Scala programming language

This material written by Nireekshan under Arjun guidelines, reviewed & corrected by Ramesh sir @DVS

Scala topic name: OOPS

Index

- Method overriding
- When should we go for overriding? (Please don’t say as I don’t know)
- Difference between Method overloading and Method overriding

- final keyword

    - final method
    - final class
    - Smart question: If we are using final keyword then are, we missing OOPs features?

✓ Abstract class

- Abstract keyword
- Types of methods
    - Implemented method
    - Unimplemented method

- Abstract method
- Abstract class
- Abstract variable
- If you have time
    - Please prepare given scenarios

✓ trait

- trait keyword
- What is trait?
- A single class can extends multiple traits
- If you have time
    - Please prepare given scenarios

✓ Normal class, Singleton object and Standalone class

- Normal class
- Singleton object
- Standalone class

✓ Singleton object

- Purpose of singleton object
- Difference between instance variable and singleton variable
- How to access singleton variable

✓ Companion object

- What is companion object
- Advantage
- Rules to define companion object

✓ Case class

- Case keyword
- Why case class?
- Advantage
- Difference between case class and normal class

OOPS

Full form of OOPS

- ✓ The full for of OOPS is “Object Oriented Programming System”
- ✓ Scala is pure Object-Oriented Programming language.
  - Scala represents everything is an object.

What is OOPS exactly?

- ✓ It’s a methodology to design a software using classes and objects.

Why should we use?

- ✓ It simplifies the software development by providing oops features.

OOPS features

- class
- object
- Data binding
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism etc.

1). class

Definition 1:

✓ A class is a specification (idea/plan/theory) of properties and actions of objects.

Definition 2:

✓ A class is a model for creating objects and it does not exist physically.

Syntax

```
class NameOfTheClass
{
        1.  constructor(s)
        2.  variables (data members)
        3.  methods (actions)
}
```

class keyword

✓ class is a keyword in scala programming language
✓ We can create a class by using class keyword

Inside class what we can define?

✓ class can contain mainly three parts,

- constructor(s)
- variables
- methods

Hey Nireekshan, what is the purpose of constructor(s), variables and methods?

✓ Yeah Good question Boss,

- Constructor purpose is to initialize instance variables
- Variables purpose is to represent data
- Methods purpose is to perform operations

Make a note

John :

- ✓ Hey Nireekshan, do I need to follow naming conventions for class while giving name to the class?

Nireekshan :

- ✓ Yes Boss, it's a good practice to follow naming convention while giving names to a class.
- ✓ class names should start with upper case and remaining letters are in lower case.
- ✓ If class name having multiple words, then every inner word should start with upper case letter.
- ✓ Examples:
  - ○ Student
  - ○ EmployeeInfo

Nireekshan :

- ✓ If you did not follow naming convention, then you will not get any error.
- ✓ But its highly recommended to follow to meet real time coding standards

Validate below names

○ Student	-	valid and highly recommended
○ student	-	valid but not recommended
○ EmployeeInfo	-	valid and highly recommended
○ empoyeeinfo	-	valid but not recommended

Program Name   Create a Student class with variables and method
Demo1.scala

```
class Student
{
	var id: Int = 10
	var name: String = "Nireekshan"

	def display()
	{
		println("Student id is: "+id)
		println("Student name is : "+name)
	}
}


object Demo1
{
	def main(args: Array[String])
	{
		println("Welcome to oops session")
	}
}
```

Compile   scalac Demo1.scala

Run   scala Demo1

Output

```
Welcome to oops session
```

Explanation about Demo1.scala

- ✓ Created Student class
- ✓ Inside Student class created two variables and one method
- ✓ Created one standalone class.
- ✓ Inside standalone class created main method

Info:

- ✓ Boss writing a class is not enough, we should learn how to access variables and methods.

How to access?

- ✓ Simple and beautiful answer is,
  - We should create an object to a class.

2). object

Info

- ✓ Please don't get confuse between,
  - object keyword
  - creating object to a class.
- ✓ Now we are discussing about creating object to class.

Then what is object keyword?

- ✓ In scala object keyword, by using object keyword we can create singleton class.
- ✓ Please hold your anxiety, we will learn full details about singleton class in upcoming chapter.
- ✓ Then let us start discussion about creating object to a class

Why should we create object for a class?

- ✓ Generally inside class we are defining variables and methods right.
- ✓ When we create an object to a class then only memory will be allocated to these variables and methods.
- ✓ So, hope you guys understand why we should create an object.
- ✓ Any questions the please…

What is an object?

Definition 1

- ✓ Instance of a class is known as an object.
- ✓ Instance
  - ○ It is a mechanism of allocating memory space for data members of a class

Definition 2

- ✓ Grouped item is known as an object.
  - ○ Object is a simple variable.
  - ○ This variable holds group of data.

Definition 3:

- ✓ Logical runtime entities are called as objects.

Definition 4:

- ✓ Real world entities are called as objects.

Syntax 1:

```
val nameofobject = new <NameOfTheClass>()
```

- ✓ We can create object for a class.
- ✓ We can create object by using new keyword

✓ nameofobject	-->	This is an object name
✓ NameOfTheClass ()	-->	This part is called as constructor.

- ✓ Regarding constructor we will learn in upcoming chapter.

Program Name

Create a Student class and object
Demo2.scala

```
class Student
{
	var id: Int = 101
	var name: String = "Nireekshan"

	def display()
	{
		println("Student id is: "+id)
		println("Student name is : "+name)
	}
}


object Demo2
{
	def main (args: Array[String])
	{
		println("Welcome to oops session")
		val s = new Student()
	}
}
```

Compile
Run

```
scalac Demo2.scala
scala Demo2
```

Output

```
Welcome to oops session
```

- ✓ Above program we have successfully created object
- ✓ Once after we create an object then happily, we can access variable and methods

Program Create a Student class and object to access variables and method
Name Demo3.scala

```
class Student
{
        var id: Int = 101
        var name: String = "Nireekshan"

        def display()
        {
                println("Student id is: "+id)
                println("Student name is : "+name)
        }
}


object Demo3
{
        def main (args: Array[String])
        {
                val s = new Student()
                s.display()
        }
}
```

Compile scalac Demo3.scala
Run scala Demo3

Output

```
Student id is: 101
Student name is: Nireekshan
```

Prasad

- ✓ Hey Nireekshan, can I create more than on object

Nireekshan

- ✓ Yes, Prasad we can create any number of objects for a class
- ✓ Make sure before creating object class should exists ☺

Program Name   Creating multiple objects to Student class
Demo4.scala

```
class Student
{
	var id: Int = 101
	var name: String = "Nireekshan"

	def display()
	{
		println("Student id is: "+id)
		println("Student name is : "+name)
	}
}


object Demo4
{
	def main (args: Array[String])
	{
		val s1 = new Student()
		val s2 = new Student()
		val s3 = new Student()

		s1.display()
		s2.display()
		s3.display()
	}
}
```

Compile
Run

```
scalac Demo4.scala
scala Demo4
```

Output

```
Student id is: 101
Student name is: Nireekshan

Student id is: 101
Student name is: Nireekshan

Student id is: 101
Student name is: Nireekshan
```

Program Name

Before creating an object class should exists otherwise, we will get error
Demo5.scala

```
class Student
{
        var id: Int = 101
        var name: String = "Nireekshan"

        def display()
        {
                println("Student id is: "+id)
                println("Student name is : "+name)
        }
}


object Demo5
{
        def main (args: Array[String])
        {
                val e = new Employee()
                e.display()
        }
}
```

Compile

```
scalac Demo5.scala
```

Run

```
scala Demo5
```

Output

```
error: not found: type Employee
```

Make a note

- ✓ An object exists physically in this world, but class does not exist.
- ✓ An object does not exist without class.
- ✓ A class can exist without an object.

