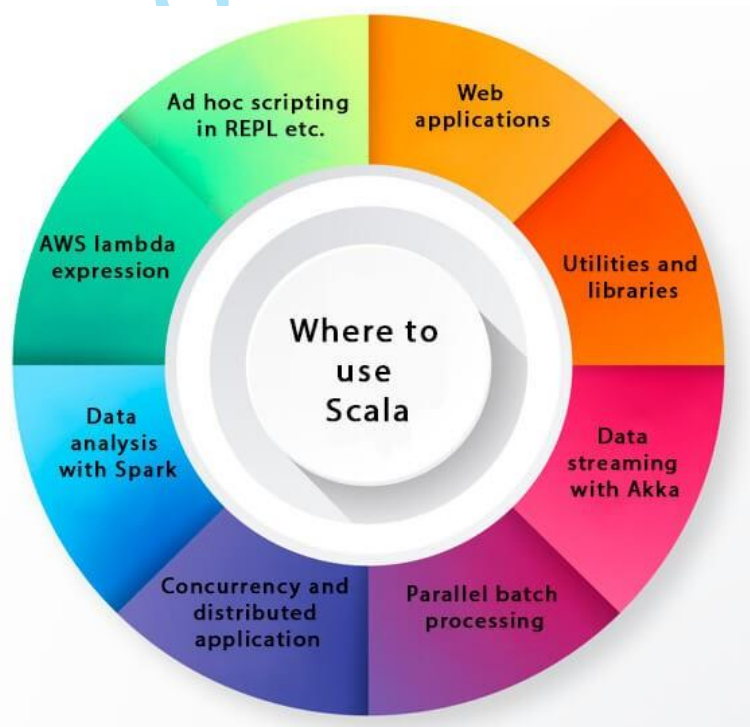


Introduction to Programming in Scala



Introduction of Scala

- Scala is a general-purpose programming language.
- It supports object oriented, functional and imperative programming approaches. It is a strong static type of language.
- In scala, everything is an object whether it is a function or a number.
- It does not have the concept of primitive data.
- File extension of scala source file may be either. scala or .sc.
- We can create any kind of application like web application, enterprise application, mobile application, desktop-based application etc.
- It was designed by Martin Odersky. It was officially released for java platform in early 2004 and for .Net framework in June 2004.



Features of Scala

- **Type Inference:** In Scala, we don't require to mention data type and function return type explicitly. Scala is enough smart to deduce the type of data. The return type of function is determined by the type of last expression present in the function.
- **Immutability:** It is the quality of being unchangeable. In scala variables are immutable by default; once assigned a value they cannot be modified by us or any other programmer.
- **Lazy Computation:** In Scala, computation is lazy by default. Scala evaluates expressions only when they are required. We can declare a lazy variable by using lazy keywords. It is used to increase performance.
- **Higher Order Function:** Higher order function is a function that either takes a function as an argument or returns a function. In other words, we can say a function which works with another function is called higher order function.
- **String Interpolation:** String interpolation in Scala is a way to create strings that include values from variables directly within the string itself. It's like combining pieces of text with the values of variables.
- **Rich Set of Collection:** Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. We can use it according to our requirements. **Scala.collection.mutable** package contains all the mutable collections. We can add, remove and update data while using this package.

Variables and Printing Methods in Scala

Mutable Variable	<p>We can create mutable variable using var keyword. It allows us to change value after declaration of variable.</p> <pre>var data = 100 data = 101 // It works, No error.</pre> <pre>var data:Int = 100 // Here, we have mentioned Int followed by : (colon)</pre>
Immutable Variable	<p>We can create immutable variable using val keyword. It allows us to change value after declaration of variable.</p>

	<pre>val data = 100 data = 101 // Error: reassignment to val</pre> <p>The above code throws an error because we have changed content of immutable variable, which is not allowed. So if we want to change content then it is advisable to use var instead of val.</p>		
Data Types	Data Type	Default Value	Size
	Boolean	False	True or false
	Byte	0	8 bit signed value (-2^7 to 2^7-1)
	Short	0	16 bit signed value (-2^{15} to $2^{15}-1$)
	Char	'\u0000'	16 bit unsigned Unicode character (0 to $2^{16}-1$)
	Int	0	32 bit signed value (-2^{31} to $2^{31}-1$)
	Long	0L	64 bit signed value (-2^{63} to $2^{63}-1$)
	Float	0.0F	32 bit IEEE 754 single-precision float
	Double	0.0D	64 bit IEEE 754 double-precision float
	String	Null	A sequence of characters

Scala Conditional Expressions

If statement	<pre>var age:Int = 20; if(age > 18){ println ("Age is greate than 18") }</pre>
---------------------	--

If Else statement	<pre> var number:Int = 21 if(number%2==0){ println("Even number") }else{ println("Odd number") } </pre>
If Else If statement	<pre> var number:Int = 85 if(number>=0 && number<50){ println ("fail") } else if(number>=50 && number<60){ println("D Grade") } else if(number>=60 && number<70){ println("C Grade") } else if(number>=70 && number<80){ println("B Grade") } else if(number>=80 && number<90){ println("A Grade") } else if(number>=90 && number<=100){ println("A+ Grade") } else println ("Invalid") </pre>

Scala Pattern Matching

```
object MainObject {  
  def main(args: Array[String]) {  
    var a = 1  
    a match {  
      case 1 => println("One")  
      case 2 => println("Two")  
      case _ => println("No")  
    }  
  }  
}
```

Here, match using a variable named a. This variable matches with best available case and prints output. Underscore () is used in the last case for making it default case.

```
object MainObject {  
  def main(args: Array[String]) {  
    var result = search ("Hello")  
    print(result)  
  }  
  def search (a:Any):Any = a match {  
    case 1  => println("One")  
    case "Two" => println("Two")  
    case "Hello" => println("Hello")  
    case _ => println("No")  
  }  
}
```

