworks which worked in a spectacular way.

Now we know the capabilities of Scala, Let us now understand its powerful features:

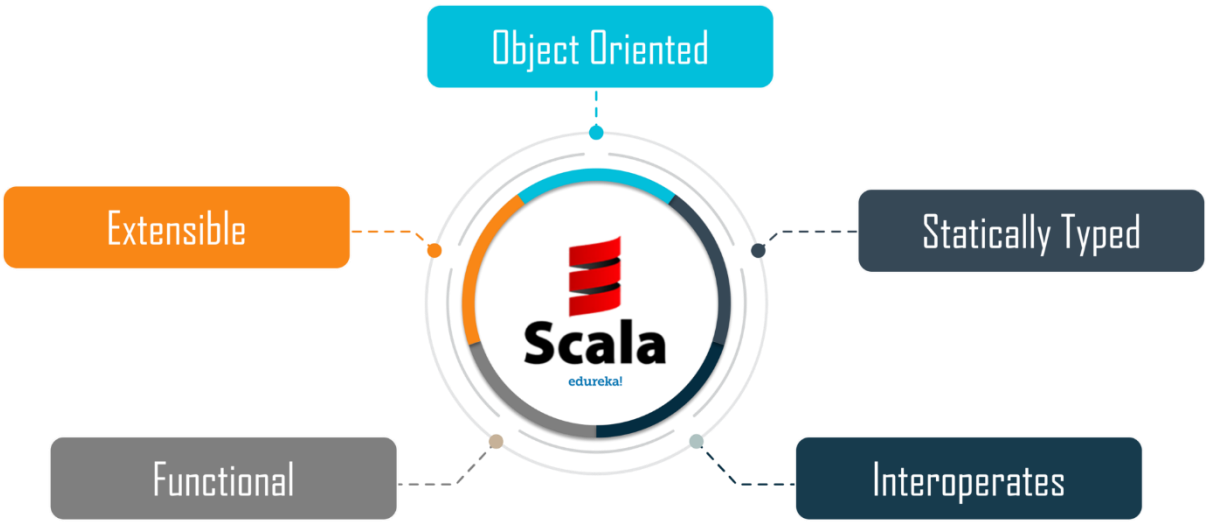
Features of Scala

- **Object-oriented Programing Language:**

Scala is both a functional Programming Language and an object-oriented programming Language. Every variable and value which is used in Scala is implicitly saved as an **object** by default.

- **Extensible Programming Language:**

Scala can support multiple language constructs without the need of any **Domain Specific Language** (**DSL**)Extensions, **Libraries,** and **APIs**.



- **Statically Typed Programming Language:**

Scala binds the Datatype to the variable in its entire **scope**.

- **Functional Programming Language:**

Scala provides a lightweight syntax for defining functions, it supports **higher-order functions**, it allows functions to be **nested**.

- **Interoperability:**

Scala compiles the code using **scala compiler** and converts code into **Java Byte Code** and Executes it on **JVM**.

These were the Features of Scala and let us get into few of the frameworks of Scala is capable to support.

Frameworks of Scala

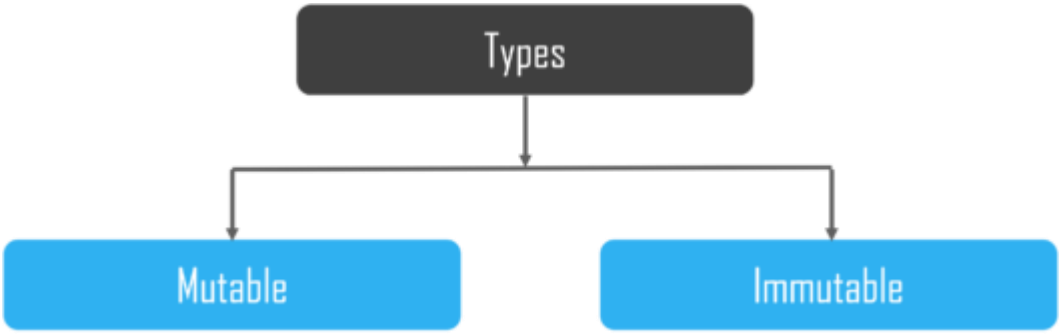
Akka, Spark, Play, Neo4j, Scalding are some of the major frameworks that Scala can support.

