**1. What is the difference between a var, a val and def?**

A var is a variable. It’s a mutable reference to a value. Since it’s mutable, its value may change through the program lifetime. Keep in mind that the variable type cannot change in Scala. You may say that a var behaves similarly to Java variables.

A val is a value. It’s an immutable reference, meaning that its value never changes. Once assigned it will always keep the same value. It’s similar to constants in another languages.

A def creates a method (and [a method is different from a function](https://tpolecat.github.io/2014/06/09/methods-functions.html) - thanks to AP for his comment). It is evaluated on call.

var x = 3 // x is of type Int. If you force it to be of type Any then this example would work

x = 4 // accepted by the language/compiler

x = "error" // not accepted by the compiler

val y = 3

y = 4 // would produce an error 'error: reassignment to val'

def fun(name: String) = "Hey! My name is: " + name

fun("Scala") // "Hey! My name is: Scala"

fun("Java") // "Hey! My name is: Java"

**Bonus:** what’s a lazy val? It’s almost like a val, but its value is only computed when needed. It’s specially useful to avoid heavy computations (using [short-circuit](https://en.wikipedia.org/wiki/Short-circuit_evaluation) for instance).

lazy val x = {

println("computing x")

3

}

val y = {

println("computing y")

10

}

y + y // x was still not computed, "computing x" was not yet printed

x + x // x is required, x is going to be computed, a "computing x" message will be printed \*once\*

**2. What is the difference between a trait and an abstract class?**

The first difference is that a class can only extend one other class, but an unlimited number of traits.

While traits only support type parameters, abstract classes can have constructor parameters.

Also, abstract classes are interoperable with Java, while traits are only interoperable with Java if they do not contain any implementation.

**3. What is the difference between an object and a class?**

An object is a singleton instance of a class. It does not need to be instantiated by the developer.

If an object has the same name that a class, the object is called a companion object (check [Q7 for more details](http://pedrorijo.com/blog/scala-interview-questions/#what-is-a-companion-object)).

class MyClass(number: Int, text: String) {

def classMethod() = ???

}

object MyObject {

def objectMethod() = ???

}

new MyClass(3, "text").classMethod()

MyClass.classMethod() // won't compile

MyObject.objectMethod() // you don't need to create an instance to call the method

**4. What is a case class?**

A case class is syntactic sugar for a class that is immutable and decomposable through pattern matching (because they have an apply and unapply methods). Being decomposable means it is possible to extract its constructors parameters in the pattern matching.

Case classes contain a companion object which holds the apply method. This fact makes possible to instantiate a case class without the new keyword. They also come with some helper methods like the .copy method, that eases the creation of a slightly changed copy from the original.

Finally, case classes are compared by structural equality instead of being compared by reference, i.e., they come with a method which compares two case classes by their values/fields, instead of comparing just the references.

Case classes are specially useful to be used as DTOs.

case class MyCaseClass(number: Int, text: String, others: List[Int])

val dto = MyCaseClass(3, "text", List.empty)

dto.copy(number = 5) // will produce an instance equal to the original, with number = 5 instead of 3

val dto2 = MyCaseClass(3, "text", List.empty)

dto == dto2 // will return true even if different references

class MyClass(number: Int, text: String, others: List[Int]) {}

val c1 = new MyClass(1, "txt", List.empty)

val c2 = new MyClass(1, "txt", List.empty)

c1 == c2 // will return false because they are different references

**5. What is the difference between a Java future and a Scala future?**

This one I had to google a little about it. I have never used Java futures, so it was impossible for me to answer.

Obviously, I was not the first to search for the differences between both futures. I found a [really clean and simple answer on StackOverflow](http://stackoverflow.com/a/31368177/4398050) which highlights the fact that the Scala implementation is in fact asynchronous without blocking, while in Java you can’t get the future value without blocking.

Scala provides an API to manipulate the future as a monad or by attaching callbacks for completion. Unless you decide to use the [Await](http://docs.scala-lang.org/overviews/core/futures.html#blocking-outside-the-future), you won’t block your program using Scala futures.

**6. What is the difference between unapply and apply, when would you use them?**

unapply is a method that needs to be implemented by an object in order for it to be an [extractor](http://docs.scala-lang.org/tutorials/tour/extractor-objects.html). Extractors are used in pattern matching to access an object constructor parameters. It’s the opposite of a constructor.