Scala Loops

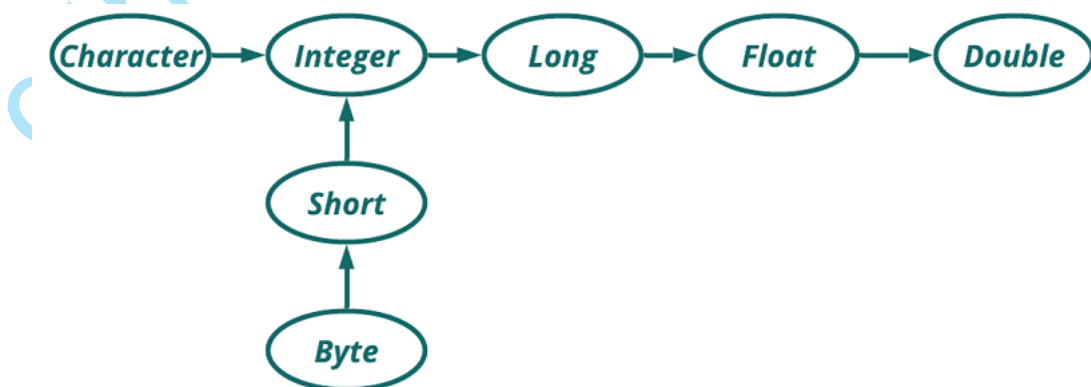
While loop

```
while (boolean expression) {  
  // Statements to be executed  
}
```

```
object MainObject {  
  def main(args: Array[String]) {  
    var a = 10;           // Initialization  
    while ( a<=20 ){      // Condition  
      println(a);  
      a = a+2             // Incrementation  
    }  
  }  
}
```

For loop	<pre>for(i <- range){ // statements to be executed }</pre>	<pre>object MainObject { def main(args: Array[String]) { for(a <- 1 to 10){ println(a); } } }</pre> <pre>object MainObject { def main(args: Array[String]) { for(a <- 1 until 10){ println(a); } } }</pre>
----------	--	--

Typecasting in Scala



Conversion
between
Numeric
Types

- toInt: Converts a numeric value to an Int.
- toLong: Converts a numeric value to a Long.
- toFloat: Converts a numeric value to a Float.
- toDouble: Converts a numeric value to a Double.
- toByte: Converts a numeric value to a Byte.
- toShort: Converts a numeric value to a Short.

```
val doubleValue: Double = 3.14
val intValue: Int = doubleValue.toInt // intValue will be 3
```

```
val intValue: Int = 42
val longValue: Long = intValue.toLong // longValue will be 42
```

```
val intValue: Int = 42
val floatValue: Float = intValue.toFloat
```

```
val intValue: Int = 42
val doubleValue: Double = intValue.toDouble
```

```
val intValue: Int = 42
val byteValue: Byte = intValue.toByte
```

```
val intValue: Int = 42
val shortValue: Short = intValue.toShort
```

Collection Conversions	<pre> val array: Array[Int] = Array(1, 2, 3) val list: List[Int] = array.toList // Convert Array to List val list: List[Int] = List(1, 2, 3) val array: Array[Int] = list.toArray // Convert List to Array val list: List[Int] = List(1, 2, 2, 3, 3, 3) val set: Set[Int] = list.toSet // Convert List to Set val pairs: List[(String, Int)] = List(("Alice", 30), ("Bob", 25)) val map: Map[String, Int] = pairs.toMap // Convert List of tuples to Map val list: List[Int] = List(1, 2, 3) val seq: Seq[Int] = list.toSeq // Convert List to Seq val list: List[Int] = List(1, 2, 3) val stream: Stream[Int] = list.toStream // Convert List to Stream </pre>
Type Casting via "asInstanceOf"	<p>The asInstanceOf method in Scala is a general-purpose mechanism for explicit type casting. It allows us to cast an object to a different type, but it is not type-safe. It means that the compiler does not perform any type checking during compile-time, and if the actual type of the object is not compatible with the target type, a ClassCastException will be thrown at runtime.</p> <p>The syntax of asInstanceOf is as follows:</p>

	<pre>val result = value.asInstanceOf[TargetType]</pre> <p>Example</p> <pre>val anyValue: Any = 42</pre> <pre>val intValue: Int = anyValue.asInstanceOf[Int]</pre>
--	---

Operators in Scala

Arithmetic Operators	<pre>val a = 10</pre> <pre>val b = 5</pre> <pre>val addition = a + b // 15</pre> <pre>val subtraction = a - b // 5</pre> <pre>val multiplication = a * b // 50</pre> <pre>val division = a / b // 2</pre> <pre>val modulus = a % b // 0</pre> <pre>Arithmetic Operators: Addition: 15 Subtraction: 5 Multiplication: 50 Division: 2 Modulus: 0</pre>
Relational Operators	<pre>val x = 10</pre> <pre>val y = 20</pre>

	<pre> val isEqual = x == y // false val isNotEqual = x != y // true val isGreater = x > y // false val isLess = x < y // true val isGreaterOrEqual = x >= y // false val isLessOrEqual = x <= y // true Comparison Operators: Is Equal: false Is Not Equal: true Is Greater: false Is Less: true Is Greater Or Equal: false Is Less Or Equal: true </pre>
Logical Operators	<pre> val p = true val q = false val andResult = p && q // false val orResult = p q // true val notResult = !p // false Logical Operators: AND Result: false OR Result: true NOT Result: false </pre>
Bitwise Operators	<pre> val a = 5 val b = 3 </pre>

	<pre> val bitwiseAnd = a & b // 1 (0101 & 0011 = 0001) val bitwiseOr = a b // 7 (0101 0011 = 0111) val bitwiseXor = a ^ b // 6 (0101 ^ 0011 = 0110) val bitwiseComplement = ~a // -6 (complement of 0101 is 1010) val leftShift = a << 1 // 10 (0101 << 1 = 1010) val rightShift = a >> 1 // 2 (0101 >> 1 = 0010) val zeroFillRightShift = a >>> 1 // 2 (0101 >>> 1 = 0010) Bitwise Operators: Bitwise AND: 0 Bitwise OR: 15 Bitwise XOR: 15 Bitwise Complement: -11 Left Shift: 20 Right Shift: 5 Zero-fill Right Shift: 5 </pre>
Assignment Operators	<pre> var x = 10 var y = 5 x += y // x = x + y x -= y // x = x - y x *= y // x = x * y x /= y // x = x / y x %= y // x = x % y </pre>

	<pre>Result after addition assignment: 15 Result after subtraction assignment: 12 Result after multiplication assignment: 24 Result after division assignment: 6 Result after modulus assignment: 0</pre>
String Concatenation Operator	<pre>val firstName = "John" val lastName = "Doe" val fullName = firstName + " " + lastName // "John Doe"</pre> <p>The + operator can also be used for concatenating strings.</p>