3. Data Hiding:

What is data hiding?

- ✓ Data hiding is nothing but hiding of the data.

Why should we hide?

- ✓ Based on requirement sometimes we need to hide the data
- ✓ If we hide the data, then outside class can't access our data directly.

How to hide the data?

- ✓ By using private modifier, we can implement data hiding.
- ✓ The main advantage of data hiding is we can achieve security.

Program Name		Without using private keyword
Demo6.scala

```
class SbiAccount
{
	val balance: Double = 500
}

class HdfcBank
{
	def bankBalance()
	{
		val s = new SbiAccount()
		println(s.balance)
	}
}


object Demo6
{
	def main(args: Array[String])
	{
		val h = new HdfcBank()
		h.bankBalance ()
	}
}
```

Compile
Run

```
scalac Demo6.scala
scala Demo6
```

Output

```
500.0
```

Program Name
Data hiding by using private keyword
Demo7.scala

```
class SbiAccount
{
        private val balance: Double = 500;
}

class HdfcBank
{
        def bankBalance()
        {
                val a = new SbiAccount()
                println(a.balance)
        }
}

object Demo7
{
        def main(args: Array[String])
        {
                val h = new HdfcBank()
                h.bankBalance ()
        }
}
```

Compile
Run

```
scalac Demo7.scala
scala Demo7
```

Output

```
error: value balance in class SbiAccount cannot be accessed in SbiAccount
```

4. Abstraction

Definition 1:

- ✓ Abstraction means hiding the unnecessary data from the user.

Definition 2:

- ✓ Technically speaking abstraction means
    - **H**iding internal implementation details

      &

    - **H**ighlight the set of services what are offering.

Example:

- ✓ In bank ATM application, its highlight the set of services,
    - withdraw
    - balance
    - mini statement
- ✓ In bank ATM application used to hide,
    - Internal implementation.
- ✓ The main advantage of abstraction is we can achieve security.

5. Encapsulation:

- ✓ Binding of the data and corresponding methods into a single unit is called “Encapsulation”.
- ✓ Encapsulation = Data Hiding + Abstraction.
- ✓ If any scala class follows Data hiding & abstraction such type of class is called as an Encapsulated class.
- ✓ Example: A class is best example for Encapsulation.
- ✓ The central concept of Encapsulation is hiding data behind methods.

Methods

- ✓ We can define a method by using def keyword
- ✓ The purpose of method is to perform operations in class.
- ✓ Terminology related to methods,

    - ○ def keyword
    - ○ method name
    - ○ parenthesis
    - ○ parameters (if required)
    - ○ method body
    - ○ return type (if required)
    - ○ = symbol

- ✓ After creating the method then we need to call that method to do operation.

Make a note

- ✓ Method name along with its parameters is called method signature.

Types of methods

- ✓ Based on parameters methods are divided into two types,

1. Zero parameterised methods
2. Parameterized methods

Zero parameterized methods

- ✓ If method having no parameters, then those methods are called as zero parameterized method.

Program Name: Creating zero parameterised method and accessing by using object

Demo8.scala

```
class Test
{
	def m()
	{
		println("Welcome to methods concept")
	}
}

object Demo8
{
	def main(args: Array[String])
	{
		val t = new Test()
		t.m()
	}
}
```

Compile / Run

```
scalac Demo8.scala
scala Demo8
```

Output

```
Welcome to methods concept
```

Program Name

Creating zero parameterised method and accessing by using object
Demo9.scala

```
class Test
{
    def m()
    {
        var a=10

        if(a==10)
        {
            println(“a value is: “+a)
        }

        else
        {
            println(“a value is not 10”)
        }
    }
}

object Demo9
{
    def main(args: Arrays[String])
    {
        val t = new Test()
        t.m()
    }
}
```

Compile
Run

```
scalac Demo9.scala
scala Demo9
```

Output

```
a value is: 10
```

Make a note

- ✓ If method having no parameters, then we can ignore parenthesis while calling method.

Program Name

If method having no parameters then parenthesis is options while calling
Demo10.scala

```
class Test
{
	def m()
	{
		println("Welcome to methods concept")
	}
}

object Demo10
{
	def main(args: Array[String])
	{
		val t = new Test()
		t.m
	}
}
```

Compile
Run

```
scalac Demo10.scala
scala Demo10
```

Output

```
Welcome to methods concept
```

Parameterized methods

- ✓ If method having parameters, then those methods called as parameterized methods.
- ✓ If method having parameters, then while calling those methods we need to pass values

Program Creating parameterised method and accessing by using object
Name Demo11.scala

```
class Test
{
        def display(x: Int, y: Int)
        {
                println(x)
                println(y)
        }
}

object Demo11
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.display(10, 20)
        }
}
```

Compile
```
scalac Demo11.scala
```
Run
```
scala Demo11
```

Output

```
10
20
```

Program Name

Creating parameterised method and accessing by using object
Demo12.scala

```scala
class Test
{
	def display(x: String, y: String)
	{
		println(x)
		println(y)
	}
}

object Demo12
{
	def main(args: Array[String])
	{
		val t = new Test()
		t.display(10, 20)
	}
}
```

Compile
Run

```
scalac Demo12.scala
scala Demo12
```

Output

```
error: type mismatch
found: Int
required: String
```

Program Name

Creating parameterised method and accessing by using object
Demo13.scala

```
class Student
{
        def display(fname: String, lname: String)
        {
                println(‘First name is: ’+fname)
                println(‘Last name is: ’+lname)
        }
}

object Demo13
{
        def main(args: Array[String])
        {
                val s = new Student()
                s.display(“Nireekshan”, “Kasagani”)
        }
}
```

Compile scalac Demo13.scala
Run scala Demo13

Output

```
First name is: Nireekshan
Last name is: Kasagani
```

Program Name

Creating parameterised method and accessing by using object
Demo14.scala

```
class Student
{
        def display(name: String, age: Int)
        {
                println(‘Name is : ’+name)
                println(‘age is: : ’+age)
        }
}

object Demo14
{
        def main(args: Array[String])
        {
                val s  = new Student()
                s.display(“Nireekshan”, 16)
        }
}
```

Compile
Run

```
scalac Demo14.scala
scala Demo14
```

Output

```
Name is: Nireekshan
Age is: 16
```

Program Name   Creating parameterised method and accessing by using object
Demo15.scala

```
class Test
{
    def display(x: Int, y: Int)
    {
        if(x>y)
        {
            println(x)
        }

        else
        {
            println(y)
        }
    }
}


object Demo15
{
    def main(args: Array[String])
    {
        val t = new Test()
        t.display(10, 20)
    }
}
```

Compile   `scalac Demo15.scala`
Run   `scala Demo15`

Output

```
20
```

Sometimes Method may not be having curly braces

- ✓ This is purely for simplicity.
- ✓ Whenever code of the method is small then we can ignore the braces.
- ✓ When the code of the method is bigger then, it’s good to write within curly braces.