The apply method is a special method that allows you to write someObject(params) instead of someObject.apply(params). This usage is common in case classes, which contain a companion object with the apply method that allows the nice syntax to instantiate a new object without the new keyword.

**7. What is a companion object?**

If an object has the same name that a class, the object is called a companion object. A companion object has access to methods of private visibility of the class, and the class also has access to private methods of the object. Doing the comparison with Java, companion objects hold the “static methods” of a class.

Note that the companion object has to be defined in the same source file that the class.

class MyClass(number: Int, text: String) {

private val classSecret = 42

def x = MyClass.objectSecret + "?" // MyClass.objectSecret is accessible because it's inside the class. External classes/objects can't access it

}

object MyClass { // it's a companion object because it has the same name

private val objectSecret = "42"

def y(arg: MyClass) = arg.classSecret -1 // arg.classSecret is accessible because it's inside the companion object

}

MyClass.objectSecret // won't compile

MyClass.classSecret // won't compile

new MyClass(-1, "random").objectSecret // won't compile

new MyClass(-1, "random").classSecret // won't compile

**8. What is the difference between the following terms and types in Scala: Nil, Null, None, Nothing?**

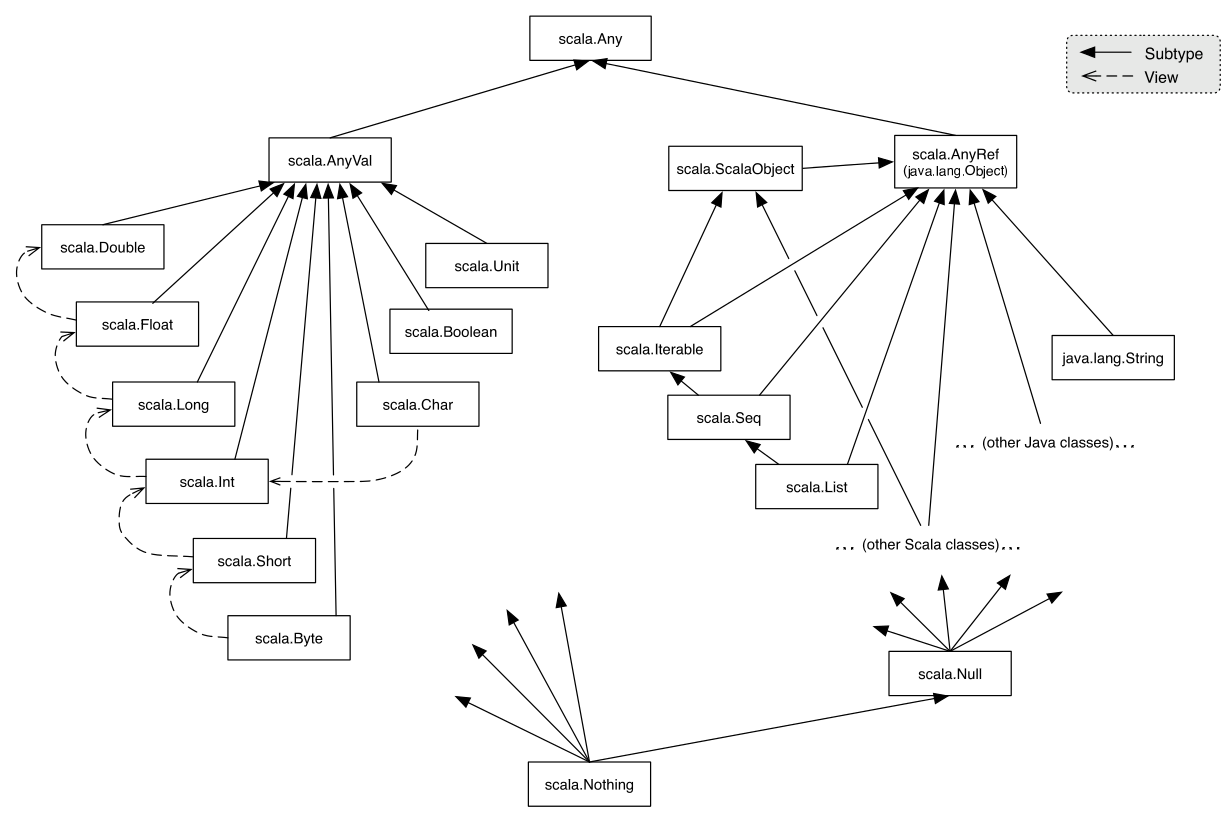
The None is the empty representation of the Option monad ([see answer #11](http://pedrorijo.com/blog/scala-interview-questions/#define-uses-for-the-option-monad-and-good-practices-it-provides)).