Type Inference in Scala

Type Inference

keyword `variableName`: **DataType** = Initial Value



keyword `variableName` = Initial Value

Type inference in Scala refers to the compiler's ability to automatically deduce the data types of variables and expressions based on context, without explicit type annotations. This feature helps reduce verbosity in code while maintaining strong static typing.

```
object TypeInferenceDemo extends App {
```

```
  // Variable declaration with type inference
```

```
  val x = 10 // Compiler infers type Int
```

```
  // Function declaration with type inference
```

```
  def square(y: Int) = y * y // Compiler infers return type Int
```

```
  // List declaration with type inference
```

```
  val numbers = List(1, 2, 3, 4, 5) // Compiler infers type List[Int]
```

```
  // Map declaration with type inference
```

```
  val userInfo = Map("name" -> "Alice", "age" -> 30) // Compiler infers type Map[String, String]
```

```
  // Conditional expression with type inference
```

```
  val result = if (x > 5) "greater than 5" else "less than or equal to 5" // Compiler infers type String
```

```
  // Pattern matching with type inference
```

```
  val message = x match {
```

```
    case 0 => "zero"
```

```
    case n if n > 0 => "positive"
```

```
    case _ => "negative"
```

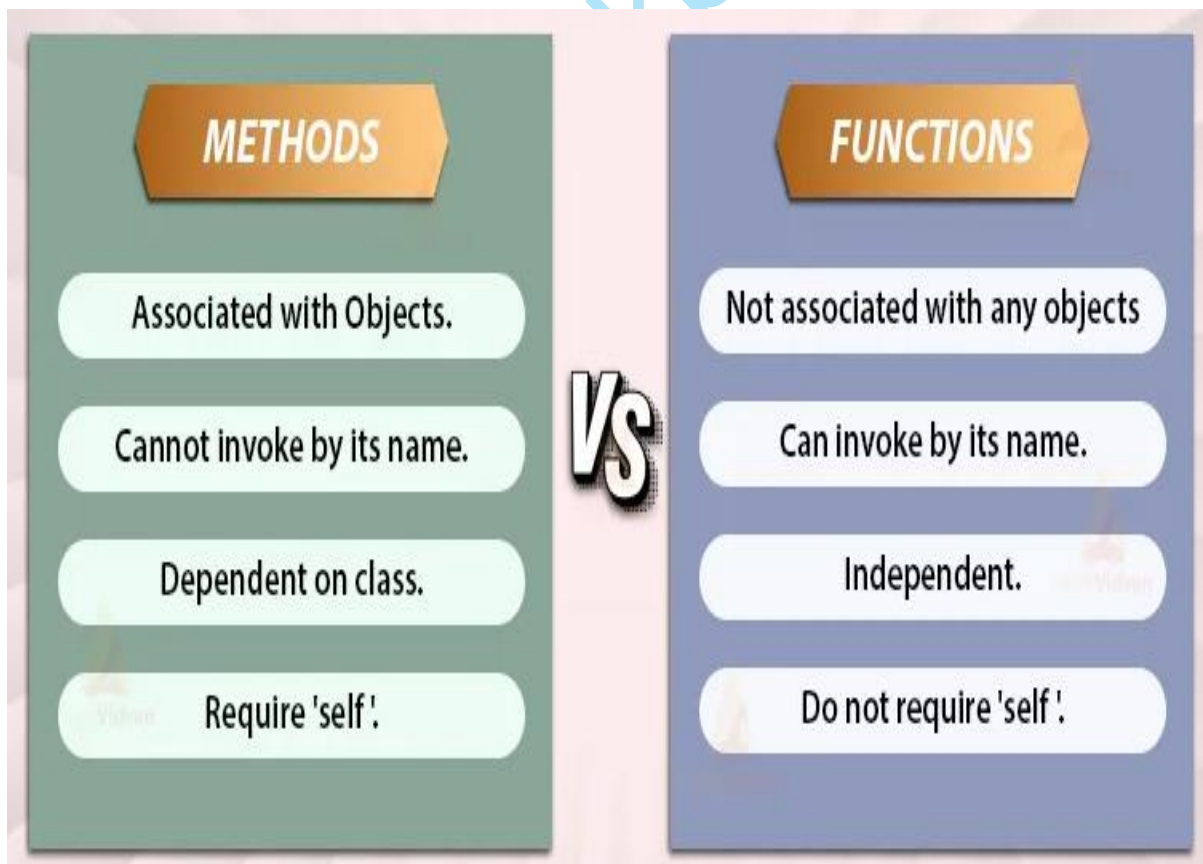
```
  } // Compiler infers type String
```

```
println(s"x: $x")
println(s"Square of x: ${square(x)}")
println(s"Numbers: $numbers")
println(s"User info: $userInfo")
println(s"Result: $result")
println(s"Message: $message")
}
```

Functions and Methods in Scala

A function is a set of instructions or procedures to perform a specific task, and a method is a set of instructions that are associated with an object.

Some differences between a function and method are listed below:



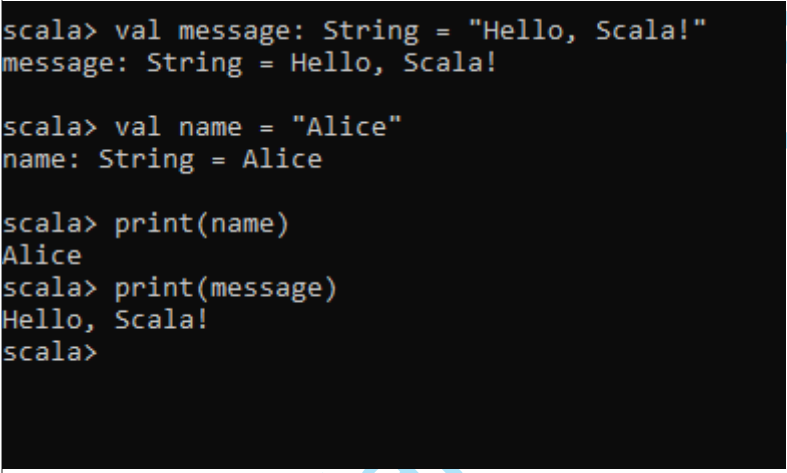
```
class Calculator {  
  def add(x: Int, y: Int): Int = {  
    x + y  
  }  
}  
  
val calc = new Calculator()  
val result = calc.add(3, 5) // Method invocation  
println(result) // Output: 8
```

In this example, add is a method defined within the Calculator class. It takes two integer parameters x and y, and returns their sum. The add method is invoked on an instance of the Calculator class (calc), and it operates on the state of that instance.

```
val add: (Int, Int) => Int = (x, y) => x + y  
  
val result = add(3, 5) // Function invocation  
println(result) // Output: 8
```

In this example, add is a function that takes two integer parameters x and y, and returns their sum. The function is defined independently of any class or object. It can be assigned to a variable (add) and invoked directly without needing an instance of any class. The function operates only on its input parameters and does not have access to any object's state.