- **Akka** is a toolkit on runtime for building highly concurrent, distributed, and fault-tolerant applications on the JVM. **Akka** is written in Scala, with language bindings provided for both Scala and Java.
- **Spark** Framework is designed to handle, and process big-data and it solely supports **Scala**.
- **Play** framework is designed to create web applications and it uses Scala in the process in order to obtain the best in class performance.
- **Scalding** is a **domain-specific language** (**DSL**)in the Scala programming language, which integrates **Cascading**. It is a functional programming paradigm used in Scala which is much closer than Java to the original model for **MapReduce** functions.
- **Neo4j** is a **java spring framework** supported by Scala with domain-specific functionality, analytical capabilities, graph algorithms, and many more.

These were the popular Frameworks supported by Scala, Now let us understand the variables and data types in Scala.

Variables in Scala

Variables can be defined as the reserved memory locations used to store the values. Similarly, we do have variables in Scala Programming Language as well. The Variables in Scala are divided into two types.



Mutable Variables

These variables allow us to **change** a value after the declaration of a variable. **Mutable** variables are defined by using the **var** keyword. The first letter of data type should be in capital letter because in Scala data type is treated as an object.

```
1                    var b = "Edureka"
2                  b = "Brain4ce Organisation"
```

output:

b: String = Edureka
b: String = Brain4ce Organisation

In this case, the variable will accept the new string and displays it.

Immutable Variable

These variables do not allow you to change a value after the declaration of a variable. **Immutable variables** are defined by using the **val** keyword. The first letter of data type should be in

capital letter because in the **Scala** data type is treated as objects.

```
1                    val a = "hello world"
2                      a = "how are you"
```

output:

a: String = hello world
<console>:25: error: reassignment to val
a = “how are you”
^

This code will give an **error** and the new value will not be accepted by the variable a.

Lazy Evaluation

```
1                    lazy val x = 100
2                          x*2
```

output:

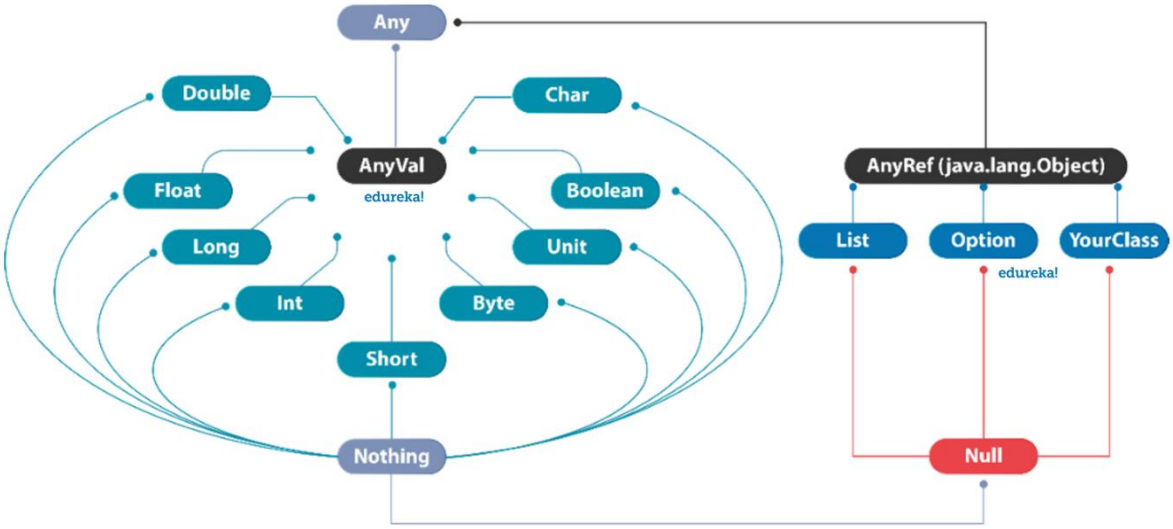
x: Int = <lazy>
res: Int = 200

Lazy Evaluation is the primary feature of Scala which bought it the dignity of a whole new level. Here, the declared variable **will not be accessed** or **any operation is not performed** on to the

variable unless the programmer particularly **accesses** it and **performs an operation** on to it.

