King Abdul-Aziz University
Faculty of Computing and Information Technology
Computer Science Department
CPCS-301
Term 1- Fall 2022

GROUP 3		
Name	Id	Section
Nsreen Hujatullah Asadullah		B9A
Aya		B9A
Noor		B9A
Ghadeer		B9A

Table of Contents

Table of tables

Table of figures

1. Project Overview

Through this project, we will discuss the Scala programming language in all its details through that we can understand it more accurately. We'll start the research with an overview of when Scala began, for what reason its programmers developed it, its features, and where to use it. After that, we will study the implementation method used to develop Scala. For more, we will discuss the domain and fields in which Scala is used, examples of which are when designing web applications and most importantly, it is mainly used in big data processing.

We will also discuss topics that advance our knowledge of Scala such as BNF and EBNF identifiers, variable declarations, expressions, select statements, and loops that the compiler uses to evaluate developed programs. To evaluate Scala, we will study three main criteria: readability, writability, and reliability, and we will discuss in detail all the properties that affect each of these criteria. Of course, we will not forget by defining the types of variables that Scala provides, then the types of arrays available in Scala, and finally, we will finish our research by defining the types of scopes used in Scala.

2. Background

Scala is a measurable language. It is a modern, multi-paradigm programming language that is designed to express common programming patterns in an elegant, concise, and type-safe manner. It is a very expressive language and most importantly, it is highly scalable as it is easy to use and learn, which contributes to being abreast of changes in various fields. This language seamlessly integrates the features of object-oriented and functional languages. The name “Scala “is derived from the word scalable which means it can grow with the need of users. (Introduction, n.d.) (Scala Tutorial, n.d.)

2.1. Historical overview

Scala is a modern programming language. The initial design of Scala began in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) and was created by Martin Odersky. Before developing Scala, Martin had experience working on Sun's Java compiler, Generic Java and javac. (History of Scala - Javatpoint, 2010)

The language was first released in 2003 and this was accompanied by the first public release of Scala at the beginning of 2004. Scala was officially released with some better features on the Java and NET platforms in June 2004. After that, version 2.0 of the language was published in March 2006. Developments in the language led to the official discontinuation of Net support in 2012. (Scala, 2016)

Scala's design has been influenced by many ideas in programming research and programming languages . In fact, Martin himself said: "that very few of Scala's features can actually be described as new," and in Scala, "the innovation comes mainly from how its architectures are put together." It was basically designed to be a better language". (Scala: A History - Learn Scala from Scratch, n.d.)

As a great move, Typesafe which is an organization with the objective of providing business support, training and other services associated with Scala. Because of the consistent developments made in the language, the Scala team has been awarded a research grant of €2.3 million from the European Research Council. (Scala, 2016)

2.2. Why Scala?

- It combines the performance and extensive maintainability associated with compiled languages, tools, the Java language ecosystem, and a virtual machine. (From First Principles: Why Scala?, n.d.)
- It allows you to reuse your existing skills so you can focus your attention on the actual task at hand. (From First Principles: Why Scala?, n.d.)
- Scala has the potential to grow in the technical aspects of the software, as it provides many features that can be used in the field of data analysis that a wide range of companies may use, such as LinkedIn, Intel, and Twitter. (Scala, 2016)

- It provides a very powerful static system supporting functional programming.
- The language is designed in such a way that it is concise and practical.
- Java's pitfalls have been used as a basis for developing a number of design decisions for Scala. (Scala, 2016)
- Scala is easy to learn because it is a high-level programming language, similar to Java, C and C++. (Scala, 2016)
- The ability to run this language on a JVM allows Scala to handle many operational and monitoring tools.

2.3. The difference between Scala and Java

Factors	Scala	Java
Type	Scala is a combination of both object-oriented and functional programming.	Java is a general-purpose object-oriented language.
Readability	Readability is less in Scala.	Readability is more in Java.
Code compilation	The process of compiling source code into bytes is slow.	The process of compiling source code into byte code is fast.
Lazy evaluation	Scala supports lazy evaluation.	Java does not support lazy evaluation.
Backward Compatible	Scala is not backward compatible, which means the code runs in the latest version only.	Java is backward compatible, it allows code to run on older and newer versions also without any error.
Operator Overloading feature	Scala support operator overloading.	Java does not support operator overloading.
Function- type	In Scala, every method or function is treated as a variable.	Java treats every function as an object.
Variables-type	In Scala, variables are by default of immutable type.	In Java, variables are by default of mutable type.
Code structure	Here, the code is written in a compact form.	Here, the code is written in the long-form.
Operator	All the operators on entities in Scala are done by using method calls.	In Java, all the operators are treated uniquely and are not done with a method call.

Table 1 The difference between Scala and Java

(Scala vs Java | Comparison between Scala and Java, n.d.)

2.4. Implementation method

There are some methods of Implementing computer programs because they should be represented using machine language "0 and 1".These methods are compiler and interpreter and both translate high-level language to low-level language to help the computer understand the instructions we write. They do the same Job but in different ways, So we will discuss their meaning, examples, advantages, and disadvantages for both compiler and interpreter to understand which one is good at what. (FreeCodeCamp, 2020)

2.4.1. Compiled languages

The compiled languages are converted directly into machine code that the processor can execute. So, it is faster and more efficient to implement than interpreted languages. It also gives the developer control over hardware aspects, such as CPU usage and memory management. Compiled languages need a 'create' step, as they must be compiled manually first. You also need to "rebuild" the program every time you need to make a change. Examples of purely compiled languages are C, C++, Erlang, Haskell, Rust, and Go. (FreeCodeCamp, 2020)

2.4.1.1. Compilation advantages

- Since the source code is already compiled and we only need to execute the generated executable it runs faster than the interpreter.
- It is often the machine code of an executable file created with a native machine help interpreter for the target machine that is well-optimized and runs faster.
- It makes your software unhackable, secure and private, as executable files generated from the compiler can be executed on any of your clients or other systems without the need for actual source code.
- In order to execute the common executable file for your source code. Neither your customer nor anyone else will need any translator, interpreter or third-party software to be installed in their system. (Advantages and Disadvantages of Compiler and Interpreter - Buggy Programmer, n.d.)

2.4.1.2. Compilation disadvantages

- It takes up additional memory since it needs to create a new file.
- Unlike the interpreter, we can't run our source code directly, we also need to run the executable.
- Since its code is optimized for the system on which it is implemented, this can lead to system incompatibility issues even if it is running on the same operating system.
- After reading the entire code, it returns all errors at once if available, making it difficult to fix and maintain. (Advantages and Disadvantages of Compiler and Interpreter - Buggy Programmer, n.d.)

2.4.2. Interpreted Languages

Interpreted languages are programming languages for which instructions are not precompiled for the target machine in a machine-readable form. For more, these languages are assisted by an interpreter. An interpreter is a program that translates high-level, human-readable source code into low-level, machine-readable target code line by line. It is slower and less efficient than compiled languages because the interpreter must be present for the entire process, but these languages are also highly adaptable. So, with the development of just-in-time compilation, that gap is shrinking. Examples of commonly interpreted languages are PHP, Ruby, Python, and JavaScript. (FreeCodeCamp, 2020)

2.4.2.1. Advantages Of Interpreter

- In the interpreted language, we share the source code directly which contributes to its running on any system without any problem of system incompatibility.
- Because it reads the code line by line it is easier to debug the code in the interpreters, and it returns the error message immediately. Also, the customer who owns the source code can easily debug or modify the code.

- Unlike a compiler, interpreters do not create new separate files. So there is no need for additional memory and not even one extra step to execute the source code, it is executed immediately.
- To be able to stop execution and release the code at any time the interpreter is reading the code line by line, which is not possible in a compiled language. (Advantages and Disadvantages of Compiler and Interpreter - Buggy Programmer, n.d.)

2.4.2.2. Disadvantages of the Interpreter

- Since it reads analyzes and converts code line by line, the interpreter is often slower than the compiler.
- In order to execute the code, the client or anyone with the shared source code needs to have an interpreter installed in their system.
- Unlike compiled languages, the interpreter does not create any executable file, so when you want to share the program with others, you need to share your source code which is not secure and private. So it is not very good in terms of privacy. (Advantages and Disadvantages of Compiler and Interpreter - Buggy Programmer, n.d.)

2.5. PL Domains

Scala is considered to be a relatively new language in the tech market, because it has so many features, like the fact that it allows programmers to explore both sides of OOP simultaneously functional programming . It can also run on a java virtual machine (JVM), that means it enables the users of Scala to use all libraries of java directly from the Scala code and to alternate easily between the two languages and write segments of codes using both languages without complexity (Pedamkar, 2018).