Strings in Scala

Creating String	<pre>val message: String = "Hello, Scala!" val name = "Alice" // Type inference infers String type print(name) print(message)</pre>  <pre>scala> val message: String = "Hello, Scala!" message: String = Hello, Scala! scala> val name = "Alice" name: String = Alice scala> print(name) Alice scala> print(message) Hello, Scala! scala></pre>
Concatenating Strings	<pre>val firstName = "John" val lastName = "Doe" val fullName = firstName + " " + lastName // Using + val fullNameInterpolated = s"\$firstName \$lastName" // Using string interpolation</pre>

	<pre>scala> val firstName = "John" firstName: String = John scala> val lastName = "Doe" lastName: String = Doe scala> val fullName = firstName + " " + lastName fullName: String = John Doe scala> val fullNameInterpolated = s"\$firstName \$lastName" fullNameInterpolated: String = John Doe scala></pre>
Accessing Characters	<pre>val str = "Scala" val firstChar = str(0) // 'S' val secondChar = str(1) // 'c'</pre> <pre>scala> val str = "Scala" str: String = Scala scala> val firstChar = str(0) firstChar: Char = S scala> val secondChar = str(1) secondChar: Char = c scala></pre>
String Length	<pre>val str = "Scala" val length = str.length // 5</pre>

	<pre>scala> val str = "Scala" str: String = Scala scala> val length = str.length length: Int = 5 scala></pre>
String Methods	<pre>val str = "Hello, Scala!" val upperCase = str.toUpperCase // "HELLO, SCALA!" val lowerCase = str.toLowerCase // "hello, scala!" val startsWithHello = str.startsWith("Hello") // true val endsWithScala = str.endsWith("Scala!") // true val containsComma = str.contains(",") // true val parts = str.split(",") // Array("Hello", " Scala!") val substring = str.substring(7, 12) // "Scala"</pre>

	<pre>scala> :paste // Entering paste mode (ctrl-D to finish) val str = "Hello, Scala!" val upperCase = str.toUpperCase // "HELLO, SCALA!" val lowerCase = str.toLowerCase // "hello, scala!" val startsWithHello = str.startsWith("Hello") // true val endsWithScala = str.endsWith("Scala!") // true val containsComma = str.contains(",") // true val parts = str.split(",") // Array("Hello", " Scala!") val substring = str.substring(7, 12) // "Scala" // Exiting paste mode, now interpreting. str: String = Hello, Scala! upperCase: String = HELLO, SCALA! lowerCase: String = hello, scala! startsWithHello: Boolean = true endsWithScala: Boolean = true containsComma: Boolean = true parts: Array[String] = Array(Hello, " Scala!") substring: String = Scala scala></pre>
Immutable Properties	<p>Strings in Scala are immutable, meaning once created, their values cannot be changed. Any operation that appears to modify a string actually creates a new string with the modified value.</p> <pre>var str = "Hello" str = str + ", Scala!" // Creates a new string</pre>

	<pre>scala> :paste // Entering paste mode (ctrl-D to finish) var str = "Hello" str = str + ", Scala!" // Exiting paste mode, now interpreting. str: String = Hello, Scala! str: String = Hello, Scala! scala></pre>
String Interpolation	<p>s Interpolation: This type of interpolation is denoted by the prefix s. It allows us to embed expressions directly within a string by prefixing them with the \$ symbol. Any valid Scala expression can be used inside the <code>\${} </code> block.</p> <pre>val name = "Alice" val age = 30 val message = s"Hello, my name is \$name and I am \$age years old." println(message) // Output: Hello, my name is Alice and I am 30 years old.</pre>

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val name = "Alice"
val age = 30
val message = s"Hello, my name is $name and I am $age years old."
println(message)

// Exiting paste mode, now interpreting.

Hello, my name is Alice and I am 30 years old.
name: String = Alice
age: Int = 30
message: String = Hello, my name is Alice and I am 30 years old.

scala>
```

f Interpolation: This type of interpolation allows us to format the interpolated values using printf-style format specifiers. It is denoted by the prefix `f`, followed by a string with embedded expressions. Inside the expressions, you can use format specifiers like `%s` for strings, `%d` for integers, `%f` for floating-point numbers, etc.

```
val name = "Bob"
val age = 25
val height = 175.5
val message = f"Hello, my name is $name, I am $age%d years old, and I am $height%.2f cm tall."
println(message) // Output: Hello, my name is Bob, I am 25 years old, and I am 175.50 cm tall.
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val name = "Bob"
val age = 25
val height = 175.5
val message = f"Hello, my name is $name, I am $age%d years old, and I am $height%.2f cm tall."
println(message)

// Exiting paste mode, now interpreting.

Hello, my name is Bob, I am 25 years old, and I am 175.50 cm tall.
name: String = Bob
age: Int = 25
height: Double = 175.5
message: String = Hello, my name is Bob, I am 25 years old, and I am 175.50 cm tall.

scala>
```

raw Interpolation: This type of interpolation is similar to s interpolation, but it performs minimal processing of escape characters like \n or \t. It is denoted by the prefix raw.