Program Name

Sometimes method may not be having curly braces
Demo16.scala

```scala
class Demo1
{
        def max(x:Int, y:Int): Int = if (x>y) x else y
}

object Demo16
{
        def main(args: Array[String])
        {
                val d = new Demo1()
                println(d.max(10, 20))
        }
}
```

Compile

```
scalac Demo16.scala
```

Run

```
scala Demo16
```

Output

```
20
```

return keyword

- return is a keyword.
- Writing a program only by using method is valid
- Writing a program method + return also valid

Syntax

```
class NameOfTheClass
{
        def methodName(): DataType=
        {
                return 100
        }
}
```

- If method having return statement,
    - We need to write a data type to method by using colon separator.
- After data type we need to write equals (=) symbol
- We can return any type of data type

Example 1

```
class Student
{
        def name(): String=
        {
                return "Nireekshan"
        }
}
```

Example 2

```
class Bank
{
        def balance(): Int=
        {
                return 100
        }
}
```

Program Name   Creating Bank class and method
Demo17.scala

```
class Bank
{
        def balance()
        {
                println("My balance is:")
        }
}


object Demo17
{
        def main(args: Array[String])
        {
                val b = new Bank()
                b.balance()
        }
}
```

Compile

```
scalac Demo17.scala
```

Run

```
scala Demo17
```

Output

```
My balance is:
```

Program Name — using return type
Demo18.scala

```
class Bank
{
        def balance(): Int=
        {
                print(“My balance is:”)
                return 100
        }
}


object Demo18
{
        def main(args: Array[String])
        {
                val b = new Bank()
                val bal = b. balance()
                print(bal)
        }
}
```

Compile — scalac Demo18.scala
Run — scala Demo18

Output

```
My balance is: 100
```

Make a note

- ✓ If method having return statement, then method calling we need to assign to a variable.
- ✓ This assigned variable holds the return value.

Why we need to assign Nireekshan?

- ✓ Good question.
- ✓ That assigned variable we can use further level in program
- ✓ Just observe below program

Program Name

using return type
Demo19.scala

```
class Bank
{
        def balance(): Int=
        {
                return 100
        }
}

object Demo19
{
        def main(args: Array[String])
        {
                val b = new Bank()

                val bal = b. balance()


                if(bal==0)
                {
                        println(“Balance is zero”)
                }

                else if(bal<0)
                {
                        println(“Balance is negative”)
                }

                else
                {
                        println(“Balance is:”+bal)
                }
        }
}
```

Compile
Run

```
scalac Demo19.scala
scala Demo19
```

Output

```
Balance is: 100
```

3. Constructors in scala

3.1 Purpose of constructor

- ✓ To initialize the instance variables. (Demo21.scala)

3.2 When constructor will get execute?

- ✓ We no need to call constructor explicitly.
- ✓ Constructor executes automatically at the time of object creation. (Demo22.scala)

3.3. How many times constructor will get execute?

- ✓ How many times we create objects that many times constructor will get execute.
- ✓ If we create 10 objects, then 10 times it executes.

How to define constructor?

- ✓ In scala, the syntax of first constructor used to define along with class only.

Program Name
Constructor
Demo20.scala

```
class Student()
{
        println("Constructor")
}


object Demo20
{
        def main(args: Array[String])
        {
                println("Welcome to main method")
        }
}
```

Compile
Run

```
scalac Demo20.scala
scala Demo20
```

Output

```
Welcome to main method
```

Program Name
Constructor
Demo21.scala

```
class Student()
{
        println("Constructor")
}


object Demo21
{
        def main(args: Array[String])
        {
                val s= new Student()
        }
}
```

Compile
Run

```
scalac Demo21.scala
scala Demo22
```

Output

```
Constructor
```

Program Name
Constructor
Demo22.scala

```
class Student()
{
        println("Constructor")
}

object Demo22
{
        def main(args: Array[String])
        {
                val s1 = new Student()
                val s2 = new Student()
        }
}
```

Compile
Run

```
scalac Demo22.scala
scala Demo22
```

Output

```
Constructor
Constructor
```

Make a note

- ✓ Developer no need to call explicitly.
- ✓ At the time of object creation, it executes automatically.

Make a note

- ✓ Developer need to call methods explicitly, but not constructor.

3.2 Types of constructor

- ✓ Primary constructor
  - without parameters
  - with parameters
- ✓ Auxiliary constructor

3.1.1 Primary constructor without parameters

- ✓ In scala, the syntax of first constructor used to define along with class only.
- ✓ It helps to optimize code.
- ✓ If constructor having no parameters, the it is called as zero parameterized constructor.

Program Name: Constructor Demo23.scala

```
class Student()
{
        println("Constructor")
}


object Demo23
{
        def main(args: Array[String])
        {
                val s=new Student()
        }
}
```

Compile

```
scalac Demo23.scala
```

Run

```
scala Demo23
```

Output

```
Constructor
```

Make a note:

- ✓ In scala, if you don't specify primary constructor then compiler creates a constructor automatically. (practically you can check by using scalap command)
- ✓ Based on requirement a class can contains any number of constructors.

3.1.2 Primary constructor with parameters

- ✓ If constructor having parameters, then we can called as parameterised constructor.
- ✓ If constructor having parameters, then during object creation we need to pass values to that parameterised constructor.

Program Name

Constructor with parameters
Demo24.scala

```
class Employee(name: String, age: Int)
{
        println(“Name is:” +name)
        println(“Age is sweet:” +age)
}


object Demo24
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
        }
}
```

Compile
Run

```
scalac Demo24.scala
scala Demo24
```

Output

```
Name is: Nireekshan
Age is sweet: 16
```

Program Constructor with parameters
Name Demo25.scala

```
class Employee(name: String, age: Int)
{
        def showDetails()
        {
                println(“Name is:” +name)
                println(“Age is sweet:” +age)
        }
}


object Demo25
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
                e.showDetails()
        }
}
```

Compile scalac Demo25.scala
Run scala Demo25

Output

```
Name is: Nireekshan
Age is sweet: 16
```

2 Auxiliary Constructor

- ✓ Auxiliary constructor also called as Secondary constructor.
- ✓ Based on requirement we can create more than one constructor in a class
- ✓ By using this, we can create Auxiliary constructors.

Rules to define Auxiliary constructor

- ✓ We can create Auxiliary constructor by using this
- ✓ We must call primary constructor from auxiliary constructor.
- ✓ By using this keyword, we can call the constructor from one to another.
- ✓ Whenever we are calling another constructor then the calling code should be first piece of code.

Program Name   Auxiliary Constructor with parameters
Demo26.scala

```
class Employee(id: Int, name: String)
{
        var age: Int = 0

        def this(id: Int, name: String, age: Int)
        {
                this(id, name)    // Calling primary constructor

                this.age = age
        }


        def showDetails()
        {
                println(“id is: ”+id)
                println(“Name is: ”+name)
                println(“Age is sweet: ”+age)
        }
}


object Demo26
{
        def main(args: Array[String])
        {
                var emp = new Employee(101,"Nireekshan",16);
                emp.showDetails()
        }
}
```

Compile   scalac Demo26.scala
Run   scala Demo26

Output

```
Id is: 101
Name is: Nireekshan
Age is sweet: 16
```

Make a note

- ✓ If instance variable name and parameter names are same, then to define instance variables we need to use this keyword on variables (Please observe above example)

Difference between constructor and method

Method	Constructor
✓ Purpose: Methods are used to perform operations	✓ Purpose: Constructors are used to initialize the instance variables.
✓ Name: Method name can be any name.	✓ Name: If auxiliary constructor then name should be this()
✓ Access: Methods we should call explicitly to execute	✓ Access: Constructor automatically executed at the time of object creation.