2.5.1. Where is Scala used?

Scala is mostly used by software engineers and data engineers. You'll see some data scientists use it with Apache Spark to process big data m so the top uses of Scala are: (Team, 2021)

- **Multi-paradigm language**

  Scala is a language that supports both object-oriented programming and functional programming. Learning this builds imperative, logical, functional, and OOP skills. Scala enables you to define different types associated with both data and behavioural attributes. (Uses of Scala | Top 10 Useful Uses of Scala in Real World, 2018)
- **It can be used in integration with Java**

  Scala runs on the Java Virtual Machine (JVM). This allows the Scala developer to use all the Java libraries directly from the Scala code and vice versa. This feature allows the user to write code in Java and Scala and work together in both languages. Scala has good libraries, and good online documentation and is used by a lot of people in the industry. (Uses of Scala | Top 10 Useful Uses of Scala in Real World, 2018)
- **Built-in language patterns**

  Scala aims to create new innovations in programming language research to popularize languages such as Java. This language already has some best practices and patterns built into the language. In addition to that, it also offers to adopt new languages like Python, Ruby, etc. to implement functional programming. (Uses of Scala | Top 10 Useful Uses of Scala in Real World, 2018)
- **High market demand**

  The main reason or reason to use Scala is for better growth and function. Learning Scala will increase demand and make you more marketable. Many companies like Twitter, LinkedIn, Foursquare and others use Scala. All investment banks and financial institutions use Scala due to its scalable nature. There are many companies that share effective ways to use Scala. (Uses of Scala | Top 10 Useful Uses of Scala in Real World, 2018)
- **The statically typed language**

  A statically typed language avoids errors in the code, helps programmers to write appropriate code, and enables them to debug the code easily. Scala's uses provide the best of both static and dynamic languages. Scala provides type inference for variables and functions, which is much better than limited type inference in Java and C#. (Uses of Scala | Top 10 Useful Uses of Scala in Real World, 2018)

- **Grammar accuracy**

  Another use of Scala is that it has a very fine structure. Java has a very long syntax. Scala is more readable and concise at the same time.

3. BNF/EBNF

BNF (Backus-Naur Form) is a formal way for explaining the syntax of any programming languages. In addition, it's a language that explains another language. It's firstly appeared in 1960 by John Bakus ans Peter Naur (BNF Notation in Compiler Design, 2020)

3.1. Identifiers

Identifiers: Naming the entity of your language' contents such as variables, labels, functions, and types is the definition of identifier. One of the main rules to name an identifier is to make it referring to the meaning of the value it's holding

The rules in Scala BNF of identifier:

1 -It should start with only:

a- A letter (lower-case or upper-case)

b- Underscore"_"

c- Dollar sign"$"

2 -After the first character, the user can add :

a- Letters (lower-case or upper-case)

b- Digits

c- Underscores"_"

b- Dollar signs"$"

(corob-msft, n.d.)

BNF of identifiers:
<Identifier> → <Begin> | <Begin> <More>
< Begin> → <Letter> | “$” | “_”
<More> → <End> | <More> <End>
<End> → <Letter> | “$” | “_”
<Letter> → “a” | “b” | “c” | “d” | ….. | “z” | “A” | “B” | ….. | “Z”
<Digit> → “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” |

Table 2 BNF of identifiers

For instance, “63ghadeer” is a not valid Scala identifier

Here is an example of how to use the BNF above when evaluating the following identifier: G22

<Identifier> → <begin><more>

→ <letter><more>

→ G<more>

→ G<more><end>

→ G<end><end>

→ G<digit><end>

→ G2<end>

→ G2<digit>

→ G22

And here is an instance of identifiers in coding in Scala:

```
object identifier

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

var num1 = 20

var num_2 = 30 println("The total of two values is " + (num1 + num_2)) } }
```

Output --> The sum of two values is 50

All identifiers in this program are: num1, num_2, identifier,main, args.14

3.2. Variable declaration:

Variables are the computer memory locations that you naming them and storing values in the program, In Scala, declaring variables has two types:

1- Define it as a value, using the keyword "val", and the value on it can't be modified, so it's an immutable variable, and the first letter of it should be in lower-case, for example:

val name : String = "Ghadeer";

2- Define it as a variable, using the keyword "var", and the value on it can be modified, so it's an mutable variable, for example:

var name : String = "Ghadeer";

3- Constant variable, using the keyword "val", the value of it couldn't be modified, the first letter of the constant variable should be in upper-case, for example:

val Name : String = "Ghadeer";

Specifying the type of variable is declared before the equal sign "=" and after variable name. In addition, there are the ability to define the type of variable and specify it like this:

var age : Int = 10;

It's also valid in variables to not give an initial value, for instance:

var age :Int;

val name :String;

And as we said before, Scala declare the type of variable by the given value of it

The rules of naming variables:

- Should start in lower case letter
- It could contain letters, dollar signs "$", underscores "_", and digits
- It couldn't be a reserved word in the programing language or a keyword
- It should start with a letter in alphabet
- It shouldn't contain any white space

Note: multiple assignments could be used in Scala, but it's just allowed in immutable variables (Variables in Scala, 2019)

BNF of variable declaration:
<Declare> → <Immutability> <Immutability> → “val” <Varname> | “var” <Varname> < Varname > → name”:” < DataType > “=” value[“;”] |name “=” value [“;”] <DataType> → “Boolean” | “Byte” | “Short” | “Char” | “Int” | “Long” | “Float” | “Double” | “String”

Table 3 BNF of variable declaration

Here is an example of how to use the BNF in variables, for instance: sum = 150

val sum: Int = 150

< Declare > → < Immutability >

→ val < var name >

→ val name: < datatype > = value

→ val sum: < datatype > = value

→ val sum: : < datatype > = 150

→ val sum: Int = 150

3.3. Expressions:

Operator is a symbol that indicates a specific operation that performed and excecuted in the program.

There are 3 types of operations:

- Arithmatic
- Logical
- Relationa

Arithmetic operation: is doing mathematical operations on a specified operand (Scala - Operators, n.d.)

For example:

X = 30

Y = 5

Operator	Description	Example
+	Adds two operands	X + Y equals 35
-	Subtracts second operand from the first	X - Y equals 25
*	Multiplies both operands	X * Y equals 150
/	Divides numerator by de-numerator	X / Y equals 6
%	Modulus operator finds the remainder after division of one number by another	X % Y equals 0

Table 4 Arithmetic operations

BNF of Arithmetic operations:
<Expression> → <Expression> + <Term> | <Expression> - <Term> | <Term>
<Term> → <Term> * <Factor> | <Term> / <Factor> | <Term> % <Factor> | <Factor>
<Factor> → (<Expression>) | Number | Variable

Table 5 BNF of Arithmetic operations

The highest precedence operations should be at the bottom, lowest precedence operations should be at the top of the BNF to avoid ambiguity and enforce precedency

Example in BNF:

6-(4*y)

< Expression > ⟶ < Expression > - <term>

⟶ <term > - <term>

⟶ <Factor > - <term>

⟶ number - <term>

⟶ 6 - <term>

⟶ 6 - <Factor>

⟶ 6 – (<Expression>)

⟶ 6 - (<Term>)

⟶ 6 - (<Term>*<Factor>)

⟶ 6 - (<Factor>*<Factor>)

⟶ 6 – (number*<Factor>)

⟶ 6 - (4*<Factor>)

⟶ 6 - (4*variable)

⟶ 6 - (4*y)

Example in coding:

```
object hop
{
def main(args: Array[String])
{
    var a = 30;
    var b = 5;
   println("The result in a + b = " + (a + b));
    println("The result in of a - b = " + (a - b));
    println("The result in of a * b = " + (a * b));
    println("The result in of a / b = " + (a / b));
    println("Modulus of a % b = " + (a % b));
}
}
```

Output:

```
The result in a + b = 35
The result in of a - b = 25
The result in of a * b = 150
The result in of a / b = 6
Modulus of a % b = 0
```

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Table 6 relational operations

(Scala - Operators, n.d.)

BNF of relational operations:
<Expression> → <Factor> < Operator> <Factor>   < Factor > → Number | Variable   < Operator > → “==” | “!=” | “>” | “<” | “>=” | “<=”

Table 7 BNF of relational operations

Example in BNF:

< Expression > → < Factor > <Operator> < Factor >

→ Variable <Operator> < Factor >

→ Sum <Operator> < Factor >

→ Sum >= <Factor>

→ Sum >= Number

→ Sum >= 15

Example in coding:

```
object relation
{
def main(args: Array[String])
{
   var a = 30;
   var b = 5;
   println("a == b result is: " + (a == b));
   println("a != b result is: " + (a != b));
   println("a > b result is: " + (a > b));
   println("a < b result is: " + (a < b));
   println("a >= b result is: " + (a >= b));
   println(" a <= b result is: " + (a <= b));
}
}
```

Output:

```
a == b result is: false
a != b result is: true
a > b result is: true
a >= b result is: true
a <= b result is: false
```

Logical operator: is a word or a symbol that connect two or more values or expressions, that relies on the meaning of the operator and the meaning of the expression.

The most popular logical operations:

- AND
- OR
- NOT

(Busbee & Braunschweig, 2018)

Operator	Description	Example
&&	It is called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	It is called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	It is called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Table8 logical operations

(Scala - Operators, n.d.)

BNF of logical operations
<Expression> → ! <Term> < Term>
< Term > → <Factor> && < Expression> | <Factor> || < Expression> | <Factor>
<Factor> → “(“< Expression>”)” | “false” | “true | Variable

Table 9 BNF of logical operations

Example in BNF: (y || x)

< Expression > → < Term >
→ <Factor>
→ (<Expression>)
→ (<Term>)
→ (<Factor> || <Expression>)
→ (Variable|| <Expression>)
→ (y|| <Expression>)
→ (y || <Term>)
→ (y || <Factor>)
→ (y || Variable)
→ (y|| x)

Example in coding:

```
object Logical
{
def main(args: Array[String])
{
   var a = true
   var b = false
   println("Not !(a && b) = " + !(a && b));
   println("Or a || b = " + (a || b));
```

```
    println("And of a && b = " + (a && b));
}
}
```

Output:

```
Not !(a && b) = true
Or a || b = true
And of a && b = false
```

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Table 10 assignment operations

(Scala - Operators, n.d.)