```
val path = raw"c:\user\numbers.txt"
println(path) // Output: c:\user\numbers.txt
```

**Finding
Pattern in
String**

"ab*c" "ab{2,5}c" "[a-zA-Z]"

"ab+c" "[abc]" "[1-9]"

"ab{2}c" "[abc]+" "[1-9]+"

ab*c	a: Matches the character 'a' literally.
------	---

	<p>b*: Matches 'b' zero or more times.</p> <p>c: Matches the character 'c' literally.</p> <p>This pattern will match strings like "ac", "abc", "abbc", "abbbc", and so on.</p>
ab+c	<p>a: Matches the character 'a' literally.</p> <p>b+: Matches 'b' one or more times.</p> <p>c: Matches the character 'c' literally.</p> <p>This pattern will match strings like "abc", "abbc", "abbbc", and so on, but not "ac".</p>
ab{2}c	<p>a: Matches the character 'a' literally.</p> <p>b{2}: Matches 'b' exactly 2 times.</p> <p>c: Matches the character 'c' literally.</p> <p>This pattern will match strings like "abbc"</p>
ab{2,5}c	<p>a: Matches the character 'a' literally.</p> <p>b{2,5}: Matches 'b' between 2 and 5 times.</p> <p>c: Matches the character 'c' literally.</p> <p>This pattern will match strings like "abbc", "abbbc", "abbbbc", "abbbbbc", but not "abc" or "abbbbbbc".</p>
[abc]	<p>[abc]: Matches any character in the set {'a', 'b', 'c'}.</p> <p>This pattern will match any single character among 'a', 'b', and 'c'.</p>
[abc]+	<p>[abc]+: Matches one or more occurrences of any character in the set {'a', 'b', 'c'}.</p> <p>This pattern will match strings like "a", "abc", "abca", "b", "bc", "cc", and so on.</p>
[a-zA-Z]	<p>[a-zA-Z]: Matches any alphabetic character, either uppercase or lowercase.</p> <p>This pattern will match any single alphabetic character.</p>
[1-9]	<p>[1-9]: Matches any digit from 1 to 9.</p> <p>This pattern will match any single digit except 0.</p>

[1-9]+

[1-9]+: Matches one or more occurrences of any digit from 1 to 9.
This pattern will match strings like "1", "123", "9", and so on.

REGULAR EXPRESSION

```
val expressionToFind = "the".r
```

```
val stringToFindExpression = "the Big Data Channel"
```

```
val match1 = expressionToFind.findFirstIn(stringToFindExpression)
```

```
match1.foreach(println)
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val expressionToFind = "the".r
val stringToFindExpression = "the Big Data Channel"
val match1 = expressionToFind.findFirstIn(stringToFindExpression)
match1.foreach(println)

// Exiting paste mode, now interpreting.

the
expressionToFind: scala.util.matching.Regex = the
stringToFindExpression: String = the Big Data Channel
match1: Option[String] = Some(the)

scala>
```

```
val expressionToFind = "[1-5]+".r
```

```
val stringToFindExpression = "12 67 93 48 51"
```

```
val match1 = expressionToFind.findAllIn(stringToFindExpression)
```

```
match1.foreach(println)
```



```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val expressionToFind = "[1-5]+".r
val stringToFindExpression = "12 67 93 48 51"
val match1 = expressionToFind.findAllIn(stringToFindExpression)
match1.foreach(println)

// Exiting paste mode, now interpreting.

12
3
4
51
expressionToFind: scala.util.matching.Regex = [1-5]+
stringToFindExpression: String = 12 67 93 48 51
match1: scala.util.matching.Regex.MatchIterator = <iterator>

scala>
```

```
val expressionToFind = "[1-5]{2}+".r
val stringToFindExpression = "12 67 93 48 51"
val match1 = expressionToFind.findAllIn(stringToFindExpression)
match1.foreach(println)
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val expressionToFind = "[1-5]{2}+".r
val stringToFindExpression = "12 67 93 48 51"
val match1 = expressionToFind.findAllIn(stringToFindExpression)
match1.foreach(println)

// Exiting paste mode, now interpreting.

12
51
expressionToFind: scala.util.matching.Regex = [1-5]{2}+
stringToFindExpression: String = 12 67 93 48 51
match1: scala.util.matching.Regex.MatchIterator = <iterator>

scala>
```

Replacing Patterns in Strings

```
String Literal.replaceFirst("SearchExpression","ReplaceExpression")
```

```
Regex.replaceFirstIn("SearchString","ReplaceExpression")
```

```
String Literal.replaceAll("SearchExpression","ReplaceExpression")
```

```
Regex.replaceAllIn("SearchString","ReplaceExpression")
```

REPLACING STRING

```
val replaceIn = "8201530"
```

```
val replaced = replaceIn.replaceFirst("[01]","X")
```

```
println(replaced)
```

```
scala> val replaceIn = "8201530"
replaceIn: String = 8201530

scala> val replaced = replaceIn.replaceFirst("[01]", "X")
replaced: String = 82X1530

scala>

scala> println(replaced)
82X1530

scala>
```

```
val regularExp = "H".r
val replaceIn = "Hello World!"
val replaced = regularExp.replaceFirstIn(replaceIn, "J")
println(replaced)
```

```
scala> val regularExp = "H".r
regularExp: scala.util.matching.Regex = H

scala> val replaceIn = "Hello World!"
replaceIn: String = Hello World!

scala> val replaced = regularExp.replaceFirstIn(replaceIn, "J")
replaced: String = Jello World!

scala> println(replaced)
Jello World!

scala>
```

```
val replaceIn = "8201530"
val replaced = replaceIn.replaceAll("[01]", "X")
println(replaced)
```

```
scala> val replaceIn = "8201530"
replaceIn: String = 8201530

scala> val replaced = replaceIn.replaceAll("[01]","X")
replaced: String = 82XX53X

scala> println(replaced)
82XX53X

scala>
```

```
val regularExp = "[a-z]+".r
val replaceIn = "dk79rx5c4lj2c8ge"
val replaced = regularExp.replaceAllIn(replaceIn,"1")
println(replaced)
```

```
scala> val regularExp = "[a-z]+".r
regularExp: scala.util.matching.Regex = [a-z]+

scala> val replaceIn = "dk79rx5c4lj2c8ge"
replaceIn: String = dk79rx5c4lj2c8ge

scala> val replaced = regularExp.replaceAllIn(replaceIn,"1")
replaced: String = 179151412181

scala> println(replaced)
179151412181

scala>
```

Methods for Comparing Strings

```
variableName.matches(regular expression)
```

```
variableName.equals(variableName)
```

```
String1.compareTo(String2)
```

```
variableName.equalsIgnoreCase(variableName)
```

COMPARING STRING

```
val stringMatch = "ab7"
```

```
val match1 = stringMatch.matches("[a-zA-Z0-9]{4}")
```

```
println(match1)
```

```
scala> val stringMatch = "ab7"  
stringMatch: String = ab7
```

```
scala> val match1 = stringMatch.matches("[a-zA-Z0-9]{4}")  
match1: Boolean = false
```