Inheritance

What is inheritance?

- ✓ Creating new classes from already existing classes is called as inheritance.
- ✓ The existing class is called a super class or base class or parent class.
- ✓ The new class is called as sub class or derived class or child class.
- ✓ Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.
    - ✓ Except the private variables and methods.
- ✓ One class can extend only one class at a time.
- ✓ One class cannot extend more than one class, because scala does not support multiple inheritance.

Make a note

- ✓ Without Inheritance we can't write even a simple Scala program also.
- ✓ Our First HelloWorld program is a child class to **Any** class in scala.
- ✓ Any class is pre-defined super class for every class in scala.
    - ○ Any super class is available in scala package.

How to implement inheritance?

- ✓ By using extends keyword we can implement the inheritance.

Advantages of Inheritance:

- ✓ Application development time is very less.
- ✓ Redundancy (repetition) of the code is reducing.

Tip

- ➢ Frankly tell me Boss, did you understand inheritance or not.
- ➢ If not, then please read it one more time after having cup of coffee.

Program Creating two class and applying inheritance concept
Name Demo27.scala

```
class One
{
	def m1()
	{
		println("m1 method from parent class")
	}
}

class Two extends One
{
	def m2()
	{
		println("m2 method from child class")
	}
}


object Demo27
{
	def main(args: Array[String])
	{
		val t = new Two()

		t.m1()
		t.m2()
	}
}
```

Compile
```
scalac Demo27.scala
```

Run
```
scala Demo27
```

Output

```
m1 method from parent class
m2 method from child class
```

Types of Inheritance:

1. Single Inheritance
2. Multilevel inheritance
3. Multiple inheritance

1. Single Inheritance:

✓ Creating a sub class from a single super class is called single inheritance.

Program	Creating two class and applying inheritance concept
Name	Demo28.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }
}

class Child extends Parent
{
        def study()
        {
                println("Studies done and waiting for job to get marriage")
                println("Requesting please do prayer for my job")
        }
}


object Demo28
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
        }
}
```

Compile	scalac Demo28.scala
Run	scala Demo28

Output

```
money + land + gold
Studies done and waiting for job to get marriage
Requesting please do prayer for my job
```

Program Creating two class and applying inheritance concept
Name Demo29.scala

```
class Parent
{
        var a: Int = 10
        var b: Int = 20

        def m1()
        {
                println("a value from parent: "+a)
                println("b value from parent: "+b)
        }
}


class Child extends Parent
{
        var d: Int = 30
        var e: Int = 40

        def m2()
        {
                println("d value from child: "+d)
                println("e value from child: "+e)
        }
}


object Demo29
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.m1()
                c.m2()
        }
}
```

Compile scalac Demo29.scala
Run scala Demo29

Output

```
a value from parent: 10
b value from parent: 20
d value from child: 30
e value from child: 40
```

Make a note

- ✓ Private data members not involve in Inheritance

Program Name

Creating two class and applying inheritance concept
Demo30.scala

```
class Parent
{
	private def m1()
	{
		println("private method m1 from parent class")
	}
}


class Child extends Parent
{
	def m2()
	{
		println("m2 method from child class")
	}
}


object Demo30
{
	def main(args: Array[String])
	{
		val c = new Child()

		c.m1()
		c.m2()
	}
}
```

Compile
Run

```
scalac Demo30.scala
scala Demo30
```

Output

```
error: value m1 is not a member of Child
```

2. Multi-level Inheritance:

✓ A class is derived from another derived class is called multi-level inheritance

Program Name   Creating two class and applying inheritance concept
Demo31.scala

```
class GrandFather
{
        def gfProperties()
        {
                println("only land from grandfather")
        }
}

class Father extends GrandFather
{
        def fProperties()
        {
                println("money + land + gold from father")
        }
}


class Child extends Father
{
        def study()
        {
                println("Studies done and waiting for job to get marriage")
                println("Requesting please do prayer for my job")
        }
}


object Demo31
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.gfProperties()
                c.fProperties()
                c.study()
        }
}
```

Compile
Run

```
scalac Demo31.scala
scala Demo31
```

Output

```
only land from grandfather
money + land + gold
Studies done and waiting for job to get marriage
Requesting please do prayer for my job
```

Program	Creating two class and applying inheritance concept
Name	Demo32.scala

```
class A
{
        var p: Int = 10
        var q: Int = 20;

        def m1()
        {
                println("p value : "+p)
                println("q value : "+q)
        }
}

class B extends A
{
        var r: Int = 30
        var s: Int = 40

        def m2()
        {
                println("r value : "+r)
                println("s value : "+s)
        }
}

class C extends B
{
        var t: Int = 50
        var u: Int = 60

        def m3()
        {
                println("t value : "+t)
                println("u value : "+u)
        }
}

object Demo32
{
        def main(args: Array[String])
        {
                val d = new C()

                d.m1()
                d.m2()
                d.m3()
        }
}
```

Compile	scalac Demo32.scala
Run	scala Demo32

Output

```
p value : 10
q value : 20
r value : 30
s value : 40
t value : 50
u value : 60
```

3. Multiple Inheritance:

- ✓ Creating a sub class from multiple super classes is called multiple inheritance.
- ✓ But java and Scala does not support multiple inheritance.

Why multiple inheritance is not supporting?

- ✓ There may be a chance of, two super classes may be having same variables or methods names, then the child will get ambiguity while accessing.

Program Trying to create a class from two parent classes
Name Demo33.scala

```
class A
{
        var i: Int = 10
}


class B
{
        var i: Int = 10
}


class C extends A, B
{
        var k=20
}


class Demo33
{
        def main(args: Array[String])
        {
                val c = new C()
                print(c.i)
        }
}
```

Compile scalac Demo33.scala
Run scala Demo33

Output

```
error: ';' expected but ',' found.

class C extends A, B
                 ^
one error found
```

Polymorphism

What is Polymorphism?

- ✓ The process of representing “one form in many forms”.
- ✓ Poly means many.
- ✓ Morphs means forms.
- ✓ Polymorphism means 'Many Forms'.

What is polymorphism

- ✓ The ability to exists in different forms is called “Polymorphism”.
- ✓ In scala an object or a method can exist in different forms, thus performing various tasks depending on the context.

Make a note

- ✓ This point is only for Java guys, remaining guys please get relax.
- ✓ In scala there is no static polymorphism, because no static keyword in scala.
- ✓ In scala only one polymorphism that is dynamic polymorphism.

Method parameters

- ✓ We can create a method which having parameters as well.

Program Name Method can contain parameters
Demo34.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

}


object Demo34
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
        }
}
```

Compile

```
scalac Demo34.scala
```

Run

```
scala Demo34
```

Output

```
Sum of two numbers: 30
```

Make a note

- ✓ In above program **add** is a method name **a** and **b** are called as parameters

Dynamic Polymorphism

- ✓ This is also called run time polymorphism.
- ✓ The polymorphism which is exhibited at runtime is called dynamic binding.
- ✓ The JVM only knows which one (variable or method) supposed to be execute at run time.