BNF of assignment operations:
<Expression> → <Term> <Operation> <Term> <Term> → Number | Variable <Operation> → "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=" | "&=" | "| =" | "^="

Table 11 BNF of assignment operations

Example in BNF: y *= 5

< Expression > → < Term > <Operator> < Term >
→ Variable <Operator> < Term >
→ y <Operator> < Term >
→ y *=< Term >
→ y *= Number
→ y *= 5

Example in coding:

```
object Assignment
{
def main(args: Array[String])
{
    var a = 50;
    var b = 40;
    var c = 0;
         c = a + b;
    println("c= a + b = " + c);
    c += a;
    println("c += a = " + c);
    c -= a;
    println("c -= a = " + c);
         c *= a;
    println("c *= a = " + c);
        c /= a;
    println("c /= a = " + c);
    c %= a;
    println("c %= a = " + c);
    c <<= 3;
    println("c <<= 3 = " + c);
    c >>= 3;
    println("c >>= 3 = " + c);
    c &= a;
    println("c &= 3 = " + c);
    c ^= a;
    println("c ^= a = " + c);
    c |= a;
    println("c |= a = " + c);
}
}
```

Output:

```
c= a + b = 90
c += a = 140
c -= a = 90
c *= a = 4500
c /= a = 90
c %= a = 40
c <<= 3 = 320
c >>= 3 = 40
c &= 3 = 32
c ^= a = 18
c |= a = 50
```

3.4. Selection (2-way if):

If statements are logical blocks performed in most programming languages. They are conditional statements which tell the program what to perform with specified information or data. They make a program take 'decisions' during it's running. They are consisted of two parts at minimum, 'if' and then.

If-else statement syntax in Scala:

```
if (condition) {
 //true condition
// Executes the block
}
else {
//false condition
// Executes the block }
```

(A Beginner's Guide to "IF" Statements, 2019)

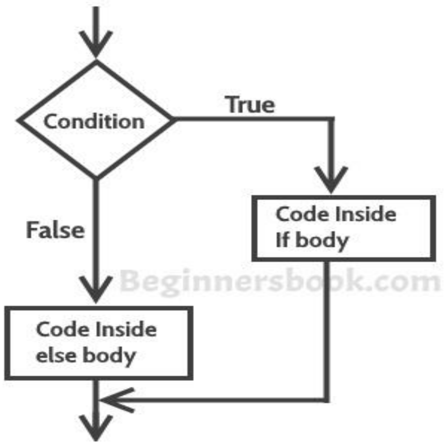


Figure 1 if else - structure

(C - If..Else, Nested If..Else and Else..If Statement with Example, 2017)

Example in coding:

```
object ScalaLanguage {
def main(args: Array[String]) {
var sum: Int = 10
if (sum > 11) {
//if sum > 11 is true , execute the statement
println("The sum is bigger than 11")
}
else {
// if sum  > 11 is false , execute the statement
println("The sum is less than or equal 11")
}
}
}
```

Output:

```
The sum is less than or equal 11
```

BNF of 2 way if-else:
<Selection> → <If> <Else> | <If>
<if> → "if" "(" condition ")" <Statement>
<Else> → "else" <Statement>
<Statement> → ["{"} statement ["]"] | "{" multiple statements "}"

Table 12 BNF of 2 way if-else

<Selection> → <If> <Else>

→ if (condition) <Statement> <Else>

→ if (sum > 11) <Statement> <Else>

→ if (sum > 11) {statement} <Else>

→ if (sum > 11) { println("The sum is bigger than 11") } <Else>

→ if (sum > 11) { println("The sum is bigger than 11") } else
<Statement>

→ if (sum > 11) { println("The sum is bigger than 11") } else
{statement}

→ if (sum > 11) {
println("The sum is bigger than 11") }
else {
println("The sum is less than or equal 11")
}

3.5. While Loop:

A loop is a series of instructions that is frequently repeated until a specified condition is achieved. Typically, a specified proceed is done, for example, earn an item of data and modifying it, then a condition is checked, for example, a counter has reached a specific number. If it hasn't reached it yet, the next instruction in the series is an instruction to go back to the beginning instruction in the series and iterate the series. If the condition has been done, the next instruction go to the next sequent instruction or go outside the loop.
(What Is Loop? - Definition from WhatIs.com, n.d.)

Example in coding:

```
def main(args: Array[String]) {
var i = 0;
var phoneNumber= Array(0555983241, 0540043700, 0544779933, 0540027024)
// It will exit if it reached the length of the array
while (i < phoneNumber.length) {
println("- Phone number: " + phoneNumber(i));
// The value of I will be incremented
i = i +1; }
```

Output:

```
- Phone number: 0555983241
- Phone number: 0540043700
- Phone number: 0544779933
- Phone number: 0540027024
```

(Team, 2018)

BNF of while loop:
<While Loop> → “while” “(“ condition “)” “{“ statement “}”

Table 13 BNF of while loop

Example in BNF:

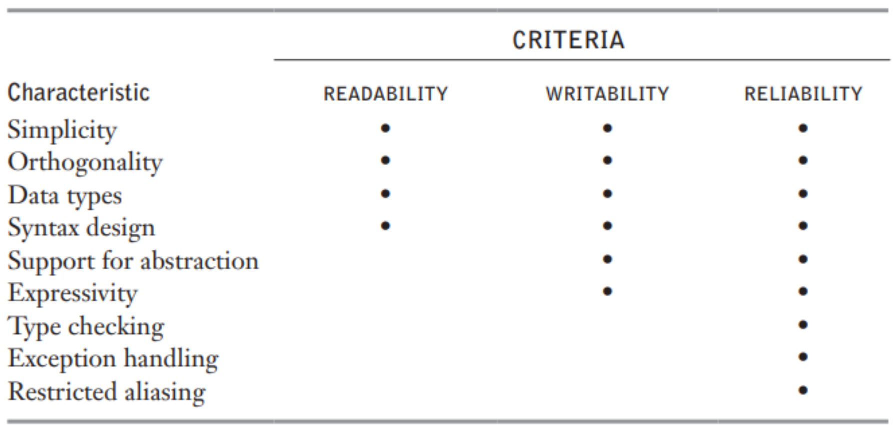
< While Loop > → while (condition) {statement}

→ while (i < phoneNumber.length) {statement}

→ while (i < phoneNumber.length) {println("- Phone number: " + phoneNumber(i)); i = I +1; }

4.Evaluation

Many features of programming languages affect the software development process, especially the maintenance process. We will discuss a set of criteria for evaluating these features and their impact on language, namely, readability, writability, and finally reliability. In addition, there are several characteristics that affect the criteria themselves. In our project, we will discuss the criteria and properties that affect the criteria in Scala. The table below summarizes all the characteristics that affect each criterion, which we will discuss further in the following sections. (Sebesta, 2019).



	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Table 14 The table below summarizes all the characteristics that affect each criterion

4.1. Readability

The ease with which programs can be read and understood is one of the most important criteria for judging a programming language. The readability of programs determines the ease of maintenance in large part. because the maintenance process requires the programmer to read and understand the code and try to improve and modify the code, so ease of maintenance is very important to reduce the cost as well. Readability has become an important measure of the quality of programs and programming languages. This was a watershed moment in the evolution of programming languages. The subsections that follow describe characteristics that contribute to a programming language's readability. (Sebesta, 2019).

4.1.1. Overall simplicity

The readability of a programming language is heavily influenced by its overall simplicity. A language with a large number of fundamental constructs is more difficult to learn than one with fewer. There are some factors that play role in it:

- Feature multiplicity: means having more than one way to write a particular operation. For example, in Scala, the operation decrementing the counter by one is written in two ways like
  ```
  count = count – 1;
  count - = 1;
  ```
  This is a feature that can reduce readability, as the reader of the code may not know one of these two methods, and it becomes difficult for him to understand the program. Fortunately, the Scala programming language does not support unary (++ or - -) operators. In Scala, binary operators are used to increment and decrement an integer. (Scala Has No ++ or – Operator, How to Increment or Decrement an Integer?, n.d.)
- Operator overloading: means a single operator symbol has more than one meaning, it can reduce readability if programmers can create their own overloading. For example, in Scala, we can overload an operator '+' in a class like String so that we can concatenate two strings by simply using + between them. (String Concatenation in Scala, 2019) As in the following code: `print ("Hello"+" World");`

4.1.2. Orthogonality

Orthogonality in Programming Languages We measure a language`s orthogonality by the degree to which we can combine its built-in constructs without side effects. Based on this, a programming language belongs to one of the following three classes:

- Fully orthogonal
- Partly or quasi-orthogonal
- Non-orthogonal

This classification is depicted in the diagram below, along with some examples of each class.