Null is a Scala trait, where null is its only instance. The null value comes from Java and it’s an instance of any object, i.e., it is a subtype of all reference types, but not of value types. It exists so that reference types can be assigned null and value types (like Int or Long) can’t.

Nothing is another Scala trait. It’s a subtype of any other type, and it has no subtypes. It exists due to the complex type system Scala has. It has *zero* instances. It’s the return type of a method that never returns normally, for instance, a method that always throws an exception. [The reason Scala has a bottom type is tied to its ability to express variance in type parameters.](http://james-iry.blogspot.pt/2009/08/getting-to-bottom-of-nothing-at-all.html).

Finally, Nil represents an empty List of anything of size zero. Nil is of type List[Nothing].

All these types can create a sense of emptiness right? Here’s a little help [understanding emptiness in Scala](http://www.nickknowlson.com/blog/2013/03/31/representing-empty-in-scala/).



**9. What is Unit?**

Unit is a type which represents the absence of value, just like Java void. It is a subtype of scala.AnyVal. There is only one value of type Unit, represented by (), and it is not represented by any object in the underlying runtime system.

**10. What is the difference between a call-by-value and call-by-nameparameter?**

The difference between a call-by-value and a call-by-name parameter, is that the former is computed before calling the function, and the later is evaluated when accessed.

Example: If we declare the following functions

def func(): Int = {

println("computing stuff....")

42 // return something

}

def callByValue(x: Int) = {

println("1st x: " + x)

println("2nd x: " + x)

}

def callByName(x: => Int) = {

println("1st x: " + x)

println("2nd x: " + x)

}

and now call them:

scala> callByValue(func())

computing stuff....

1st x: 42

2nd x: 42

scala> callByName(func())

computing stuff....

1st x: 42

computing stuff....

2nd x: 42

As it may be seen, the call-by-name example makes the computation **only when needed**, and **every time** it is called, while the call-by-value **only computes once**, but it **computes before invoking** the function (callByName).

**11. Define uses for the Option monad and good practices it provides.**

The Option monad is the Scala solution to the null problem from Java. While in Java the absence of a value is modeled through the null value, in Scala its usage is discouraged, in flavor of the Option.

Using null values one might try to call a method on a null instance, because the developer was not expecting that there could be no value, and get a NullPointerException. Using the Option, the developer always knows in which cases it may have to deal with the absence of value.

val person: Person = getPersonByIdOnDatabaseUnsafe(id = 4) // returns null if no person for provided id

println(s"This person age is ${person.age}") // if null it will throw an exception

val personOpt: Option[Person] = getPersonByIdOnDatabaseSafe(id = 4) // returns an empty Option if no person for provided id

personOpt match {

case Some(p) => println(s"This person age is ${p.age}")

case None => println("There is no person with that id")

}

**12. How does yield work?**

yield generates a value to be kept in each iteration of a loop. yield is used in for comprehensions as to provide a syntactic alternative to the combined usage of map/flatMap and filter operations on monads ([see answer #14](http://pedrorijo.com/blog/scala-interview-questions/#what-operations-is-a-for-comprehension-syntactic-sugar-for)).

scala> for (i <- 1 to 5) yield i \* 2

res0: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

**13. Explain the implicit parameter precedence.**

Implicit parameters can lead to unexpected behavior if one is not aware of the precedence when looking up.

So, what’s the order the compiler will look up for implicits?

1. implicits declared locally
2. imported implicits
3. outer scope (implicits declared in the class are considered outer scope in a class method for instance)
4. inheritance
5. package object
6. implicit scope like companion objects

A nice [set of examples can be found here](http://eed3si9n.com/implicit-parameter-precedence-again).

**14. What operations is a for comprehension syntactic sugar for?**

A for comprehension is a alternative syntax for the composition of several operations on monads.

A for comprehension can be replaced by foreach operations (if no yield keyword is being used), or by map/flatMap and filter (actually, while confirming my words I [found out about the withFilter method](http://docs.scala-lang.org/tutorials/FAQ/yield.html#translating-for-comprehensions)).

for {

x <- c1

y <- c2

z <- c3 if z > 0

} yield {...}

is translated into

c1.flatMap(x => c2.flatMap(y => c3.withFilter(z => z > 0).map(z => {...})))

[More examples by Loïc Descotte](https://gist.github.com/loicdescotte/4044169).