Program Name　　Dynamic polymorphism
Demo35.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

        def add(a: Int, b: Int, c: Int)
        {
                println("Sum of three numbers: "+(a+b+c))
        }
}


object Demo35
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
                s.add(10,20,30)
        }
}
```

Compile　　scalac Demo35.scala
Run　　scala Demo35

Output

```
Sum of two numbers: 30
Sum of three numbers: 60
```

Examples for dynamic Polymorphism

- ✓ Method overloading
- ✓ Method overriding

Method Overloading:

- ✓ In a class writing two or more methods with the same name but with difference parameters is called method overloading.

Program Name   Method overloading
Demo36.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

        def add(a: Int, b: Int, c: Int)
        {
                println("Sum of three numbers: "+(a+b+c))
        }
}

object Demo36
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
                s.add(10,20,30)
        }
}
```

Compile
Run

```
scalac Demo36.scala
scala Demo36
```

Output

```
Sum of two numbers: 30
Sum of three numbers: 60
```

Cases in overloading:

- ✓ In method overloading three cases are available

1. Difference in number of parameters
2. Difference in type of parameters
3. Difference in order of parameters

Case 1: Difference in number of parameters

- ✓ In overloading we can define two methods having same name with different number of parameters

Program Name: Case 1: Difference in number of parameters
Demo37.scala

```
class Addition
{
        def add(a: Int, b: Int)
        {
                println(a + b)
        }


        def add(a: Int, b: Int, c: Int)
        {
                println(a + b + c)
        }
}


object Demo37
{
        def main (args: Array[String])
        {
                val a = new Addition()

                a.add(40,40)
                a.add(20,20,20)
        }
}
```

Compile

```
scalac Demo37.scala
```

Run

```
scala Demo37
```

Output

```
80
60
```

Case 2: Difference in type of parameters

- ✓ In overloading we can define two methods having same name with different type of parameters

Program Name Case 2: Difference in type of parameters
Demo38.scala

```
class Addition
{
        def add(a: Int, b: Int)
        {
                println(a + b)
        }


        def add(a: Double, b: Double)
        {
                println(a + b)
        }
}


object Demo38
{
        def main(args: Array[String])
        {
                val a = new Addition()

                a.add(40, 40)
                a.add(20.1, 20.3)
        }
}
```

Compile
Run

```
scalac Demo38.scala
scala Demo38
```

Output

```
80
40.400
```

Case 3: Difference in order of parameters

- ✓ In overloading we can define two methods having same name with different order of parameters

Program Name Case 3: Difference in order of parameters
Demo39.scala

```
class Addition
{
        def add (a: Int, b: Double)
        {
                println(a + b)
        }

        def add (a: Double, b: Int)
        {
                println(a + b)
        }
}

object Demo39
{
        def main (args: Array[String])
        {
                val a = new Addition()

                a.add(40, 40.12)
                a.add(20.56, 20)
        }
}
```

Compile
Run

```
scalac Demo39.scala
scala Demo39
```

Output

```
80.12
40.56
```

Can we overload main () method?

- ✓ Yes, we can overload main method but JVM will always search for signature which having like main(args: Array[String]) to start program execution.
- ✓ The other user defined main method we need to call explicitly

Program Name
Overloading main method
Demo40.scala

```
object Demo40
{
        def main(args: Array[Int])
        {
                println(“Dupe Hero”)
        }


        def main(args: Array[String])
        {
                println("Original Hero")

        }
}
```

Compile
scalac Demo40.scala

Run
scala Demo40

Output

```
Original Hero
```

Program Overloading main method
Name Demo41.scala

```
object Demo41
{
	def main(a: Array[Int])
	{
		println(“Dupe main method with Array of Int”)
	}


	def main(args: Array[String])
	{
		println("Original main method")

		val b = Array(1,2,3)
		main(b)
	}
}
```

Compile

Run

```
scalac Demo41.scala
scala Demo41
```

Output

```
Original main method
Dupe main method with Array of Int
```

Method overriding

How to implement method overriding?

- ✓ We can implement method overriding by using override keyword

What is method overriding?

- ✓ Writing a method in super class and sub class which having same name and same parameters.

Program Name: Creating two class and applying inheritance concept
Demo42.scala

```
class Parent
{
        def m1()
        {
                println("Parent - m1")
        }
}


class Child extends Parent
{
        override def m1()
        {
                println("Child - m1")
        }
}


object Demo42
{
        def main(args: Array[String])
        {
                val c = new Child()
                c.m1()
        }
}
```

Compile:
```
scalac Demo42.scala
```
Run:
```
scala Demo42
```

Output

```
Child – m1
```

When should we go for overriding?

- ✓ If child class won’t like parent class method implementation, then happily child class can override parent class method.

Program Name   Creating two class and applying inheritance concept
Demo43.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }

        def marriage()
        {
                println(“Father decided Child marriage with uncle daughter: Her
                name is Subbalaxmi”)
        }
}


class Child extends Parent
{
        def study()
        {
                println("Studies done and got job")
                println("Thank you all for your prayers")
        }

}

object Demo43
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
                c.marriage()
        }
}
```

Compile   `scalac Demo43.scala`
Run   `scala Demo43`

Output

```
money + land + gold
Studies done and got job
Thank you all for your prayers
Father decided Child marriage with uncle daughter: Her name is Subbalaxmi
```

Program Name   Creating two class and applying inheritance concept
Demo44.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }


        def marriage()
        {
                println(“Father decided Child marriage with uncles daughter: Her
                name is Subbalaxmi”)
        }
}

class Child extends Parent
{
        def study()
        {
                println("Studies done and got job")
                println("Thank you all for your prayers")
        }

        override def marriage()
        {
                println(“Child wont like father decision about regarding
                marriage, so planning to marry Anushka in Banglore”)
        }
}

object Demo44
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
                c.marriage()
        }
}
```

Compile   `scalac Demo44.scala`
Run   `scala Demo44`

Output

```
money + land + gold
Studies done and got job
Thank you all for your prayers
Child wont like father decision about regarding marriage, so planning to marry
Anushka in Banglore
```

Program Name — Creating two class and applying inheritance concept
Demo45.scala

```
class Commercial
{
        def calculateBill(units: Int)
        {
                println ("Commercial Bill amount: "+units*5.00);
        }
}


class Domestic extends Commercial
{
        override def calculateBill(units: Int)
        {
                println("Domestic Bill amount: "+units*2.00);
        }
}


object Demo45
{
        def main (args: Array[String])
        {
                val c = new Commercial()
                c.calculateBill(100)

                val d=new Domestic()
                d.calculateBill(100)
        }
}
```

Compile — `scalac Demo45.scala`
Run — `scala Demo45`

Output

```
Commercial Bill amount: 500.0
Domestic Bill amount: 200.0
```

Difference between Method overloading and Method overriding

Overloading	Overriding
✓ Writing two or more methods with the same name but different parameters is called method overloading.	✓ Writing two or more methods with the same name with same parameters is called method overriding.
✓ No keyword is required.	✓ By using override keyword.
✓ Method overloading is done in the same class.	✓ Method overriding is done in super and sub classes, so here inheritance involves.
✓ In method overloading method return type can be same or different	✓ In method overriding method return type should be same.

final keyword

- ✓ In scala final keyword we can apply on two concepts,

  1. method
  2. class

- ✓ So, in scala,

  1. A method can be final
  2. A class can be final

1. final method

- ✓ In super class, if we declare a method as a final then, it is not possible to override this method in child class.
- ✓ So, final methods cannot be overridden