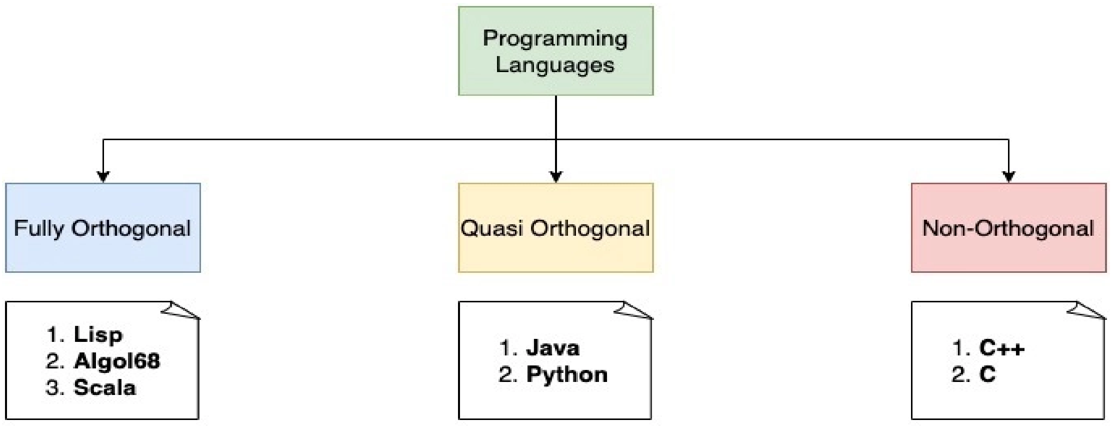


Figure 2 classification Orthogonality example

A programming language is fully orthogonal if the majority of its constructs can be combined in a variety of ways with no side effects. Scala is considered a fully orthogonal language. In Scala, for example, functions can be used as local variables, fields, or parameters to other functions, just like any other object. We experience no side effects as a result of this. Passing a function as an argument, for example, does not invoke it. Scala is orthogonal with both object-oriented and functional programming paradigms. As a result, we can freely use both of these frameworks in Scala without being concerned about any side effects. (Bhargav, 2022)

4.1.3. Data Types and Structures

A data type is a classification of data that tells the compiler what type of value a variable has. The presence of suitable and meaningful data types in a language is a powerful aid to readability. For example, if a variable has an int data type, it holds a numeric value. Scala data types are similar to Java in terms of length and storage. Data types are treated as objects in Scala, so the first letter of the data type is capitalized. Some examples of data types in Scala. (Data Types in Scala, 2019).

- A data type of Boolean (true or false).
- A data type of integer (32-bit signed value).
- A data type of string (sequence of characters).

4.1.4. Syntax design

The syntax of the elements of a language has a significant effect on the readability of programs. Here are some examples of syntactic design:

- Special words (Reserved words) are defined as part of the Scala language and cannot be used as identifiers. such as (case, def, finally). (Reserved Words - Learning Scala [Book], n.d.)
- Scala uses braces {} to define the structure of the block, this makes it easier to read. (Effective Scala, n.d.)

4.2. Writability

Writability is a measure of how efficiently and easily language can be used to create programs. Writability, like readability, must be considered in the context of the target language problem domain, it is not fair to compare the writability of two languages in two different domains. (Sebesta, 2019). Some of its factors have been discussed in detail previously. Now we will discuss the effect of support for abstraction, and expressivity.

4.2.1. Support for abstraction

Abstraction is the process of concealing internal details and displaying only functionality. An abstract class is used in Scala to achieve abstraction. The Scala abstract class works similarly to the Java abstract class. The abstract keyword is used to create an abstract class in Scala. It has both abstract and non-abstract methods but does not support multiple inheritances. A class can only extend one abstract class. We can define abstract classes using the following syntax:

```
abstract class class_name
{
// code...
}
```

The abstract methods of an abstract class are those that do not have any implementation. In other words, an abstract method does not contain anybody.

```
def function_name()
```

In Scala, an abstract class is used when we want to create a base class that requires constructor arguments. Attributes are also used to achieve abstraction. (Abstract Classes in Scala, 2019)

4.2.2. Expressivity

Expressivity implies that the language implements extremely powerful operators, allowing for large computations with small programs. The greater the expressiveness of a language, the greater the variety and quantity of ideas it can represent. Scala, on the whole, is more expressive than other languages. Developers enjoy writing code in Scala, especially after learning Java. Let us investigate the expressiveness of the Scala language by running a small program written in Java and Scala that counts words. A single line in Scala where Java has taken more than 10 lines to achieve. (Abstract Classes in Scala, 2019)

Code in Java:

```
Public class WordCount {
    Public static void main(String [] args){
        StringToken st= new StringToken(args[0]);
        Map<String, Integer> map= new HashMap<String,
Integer>();
        while(st.hasMoreTokens()) {
            String word= st.nextToken();
            Integer count= map.get(word);
                If(count == null)
                    map.put(word, count+1);
         }
        System.out.println(map);
    }
}
```

Code in Scala:

```
Object WordCountScala extends App {
    Println( args(0).split(“ ”).groupBy(x => x).map(t => t._1 ->
t._2.length))
}
```

4.3. Reliability

The importance of reliability is equal to that of readability and writability. A system is reliable if it has few errors and performs as expected. Some concepts that influence reliability, which is: Type Checking, Exception Handling, and Aliasing, and Readability, and writability. (Karimov, 2020).

4.3.1. Type checking

Type errors are checked by the compiler or during program execution. Compile-time type checking is preferable to run-time type checking because it is less expensive. Not every programming language is intended to support type-checking. There are two different types of type checking: dynamic type checking and static type checking. The dynamic type of type checking occurs at runtime, whereas the compile type occurs at compile time. Scala is a statically typed language, which means that all type-checking occurs during the compilation process. In a programming language, type-checking prevents programmers from writing bad code that produces unexpected results. Scala can detect errors during compilation, providing early protection. Scala also improves runtime performance because its compiler can perform various optimizations and type-checking is not required during runtime (Joy, 2022).

4.3.2. Exception handling

The ability to intercept run-time errors, take corrective measures, and then continue is called exception handling. Scala's exception handling is different, but it behaves exactly like Java and works seamlessly with existing Java libraries. Exceptions in Scala work in the same way that they do in C++ or Java.

However, the try/catch construct in Scala differs from that in Java, try/catch in Scala is an expression. Instead of providing a separate catch block for each different exception, the exception in Scala that results in a value can be pattern matched in the catch block. Because try/catch is an expression in Scala. Here is an example of exception handling using the traditional try-catch block in Scala. (Scala | Exception Handling, 2019).

```
// Scala program of finally Exception
// Creating object
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        try
        {
            var N = 5/0
        }
        catch
        {
            // Catch block contain case.
            case ex: ArithmeticException =>
            {
                println("Arithmetic Exception occurred.")
            }
        }
        finally
        {
            // Finally block will execute
            println("This is final block.")
        }
    }
}
```

Output:

```
Arithmetic Exception occurred.
This is final block.
```

4.3.3. Aliasing

It refers to the situation in which the same memory cell can be accessed by two or more distinct names in a program. Aliasing by pointers (referencing) to the same variable is supported in programming languages such as C++. Remember that if two pointers point to the same variable, changing the value of one will change the value referenced by the others. (Karimov, 2020) Scala supports aliasing by pointers (referencing) to the same variable, which reduces its reliability. (Arrays - How to Get Pointer/Reference Semantics in Scala, n.d.)

4.3.4. Readability and Writability

Both have an impact on reliability. Readability influences reliability during both the writing and maintenance phases of the life cycle. Difficult-to-read programs are difficult to write and modify. As a result, they are less reliable because they are prone to more errors. (Sebesta, 2019). As explained previously, Scala's readability and writability, as well as all of the characteristics that influence them, have a direct impact on its reliability.

5. Types of variables

5.1. What are the variables?

Variables are names given to computer memory locations to store data in a program. Every variable has its own data type, a name, and a value assigned to it. We will mention some of the features of Scala:

- Scala is a statically typed language, meaning that the variable's data type is defined before it is used. Type checking occurs at compile time rather than run time.
- It is also a "Strongly Typed" language, which means that variables are checked before having an operation in it.
- It also supports Type Inference, which allows the compiler to infer the type of the variable from the expression or literals, making declaring the type of variable optional. (Variables in Scala, n.d.)

5.2. Name

Rules for naming variables in Scala:

- The variable name should be written in lowercase.
- A variable name can contain a letter, a digit, and two special characters (the underscore (_) and the dollar sign.($)
- The keyword or reserved word must not appear in the variable name.
- The variable name should begin with an alphabet letter.
- Variable names cannot contain white space.

Examples of valid variable names: $class, name$, student_name.

Examples of invalid variable names: 1class, name%, dev. (Variables in Scala, 2019)

5.3. Declaration

In order to clarify the declaration, we must know the types of variables in Scala. Variables in Scala are classified into two types:

- Mutable Variables
- Immutable Variables

Mutable Variables

Mutable variables are those that allow us to change the value of a variable after it has been declared. The var keyword is used to define mutable variables. In addition, data types are treated as objects in Scala. The syntax is as follows:

```
var Variable_name: Data_type = "value";
```

Example:

```
var name: String = "Final project 301 ";
```

Here, name is the variable's name, string is the variable's data type, and "Final project 301" is the value stored in memory.

Another way:

```
var variable_name = value
```

Scala compiler automatically determines which values belong to which data types. We can use this method in both types, but we pay attention to the keyword.

Example:

```
var value = 40
//it works without error
```

Here value is the name of the variable, is by default int type, and 40 is the value stored in memory.

Immutable Variables

These are variables that do not allow you to change their values after they have been declared. The val keyword is used to define immutable variables. In addition, data types are treated as objects in Scala. The syntax is as follows:

```
val Variable_name: Data_type =  "value";
```

Example:

```
val name: String = "Final project 301 ";
```

Here, name is the variable's name, string is the variable's data type, and "Final project 301" is the value stored in memory. (Variables in Scala, 2019)

5.4. Value assigning

Variable assignment in Scala, like in many other languages, is divided into two parts: a declaration and a value assignment separated by an assignment operator, which is an equal sign. Variables and fundamental types Scala is very similar to Java and other languages. The syntax is as follows: (Singh & Singh, 2021)

```
var Variable_name: Data_type = "value";
```

Example:

```
var number: Int = 301
```

This is a very simple example of declaring and assigning a variable in one line.