In simple words, it is an **On-Demand execution** of an operation which saves a lot of **memory** and **processing resources in real-time.**

Datatypes supported in Scala are as follows.



Collections in Scala

Arrays

An array is a data structure which stores a **fixed-size** sequential collection of elements of the **same data type.**

We shall look into a few examples of arrays in Scala

- Here we are creating an Array of integer type which is empty and will store 0 as a default value in all its memory locations.

```
1                    val array = new Array[Int](10)
```

output:

array: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

- We can access the **memory location** and **ingest** a value into the array.

```
1                    array(0) = 10
2                    array(1) = 2
```

output:

array: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
res: Array[Int] = Array(10, 2, 0, 0, 0, 0, 0, 0, 0, 0)

- Let us create an array of **String Datatype** and print each of the values in a new line using a for loop.

```
1    val mystring = Array("Edureka", "Brain4ce",
2                  "Organisation")
3                for ( x <- mystring )
4                       {
5                    println( x )
                        }
```

output:

Edureka
Brain4ce
Organisation

- Let us now learn about **ArrayBuffer**. We need to import the following **Library** before we perform any of the **operations** on an **Array Buffer**

```
1    import scala.collection.mutable.ArrayBuffer
```

output:

import scala.collection.mutable.ArrayBuffer

- Let us **create** an **Array Buffer** of **Integer type**.

```
1                    val a = ArrayBuffer[Int]()
```

output:

a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

- Let us **insert** an element into the **Array Buffer**

```
1                              a += 1
```

output:

res: a.type = ArrayBuffer(1)

- Let us **insert Multiple elements** into the **Array Buffer.**

```
1                           a += (2,3,4,5)
```

output:

res: a.type = ArrayBuffer(1, 2, 3, 4, 5)

- Let us **insert** a **complete Array** into the **Array Buffer**

```
1                        a ++= Array(6, 7, 8)
```

output:

res: a.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)

- Let us perform some **basic operations** on the **Array Buffer**.
- This function is used to **trim** the last two elements of the **Array Buffer**.

```
1                                   a.trimEnd(2)
```

output:

res: a.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)
res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6)

- This function is used to **insert element 9** in the **2nd memory location** of the **Array Buffer.**

```
1                                   a.insert(2, 9)
```

output:

res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6)

res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 9, 3, 4, 5, 6)

- This function is used to **insert elements (0,9,6,1)** in the **2nd memory location** of the **Array Buffer**.

```
1                    a.insert(2,0,9,6,1)
```

output:

res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 9, 3, 4, 5, 6)
res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 0, 9, 6, 1, 9, 3, 4, 5, 6)

This function is used to **remove** an element in the **second memory location** of the **Array Buffer**.

```
1                    a.remove(2)
```

output:

res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 0, 9, 6, 1, 9, 3, 4, 5, 6)
res: Int = 0
res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 9, 6, 1, 9, 3, 4, 5, 6)

- This function is used to **remove three elements** from the **Array buffer**, starting from the **second memory location**.

```
1                    a.remove(2, 3)
```

output:

res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 9, 6, 1, 9, 3, 4, 5, 6)
res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 9, 3, 4, 5, 6)

- Let us perform some common operations on the Array Buffer
- Sum function is used to perform the summation of all the elements in the Array Buffer.

```
1                    Array(1,2,3,4).sum
```

output:

res: Int = 10

- **Max** function is used to find the element with the maximum in the **Array Buffer**.

```
1                    Array(1,2,3,4).max
```

output:

res: Int = 4

- Let us perform a **quicksort** on the **Array Buffer**.

```
1                      val a = Array(1,5,3,2,4)
2                  scala.util.Sorting.quickSort(a)
```

output:

a: Array[Int] = Array(1, 5, 3, 2, 4)
res: Array[Int] = Array(1, 2, 3, 4, 5)

Lists

The **Scala List** is an immutable sequence of elements, implemented as a linked **list**. Unlike an array, a linked **list** consists of many small objects, each containing a reference to an object as well as a reference to the rest of the **list**

Now we shall look into few examples of lists in Scala.

