1.What is a trait ? when to use ?

A trait is a concept predominantly used in object-oriented programming, which can extend the functionality of a class using a set of methods. Traits are similar in spirit to interfaces in java programming language. Unlike a class, scala traits cannot be instantiated and have no arguments or parameters. However, you can inherit (extend) them using classes and objects. Having said that, traits provide a specific set of method/functions that execute behavior to a class and expect that the class and expect that the class implements a set of methods that then parameterize the provided behavior.

A trait is used to define an object created as a mixture of methods that can be used by different classes without requiring multiple inheritances. However, there can be outliers wherein two traits have a method with the same name (naming collision), which have to be used by a class, in which case the ambiguity has to be resolved explicitly.

Traits can have both abstract and non-abstract methods, fields as its members. When you do not initialize a method in a trait, then they are abstract, while the ones that are initialized are called non-abstract. In the abstract methods, the class that implements the trait takes care of the initialization.

Scala trait syntax:

Trait Trait\_Name{

//Variables

//Methods

}

When to use a trait:

If you want to reuse some code between multiple classes, using a trait is an alternative for extending the class. In that case the trait may be the better option because it doesn't become part of the type of hierarchy, there for class that uses a trait in not an instance of that trait.

A trait can save you some manual copy/paste by offering compile-time copy/paste instead.

2.Difference between trait and sealed trait?

Both normal traits and sealed traits are common in scala applications: normal traits for interfaces which may have any number of subclasses, and sealed traits where the number of subclasses is fixed.

Traits :

Traits provide a way to reuse code and promote modular code design. They can be mixed in with classes to add functionality without using inheritance, which leads to more flexible and reusable code. Scala allows for multiple traits to be mixed in with a single class, which allows for more complex and powerful class hierarchies. This is in contrast to java, which only allows for single inheritance. Traits can contain both abstract and concrete methods, which allows for more flexible and modular code design. Concrete methods can be overridden by classes that mix in the trait, while abstract methods must be implemented.

Sealed trait: sealed provides exhaustive checking for our application. Exhaustive checking allows to check that all members of a sealed trait must be declared in the same file as of the source file, that means that all the possible known members of a trait that must be included are known by the compiler in advance. So, this gives us an advantage to prevent mistakes in our code.

3.What is an abstract class?

Abstract classes are classes that contain one or more abstract behaviors or methods. Objects or classes can be abstracted, which means that they are summarized into characteristics relevant to the current program's operation. Abstracts classes are used in all object-oriented (OOP) languages, including Java, C++, C# and VB.NET.

4.What is the difference between an java interface and a scala trait?

Java interfaces: In java, an interface is very similar to an abstract class,

* Interfaces are declared using the interface keyword, rather than the class keyword.
* A class can implement an interface using the implements keyword, rather than the extends keyword.
* Functions are declared as abstract methods, therefore. They don't provide an implementation.
* Functions are automatically declared as public visibility.
* Variables are automatically declared as public, static and final, since an interface cannot be directly instantiated.

The cool thing about interfaces is that they provide a way to perform a sort of multiple inheritance, since a class can implement more than one interface, while also eventually extending a single class.

public interface A {  
 public void foo();  
}  
  
public interface B {  
 public void bar();  
}  
  
public class C implements A, B {  
 public void baz();  
}

An interface can also extend another interface, by means of the same inheritance rules applied to standard java classes.

public interface A {  
 public void foo();  
}  
  
public interface B extends A {  
 public void bar();  
}

Scala traits: scala traits are very similar to java interfaces, since they provide a way to inject specific behaviors into a class, using a set of methods defined in the implemented trait. These behaviors should then be parametrized, according to the actual mission of the class. Scala traits could be seen as rich interfaces, since they provide a way to implement both abstract and non-abstract function definitions, out of the box. A scala class or object can inherit from multiple traits at the same time. For the single inheritance problem,

trait A {  
 def foo() = println("foo")  
}  
  
class B() extends A {  
 def bar() = println("bar")  
}

For the multiple inheritance problem,

trait A {  
 override def foo() = println("foo")  
}  
  
trait B {  
 override def foo() = println("bar")  
}  
  
class C() extends A with B {  
 def bar() = println("bar")  
}

Alternatively to class C() extends A with B, we could have also written class C() with A with B.

Beware that the order in which traits are listed actually matters, since the last trait will be considered first in case of traits that override methods (in our example, calling foo() on an instance of type C will print out “bar” instead of “foo”).

5.What is a singleton

Singleton pattern is probably the most widely used design pattern. It is a simple pattern, easy to understand and to use. Sometimes it is used in excess and in scenarios where it is not required. In such cases, the disadvantages of using it outweigh the advantages it brings. For this reason, the singleton pattern is sometimes considered an antipattern or pattern singleton.

The singleton method or singleton design pattern is one of the simplest design patterns. It ensures a class only has one instance and provides a global point of access to it.

When to use singleton method design:

* There must be exactly one instance of a class and it must be accessible to clients from a well-known access point.
* When the sole instance should be extensible by subclassing and clients should be able to use an extended instance without modifying
* Singleton classes are used for logging, driver objects, caching, thread pool, database connections.

6.What is a higher order function?

The functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. How to be a consumer of methods that take other functions as input parameters, such as using HOFs like map() and filter(). The syntax you use to define function input parameters and how to call a function once you have a reference to it. As a beneficial side effect of this discussion, once you are comfortable with this syntax, you will use it to define function parameters, anonymous functions, and function variables, and it also becomes easier to read the scaladoc for HOFs.