Multiple declarations are possible in Scala by using the 'var' keyword followed by the variable name separated by commas and the "=" sign followed by the variable value. As the following example:

```
var (x, y, z) = (5, 4.5, "Sit")
```

Multiple assignments are possible in Scala in a single line by using the 'val' keyword with variable names separated by commas and the "=" sign with the variable's value. As the following example: (Variables in Scala, n.d.)

```
val a, b, c = 1;
```

We can declare and assign each variable on a line, but this way of writing helps in reducing the number of codes and thus will improve readability and ease of writing as well.

5.5. Data types

Scala supports all of Java's data types, with the same memory footprint and precision. The table below contains information on all of the data types available in Scala. The table below shows data types in Scala. (Data Types in Scala, 2019)

DataType	Default value	Description
Boolean	False	True or False
Byte	0	8 bit signed value. Range:-128 to 127
Short	0	16 bit signed value. Range:-2^{15} to 2^{15}-1
Char	'\u000'	16 bit unsigned unicode character. Range:0 to 2^{16}-1
Int	0	32 bit signed value. Range:-2^{31} to 2^{31}-1
Long	0L	64 bit signed value. Range:-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 character
Unit	–	Coinsides to no value.
Nothing	–	It is a subtype of every other type and it contains no value.
Any	–	It is a supertype of all other types
AnyVal	–	It serve as value types.
AnyRef	–	It serves as reference types.

Table 15 Data type of Scala

5.5. lifetime, and storage

A variable's lifetime is the amount of time that it is bound to a specific memory cell. It is also the time when a variable is available. A variable's lifetime typically begins when it is bound to a memory cell and ends when it is unbound from that cell (Sebesta, 2019).

```
var globalVar = 10
def sub}()
var difference = 9-4;
{
print(difference)//Error
print(globalVar)//Valid
```

The 'difference' variable is in Local Scope in the above code because it is declared inside the 'sub' method, and subtraction will produce valid results, but printing the value outside the method 'sub' will result in an error. There is also a Global Scope in which the variable 'globalVar' can be accessed from anywhere in the program and any operation can be performed on it. (Variables in Scala, n.d.)

The variable is allocated when storage is associated with it. Allocation for a given variable can occur statically (before the program is executed) or dynamically (during execution). (Storage Classes, Allocation, and Deallocation, n.d.)

Scala stores variables in different memory areas. For example, local variables are stored in a stack, while global variables are stored in heap memory (How Is Memory Managed in Scala? – Quick-Advisors.com, n.d.).

6. Array Types

The array, a data structure offered by Scala, is used to store a fixed-size sequential collection of the same kind of elements. It is important to think of an array as a collection of variables of the same type even though it is used to store a collection of data. You define one array variable, such as weeks, and use weeks [0], weeks [1], and..., weeks [99] to represent individual variables rather than specifying individual variables, such as week0, week1..., and week99. The index of the first element of an array is zero and the last element's index is equal to the number of elements minus one. Both single-dimension and multi-dimension arrays are supported by Scala. A two-dimensional array is actually a matrix of dimensions (n * m), whereas a single-dimension array simply has one row and n columns (Scala - Arrays, n.d.).

6.1. Some Important Points:

- Scala arrays can be generic. which means we can have an Array[T], where T is a type parameter or abstract type.
- Scala arrays are compatible with Scala sequences – we can pass an Array[T] where a Seq[T] is required.
- Scala arrays also support all sequence operations (Scala Array - Javatpoint, n.d.).

The next illustration demonstrates how values can be successively stored in an array:

10	20	30	40	50	60	70	80	90	
0	1	2	3	4	5	6	7	8	<-Array indices

Array length = 9, First index = 0, Last index = 8

6.2. The Syntax for Declaring a Single-Dimensional Array:

var arrayName : Array[arrayType] = new Array[arrayType](arraySize);   or

var arrayName = new Array[arrayType](arraySize) or

var arrayName : Array[arrayType] = new Array(arraySize);   or

var arrayName = Array(element1, element2 ... elementN)

To explain that, arrayName is the name of the variable array, which is used to reference the array, arrayType is a specific type of data elements being stored, and arraySize is a specific number of elements in the array.

6.2.1. Scala Single-Dimensional Array Example:

```
class ArrayExample{
   var arr = Array(Aya,Ghadeer,Noor,Nsreen)      // Creating single dimensional array

   def show(){
      for(a<-arr)                      // Traversing array elements
         println(a)
      println("Third Element  = "+ arr(2))       // Accessing elements by using index
   }
}

object MainObject{
   def main(args:Array[String]){
      var a = new ArrayExample()
      a.show()
   }
}
```

Output:

```
Aya
Ghadeer
Noor
Nsreen
Third Element  = Noor
```

Scala Single-Dimensional Array Example2:

In this example, we have created an array by using a new keyword which is used to initialize memory for the array. The entire array of elements is set to the default value, we can assign that later in our code.

```
   var arr = new Array[Int](4)       // Creating single dimensional array
   def show(){
      for(a<-arr){                    // Traversing array elements
         println(a)
      }
         // Accessing elements by using index
      println("Third Element before assignment = "+ arr(2))
      arr(2) = 100                                 // Assigning new element at 2 index
      println("Third Element after assignment = "+ arr(2))
   }
}
object MainObject{
   def main(args:Array[String]){
      var a = new ArrayExample()
      a.show()
   }
         }
```

Output:

```
0
0
0
0
Third Element before assignment = 0
Third Element after assignment = 100
```

Scala updating an element in Array Example:

```
// Scala program to updating an array of the string as name
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        // allocating memory of 1D Array of string
        var name = Array("Aya", "Rania", "Noor", "Nsreen" )

        // Updating an element in an array
        name(1)="Ghadeer"
        println("After updating array elements are: ")

        for ( m1 <-name )
        {
            println(m1 )
        }
    }
}
```

Output:

```
After updating array elements are :
Aya
Ghadeer
Noor
Nsreen
```

Scala adding elements in Array Example:

```
// Scala program to adding elements in an array of the string as name
object GFG
{
   // Main method
   def main(args: Array[String])
   {
      var name = new Array[String](4)

      // Adding element in an array
      name(0)="Aya"
      name(1)="Ghadeer"
      name(2)="Noor"
      name(3)="Nsreen"
      println("After adding array elements : ")

      for ( m1 <-name )
      {
         println(m1 )
      }

   }
}
```

Output:

```
After adding array elements :

Aya
Ghadeer
Noor
Nsreen
```

Scala Passing Array into Function Example:

we can pass an array as an argument to function during the function call. The following example illustrates the process of how we can pass an array to the function.

```
class ArrayExample{
    def show(arr:Array[Int]){
        for(a<-arr)                // Traversing array elements
            println(a)
        println("Third Element = "+ arr(2))        // Accessing elements by using index
    }
}

object MainObject{
    def main(args:Array[String]){
        var arr = Array(10,20,30,40,50,60)    // creating single dimensional array
        var a = new ArrayExample()
        a.show(arr)                    // passing array as an argument in the function
    }
}
```

Output:

```
10
20
30
40
50
60
Third Element = 30
```

Scala Concatenate Arrays Example:

We can concatenate two arrays by using concat() method. In concat() method we can pass more than one array as arguments.

```
// Scala program to concatenate two arrays by using concat() method
import Array._
// Creating object
object GFG
{
    // Main method
def main(args: Array[String])
{
    var arr1 = Array(10, 20, 30, 40,50)
    var arr2 = Array(60, 70, 80, 90,100)
    var arr3 = concat( arr1, arr2)
    // Print all the array elements
    for ( x <- arr3 )
    {
        println( x )
    }
}
}
```

Output:

```
10
20
30
40
50
60
70
80
90
100
```

To explain the code:

arr1 is an array of five elements and arr2 is another array of five elements now we concatenate these two array in arr3 by using concat() method.

Scala Array Example Iterating By using Foreach Loop:

We can also iterate array elements by using foreach loop

```
class ArrayExample{
   var arr = Array(10,20,30,40,50,60,70,80)// Creating single dimensional array
   arr.foreach((element:Int)=>println(element))//Iterating by using foreah loop
}

object MainObject{
   def main(args:Array[String]){
      new ArrayExample()
   }
}
```

Output:

```
10
20
30
40
50
60
70
80
```

6.3. Multidimensional Arrays

Scala has a method Array.ofDim to create Multidimensional arrays. Multi-dimensional arrays can be used in structures such as matrices and tables.

6.3.1. The Syntax for Declaring a Multi-Dimensional Array:

var array_name = Array.ofDim[ArrayType](n, m)

or

var array_name = Array(Array(elements), Array(elements)

To explain that, this is a Two-Dimensional array. where n is the number of rows and m is the number of Columns.