- **Creating** a Scala **List** object

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

output:

list: List[Int] = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

- **Adding Element** to List

```
val x = List(2)
val y = 1 :: x
val z = 0 :: y
```

output:

x: List[Int] = List(2)
y: List[Int] = List(1, 2)
z: List[Int] = List(0, 1, 2)

- **delete** elements from a Scala **List** or **ListBuffer**

```
val originalList = List(5, 1, 4, 3, 2)
val newList = originalList.filter(_ > 2)
```

output:

originalList: List[Int] = List(5, 1, 4, 3, 2)
newList: List[Int] = List(5, 4, 3)

- Let us **import List Buffer**

```
1          import scala.collection.mutable.ListBuffer
```

output:

import scala.collection.mutable.ListBuffer

- Creating List Buffer

```
1       val x = ListBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

output:

x: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)

- You can **delete** one element at a time, **by value**:

```
1                                  x -= 5
```

output:

res: x.type = ListBuffer(1, 2, 3, 4, 6, 7, 8, 9)

- You can **delete** elements by **position**:

```
1                                x.remove(0)
```

output:

res: Int = 1
res: scala.collection.mutable.ListBuffer[Int] = ListBuffer(2, 3, 4, 6, 7, 8, 9)

- The List '**range**' method

```
1                    val x = List.range(1,10)
```

output:

x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

- The **range function** can also take a third argument which serves as a "**step**"

```
1                    val x = List.range(0,10,2)
```

output:

x: List[Int] = List(0, 2, 4, 6, 8)

- The List '**fill**' method

```
1                    val x = List.fill(3)("Apple")
```

output:

x: List[String] = List(Apple, Apple, Apple)

- The List class '**tabulate**' method
- The tabulate method creates a new List whose elements are created according to the **function** you apply

```
1                    val x = List.tabulate(5)(n = n + n)
```

output:

x: List[Int] = List(0, 2, 4, 6, 8)

- **prepend** items to a List

- create a List

```
1                    val x = List(1,2,3)
```

output:

x: List[Int] = List(1, 2, 3)

- **prepend** an element to the list

```
1                    val y = 0 :: x
```

output:

y: List[Int] = List(0, 1, 2, 3)

- **Appending** and merging Lists

```
1                    val a = List(1,2,3)
2                    val b = List(4,5,6)
3                     val c = a ::: b
```

output:

y: List[Int] = List(0, 1, 2, 3)
a: List[Int] = List(1, 2, 3)
b: List[Int] = List(4, 5, 6)
c: List[Int] = List(1, 2, 3, 4, 5, 6)

- You can also **merge** two Scala lists using the List's **concat** method:

```
1                    val c = List.concat(a, b)
```

output:

c: List[Int] = List(1, 2, 3, 4, 5, 6)

- **Iterating** lists with foreach

```
1                    val x = List(1,2,3)
2                   x.foreach { println }
```

output:

x: List[Int] = List(1, 2, 3)
1
2
3

- **add one element** at a time to the **ListBuffer**

```
1          var flowers = new ListBuffer[String]()
2                   flowers += "Rose"
3                  flowers += "Lilly"
4                  flowers += "Tulip"
```

output:

flowers: scala.collection.mutable.ListBuffer[String] = ListBuffer()
res: scala.collection.mutable.ListBuffer[String] =
ListBuffer(Rose)
res: scala.collection.mutable.ListBuffer[String] =
ListBuffer(Rose, Lilly)

- **add multiple elements**

```
1       flowers += ("Daisy", "Sunflower", "Jasmine")
```

output:

res: scala.collection.mutable.ListBuffer[String] =
ListBuffer(Rose, Lilly, Tulip, Daisy, Sunflower, Jasmine)

- remove one element

```
1                    flowers -= "Rose"
```

output:

res: scala.collection.mutable.ListBuffer[String] = ListBuffer(Lilly, Tulip, Daisy, Sunflower, Jasmine)

- **remove multiple elements**

```
1                flowers -= ("Lilly", "Tulip")
```

output:

res: scala.collection.mutable.ListBuffer[String] = ListBuffer(Daisy, Sunflower, Jasmine)

- **remove multiple elements specified by another sequence**

```
1            flowers --= Seq("Daisy", "Sunflower")
```

output:

res: scala.collection.mutable.ListBuffer[String] = ListBuffer(Jasmine)

- **convert** the **ListBuffer** to a **List** when you need to

```
1              val flowersList = flowers.toList
```

output:

flowersList: List[String] = List(Jasmine)

Sets

A **Set** is a collection that contains no duplicate elements. By default, **Scala** uses the immutable **Set**. If you want to use the mutable **Set**, you'll have to import the library called,

import scala.collection.mutable.Set class explicitly.

We shall try out some examples of sets in Scala.

- **Set** is a collection that contains no **duplicate** elements. There are two kinds of Sets, the **immutable** and the **mutable**.

