**What is Scala?**

* Scala is a modern *multi-paradigm* programming language designed to express common programming patterns in a concise, elegant, and *type-safe* way.
* It smoothly integrates features of **object-oriented** and **functional languages**.

Definations:

1. Paradigm : Paradigm can also be termed as method to solve some problem. Programming paradigm is an approach to solve problem using some programming language.
2. multi-paradigm: Multi-paradigm progamming language supports more than one programming paradigm.In programming, there is no one best method or efficient method to solve a problem. Every problem can be tackled through different approaches. With this in mind, some programming languages are designed in such a way that they support different programming paradigms. For example, C++ supports functional paradigm, object-oriented paradigm, imperative paradigm, etc.
3. Type-safety :  Type safety is the extent to which a programming language discourages or prevents type errors. A type error is erroneous or undesirable program behaviour caused by a discrepancy between differing data types for the program's constants, variables, and methods (functions), e.g., treating an integer (int) as a floating-point number (float). Type safety means that the compiler will validate types while compiling, and throw an error if you try to assign the wrong type to a variable. A type-safe language is one where the only operations that one can execute on data are the ones that are condoned by the data's type. That is, if your data is of type X and X doesn't support operation y, then the language will not allow you to to execute y(X).
4. **Scala is object-oriented**

* Scala is a pure object-oriented language in the sense that every value is an object.
* Types and behavior of objects are described by classes and traits.
* Classes are extended by subclassing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

1. **Scala is functional**

* Scala is also a functional language in the sense that every function is a value.
* Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying.
* Scala’s case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages.
* Singleton objects provide a convenient way to group functions that aren’t members of a class.

1. **Scala is statically typed**

* Scala is equipped with an expressive type system that enforces at compile-time that abstractions are used in a safe and coherent manner. In particular, the type system supports:
  1. generic classes
  2. variance annotations
  3. upper and lower type bounds,
  4. inner classes and abstract type members as object members
  5. compound types
  6. explicitly typed self references
  7. implicit parameters and conversions
  8. polymorphic methods
* Type inference means the user is not required to annotate code with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

**Blocks**

You can combine expressions by surrounding them with {}. We call this a block.

**Function vs Method**

* A function is a group of statements that perform a task.
* Scala has both functions and methods and we use the terms method and function interchangeably with a minor difference.
* A Scala method is a part of a class which has a name, a signature, optionally some annotations, and some bytecode where as a function in Scala is a complete object which can be assigned to a variable.
* In other words, a function, which is defined as a member of some object, is called a method.
* A function definition can appear anywhere in a source file and Scala permits nested function definitions, that is, function definitions inside other function definitions

**Unit Vs Void**

* A difference is that because every Scala expression must have some value, there is actually a singleton value of type Unit, written (). It carries no information.

**Case Classes**

* Scala has a special type of class called a “case” class. By default, case classes are immutable and compared by value. You can define case classes with the case class keywords.

case class Point(x: Int, y: Int)

* You can instantiate case classes without new keyword.
* And they are compared by value.
* Case classes can be seen as plain and immutable data-holding objects that should exclusively depend on their constructor arguments.
* This functional concept allows us to
  + use a compact initialisation syntax (Node(1, Leaf(2), None)))
  + decompose them using pattern matching
  + have equality comparisons implicitly defined
* In combination with inheritance, case classes are used to mimic algebraic datatypes.
* If an object performs stateful computations on the inside or exhibits other kinds of complex behaviour, it should be an ordinary class.
* In Scala, by default, comparing objects will compare their identity, but in the case of case class instances, the equality is redefined to compare the values of the aggregated information.

**Lazy val**

The difference between them is, that a val is executed when it is defined whereas a lazy val is executed when it is accessed the first time.

var x = { println("x"); 15 }

lazy val y = { println("y"); x+1 }

println("-----")

x = 17

println("y is: " + y)

Output of above code is:

x

-----

y

y is: 18

**Objects**

* Objects are single instances of their own definitions. You can think of them as singletons of their own classes.
* You can define objects with the object keyword.

object IdFactory {

private var counter = 0

def create(): Int = {

counter += 1

counter

}

}

* You can access an object by referring to its name.

val newId: Int = IdFactory.create()

println(newId) // 1

val newerId: Int = IdFactory.create()

println(newerId) // 2

**Singleton Objects**

* An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val.
* As a top-level value, an object is a singleton.
* As a member of an enclosing class or as a local value, it behaves exactly like a lazy val.

**Companion objects**

* An object with the same name as a class is called a companion object.
* Conversely, the class is the object’s companion class.
* A companion class or object can access the private members of its companion.
* static members in Java are modeled as ordinary members of a companion object in Scala.
* When using a companion object from Java code, the members will be defined in a companion class with a static modifier. This is called static forwarding. It occurs even if you haven’t defined a companion class yourself.

(factory methods???)

**Traits**

* Traits are used to share interfaces and fields between classes.
* They are similar to Java 8’s interfaces.
* Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters.
* Use the extends keyword to extend a trait. Then implement any abstract members of the trait using the override keyword
* **Subtyping**: Where a given trait is required, a subtype of the trait can be used instead.