6.3.2. Scala Multi-Dimensional Array Example:

```
// Scala program to creating a multidimension array of the string as
// names, store values in the names, and prints each value
object GFG
{
    // Main method
    def main(args:Array[String])
    {
        val rows = 2
        val cols = 3
        // Declaring Multidimension array
        val names = Array.ofDim[String](rows, cols)
        // Allocating values
        names(0)(0) = "Hello"
        names(0)(1) = "World"
        names(0)(2) = "Using"
        names(1)(0) = "Scala"
        names(1)(1) = "Language"
        names(1)(2) = "+_+"
        for
        {
            i <- 0 until rows
            j <- 0 until cols
        }
        // Printing values
        println(s"($i)($j) = ${names(i)(j)}")
    }
}
```

Output:

```
(0)(0) = Hello
(0)(1) = World
(0)(2) = Using
(1)(0) = Scala
(1)(1) = Language
(1)(2) = +_+
```

6.4. Append and Prepend elements to an Array in Scala:

Use these operators (methods) to append and prepend elements to an array while assigning the result to a new variable:

Method	Function	Example
:+	append 1 item	old_array :+ e
++	append N item	old_array ++ new_array
+:	prepend 1 item	e +: old_array
++:	prepend N items	new_array ++: old_array

Table 16 Append and Prepend elements to an Array in Scala

6.4.1. Examples using Scala to show how to use the above methods to append and prepend elements to an Array:

```
object GFG
{
 // Main method
 def main(args: Array[String])
 {
  // Declaring an array
  val a = Array(10, 20, 30)
  println("Array a ")
  for ( x <- a )
  {
   println( x )
  }
  // Appending 1 item
  val b = a :+ 40
  println("Array b ")
  for ( x <- b )
  {
   println( x )
  }
```

```
    // Appending 2 item
    val c = b ++ Array(50, 60)
    println("Array c ")
    for ( x <- c )
    {
       println( x )
    }
    // Prepending 1 item
    val d = 70 +: c
    println("Array d ")
    for ( x <- d )
    {
      println( x )
    }

    // Prepending 2 item
    println("Array e ")
    val e = Array(80, 90) ++: d
    for ( x <- e )
    {
      println( x )
    }
  }
}
```

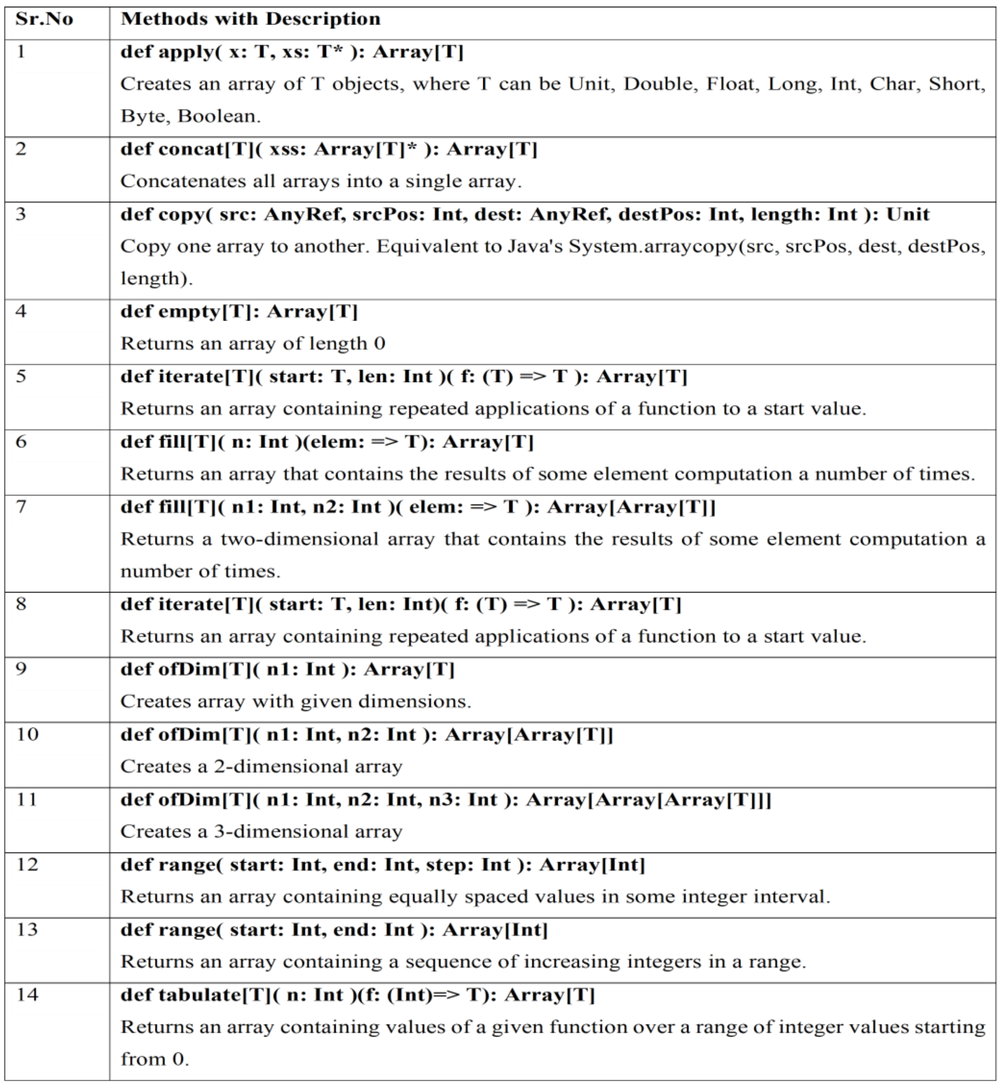
Output:

```
Array a
10
20
30
Array b
10
20
30
40
Array c
10
20
30
40
50
60
Array d
70
10
20
30
40
```

```
50
60
Array e
80
90
70
10
20
30
40
50
60
```

6.5. Scala Array Methods

Here are the important methods, which you can use while dealing with the array. You should import Array. package before using any of these methods.



Sr.No	Methods with Description
1	**def apply(x: T, xs: T*): Array[T]** Creates an array of T objects, where T can be Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean.
2	**def concat[T](xss: Array[T]*): Array[T]** Concatenates all arrays into a single array.
3	**def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit** Copy one array to another. Equivalent to Java's System.arraycopy(src, srcPos, dest, destPos, length).
4	**def empty[T]: Array[T]** Returns an array of length 0
5	**def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]** Returns an array containing repeated applications of a function to a start value.
6	**def fill[T](n: Int)(elem: => T): Array[T]** Returns an array that contains the results of some element computation a number of times.
7	**def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]** Returns a two-dimensional array that contains the results of some element computation a number of times.
8	**def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]** Returns an array containing repeated applications of a function to a start value.
9	**def ofDim[T](n1: Int): Array[T]** Creates array with given dimensions.
10	**def ofDim[T](n1: Int, n2: Int): Array[Array[T]]** Creates a 2-dimensional array
11	**def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]]** Creates a 3-dimensional array
12	**def range(start: Int, end: Int, step: Int): Array[Int]** Returns an array containing equally spaced values in some integer interval.
13	**def range(start: Int, end: Int): Array[Int]** Returns an array containing a sequence of increasing integers in a range.
14	**def tabulate[T](n: Int)(f: (Int)=> T): Array[T]** Returns an array containing values of a given function over a range of integer values starting from 0.

Table 17 Scala Array Methods

7. Types of Scopes

The number of statements that can utilize and access a variable is referred to as its scope. Statements within the scope of a variable's visibility may refer to it. The scope of n for example is in the region where n is used, such as when n is used as the declaration for a variable. Maintaining unique variables in various parts of the program is one of the key justifications for utilizing the scope concept. Even if another variable with the same name is used in different program statements, we can still protect the value of the variable from changes (Static and Dynamic Scoping - GeeksforGeeks, 2017).

In programming languages there are two types of scopes:

- Dynamic Scope.
- Static Scope is also known as (lexical scope).

7.1. Dynamic Scope

In programming languages with dynamic scope, variables are mentioned and defined based on how recently they have been called. Because it occurs during run time, it is referred to be dynamic.

When using the dynamic scoping rule, we initially look for a variable's local definition. If it isn't found, we look for a definition in the calling stack. Dynamic scoping is more concerned with how the code works than how it is written. A new scope is added to the stack each time a new function is performed.

This form of scope is rarely used in programming languages; instead, the static (or lexical) scope is the norm. The second type of scope, which we will go over in more detail, is used by the Scala language (Static and Dynamic Scoping - GeeksforGeeks, 2017).

7.2. Static Scope

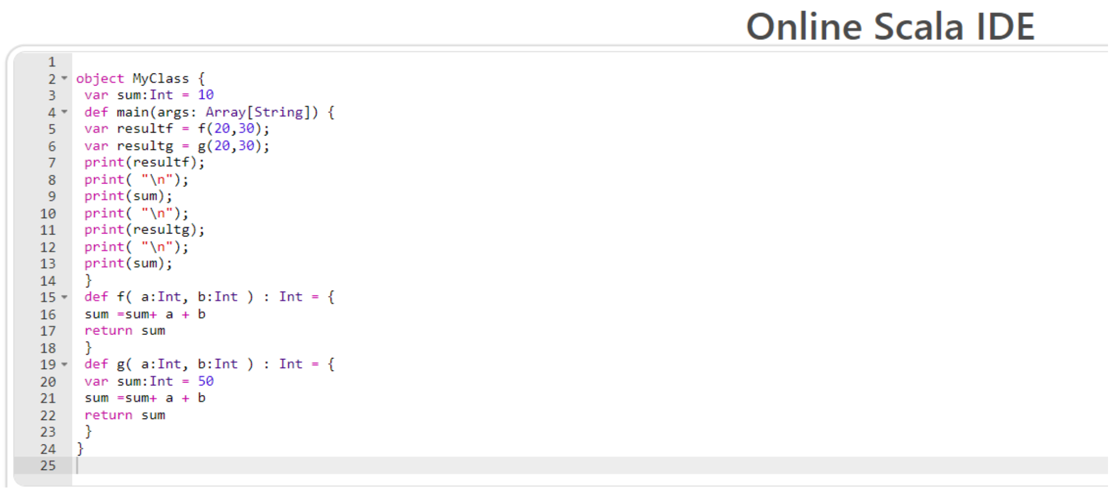
The static scope is also called lexical scope. This indicates that a variable's scope will be determined at compile time. Static scope depends on the logical structure and organization of the program. Each block has its own scope. Variables can be defined in that scope and are concealed from the outside blocks. Variables outside the scope (beyond the block's parentheses) are invisible. The language rules determine the scope of each type of declaration since variables may only be called (referred) from inside the block of code to which they are declared/visible to.