```
1                   var s : Set[Int] = Set()
2             var s : Set[Int] = Set(1,1,5,5,7)
```

output:

var s : Set[Int] = Set()
var s : Set[Int] = Set(1,1,5,5,7)

- Immutable set

```
1                       var s = Set(1,3,5,7)
```

output:

s: scala.collection.immutable.Set[Int] = Set(1, 3, 5, 7)

- Basic operational methods

```
1       val fruit = Set("apples", "oranges", "pears")
2         println( "Head of fruit : " + fruit.head )
3         println( "Tail of fruit : " + fruit.tail )
4  println( "Check if fruit is empty : " + fruit.isEmpt
5                 val nums: Set[Int] = Set()
6   println( "Check if nums is empty : " + nums.isEmpty
```

output:

fruit: scala.collection.immutable.Set[String] = Set(apples, oranges, pears)
Head of fruit : apples
Tail of fruit : Set(oranges, pears)
Check if fruit is empty : false
nums: Set[Int] = Set()
Check if nums is empty : true

- **Concatenating** Sets
- You can use either ++ operator or Set.++() method to **concatenate** two or more sets, but while adding sets it will remove **duplicate** elements.

```
1      val fruit1 = Set("apples", "oranges", "pears")
2    val fruit2 = Set("mangoes", "banana", "oranges")
3             val fruit = fruit1 ++ fruit2
4                          fruit
```

output:

fruit1: scala.collection.immutable.Set[String] = Set(apples, oranges, pears)
fruit2: scala.collection.immutable.Set[String] = Set(mangoes, banana, oranges)
fruit: scala.collection.immutable.Set[String] = Set(banana, apples, mangoes, pears, oranges)

- use two or more sets with **++** as **operator**
- Mutable set
- Add elements to a mutable Set with the +=, ++=, and add methods:
- use **var** with mutable

```
1      var set = scala.collection.mutable.Set[Int]()
```

output:

set: scala.collection.mutable.Set[Int] = Set()

- add **one** element

```
1                                   set += 1
```

output:

res: scala.collection.mutable.Set[Int] = Set(1)

- add **multiple** elements

```
1                                 set += (2, 3)
2                                   set.add(6)
3                                   set.add(2)
```

output:

res: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
res: Boolean = true
res: Boolean = false

Maps

A **Scala Map** is a collection of a Key-value pair. A **map** cannot have duplicate keys, but different keys can have the same values

We shall try a few examples of maps in Scala.

```
1 val colors1 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF",
2 "peru" -> "#CD853F")
3 val colors2 = Map("blue" -> "#0033FF", "yellow" ->
4 "#FFFF00", "red" -> "#FF0000")
5 colors1.keys
6 colors1.values
  colors1.isEmpty
```

```
var colors = colors1 ++ colors2
```

output:

res: Iterable[String] = Set(red, azure, peru)
res: Iterable[String] = MapLike(#FF0000, #F0FFFF, #CD853F)
res: Boolean = false

Let us check **another example** in maps.

```
1 val mapping = Map("virat" -> "kohili", "mahendra" -> "singhdh
2 val mapping = scala.collection.mutable.Map("virat" -> "kohil
3 "mahendra" -> "singhdhoni")
4 mapping("virat")
5 mapping -= "mahendra"
6 mapping += ("ajay"-> "sharma")
  mapping.getOrElse("virat", 0)
```

output:

mapping: scala.collection.immutable.Map[String,String] =
Map(virat -> kohili, mahendra -> singhdhoni)
mapping: scala.collection.mutable.Map[String,String] =
Map(mahendra -> singhdhoni, virat -> kohili)
res: String = kohili
res: mapping.type = Map(virat -> kohili)
res: mapping.type = Map(virat -> kohili, ajay -> sharma)
res: Any = kohili

Tuples

Scala tuple combines a fixed number of items together so that they can be passed around. Unlike an array or list, a **tuple** can hold objects with different types, but they are also **immutable**. The following is an example of a **tuple** holding an integer, a string.

We shall try a few examples of tuples in Scala.