**15. Streams: What consideration you need to have when you use Scala’s Streams? What technique does the Scala’s Streams use internally?**

While Scala Streams can be really useful due to its lazy nature, it may also come with some unexpected problems.

The biggest problem is that Scala Streams can be infinite, but your memory isn’t. If used wrongly, streams can lead to memory consumption problems. One must be cautious when saving references to a stream. One common guideline, is not to assign a stream (head) to a val, but instead, make it a def.

This is a consequence of the technique behind streams: [memoization](https://en.wikipedia.org/wiki/Memoization)

**16. What is a value class?**

Have you ever had one of those nasty bugs where you were using an integer thinking it would represent something but it actually represented a totally different thing ? For instance, an integerrepresenting an age, and another representing an height getting mixed (180 years old and 25 centimetres tall do look weird no?).

Because of that, it’s considered a good practice to wrap primitive types into more meaningful types.

[Value classes](http://docs.scala-lang.org/overviews/core/value-classes.html) allow a developer to increase the program type safety without incurring into penalties from allocating runtime objects.

There are some [constraints](http://docs.scala-lang.org/overviews/core/value-classes.html#when-allocation-is-necessary) and [limitations](http://docs.scala-lang.org/overviews/core/value-classes.html#limitations), but the basic idea is that at compile time the object allocation is removed, by replacing the value classes instance by primitive types. [More details can be found on its SIP](http://docs.scala-lang.org/sips/completed/value-classes.html#expansion-of-value-classes).

**17. Option vs Try vs Either**

All of these 3 monads allow us to represent a computation that did not executed as expected.

An Option, as explained on [answer #11](http://pedrorijo.com/blog/scala-interview-questions/#define-uses-for-the-option-monad-and-good-practices-it-provides), represents the absence of value. It can be used when searching for something. For instance, database accesses often return Option in lookup queries.

Try is the monad approach to the Java try/catch block. It wraps runtime exceptions.

If you need to provide a little more info about the reason the computation has failed, Either may be very useful. With Either you specify two possible return types: the expected/correct/successful and the error case which can be as simple as a String message to be displayed to the user, or a full [ADT](https://en.wikipedia.org/wiki/Algebraic_data_type).

def personAge(id: Int): Either[String, Int] = {

val personOpt: Option[Person] = DB.getPersonById(id)

personOpt match {

case None => Left(s"Could not get person with id: $id")

case Some(person) => Right(person.age)

}

}

**18. What is function currying?**

Currying is a technique of making a function that takes multiple arguments into a series of functions that take a part of the arguments.

def add(a: Int)(b: Int) = a + b

val add2 = add(2)(\_)

scala> add2(3)

res0: Int = 5

Currying is useful in many different contexts, but most often when you have to deal with Higher order functions.

**19. What is Tail recursion?**

In “normal” recursive methods, a method calls itself at some point. This technique is used in the naive implementation of the [Fibonacci number](https://en.wikipedia.org/wiki/Fibonacci_number), for instance. The problem with this approach is that at each recursive step, another chunk of information needs to be saved on the stack. In some cases, an huge number of recursive steps can occur, leading to stack overflow errors.

Tail recursion solves this problem. In tail recursive methods, all the computations are done before the recursive call, and the last statement is the recursive call. Compilers can then take advantage of this property to avoid stack overflow errors, since tail recursive calls can be optimized by not inserting info into the stack.

You can ask the compiler to enforce tail recursion in a method with [@tailrec](https://www.scala-lang.org/api/current/scala/annotation/tailrec.html)

def sum(n: Int): Int = { // computes the sum of the first n natural numbers

if(n == 0) {

n

} else {

n + sum(n - 1)

}

}

@tailrec // just to ensure at compile time

def tailSum(n: Int, acc: Int = 0): Int = {

if(n == 0) {

acc

} else {

tailSum(n - 1, acc + n)

}

}

if we now run both versions, what would happen?

> sum(5)

sum(5)

5 + sum(4) // computation on hold => needs to add info into the stack

5 + (4 + sum(3))

5 + (4 + (3 + sum(2)))

5 + (4 + (3 + (2 + sum(1))))

5 + (4 + (3 + (2 + 1)))

15

tailSum(5) // tailSum(5, 0) because the default value

tailSum(4, 5) // no computations on hold

tailSum(3, 9)

tailSum(2, 12)

tailSum(1, 14)

tailSum(0, 15)

15

**20. What are High Order Functions?**

High order functions are functions that can receive or return other functions. Common examples in Scala are the .filter, .map, .flatMap functions, which receive other functions as arguments.