In most programming languages including C, C++, and Java, variables are always statically (or lexically) scoped i.e., the binding of a variable can be determined by program text and is independent of the run-time function call stack .

In most programming languages static scoping is dominant. This is simply because in static scoping it's easy to reason about and understand just by looking at code. We can see what variables are in the scope just by looking at the text in the editor.

Scala is one of the most common programming languages that use this type of scope (Static and Dynamic Scoping - GeeksforGeeks, 2017).

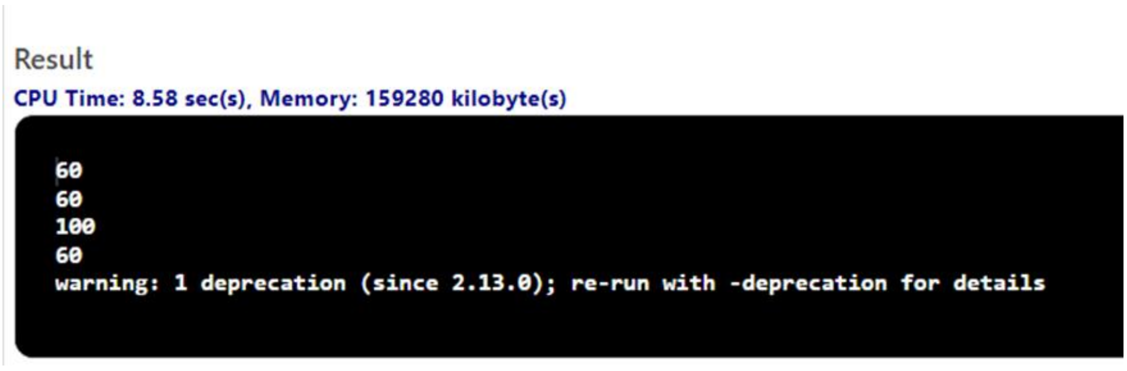
7.2.1. Below we have an example of static scope in Scala:



```scala

object MyClass {
 var sum:Int = 10
 def main(args: Array[String]) {
 var resultf = f(20,30);
 var resultg = g(20,30);
 print(resultf);
 print( "\n");
 print(sum);
 print( "\n");
 print(resultg);
 print( "\n");
 print(sum);
 }
 def f( a:Int, b:Int ) : Int = {
 sum =sum+ a + b
 return sum
 }
 def g( a:Int, b:Int ) : Int = {
 var sum:Int = 50
 sum =sum+ a + b
 return sum
 }
}
```

Output:



CPU Time: 8.58 sec(s), Memory: 159280 kilobyte(s)

```
60
60
100
60
warning: 1 deprecation (since 2.13.0); re-run with -deprecation for details
```

To explain the code:

1- start the execution from the main method

2- calling function f and send to it the value of variables (a,b)

3 - In the method f, execute the expression (sum =sum+ a + b)

First, the compiler will search for the declaration of the variable sum locally (inside the method f), since it has not found the declaration, it will trace back to the nearest static parent of the method f to find the declaration of sum, which in this case was the global variable sum with the value10. After executing method f the returned value of sum will be 60. To show that even the original value of the global value of sum has changed we have printed the sum outside the method f, and it did indeed change its original value from 10 to 60.

4- calling function g and send to it the value of variables (a,b)

5- In the method g execute the expression (sum =sum+ a + b)

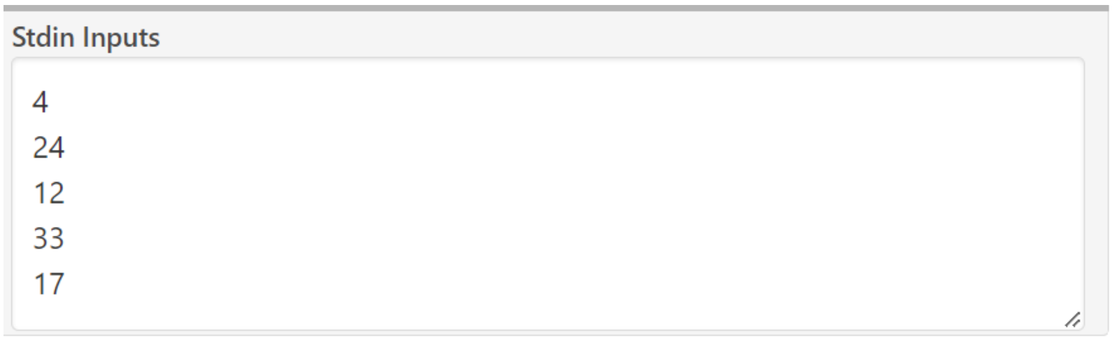
First, the compiler will search for the declaration of the variable sum locally (inside the method g), if it did not find the declaration inside the method, it needs to trace back to the nearest static parent of method g to search for the declaration of sum. In this case, the declaration of sum was found locally inside method g, so the compiler will not use the global variable. Therefore, the value of sum outside the method will not change. Method g will return a which is 100. And to prove that it did not change the value of the global sum we have printed sum from outside the method and it did not change from 60 (its value after method f).

8. Program (code and run)

The program's idea is that it will read "X" grades from the user. The grades will be stored inside an array of size "X". Then the program will compute average, minimum and maximum and display the results.We used jdoodle online compiler to run our code .

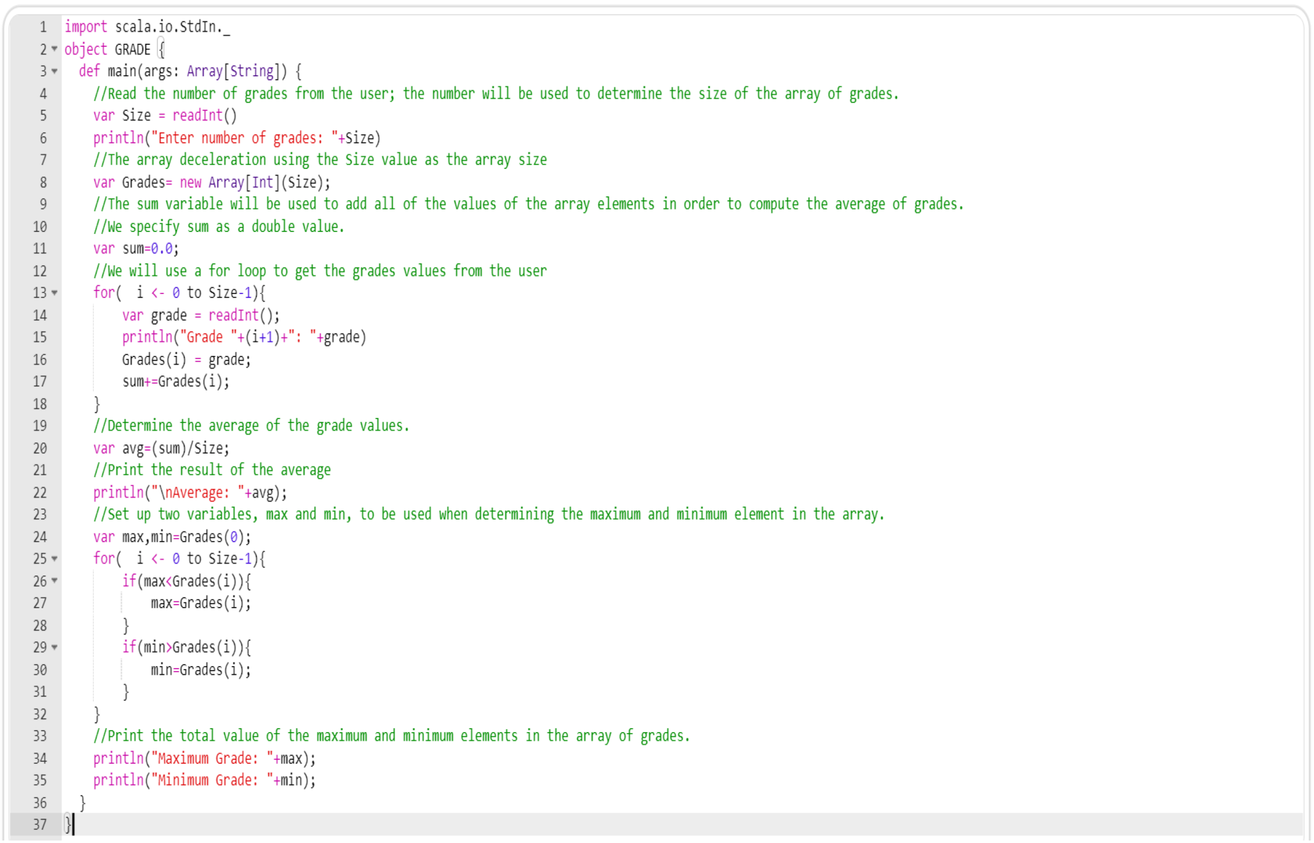
NOTE: Please keep in mind that the jdoodle compiler requires the user to enter all input values in a separate panel from the output panel. That is why we decided to print the input values in the output to show what values the user inserted.