```
val a = (1,2,"Ajay","Devgan")
a._1
a._3
```

output:

a: (Int, Int, String, String) = (1,2,Ajay,Devgan)
res: Int = 1
res: String = Ajay

These were the Collections supported by Scala, Now let us understand the **Control Statements**.

Control Statements in Scala

Now with this, we shall move into our next topic, the **control statements**.

If

An **if** statement decides to execute which of the two given **statements**.

```
1            for (i <- 0 to 5; j <- 0 to 5 if i==j)
2                    println(i + j)
```

output:

0
2
4
6
8
10

if else

An **if** statement which will be followed by an **else** statement which executes **one** of the two statements. if the **condition** provided at the if block is **true** then the if block statements will be executed. else, the statements at the **else** block will be executed.

```
var x = 5
val s = if (x > 0 && x < 6) 1 else 0
val s =if (x > 0 && x < 6) "positive" else 0
```

output:

x: Int = 5
s: Int = 1
s: Any = positive

While

While is a control statement which checks the **condition** and determines whether to **execute** a particular set of statements present in the loop or not.

```
var args = "Edureka"
println("length is " + args.length)
var i = 0
while(i < args.length)
{
println(args(i))
i+= 1
}
```

output:

E
d
u
r
e
k
a

Do while

Do While loop is similar to While loop with the only difference which is the **condition** to the loop lies at the **end of the loop** program block as shown below. The Do while loop **executes at least** for once as the condition checked after the execution of statements in the block.

```
1                    var x=7
2                    do
3                    {
4                    println(x)
5                    x=x-1
6                    }while(x > 0);
```

output:

```
7
6
5
4
3
2
1
```

For

For loop is a simple loop which has an **initializing variable**, **counter variable** and a **condition** in it. The initializing variable will be initialized to a certain value and counter variable counts the iterations and the iterations will run until the provided condition is true.

```
1                    for ( i <- 1 to 5)
2                    println(i)
3                    #Advaned For Loop
4                    for (i <- 0 to 5 if i==j)
5                    println(10*i + j)
```

output

1
2
3
4
5

output (Advanced For Loop):

0
11
22
33
44
55

For each iteration of your for loop, **yield** generates a value which will be remembered.
It's like the for loop has a **buffer** you can't see, and for each iteration of your for loop, another item is added to that buffer. When your for loop finishes running, it will return this **collection** of all the **yielded** values.

```
1                   for (i <- 1 to 5) yield i
2                 for (i <- 1 to 5) yield i * 2
```

output:

res: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5)
res: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

Foreach

Foreach control statement is similar to the for loop on the only difference is it prints each value and to iterate items in a list, we can use for loop. It operates on arrays or collections such as **ArrayList**, which can be found in the System.

```
1                    var args= "Hello"
2                 args.foreach(println(arg))
```

output:

H
e
l
l
o

Find the area of a circle