```
scala> println(match1)  
false
```

```
scala>
```

```
val string1 = "UBD"
```

```
val string2 = "UBD"
```

```
val comparingStrings = string1.equals(string2)
```

`println(comparingStrings)`

```
scala> val string1 = "UBD"
string1: String = UBD

scala> val string2 = "UBD"
string2: String = UBD

scala> val comparingStrings = string1.equals(string2)
comparingStrings: Boolean = true

scala> println(comparingStrings)
true

scala>
```

`val string1 = "This is Scala"`

`val string2 = "Hello Scala"`

`val string3 = "Hello Scala"`

`val lexiCompare1 = string1.compareTo(string2)`

`val lexiCompare2 = string2.compareTo(string3)`

```
scala> val string1 = "This is Scala"
string1: String = This is Scala

scala> val string2 = "Hello Scala"
string2: String = Hello Scala

scala> val string3 = "Hello Scala"
string3: String = Hello Scala

scala> val lexiCompare1 = string1.compareTo(string2)
lexiCompare1: Int = 12

scala> val lexiCompare2 = string2.compareTo(string3)
lexiCompare2: Int = 0

scala>
```

```
val string1 = "UBD"
val string2 = "ubd"
val comparingStrings = string1.equalsIgnoreCase(string2)
println(comparingStrings)
```

```
scala> val string1 = "UBD"
string1: String = UBD

scala> val string2 = "ubd"
string2: String = ubd

scala> val comparingStrings = string1.equalsIgnoreCase(string2)
comparingStrings: Boolean = true

scala> println(comparingStrings)
true

scala>
```

Classes and Objects in Scala

Classes

A class is a blueprint for creating objects. It defines the properties (fields) and behaviors (methods) that objects of the class will have. You can think of a class as a template or a cookie-cutter that defines the structure and behavior of objects.

Imagine an architectural blueprint for a house. It defines the structure, layout, and features of the house but doesn't exist as a physical entity. It's a template for creating actual houses.

Objects

An object is an instance of a class. It is a concrete realization of the blueprint defined by the class. Objects have their own unique state (values of fields) and behavior (execution of methods). You can create multiple objects from the same class, each with its own distinct state.

An object is like an actual house built based on the blueprint. Each house (object) may have the same layout (defined by the blueprint) but can have different furniture, decorations, and inhabitants, making each house unique.

```
// Define a class representing a Car
class Car(val make: String, var speed: Int) {

    // Method to accelerate the car
    def accelerate(incrSpeed: Int): Unit = {
        speed += incrSpeed
        println(s"The $make is accelerating. Current speed: $speed km/h")
    }

    // Method to brake the car
    def brake(decrSpeed: Int): Unit = {
        speed -= decrSpeed
        println(s"The $make is braking. Current speed: $speed km/h")
    }
}

object Main {
    def main(args: Array[String]): Unit = {
        // Create objects of the Car class
        val ferrari = new Car("Ferrari", 0)
        val tesla = new Car("Tesla", 0)
    }
}
```



```
// Accelerate the Ferrari

ferrari.accelerate(50)

ferrari.accelerate(20)


// Brake the Tesla

tesla.accelerate(30)

tesla.brake(10)

}

}
```

- We define a class Car with fields make (manufacturer) and speed, and methods accelerate and brake.
- We create two objects ferrari and tesla of type Car.
- We invoke methods on these objects to manipulate their state (speed).

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// Define a class representing a Car
class Car(val make: String, var speed: Int) {
  // Method to accelerate the car
  def accelerate(incrSpeed: Int): Unit = {
    speed += incrSpeed
    println(s"The $make is accelerating. Current speed: $speed km/h")
  }

  // Method to brake the car
  def brake(decrSpeed: Int): Unit = {
    speed -= decrSpeed
    println(s"The $make is braking. Current speed: $speed km/h")
  }
}

object Main {
  def main(args: Array[String]): Unit = {
    // Create objects of the Car class
    val ferrari = new Car("Ferrari", 0)
    val tesla = new Car("Tesla", 0)

    // Accelerate the Ferrari
    ferrari.accelerate(50)
    ferrari.accelerate(20)

    // Brake the Tesla
    tesla.accelerate(30)
    tesla.brake(10)
  }
}

// Exiting paste mode, now interpreting.

defined class Car
defined object Main
```

Collection Library in Scala

The Collection Library in Scala provides a rich set of data structures for working with collections of elements. These collections can be manipulated, transformed, and processed efficiently. The library is highly flexible and offers immutable and mutable collections, as well as sequential and parallel collections.

Immutable Collections: Immutable collections **cannot be modified** after they are created. They provide safety in concurrent programming and functional programming paradigms.

Mutable Collections: Mutable collections **can be modified** after creation, allowing for in-place modifications. While they offer mutability, they require careful handling, especially in concurrent environments.

Lists	<p>Ordered collections with fast prepend and access to the head element.</p> <pre>// Define a list of integers val numbers = List(1, 2, 3, 4, 5) // Print the first element of the list println("First element of the list: " + numbers.head) Method#1 val fruitList = List("orange","banana","apple","grape") fruitList.foreach(println) Method#2 val intList = List.range(1,10) intList.foreach(println)</pre>
--------------	---

Method#3

```
val MyList = List.fill(3)("UBD")
```

```
MyList.foreach(println)
```

//Appending Elements

Using ++ operator:

```
val list1 = List("banana", "apple")
```

```
val list2 = List("grape", "orange")
```

```
val appendedList = list1 ++ list2
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val list1 = List("banana", "apple")
val list2 = List("grape", "orange")
val appendedList = list1 ++ list2

// Exiting paste mode, now interpreting.

list1: List[String] = List(banana, apple)
list2: List[String] = List(grape, orange)
appendedList: List[String] = List(banana, apple, grape, orange)

scala> appendedList.foreach(println)
banana
apple
grape
orange

scala>
```

Using ::: method:

```
val list1 = List("banana", "apple")
```

```
val list2 = List("grape", "orange")
```

```
val appendedList = list1 ::: list2
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val list1 = List("banana", "apple")
val list2 = List("grape", "orange")
val appendedList = list1 ::: list2

// Exiting paste mode, now interpreting.

list1: List[String] = List(banana, apple)
list2: List[String] = List(grape, orange)
appendedList: List[String] = List(banana, apple, grape, orange)

scala> appendedList.foreach(println)
banana
apple
grape
orange

scala>
```