Before running the program, the following user inputs must be entered into the panel Stdin input:



```
4
24
12
33
17
```

The code

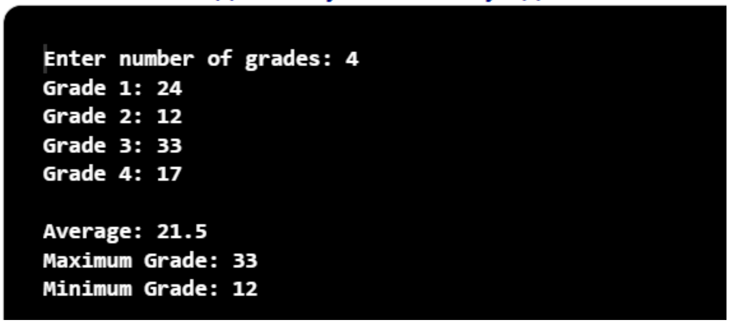


```scala
import scala.io.StdIn._
object GRADE {
  def main(args: Array[String]) {
    //Read the number of grades from the user; the number will be used to determine the size of the array of grades.
    var Size = readInt()
    println("Enter number of grades: "+Size)
    //The array deceleration using the Size value as the array size
    var Grades= new Array[Int](Size);
    //The sum variable will be used to add all of the values of the array elements in order to compute the average of grades.
    //We specify sum as a double value.
    var sum=0.0;
    //We will use a for loop to get the grades values from the user
    for(  i <- 0 to Size-1){
        var grade = readInt();
        println("Grade "+(i+1)+": "+grade)
        Grades(i) = grade;
        sum+=Grades(i);
    }
    //Determine the average of the grade values.
    var avg=(sum)/Size;
    //Print the result of the average
    println("\nAverage: "+avg);
    //Set up two variables, max and min, to be used when determining the maximum and minimum element in the array.
    var max,min=Grades(0);
    for(  i <- 0 to Size-1){
        if(max<Grades(i)){
            max=Grades(i);
        }
        if(min>Grades(i)){
            min=Grades(i);
        }
    }
    //Print the total value of the maximum and minimum elements in the array of grades.
    println("Maximum Grade: "+max);
    println("Minimum Grade: "+min);
  }
}
```

Output

Result

CPU Time: 13.29 sec(s), Memory: 185656 kilobyte(s)



```
Enter number of grades: 4
Grade 1: 24
Grade 2: 12
Grade 3: 33
Grade 4: 17

Average: 21.5
Maximum Grade: 33
Minimum Grade: 12
```

References:

- Sebesta, R. (n.d.). *GlobAl EdiTioN Concepts of Programming languages ElEVENTH EdiTioN*. https://vulms.vu.edu.pk/Courses/CS508/Downloads/Concepts%20of%20Programming%20Languages%2011th%20Ed.pdf
- *Introduction*. (n.d.). Scala Documentation. https://docs.scala-lang.org/tour/tour-of-scala.html
- Scala has no ++ or – operator, how to increment or decrement an integer? (n.d.). Www.includehelp.com. Retrieved November 5, 2022, from https://www.includehelp.com/scala/scala-has-no-plus-plus-or-minus-minus-operator-how-to-increment-or-decrement-an-integer.aspx
- *String concatenation in Scala*. (2019, July 16). GeeksforGeeks. https://www.geeksforgeeks.org/string-concatenation-in-scala/
- Bhargav, N. (2022, September 14). *Orthogonality in Computer Programming | Baeldung on Computer Science*. Www.baeldung.com. https://www.baeldung.com/cs/orthogonality-cs-programming-languages-software-databases
- *Data Types in Scala*. (2019, January 16). GeeksforGeeks. https://www.geeksforgeeks.org/data-types-in-scala/
- *Reserved Words - Learning Scala [Book]*. (n.d.). Www.oreilly.com. https://www.oreilly.com/library/view/learning-scala/9781449368814/apa.html
- *Effective Scala*. (n.d.). Twitter.github.io. Retrieved November 5, 2022, from https://twitter.github.io/effectivescala/
- *Abstract Classes in Scala*. (2019, February 5). GeeksforGeeks. https://www.geeksforgeeks.org/abstract-classes-in-scala/
- Karimov, I. (2020, November 16). *Language Evaluation Criteria — Writability and Reliability*. Medium. https://mrilyaskarimov.medium.com/language-evaluation-criteria-writability-and-reliability-1a4c8ae369a9
- *Scala | Exception Handling*. (2019, April 9). GeeksforGeeks. https://www.geeksforgeeks.org/scala-exception-handling/
- *arrays - How to get Ponter/Reference semantics in Scala*. (n.d.). Stack Overflow. Retrieved November 5, 2022, from https://stackoverflow.com/questions/2799128/how-to-get-ponter-reference-semantics-in-scala
- *Variables in Scala*. (2019, January 16). GeeksforGeeks. https://www.geeksforgeeks.org/variables-in-scala/

- *Variables in Scala*. (n.d.). Www.datacamp.com. Retrieved November 5, 2022, from https://www.datacamp.com/tutorial/variables-in-scala
- Singh, G., & Singh, G. (2021, November 12). *Variables and Basic Types In Scala*. Knoldus Blogs. https://blog.knoldus.com/variables-and-basic-types-in-scala/
- *Storage classes, allocation, and deallocation*. (n.d.). Www.ibm.com. Retrieved November 5, 2022, from https://www.ibm.com/docs/en/epfz/5.3?topic=control-storage-classes-allocation-deallocation
- *How is memory managed in Scala? – Quick-Advisors.com*. (n.d.). Quick-Advisors.com. Retrieved November 5, 2022, from https://quick-advisors.com/how-is-memory-managed-in-scala/
- Static and Dynamic Scoping - GeeksforGeeks. (2017, October 20). GeeksforGeeks. https://www.geeksforgeeks.org/static-and-dynamic-scoping/
- Scala-Arrays.(n.d.).Www.tutorialspoint.com. https://www.tutorialspoint.com/scala/scala_arrays.htm
- Scala|Arrays-GeeksforGeeks.(2019,March8).GeeksforGeeks. https://www.geeksforgeeks.org/scala-arrays/
- Scala Array - javatpoint. (n.d.). Www.javatpoint.com. Retrieved November 5, 2022, from https://www.javatpoint.com/scala-array
- Scala - Variables. (n.d.). Www.tutorialspoint.com. Retrieved November 6, 2022, from https://www.tutorialspoint.com/scala/scala_variables.htm
- corob-msft. (n.d.). C Identifiers. Learn.microsoft.com. https://learn.microsoft.com/en-us/cpp/c-language/c-identifiers?view=msvc-170
- Scala - Variables. (n.d.). Www.tutorialspoint.com. Retrieved November 6, 2022, from https://www.tutorialspoint.com/scala/scala_variables.htm
- Scala - Operators. (n.d.). Www.tutorialspoint.com. https://www.tutorialspoint.com/scala/scala_operators.htm
- Busbee, K. L. (2018, December 15). Relational Operators. Press.rebus.community. https://press.rebus.community/programmingfundamentals/chapter/relational-operators/
- Kenneth Leroy Busbee, & Braunschweig, D. (2018, December 15). Logical Operators. Rebus.community; Pressbooks.

https://press.rebus.community/programmingfundamentals/chapter/logical-operators/#:~:text=A%20logical%20operator%20is%20a

- C - If..else, Nested If..else and else..if Statement with example. (2017, September 23). Beginnersbook.com. https://beginnersbook.com/2014/01/c-if-else-statement-example/

- Redirect Notice. (n.d.). Www.google.com. Retrieved November 6, 2022, from https://www.google.com/amp/s/www.techtarget.com/whatis/definition/loop%3famp=1
- *Introduction*. (n.d.). Scala Documentation. https://docs.scala-lang.org/tour/tour-of-scala.html
- *Scala Tutorial*. (n.d.). Www.tutorialspoint.com. https://www.tutorialspoint.com/scala/index.htm
- *History of Scala - javatpoint*. (2010). Www.javatpoint.com. https://www.javatpoint.com/history-of-scala
- *Scala*. (2016, August 11). Cleverism. https://www.cleverism.com/skills-and-tools/scala/
- *Scala: A History - Learn Scala from Scratch*. (n.d.). Educative: Interactive Courses for Software Developers. https://www.educative.io/courses/learn-scala-from-scratch/39BnN6DMZxr
- *Scala*. (2016, August 11). Cleverism. https://www.cleverism.com/skills-and-tools/scala/
- *From First Principles: Why Scala?* (n.d.). Www.lihaoyi.com. https://www.lihaoyi.com/post/FromFirstPrinciplesWhyScala.html
- *Scala*. (2016, August 11). Cleverism. https://www.cleverism.com/skills-and-tools/scala/
- *Scala Vs Java | Comparison Between Scala And Java*. (n.d.). Www.knowledgehut.com. https://www.knowledgehut.com/blog/programming/scala-vs-java
- FreeCodeCamp. (2020, January 10). *Interpreted vs Compiled Programming Languages: What's the Difference?* FreeCodeCamp.org. https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

- *Advantages And Disadvantages Of Compiler And Interpreter - Buggy Programmer*. (n.d.). https://buggyprogrammer.com/advantages-and-disadvantages-of-compiler-and-interpreter/
- Pedamkar, P. (2018, October 30). Uses Of Scala | Top 10 Useful Uses Of Scala In
- Real World. EDUCBA. https://www.educba.com/uses-of-scala/?source=leftnav
- *Uses Of Scala | Top 10 Useful Uses Of Scala In Real World*. (2018, October 30). EDUCBA. https://www.educba.com/uses-of-scala/

  Team, G. L. (2021, March 10). *Scala Tutorial - Learn Scala Step-by-Step Guide*. Great Learning Blog: Free Resources What Matters to Shape Your Career! https://www.mygreatlearning.com/blog/scala-tutorial/