```
1          def area (radius: Int): Double= {
2    println("This is a function to perfrom double
3                        operations")
4                  3.14 * radius * radius
5                             }
                      area(radius)
```

output:

res: Double = 1384.74

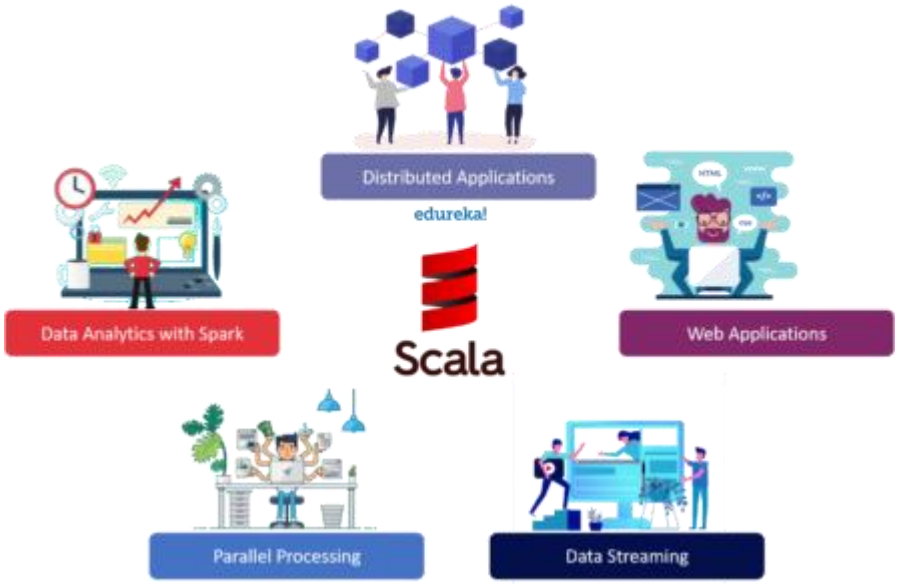
Now Let us understand the **Applications** where we require **Scala**.

Applications of Scala

Scala is a powerful programming language that has the capabilities to support multiple functionalities.

Some of the major applications of Scala are as follows:

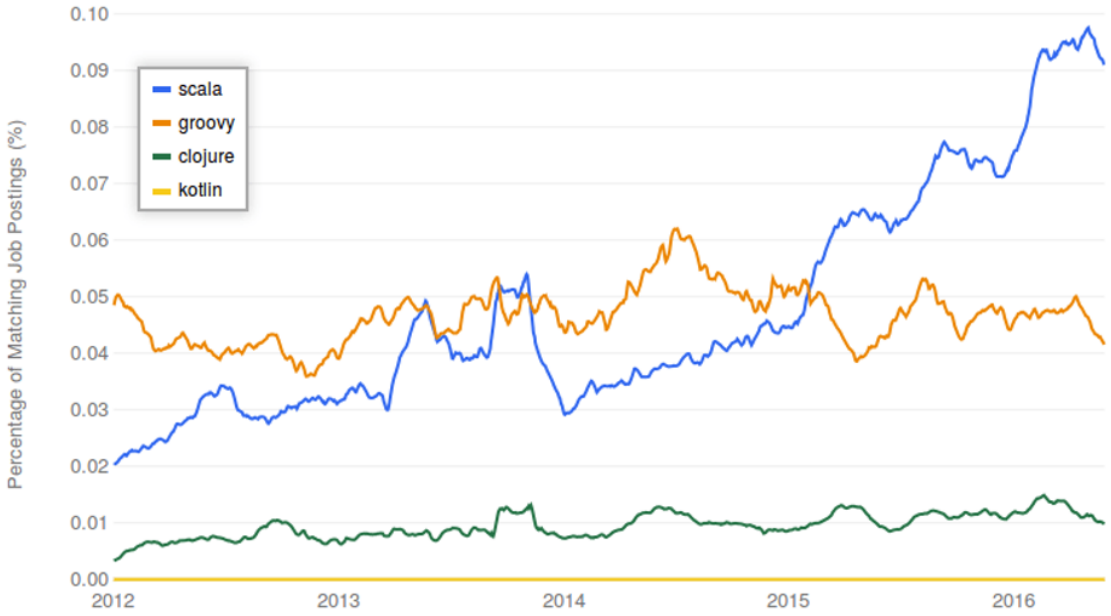
- Designing **Web Applications** and Web Pages
- **Spark Framework** uses Scala to perform **real-time data streaming**
- **Concurrency** and **Distributed data processing applications**
- **Scala** supports both **Batch Data Processing** and **Parallel Data Processing**
- **Spark Framework** uses Scala in **Data Analytics**



Now, these were a few important applications of **Scala**, Let us now look into the **scope** we have for **Scala** Programming Language.

Scope for Scala

Scala is the miracle of the 20th century in **multiple streams**. It has seen astounding growth since day one and it is for sure it is one of the programming languages which is in **higher demand**. The stats below explain more about the **scope of Scala** in the near future.



The **chart** below describes the **permanent jobs** and **Contract based jobs** available based on the knowledge of Scala Programming Language.