### What is Scala? Is it a Language or Platform? Does it support OOP or FP? Who is the father of Scala?

Scala stands for **SCA**lable **LA**nguage. Martin Odersky is the father of Scala.

Scala is a Multi-Paradigm Programming Language, which supports both Object-Oriented and Functional Programming concepts. It is designed and developed by Martin Odersky.

Scala is a Type-Safe Object-Functional Programming JVM Language. Scala runs on JVM(Java Virtual Machine).

Scala is a Hybrid Functional (Object-Oriented and Functional) Programming JVM Language. Scala has a Strong and Statically Type System. In Scala, all types are checked at compile-time.

### Is Scala Statically-Typed Language? What is Statically-Typed Language and What is Dynamically-Typed Language? What is the difference between statically typed and dynamically typed languages?

Yes, Scala is a Statically-Typed Language.

Statically-Typed Language means that Type checking is done at compile-time by compiler, not at run-time. The main Advantage of these kinds of Languages is: As a Developer, we should care about writing right code to avoid all compile-time errors. As Compiler checks many of the errors at compile-time, we don’t get much issues or bugs at run-time.

**Examples**:- Java, Scala, C, C++, Haskell etc.

Dynamically-Typed Language means that Type checking is done at run-time, not at compile-time by compiler. As a compiler won’t check any type checking at compile-time, We can expect more run-time issues or bugs.

**Example**:- Groovy, JavaScript, Ruby, Python, Smalltalk etc.

### Is Scala a Pure OOP Language? Is Java a Pure OOP Language?

Pure Object-Oriented Programming Language means that everything should be an Object.

Java is not a Pure Object-Oriented Programming (OOP) Language because it supports the following two Non-OOP concepts:

* Java supports primitive data types. They are not objects.
* Java supports Static members. They are not related to objects.

Yes, Scala is a Pure Object-Oriented Programming Language because in Scala, everything is an Object and everything is a value. Functions are values and values are Objects.

Scala does not have primitive data types and also does not have static members.

### Does Scala support all Functional Programming concepts? Does Java 8 support all Functional Programming concepts?

Yes, Scala supports all Functional Programming (FP) concepts. Java 8 has introduced some Functional Programming constructs, but it does NOT support all Functional Programming concepts.

For instance, Java 8 does not support Pattern Matching, Function Currying, Implicits etc.

### What are the major advantages of Scala Language? Are there any drawbacks of Scala Language?

If we use Scala Language to develop our applications, we can get the following benefits or advantages and drawbacks:  
**Advantages of Scala Language:-**

* Simple and Concise Code
* Very Expressive Code
* More Readable Code
* 100% Type-Safe Language
* Immutability and No Side-Effects
* More Reusable Code
* More Modularity
* Do More With Less Code
* Very Flexible Syntax
* Supports all OOP Features
* Supports all FP Features. Highly Functional.
* Less Error Prone Code
* Better Parallel and Concurrency Programming
* Highly Scalable and Maintainable code
* Highly Productivity
* Distributed Applications
* Full Java Interoperability
* Powerful Scala DSLs available
* REPL to learn Scala Basics

**Drawbacks of Scala Language:-**

* Less Readable Code
* Bit tough to Understand the Code for beginners
* Complex Syntax to learn
* Less Backward Compatibility

**NOTE:-** We can write Scala Code either more readable or less readable way.

### What is the Main drawback of Scala Language?

Apart from many benefits of Scala, it has one major Drawback: Backward Compatibility Issue. If we want to upgrade to latest version of Scala, then we need to take care of changing some package names, class names, method or function names etc.

For instance, If you are using old Scala version and your project is using BeanProperty annotation. It was available in “scala.reflect” like “scala.reflect.BeanProperty” in old versions. If we want to upgrade to new Scala versions, then we need to change this package from “scala.reflect” to “scala.beans”.

### What is the main motto of Scala Language?

Like Java’s Motto “Write Once Run Anywhere”, Scala has **“Do More With Less” or “Do More With Less Code”** Motto.  
“Do More With Less” means that we can develop more complex program or logic with less code.

### What are the popular JVM Languages available now?

Java, Scala, Groovy and Closure are most popular JVM (Java Virtual Machine) languages.