Program Name   Trying to override final method
Demo46.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }


        final def marriage()
        {
                println(“Father decided Child marriage with uncles daughter: Her
                name is Subbalaxmi”)
        }
}



class Child extends Parent
{
        def study()
        {
                println("Studies done and got job")
                println("Thank you all for your prayers")
        }

        override def marriage()
        {
                println(“Child wont like father decision about regarding
                marriage, so planning to marry Anushka in Banglore”)
        }
}


object Demo46
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
                c.marriage()
        }
}
```

Compile   scalac Demo46.scala
Run   scala Demo46

Output

```
overriding method marriage in class Parent of type ()Unit;
method marriage cannot override final member
override def marriage()
                    ^
```

2. final class

- ✓ If we declare a class as a final, then it is not possible to inherit this class.
- ✓ Final classes cannot be inherited

Program Name

Trying to inherit final class
Demo47.scala

```
final class Parent
{
        def m1()
        {
                println("m1 method from parent class")
        }
}

class Child extends Parent
{
        def m2()
        {
                println("m2 method from child class")
        }
}


object Demo47
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.m1()
                c.m2()
        }
}
```

Compile
Run

```
scalac Demo47.scala
scala Demo47
```

Output

```
error: illegal inheritance from final class Parent
class Child extends Parent
                ^
```

Summary of the story

- ✓ final methods cannot be overridden.
- ✓ final classes cannot be inherited.

Smart question: If we are using final keyword then, Are we missing OOPs features?

- ✓ Yes Boss ☹, if you are using final keyword then we are missing inheritance and overriding concepts.
- ✓ If it is really required, then only use final keyword otherwise enjoy oops features cheers.

abstract class

abstract keyword

- ✓ abstract is a keyword in scala.
- ✓ We can apply abstract keyword on three concepts,

  1. class
  2. method
  3. variable

- ✓ So, in scala,

  1. A class can be abstract
  2. A method can be abstract
  3. A variable can be abstract

Just recall once scala method

- ✓ As we discussed method have two parts,

  1. method name and parameters (if exists)
  2. method body

```
class Bank
{
        def balance()
        {
                println ("This is body of the method")
        }
}
```

There are two types of methods in-terms of implementation

1. Implemented methods.
2. Un-implemented method.

1. Implemented method

- A method which have a method name and method body then that method is called as implemented method.
- Also called as concrete method or non-abstract method

```
class Bank
{
        def balance()
        {
                println (“This is body of the method”)
        }
}
```

2. Un-implemented method

- A method which have only method name and no method body then that method is called as un-implemented method.
- Also called as non-concrete or abstract method.

```
abstract class Bank
{
        def interest()
}
```

- In above code, interest() method having no method body.
- So, this method is called as abstract method.

abstract method

- abstract class and trait can contain abstract methods.
- abstract method will not have method body.
- abstract method will be implemented in its sub class of abstract class.
- Explicitly we no need to give abstract keyword for abstract method.
- If any method having no method body means automatically that will become an abstract method.

Syntax

```
abstract class NameOfTheClass
{
        def nameOfTheMethod()
}
```

Example 1

```
abstract class Bank
{
        def interest()
        def offers()
}
```

Make a note

- If any class having abstract method, then that class should be declared as an abstract class.

abstract class

- We can create abstract class by using abstract keyword.
- A class which is declared as abstract is known as abstract class.
- abstract class can contain,

  - constructors
  - abstract variables
  - non-abstract variables
  - abstract methods
  - non-abstract methods
  - sub class

- abstract methods should be implemented in sub class of abstract class. (Demo48.scala)
- If sub class didn't provide implementation of abstract method, then we need to declare that sub class as abstract class. (Demo49.scala)
- If any class inheriting this sub class, then that sub class should provide the implementation for abstract methods. (Demo49.scala)
- *object creation is not possible for abstract class*. (Demo50.scala)

Reminder

- If any class having abstract method, then that class should be declared as an abstract class.

Syntax

```
abstract class NameOfTheClass
{
        Mainly it can contain,

        1. abstract methods
        2. non-abstract methods
}
```

Program Name	Abstract class and child class giving implementation for abstract methods
Demo48.scala

```
abstract class Bank
{
	def balanceCheck()
	{
		println(“Balance checking implementation ”)
	}


	def transfer()
	{
		println(“transfer implementation ”)
	}

	def interest()
}


class Sbi extends Bank
{
	def interest()
	{
		println("Sbi bank interest is 10 rupees")
	}
}


object Demo48
{
	def main(args: Array[String])
	{
		val s = new Sbi()

		s.balanceCheck()
		s.transfer()
		s.interest()
	}
}
```

Compile
Run

```
scalac Demo48.scala
scala Demo48
```

Output

```
Balance checking implementation
transfer implementation
Sbi bank interest is 10 rupees
```

Program Name

Abstract class and child class giving implementation for abstract methods
Demo49.scala

```
abstract class Bank
{

        def balanceCheck()
        {
                println(“Balance checking implementation ”)
        }

        def transfer()
        {
                println(“transfer implementation ”)
        }

        def interest()
}

abstract class Sbi extends Bank
{
        def offers()
        {
                println("Sbi bank having good offers")
        }
}

class Sbi1 extends Sbi
{
        def interest()
        {
                println("Sbi bank interest is 10 rupees")
        }
}

object Demo49
{
        def main(args: Array[String])
        {
                val s = new Sbi1()

                s.balanceCheck()
                s.transfer
                s.offers()
                s.interest()
        }
}
```

Compile
Run

```
scalac Demo49.scala
scala Demo49
```

Output

```
Balance checking implementation
transfer implementation
Sbi bank having good offers
Sbi bank interest is 10 rupees
```

Program Name

object creation is not possible for abstract class
Demo50.scala

```
abstract class Bank
{
        def balanceCheck()
        {
                println("Balance checking implementation ")
        }

        def transfer()
        {
                println("transfer implementation ")
        }

        def interest()
}

object Demo50
{
        def main(args: Array[String])
        {
                val s = new Bank()

        }
}
```

Compile
Run

```
scalac Demo50.scala
scala Demo50
```

Output

```
error: class Bank is abstract; cannot be instantiated
val s = new Bank()
                ^
```

abstract variable

- ✓ Abstract class can contain abstract variables which having no initialization.
- ✓ We need to initialize those variables in sub class of abstract class.

Program Name: Abstract variable
Demo51.scala

```
abstract class Bank
{
	var minBalance: Int
}


class Sbi extends Bank
{
	var minBalance: Int = 500

	def balance()
	{
		println("My balance is rupees: "+minBalance)
	}
}


object Demo51
{
	def main(args: Array[String])
	{
		val s = new Sbi()

		s.balance()
	}
}
```

Compile
Run

```
scalac Demo51.scala
scala Demo51
```

Output

```
My balance is rupees: 500
```

If you have time,

- ✓ If you have time, then please prepare these below four cases also about abstract class.

Make a note

- ✓ Syntactically all below programs are valid

Case 1

- ✓ abstract class may not contain anything

Program abstract class may not contain anything
Name Demo52.scala

```
abstract class A
{

        // No methods, no work, be cool...!!! Dude.