//Prepending Elements

```
val originalList = List("banana", "apple", "grape")
```

```
val newList = "orange" :: originalList
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val originalList = List("banana", "apple", "grape")
val newList = "orange" :: originalList

// Exiting paste mode, now interpreting.

originalList: List[String] = List(banana, apple, grape)
newList: List[String] = List(orange, banana, apple, grape)

scala> newList.foreach(println)
orange
banana
apple
grape

scala> _
```

//List Concatenation

```
val fruitList = "orange"::"banana"::"apple"::"grape"::Nil
val fruitList2 = fruitList :+ "peach"
```

```
scala> val fruitList = "orange"::"banana"::"apple"::"grape"::Nil
fruitList: List[String] = List(orange, banana, apple, grape)

scala> val fruitList2 = fruitList :+ "peach"
fruitList2: List[String] = List(orange, banana, apple, grape, peach)

scala> fruitList2.foreach(println)
orange
banana
apple
grape
peach

scala> _
```

```
val fruitList3 = "watermelon" :: fruitList2
val fruitList4 = "mango" +: fruitList3
```

```
scala> val fruitList = "orange"::"banana"::"apple"::"grape"::Nil
fruitList: List[String] = List(orange, banana, apple, grape)

scala>

scala> val fruitList2 = "peach" +: fruitList
fruitList2: List[String] = List(peach, orange, banana, apple, grape)

scala> fruitList2.foreach(println)
peach
orange
banana
apple
grape

scala> _
```

```
val twoFruits = "pear"::"apricot":Nil
val fruitList5 = twoFruits ::: fruitList4
```

```
fruitList5.foreach(println)
```

```
//Head & Tail
```

```
val getHead = fruitList5.head
```

```
val getTail = fruitList5.tail
```

```
scala> fruitList2.foreach(println)
```

```
peach
```

```
orange
```

```
banana
```

```
apple
```

```
grape
```

```
scala> val getH = fruitList2.head
```

```
getH: String = peach
```

```
scala> val getT = fruitList2.tail
```

```
getT: List[String] = List(orange, banana, apple, grape)
```

```
scala> _
```

Sets Collections of distinct elements **without duplicates**.

```
// Define a set of characters
```

```
val uniqueCharacters = Set('a', 'b', 'c', 'a', 'd')
```

```
// Print the set of unique characters
```

```
println("Set of unique characters: " + uniqueCharacters)
```

	<pre>scala> val uniqueCharacters = Set('a', 'b', 'c', 'a', 'd') uniqueCharacters: scala.collection.immutable.Set[Char] = Set(a, b, c, d) scala> println("Set of unique characters: " + uniqueCharacters) Set of unique characters: Set(a, b, c, d) scala></pre>
Maps	<p>Key-value pairs where each key is unique.</p> <pre>// Define a map of employee names and ages val employeeAges = Map("Alice" -> 30, "Bob" -> 35, "Charlie" -> 40) // Print the age of Bob println("Age of Bob: " + employeeAges("Bob"))</pre> <pre>scala> :paste // Entering paste mode (ctrl-D to finish) // Define a map of employee names and ages val employeeAges = Map("Alice" -> 30, "Bob" -> 35, "Charlie" -> 40) // Print the age of Bob println("Age of Bob: " + employeeAges("Bob")) // Exiting paste mode, now interpreting. Age of Bob: 35 employeeAges: scala.collection.immutable.Map[String,Int] = Map(Alice -> 30, Bob -> 35, Charlie -> 40) scala> █</pre>
Seq	<p>General sequence, which includes List, Array, and Vector.</p>


```
// Define a sequence of strings
val fruitsSeq: Seq[String] = Seq("Apple", "Banana", "Orange")

// Print the elements of the sequence
println("Fruits sequence: " + fruitsSeq)
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// Define a sequence of strings
val fruitsSeq: Seq[String] = Seq("Apple", "Banana", "Orange")

// Print the elements of the sequence
println("Fruits sequence: " + fruitsSeq)

// Exiting paste mode, now interpreting.

Fruits sequence: List(Apple, Banana, Orange)
fruitsSeq: Seq[String] = List(Apple, Banana, Orange)

scala>
```

Array Mutable, fixed-size sequential collections.

```
// Define an array of integers
val numbersArray = Array(1, 2, 3, 4, 5)

// Access and print the third element of the array
println("Third element of the array: " + numbersArray(2))

val array1 = Array.range(0,5)

// range(start, number of elements)
```

```
array1.foreach(println)
```

```
val array3 = Array.fill(2)("UBD")
```

```
//fill(number of elements)(value)
```

```
array3.foreach(println)
```

```
val array4 = "hello".toArray
```

```
// converts arguments to an array
```

```
array4.foreach(println)
```

```
val intArray = Array(17, 34, 23, 6, 50)
```

```
val len = intArray.length
```

```
println(len)
```

```
//new keyword
```

```
val myArray = new Array[String](3)
```

```
myArray(0) = "red"
```

```
myArray(1) = "blue"
```

```
myArray(2) = "yellow"
```

```
myArray.foreach(println)
```

```

scala> :paste
// Entering paste mode (ctrl-D to finish)

//new keyword
val myArray = new Array[String](3)
myArray(0) = "red"
myArray(1) = "blue"
myArray(2) = "yellow"
myArray.foreach(println)

// Exiting paste mode, now interpreting.

red
blue
yellow
myArray: Array[String] = Array(red, blue, yellow)

scala> _

```

Vector Immutable, indexed, and effectively constant-time random access.

```

// Define a vector of characters
val charactersVector = Vector('a', 'b', 'c', 'd', 'e')

// Access and print the fourth element of the vector
println("Fourth element of the vector: " + charactersVector(3))

// Creating a Vector
val numVector = Vector(1,2,3,4,5)
numVector.foreach(println)

```

```
//Initializing empty vector
val emptyVector = Vector.empty

//Appending an element
val pVector = Vector("a","b","c")
val pVector2 = pVector :+ "d"
pVector2.foreach(println)

//Prepending an element
val pVector3 = "z" +: pVector2

//Vector concatenation
val tempVector = Vector("e","f")
val pVector4 = pVector3 ++ tempVector
```