Scala, Groovy and Closure JVM languages supports both Object-Oriented Programming Features and Functional Programming Features.

Java SE 8 supports all Object-Oriented Programming Features. However, it supports very few Functional Programming Features like Lambda Expressions, Functions, Type Inference, Higher-Order Functions.

### Like Java’s java.lang.Object class, what is the super class of all classes in Scala?

As we know in Java, the super class of all classes (Java API Classes or User Defined Classes) is java.lang.Object. In the same way in Scala, the super class of all classes or traits is “Any” class.

Any class is defined in scala package like “scala.Any”.

### What is default access modifier in Scala? Does Scala have “public” keyword?

In Scala, if we don’t mention any access modifier to a method, function, trait, object or class, the default access modifier is “public”. Even for Fields also, “public” is the default access modifier.

Because of this default feature, Scala does not have “public” keyword.

### What is “Type Inference” in Scala?

Types can be inferred by the Scala Compiler at compile-time. It is known as “Type Inference”. Types means Data type or Result type. We use Types at many places in Scala programs like Variable types, Object types, Method/Function Parameter types, Method/Function return types etc.

In simple words, determining the type of a variable or expression or object etc at compile-time by compiler is known as “Type Inference”.

### What are the similarities and differences between Scala’s Int and Java’s java.lang.Integer? What is the relationship between Int and RichInt in Scala?

**Similarities between Scala’s Int and Java’s java.lang.Integer:**

* Both are classes.
* Both are used to represent integer numbers.
* Both are 32-bit signed integers.

**Differences between Scala’s Int and Java’s java.lang.Integer:**

* Scala’s Int class does not implement Comparable interface.
* Java’s java.lang.Integer class implements Comparable interface.

Java’s Integer is something similar to Scala’s Int and RichInt. RichInt is a final class defined in scala.runtime package like “scala.runtime.RichInt”.

In Scala, the Relationship between Int and RichInt is that when we use Int in a Scala program, it will automatically convert into RichInt to utilize all methods available in that Class. We can say that RichInt is an Implicit class of Int. (We will discuss “What is Implicit and the advantages of Implicits in my next post).

### What is Nothing in Scala? What is Nil in Scala? What is the relationship between Nothing and Nil in Scala?

In Scala, Nothing is a Type (final class). It is defined at the bottom of the Scala Type System that means it is a subtype of anything in Scala. There are no instances of Nothing.

**Use Cases of Nothing In Scala:-**  
If Nothing does not have any instances, then when do we use this one in Scala Applications?

* Nil is defined using Nothing (See below for example).
* None is defined using Nothing.

object None extends Option[Nothing]

* We can use Nothing as a return type of methods which never return.
* We can use Nothing as a return type of methods which terminates abnormally.

Nil is an object, which is used to represent an empty list. It is defined in “scala.collection.immutable” package as shown below:

object Nil extends List[Nothing]

**Example:-**

scala> Nil

res5: scala.collection.immutable.Nil.type = List()

scala> Nil.length

res6: Int = 0

### What is Null in Scala? What is null in Scala? What is difference between Null and null in Scala?

Null is a Type (final class) in Scala. Null type is available in “scala” package as “scala.Null”. It has one and only one instance that is null.

In Scala, “null” is an instance of type scala.Null type.

**Example:-**

scala> val myNullRef : Null = null

myNullRef: Null = null

We cannot assign other values to Null type references. It accepts only ‘null’ value.

Null is a subtype of all Reference types. Null is at the bottom of the Scala Type System. As it is NOT a subtype of Value types, we can assign “null” to any variable of Value type.

**Example:-**

scala> val myInt : Int = null

:10: error: an expression of type Null is ineligible for implicit conversion

val myInt : Int = null

^

Here type mismatch error. found : Null(null) but required: Int. The implicit conversions between Null and Int are not applicable because they are ambiguous.

### What is Unit in Scala? What is the difference between Java’s void and Scala’s Unit?

In Scala, Unit is used to represent “No value” or “No Useful value”. Unit is a final class defined in “scala” package that is “scala.Unit”.

Unit is something similar to Java’s void. But they have few differences.

* Java’s void does not any value. It is nothing.
* Scala’s Unit has one value ()
* () is the one and only value of type Unit in Scala. However, there are no values of type void in Java.
* Java’s void is a keyword. Scala’s Unit is a final class.