}
```

Compile scalac Demo52.scala
Run scala Demo52

Output

Case 2

- ✓ abstract class may contain all abstract methods

Program abstract class may contain all abstract methods
Name Demo53.scala

```
abstract class A
{
        def m1()
        def m2()
        def m3()
}
```

Compile scalac Demo53.scala
Run scala Demo53

Output

Case 3

- ✓ abstract class may contain abstract methods and non-abstract methods

Program abstract class may contain abstract methods and non-abstract methods
Name Demo54.scala

```
abstract class A
{
        def m1()
        {

        }

        def m2()
        def m3()
}
```

```
Compile    scalac Demo54.scala
Run        scala Demo54

Output
```

Case 4.

- ✓ abstract class may contain all implemented methods

Program abstract class may contain all implemented methods
Name Demo55.scala

```
abstract class A
{
        def m1()
        {

        }

        def m2()
        {

        }

        def m3()
        {

        }
}
```

```
Compile    scalac Demo55.scala
Run        scala Demo55

Output
```

trait

trait

- trait is a keyword in scala

- This point is for Java guys:
    - By using trait keyword, we can create trait just like an interface in java

What is trait?

- A trait is just like an interface in java.
- We can create trait by using trait keyword.
- trait can contain,

    - abstract variables
    - non-abstract variables
    - abstract methods
    - default methods (non-abstract methods)
    - sub class

- abstract methods will be implemented in sub class of trait. (Demo56.scala)
- If sub class didn’t provide implementation of abstract method, then we need to declare that sub class as abstract class. (Demo57.scala)
- If any class inheriting this sub class, then that sub class should provide the implementation for abstract methods. (Demo57.scala)
- object creation is not possible for trait (Demo58.scala)

Points to remember

- one class can extend any number of traits by using with keyword. (Demo.scala)
- one trait can extend multiple traits. (Demo.scala)
- trait cannot have constructors.
- trait is like an interface in Java.

Make a note

- In trait non-abstract methods are default methods.
- These default methods are by-default available to the child classes of traits.

Syntax

```
trait NameOfTheTrait
{
        Mainly it can contain,

        1. abstract methods
        2. default methods(non-abstract methods)
}
```

Program		Creating trait and child class for trait
Name		Demo56.scala

```
trait Bank
{
	def info()
	{
		println(“This is bank application”)
	}

	def interest()
}


class AndhraBank extends Bank
{
	def interest()
	{
		println("Interest is 10 rupees")
	}
}


object Demo56
{
	def main (args: Array[String])
	{
		val a = new AndhraBank()

		a.info()
		a.interest()
	}
}
```

Compile		scalac Demo56.scala
Run		scala Demo56

Output

```
This is bank application
Interest is 10 rupees
```

Program Name

Creating trait and child classes for trait
Demo57.scala

```
trait Bank
{
        def info()
        {
                println(“This is bank application”)
        }

        def interest()
}


abstract class TelanganaBank extends Bank
{
        def offers()
        {
                println("Giving silver coin for new customers")
        }
}


class TelanganaBankSub1 extends TelanganaBank
{
        def interest()
        {
                println("Interest is 5 rupees")
        }
}


object Demo57
{
        def main(args: Array[String])
        {
                val d = new TelanganaBankSub1()

                d.info()
                d.offers()
                d.interest()
        }
}
```

Compile
Run

```
scalac Demo57.scala
scala Demo57
```

Output

```
This is bank application
Giving silver coin for new customers
Interest is 5 rupees
```

Program Name

Object creation is not possible for trait
Demo58.scala

```
trait A
{
        def m()
        def n()
}

object Demo58
{
        def main(args: Array[String])
        {
                val d = new A()
        }
}
```

Compile
Run

```
scalac Demo58.scala
scala Demo58
```

Output

```
error: trait A is abstract; cannot be instantiated
val d = new A()
              ^
```

- ✓ A single class can extend multiple traits

Program Name	Class is inheriting two child classes Demo59.scala

```
trait Amazon
{
        def amazonShopping()

        def amazonInfo()
        {
                println("Welcome to Amazon shopping")
        }
}


trait FlipKart
{
        def flipKartShopping()

        def flipKartInfo()
        {
                println("Welcome to FlipKart shopping")
        }
}



class Customer extends Amazon with FlipKart
{
        def amazonShopping()
        {
                println("Bought Ponds powder dabba from amazon")
        }


        def flipKartShopping()
        {
                println("Bought hTC mobile from flipKart")
        }

}

object Demo59
{
        def main(args: Array[String])
        {
                val c = new Customer()

                c.amazonInfo()
                c.amazonShopping()

                c.flipKartInfo()
                c.flipKartShopping()

        }
}
```

Compile

```
scalac Demo59.scala
```

Run

```
scala Demo59
```

Output

```
Welcome to Amazon shopping
Bought Ponds powder dabba from amazon
Welcome to FlipKart shopping
Bought hTC mobile from flipKart
```

If you have time,

- ✓ If you have time, then please prepare these below four cases also about trait.

Make a note

- ✓ Syntactically all below programs are valid

Case 1

- ✓ trait may not contain anything
- ✓ trait Serializable, this is called as marker trait

Program Name: trait may not contain anything
Demo60.scala

```
trait A
{

        // No methods, no work, be cool...!!! Dude.

}
```

Compile: scalac Demo60.scala
Run: scala Demo60

Output

Case 2

- ✓ trait may contain all abstract methods

Program Name: trait may contain all abstract methods
Demo61.scala

```
trait A
{
        def m1()
        def m2()
        def m3()
}
```

Compile: scalac Demo61.scala
Run: scala Demo61

Output

Case 3

✓ trait may contain abstract methods and default methods

Program Name trait may contain abstract methods and default methods
Demo62.scala

```
trait A
{
        def m1()
        {

        }

        def m2()
        def m3()
}
```

Compile scalac Demo62.scala
Run scala Demo62

Output

Case 4.

✓ trait may contain all implemented methods

Program Name trait may contain all implemented methods
Demo63.scala

```
trait A
{
        def m1()
        {

        }

        def m2()
        {

        }

        def m3()
        {

        }
}
```

Compile scalac Demo63.scala
Run scala Demo63

Output

Hey Nireekshan, can you explain, when should we go for class, abstract class and trait?

class

- ✓ If we know complete implementation about the requirements, then we should go for class.
- ✓ A class having complete implementation.

abstract class

- ✓ If we know partial implementation about the requirements, then we should go for abstract class.
- ✓ Abstract class can contain implemented and un-implemented methods as well.

trait

- ✓ If we don’t know complete implementation about the requirements, then we should go for trait.

Normal class, Singleton object and Standalone class

Normal class

- Normal class we can create by using class keyword
- Inside normal class we can define instance variables and instance methods.

```
class NameOfTheClass
{
        // Instance variable
        // Instance method
}
```

Example

```
class NameOfTheClass
{
        var id = 101
        var name = “Nireekshan”

        def display()
        {
                println(“Id is: ”+id)
                println(“Name is: ”+name)
        }
}
```

- In above program *id* and *name* are instance variable
- display() method is an instance method
- Instance methods will use instance variables to perform operations or action.

Singleton object

- ✓ In Scala static keyword is not available, instead of static keyword we need to use singleton object to fulfil the requirement.
- ✓ Singleton object we can create by using object keyword
- ✓ Inside singleton object we can define singleton variables and singleton methods.