Queue Immutable first-in, first-out (FIFO) data structure.

```
// Import Queue class from scala.collection.immutable package
import scala.collection.immutable.Queue

// Define a queue of integers
val intQueue = Queue(1, 2, 3, 4, 5)
```

```
// Print the elements of the queue  
println("Queue of integers: " + intQueue)
```

```
scala> import scala.collection.immutable.Queue  
import scala.collection.immutable.Queue  
  
scala> val intQueue = Queue(1, 2, 3, 4, 5)  
intQueue: scala.collection.immutable.Queue[Int] = Queue(1, 2, 3, 4, 5)  
  
scala> println("Queue of integers: " + intQueue)  
Queue of integers: Queue(1, 2, 3, 4, 5)  
  
scala> _
```

Stack Immutable last-in, first-out (LIFO) data structure.

```
// Import Stack class from scala.collection.immutable package  
import scala.collection.immutable.Stack  
  
// Define a stack of strings  
val stringStack = Stack("first", "second", "third")  
  
// Print the elements of the stack  
println("Stack of strings: " + stringStack)
```

```
scala> import scala.collection.immutable.Stack
import scala.collection.immutable.Stack

scala> val stringStack = Stack("first", "second", "third")
stringStack: scala.collection.immutable.Stack[String] = Stack(first, second, third)

scala> println("Stack of strings: " + stringStack)
Stack of strings: Stack(first, second, third)

scala> _
```

Array Buffer in Scala \

In Scala, Array Buffer is a mutable sequence that **serves as a resizable array**. It provides constant-time amortized complexity for appending and updating elements at the end of the buffer, making it efficient for dynamic collections where the size is expected to change frequently.

In Scala, an ArrayBuffer is a mutable collection that stores elements in an array-like structure, allowing for efficient random access and modification. Think of it as a resizable array, where you can add, update, or remove elements dynamically without needing to reallocate memory.

Imagine you have a whiteboard where you can write down items sequentially. Initially, you allocate space for a few items, but as you run out of space, you can easily erase and write new items in the available space without needing a new whiteboard.

The advantage of an ArrayBuffer over a regular array is its dynamic resizing capability. You don't need to specify the size upfront, and it can grow or shrink as needed, making it suitable for situations where the number of elements is not known in advance or may change over time.

Commands

```
import scala.collection.mutable.ArrayBuffer

// Create an empty ArrayBuffer of integers
val numbers = ArrayBuffer[Int]()

// Append elements to the buffer
numbers += 1
numbers += 2
numbers += 3

// Append multiple elements at once
numbers ++= Seq(4, 5, 6)

// Insert an element at a specific index
numbers.insert(2, 10)

// Remove an element at a specific index
numbers.remove(3)

// Update an element at a specific index
numbers(1) = 20

// Print the elements of the buffer
println("ArrayBuffer elements: " + numbers)

// Access individual elements
println("First element: " + numbers.head)
```

```
println("Last element: " + numbers.last)

// Check if the buffer contains a specific element
println("Contains 3: " + numbers.contains(3))

// Iterate over the elements of the buffer
println("Iterating over elements:")
for (num <- numbers) {
    println(num)
}

// Create an ArrayBuffer
val numbers = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)

// Removing by Index
numbers.remove(3) // Remove element at index 3
println("After removing element at index 3: " + numbers)

// Removing by Value
numbers -= 6 // Remove all occurrences of value 6
println("After removing all occurrences of value 6: " + numbers)

// Removing a Range of Elements
numbers.remove(1, 3) // Remove elements from index 1 to index 3 (inclusive)
println("After removing elements from index 1 to index 3: " + numbers)
```



```

scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> val numbers = ArrayBuffer[Int]()
numbers: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> numbers += 1
res15: numbers.type = ArrayBuffer(1)

scala> numbers += 2
res16: numbers.type = ArrayBuffer(1, 2)

scala> numbers += 3
res17: numbers.type = ArrayBuffer(1, 2, 3)

scala> numbers += 4
res18: numbers.type = ArrayBuffer(1, 2, 3, 4)

scala> numbers += 5
res19: numbers.type = ArrayBuffer(1, 2, 3, 4, 5)

scala> numbers += 6
res20: numbers.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

scala> numbers ++= Seq(7,8,9)
res21: numbers.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> numbers.insert(2, 10)

scala> numbers.foreach(println)
1
2
10
3
4
5
6
7
8
9

scala> numbers.remove(3)
res24: Int = 3

```

```

scala> numbers.foreach(print)
1210456789
scala> numbers.foreach(println)
1
2
10
4
5
6
7
8
9

scala> numbers(1) = 20

scala> numbers.foreach(println)
1
20
10
4
5
6
7
8
9

scala> println("ArrayBuffer elements: " + numbers)
ArrayBuffer elements: ArrayBuffer(1, 20, 10, 4, 5, 6, 7, 8, 9)

scala> println("First element: " + numbers.head)
First element: 1

scala> println("Last element: " + numbers.last)
Last element: 9

scala> println("Contains 3: " + numbers.contains(3))
Contains 3: false

scala> _

```

Lazy List in Scala

In Scala, LazyList is a data structure that represents a potentially infinite sequence of elements. It's similar to Stream but designed to be more efficient and tail-recursive. LazyList evaluates elements lazily, meaning it computes elements only when they are needed, allowing for efficient handling of potentially infinite sequences without consuming memory for elements that are not accessed.

Imagine you have a library with an endless number of books, and each book represents an element in your lazy list. However, you don't have enough space in your reading room to accommodate all the books at once. So, you decide to bring in books only when you need them, read them, and then return them to the library. This way, you optimize space usage in your reading room.

```
import scala.collection.immutable.LazyList
```

```
val myFirstLazyList = 1.5 #:: 2.5 #:: 3.5 #:: LazyList.empty  
myFirstLazyList(println)
```

```
val wholeList = "orange"::"banana"::"apple"::"grape"::Nil  
print(wholeList)
```

```
val wholeLazyList = "orange" #:: "banana" #:: "apple" #:: "grape" #:: LazyList.empty  
print(wholeLazyList)
```

```
val list = List.from(1 to 1000000000)  
val lazyList = LazyList.from(1 to 1000000000)
```

```
val myFirstStream = 1.5 #:: 2.5 #:: 3.5 #:: Stream.empty
```

```
val ms = Stream.from(1).take(1000)
```