Both are used to represent a method or function is not returning anything.

### What is the difference between val and var in Scala?

In Scala, both val and var are used to define variables. However, they have some significant differences.

* var stands for variable.
* val stands for value.
* As we know, variable means changeable and value means constant.
* var is used to define Mutable variables that means we can reassign values once its created.
* val is used to define Immutable variables that means we cannot reassign values once its created.
* In simple Java terminology, var means ‘variable’ and val means ‘final variable’.

### What is REPL in Scala? What is the use of Scala’s REPL? How to access Scala REPL from CMD Prompt?

REPL stands for Read-Evaluate-Print Loop. We can pronounce it as ‘ripple’. In Scala, REPL is acts as an Interpreter to execute Scala code from command prompt. That’s why REPL is also known as Scala CLI(Command Line Interface) or Scala command-line shell.

The main purpose of REPL is that to develop and test small snippets of Scala code for practice purpose. It is very useful for Scala Beginners to practice basic programs.

We can access REPL by using “scala” command. When we type “scala” command at CMD Prompt, we will get REPL shell where we can type and execute scala code.

D:\> scala

scala>

### What are the Scala Features?

Scala Language supports the following features:

* Supports both OOP-style(Imperative-Style) and Functional-Style Programming
* Pure Object-Oriented Programming Language
* Supports all Functional Features
* REPL(Read-Evaluate-Print Loop) Interpreter
* Strong Type System
* Statically-Typed Language
* Type Inference
* Supports Pattern Matching
* Supports Closures
* Supports Persistent Data Structures
* Uses Actor Model to develop Concurrency Applications
* Interoperable with Java
* Available all development tools – IDEs, Build Tools, Web Frameworks, TDD and BDD Frameworks

### How do we implement loops functionally? What is the difference between OOP and FP style loops?

We know how to implement loops in Object-Oriented style: Using Mutable Temporary variables, update the variable value and use Loop constructs. It is very tedious and unsafe approach. It is not Thread-Safe.

Object-Oriented style uses the following constructs to implement Loops:

* Loop Constructs
* Mutability
* Side Effects

We can implement same Loops differently in Functional way. It is Thread-Safe. We can use the following two techniques to implement the loops in functional style:

* Recursion
* Tail-Recursion
* Immutability
* No Side-Effects

### What is “Application” in Scala or What is Scala Application? What is “App” in Scala? What is the use of Scala’s App?

**Scala Application:**  
In Scala, App is a trait defined in scala package like “scala.App”. It defines main method. If an Object or a Class extends this trait, then they will become as Scala Executable programs automatically because they will inherit main method from Application.

The main advantage of using App is that we don’t need to write main method. The main drawback of using App is that we should use same name “args” to refer command line argument because scala.App’s main() method uses this name.

**Example:-**  
**Without Scala App:**

object MyApp {

def main( args: Array[String]){

println("Hello World!")

}

}

**With Scala App:**

object MyApp extends App{

println("Hello World!")

}

If we observe above two examples, in second example we have not defined main method because we have inherited from Scala App(Application).

Before Scala 2.9, we have scala.Application trait. But it is deprecated by scala.App since Scala 2.9 version.

### Does Scala support Operator Overloading? Does Java support Operator Overloading?

Java does not support Operator Overloading. Scala supports Operator Overloading.

The reason is that Java does not want to support some misleading method names like “+\*/”. Scala has given this flexibility to Developer to decide which methods/functions name should use.

When we call 2 + 3 that means ‘+’ is not an operator, it is a method available in Int class (or it’s implicit type). Internally, this call is converted into “**2.+(3)**“.

### What is an Expression? What is a Statement? Difference between Expression and Statement?

**Expression:**  
Expression is a value that means it will evaluate to a Value. As an Expression returns a value, We can assign it to a variable.

Example:- Scala’s If condition, Java’s Ternary operator.

**Statement:**  
Statement defines one or more actions or operations. That means Statement performs actions. As it does not return a value, we cannot assign it to a Variable.

Example:- Java’s If condition.

### What is the difference between Java’s “If..Else” and Scala’s “If..Else”?

**Java’s “If..Else”:**  
In Java, “If..Else” is a statement, not an expression. It does not return a value and cannot assign it to a variable.