```
object NameOfTheSingleTonObject
{
        // singleton variable
        // singleton methods
}
```

What is the purpose of singleton object?

- ✓ Let us understand below example

Program Name   Instance variables
Demo64.scala

```
class Student (id: Int, name: String, collegeName: String)
{
        def showDetails()
        {
                println(id)
                println(name)
                println(collegeName)
        }
}

object Demo64
{
        def main(args: Array[String])
        {
                val s1 = new Student(1, “Arjun”, “DVS college”)
                val s2 = new Student(2, “Prasad”, “DVS college”)
                val s3 = new Student(3, “Nireekshan”, “DVS college”)

                println("First Student information")
                s1.showDetails()

                println("Second Student information")
                s2.showDetails()

                println("Third Student information")
                s3.showDetails()

        }
}
```

Compile
Run

```
scalac Demo64.scala
scala Demo64
```

Output

```
First Student information
1
Arjun
DVS college

Second Student information
2
Prasad
DVS college

Third Student information
3
Nireekshan
DVS college
```

What is instance variable?

- ✓ If value of the variable is changing from object to object such type of variable is called as instance variables.

What is singleton variable?

- ✓ If value of the variable is not changing from object to object such type of variable is called as singleton variables.
- ✓ Here, for singleton variables memory will be allocated only once and that variable we can reuse in everywhere.

Program explanation

- ✓ Above program id and name is changing from object to object.
- ✓ But college name is not changing from object to object, so this type of variable we should not declare at singleton level.
- ✓ So, to create singleton class we need to use object keyword

How to access singleton variables?

- ✓ We should access singleton variables and methods directly by using singleton object name

Program Name

Creating singleton object
Demo65.scala

```
class Student (id: Int, name: String, collegeName: String)
{
        def showDetails()
        {
                println (id)
                println (name)
                println (collegeName)
        }
}

object College
{
        val colName: String = “DVS college”
}

object Demo65
{
        def main(args: Array[String])
        {
                val s1 = new Student(1, “Arjun”, College.colName)
                val s2 = new Student(2, “Ramesh”, College.colName)
                val s3 = new Student(3, “Nireekshan”, College.colName)

                println("First Student information")
                s1.showDetails()

                println("Second Student information")
                s2.showDetails()

                println("Third Student information")
                s3.showDetails()
        }
}
```

Compile
Run

```
scalac Demo65.scala
scala Demo65
```

Output

```
First Student information
1
Arjun
DVS college

Second Student information
2
Prasad
DVS college

Third Student information
3
Nireekshan
DVS college
```

Standalone class

- Standalone class we can create by using object keyword.
- A class which can contain main method is called as Standalone class

```
object NameOfTheStandAloneClass
{
        // main method
}
```

Examples

- Till we have seen many standalone classes which having main method

Scala Companion Object

- In Scala program, syntactically it is valid if we are declaring a normal class name and singleton class name as the same name.
- If we are giving normal class name and singleton class as same, then such type of classes is called as companion object.
- The companion object is useful for implementing helper methods and factory.

Advantage

- We can use companion object to create instances for a specific class without using new keyword.

Define a normal class

```
class Animal(name: String)
{
        def display()
        {
                println("Animal name is:"+name)
        }
}
```

Define companion object for a Animal class

Rules to follow:

- We can define companion object by using object keyword.
- Name of companion object and class name should be same.
- These two should be in same source file.

Companion object responsible

- Companion object should define an apply() method.
- Internally this method will be creating object for corresponding class.

Define a companion object

```
object Animal
{
        def apply(name: String): Animal =
        {
                new Animal(name)
        }
}
```

Creating object to Animal class

- ✓ Now happily we can create object for Animal class without using new keyword.

```
val d = Animal(“Dog”)
val c = Animal(“Cat”)

d.display()
c.display()
```

Program Name   Creating companion object
                Demo66.scala

```
class Animal(name: String)
{
        def display()
        {
                println(“Animal name is: ”+name)
        }
}


object Animal
{
        def apply(name: String): Animal =
        {
                new Animal(name)
        }
}

object Demo66
{
        def main(args: Array[String])
        {
                val d = Animal(“Dog”)
                val c = Animal(“Cat”)

                d.display()
                c.display()
        }
}
```

Compile   scalac Demo66.scala
Run   scala Demo66

Output

```
Animal name is: Dog
Animal name is: Cat
```

case class

- ✓ A class which is declared with case keyword is called as case class.

Why case class?

- ✓ Its just like normal class but internally it creates companion object automatically
- ✓ By default case classes will get few methods automatically,
  - apply()
  - toString()
  - hashCode()
  - equals()
- ✓ This point if for java guys, scala case classes will helpful to reduce boiler plate code.

Why above methods are required?

- ✓ After creating objects for a class, sometimes based on requirement its required to compare the objects related stuff.
- ✓ These comparisons will be done by above methods.
- ✓ In Java programming a java developer should write these methods explicitly in their programs.
- ✓ But in scala these methods are by default available for case classes.

Case class Advantages

- ✓ By default, hashCode, equlas, toString methods are available.
- ✓ By default, classes are immutable.
- ✓ new keyword is not required to create object.

Difference between case classes and normal classes

- ✓ when you are comparing two normal classes objects with **==** operator then it will compare the addresses of those two objects.
- ✓ when you are comparing two case classes objects with **==** operator then it will compare the values of the objects.

Program Name — Creating normal class and comparing two objects
Demo67.scala

```
class Staff(name:String, age: Int)

object Demo67
{
        def main(args: Array[String])
        {
                val s1 = new Staff("David", 45)
                val s2 = new Staff(“David”, 45)

                println(s1 == s2)       //      false
        }
}
```

Compile — scalac Demo67.scala
Run — scala Demo67

Output

```
false
```

Program Name — Creating a case class comparing two objects
Demo68.scala

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

object Demo68
{
        def main(args: Array[String])
        {
                val s1 = Staff("David", 45)
                val s2 = Staff(“David”, 45)

                println(s1 == s2)
        }
}
```

Compile — scalac Demo68.scala
Run — scala Demo68

Output

```
true
```

Make a note

- ✓ We can create a parameterised constructor.
- ✓ So, these parameters we can declare as either val or var depends requirement

  - ○ val - Getter methods will create automatically
  - ○ var - Getter and Setter methods will create automatically

1. If constructor parameter declared as a val

- ✓ If parameter declared as a val the scala generates only a getter method.
- ✓ As we know val fields are immutable means we cannot change.

Program Name: If declared constructor parameter as val then
Demo69.scala

```
class Name(val name: String)

object Demo69
{
        def main(args: Array[String])
        {
                val n = new Name("Prasad")

                println(n.name)
                n.name = "Nireekshan"
        }
}
```

Compile
Run

```
scalac Demo69.scala
scala Demo69
```

Output

```
error: reassignment to val
```

2. If constructor parameter declared as a var

- If parameter declared as a var the scala generates both setter and getter methods.
- As we know var fields are mutable means we can change means we can set the value here setter methods work.

Program Name If declared constructor parameter as var then
Demo70.scala

```
class Name(var name: String)

object Demo70
{
        def main(args: Array[String])
        {
                val n = new Name("Prasad")

                println(“Before modifying name is: ”+n.name)
                n.name = "Nireekshan"
                println(“After modifying name is: ”+n.name)
        }
}
```

Compile
Run

```
scalac Demo70.scala
scala Demo70
```

Output

```
Before modifying name is: Prasad
Before modifying name is: Nireekshan
```

Thank you ☺