Example:-

int year;

if( count == 0)

year = 2014;

else

year = 2015;

**Scala’s “If..Else”:**  
In Scala, “If..Else” is an expression. It evaluates a value i.e. returns a value. We can assign it to a variable.

val year = if( count == 0) 2014 else 2015

**NOTE:-**Scala’s “If..Else” works like Java’s Ternary Operator. We can use Scala’s “If..Else” like Java’s “If..Else” statement as shown below:

val year = 0

if( count == 0)

year = 2014

else

year = 2015

### Is Scala an Expression-Based Language or Statement-Based Language? Is Java an Expression-Based Language or Statement-Based Language?

In Scala, everything is a value. All Expressions or Statements evaluates to a Value. We can assign Expression, Function, Closure, Object etc. to a Variable. So Scala is an Expression-Oriented Language.

In Java, Statements are not Expressions or Values. We cannot assign them to a Variable. So Java is not an Expression-Oriented Language. It is a Statement-Based Language.

### Tell me some features which are supported by Java, but not by Scala and Vice versa?

* Java does not support Operator Overloading, but Scala supports it.
* Java supports **++** and **—** operators , but Scala does not support them.
* Java has Checked and Unchecked Exceptions, but Scala does not have Checked Exceptions.
* Scala does not support break and continue statements, but Java uses them.
* Scala does not have explicit Type casting, but Java supports this feature.
* Scala supports Pattern Matching, but Java does not.
* Java uses Primitive Data types, but Scala does not have.
* Java supports static members, but Scala does not have static members concept.
* Scala supports Implicits and Traits, Java does not support them.

**NOTE:-**This list goes beyond one page. However, these are some important points to remember about differences in Scala and Java features to face Scala Interviews.

### What is the difference between Function and Method in Scala?

Scala supports both functions and methods. We use same syntax to define functions and methods, there is no syntax difference.

However, they have one minor difference:

* We can define a method in a Scala class or trait. Method is associated with an object (An instance of a Class). We can call a method by using an instance of a Class. We cannot use a Scala Method directly without using object.
* Function is not associated with a class or trait. It is defined in a Scala Package. We can access functions without using objects, like Java’s Static Methods.

**NOTE:-** We will discuss about Class, Trait,Package, Object etc in my coming posts.

### How many public class files are possible to define in Scala source file?

In Java, we can define at-most one public class/interface in a Source file. Unlike Java, Scala supports multiple public classes in the same source file.

We can define any number of public classes/interfaces/traits in a Scala Source file.

### Like Java, what are the default imports in Scala Language?

We know, java.lang is the default package imported into all Java Programs by JVM automatically. We don’t need to import this package explicitly.

In the same way, the following are the default imports available in all Scala Programs:

* java.lang package
* scala package
* scala.PreDef

### How many operators are there in Scala and Why?

Unlike Java and like C++, Scala supports Operator Overloading. Scala has one and only operator that is “=” (equalto) operator. Other than this all are methods only.

For instance 2 + 3, here “+” is not an Operator in Scala. “+” is method available in Int class. Scala Compiler observes 2 and 3 are Integers and tries to find that “+” method in Int class. So Scala Compiler converts “2 + 3” expression into “2.+(3)” and make a call to “+” method on integer object “2” and pass integer object “3” as parameter to “+” method.

Both “2 + 3” and “2.+(3)” are equal. It’s just Scala’s syntactic sugar to write programs in Functional style.

### Mention Some keywords which are used by Java and not required in Scala? Why Scala does not require them?

Java uses the following keywords extensively:

* ‘public’ keyword – to define classes, interfaces, variables etc.
* ‘static’ keyword – to define static members.

Scala does not required these two keywords. Scala does not have ‘public’ and ‘static’ keywords.

* In Scala, default access modifier is ‘public’ for classes,traits, methods/functions, fields etc. That’s why, ‘public’ keyword is not required.
* To support OOP principles, Scala team has avoided ‘static’ keyword. That’s why Scala is a Pure-OOP Langauge. It is very tough to deal static members in Concurrency applications.

h3>What is PreDef in Scala? What is the main purpose of PreDef in Scala?

In Scala, PreDef is an object defined in scala package as “scala.PreDef”. It is an utility object.

It defines many utility methods as shown below:

* Console IO (print,println etc)
* Collection utility methods
* String utility methods
* Implicit conversion methods
* Assertion utility methods etc.

For instance, print, println, readLine, readInt, require etc methods are defined in PreDef object.

In Scala, PreDef is available to use its methods without importing in all Scala Programs because Scala Compiler imports this object into all compilation units like Class, Object, Trait etc. automatically.