7.What is a closure

Closures are functions which use one or more free variables, and the return value of this function is dependent on these variables. The free variables are defined outside of the closure function and is not included as a parameter of this function. So the difference between a closure function and a normal function is the free variable. A free variable is any kind of variable which is not defined within the function and not passed as the parameter of the function. A free variable is not bound to a function with a valid value. The function does not contain any values for the free variable.

Example :

Val = b = 10 # free variable

//define closure

Def example(a:double) = a\*b/100

Calling the function: example(10000)

Input: b = 10

Output: double = 1000.0

A closure function can further be classified into pure and impure functions, depending on the type of the free variable. If we give the variable a type “var” then the variable tends to change the value any time throughout the entire code and thus may result in changing the value of the closure function. Thus, this closure is an impure function. On the other-hand if we declare the free variable of the type "val” then the value of the variable remains constant and thus making the closure function a pure one.

Example:

//adding two number

//create object

Object DEN{

//main method

Def main ( args: Array[String]) {

println(“sum(1) value = ” + sum(1))

println(“sum(2) value = ” + sum(2))

println(“sum(3) value = ” + sum(3))

}

var a = 4

//define closure function

Val sum = (b: Int) => b+a

}

Output

Sum(1) value = 5

Sum(2) value = 6

Sum(3) value = 7

8.What is a companion object? What are the advantages? Example

In scala, when you have a class with same name as Singlton object, it is called companion class, and the singleton object is called companion object. The companion class and its companion object both must be defined in the same source file.

Ex: class companionClass{

Def hello(){ println(“Hello, this is companion Class.”)}

}

Object companionObject{

Def main(args: Array[String]){

New CompanionClass().hello()

Println(“and this is Companion Object.”)

}

}

Output:

Hello, this is Companion Class.

And this is Companion Object.

Advantages:

Companion objects are beneficial for encapsulating things, and they act as a bridge for writing functional and object-oriented programming code. Using companion objects, the Scala programming code can be kept more concise as the static keyword need not be added to each and every attribute. Companion objects provide a clear separation between static and non-static methods in class because everything that is located inside a companion object is not a part of the classes runtime object but is available from a static context and vice versa.

Class FortStep{

Def classMethod: Unit = ???

// Lot of code

}

Object FortStep{

Def objectMethod: Unit = ???

//Some code

}

9.Nil vs Null vs null vs Nothing vs None vs Unit

Although it may seem like all these keywords are used for representing the absence of a value, each one has its own purpose.

Null and null: the null reference is used to represent an absent value, and Null with a capital ‘N’ its type.

Lets first start with the null reference, as its probably the most familiar one for most of us because of the dreaded NullPointerException it causes.

Ex: case class Car(make: String)

//Initializes an instance of Car with null reference

Val nullRefCar : Car = null

Try{ println(nullRefCar.make)}

Catch{case:npe:NullPointerException => println(“Null Pointer Error occurred: %s”. Format(npe))}

//Initialize an instance of Car type with argument as null

Val nullMakeCar = Car(null)

Println(nullMakeCar.make)

Null type: Null is the type of null reference. It extends all reference types including the custom classes and traits we define. This allows us to use the null value in place of any reference type. The Null type does not have any method or fields. It's defined as an abstract final class; therefore, it does not allow to be extended or instantiated. Null is the one the only instance of type Null.

Nil-the Empty List

Nil is an empty singleton object list that extends the List type; therefore, it has all fields and methods that any other List object has, and its usually used to initialize empty lists:

Val myList = Nil

Println(“a list is initialized with length %s”. Format(myList.length))

All List methods are applicable for Nil as well. Thus, it can be used safely in place of any regular List. A popular way of creating and populating new list is using Nil and the cons(::)operator:

Val conList = “A” :: “B” :: Nil

None: None is a subtype of option type. We have seen the problems caused when functions return null values in case, they do not have any values to return. This may cause calling programs to crash if it does not properly handle null. scala's best practice advise us to wrap the return value in the option type in cases where the function may not have a return value.

Ex:

Val stuReg:Map[Int, String] = Map(1 -> “solo”, 2 -> “mon”)

Def getStuName(stuReg:Map[Int, String], roll:Int):Option[String]={

StuReg.get(roll)

}

Def printStu(stu:Option[String]): Unit = {

Stu match {

case Some(str) => println(“Student Name is %s”.format(str))

case None => println(“No Student!!”)

}

}

Unit is the return type of functions returning nothing. Its equivalent to the void type in java except for one difference: the Unit type in scala has one singleton value that is (), but void in java does not have any value:

def functionReturnUnit:Unit = { """ do something, don't return anything """ } println("result of function returning Unit: %s".format(functionReturnUnit))

We will see an output printed in the console as “result of function returning Unit: ()”.

If we omit the return type and the “=” operator in the function definition. The scala compiler will implicitly convert it to a Unit return type:

Nothing:

Nothing is the absolute “no value” type in scala. It does not have any methods or values.

Any type is the root of the entire scala type system, and nothing extends the any type. Therefore, we can use nothing in place of any scala type both reference types and value types.

Nothing together with the Null type sits at the bottom of the type hierarchy. Therefore, it’s also a subtype of every other type in Scala — even the classes and traits we define. This property gives us a lot of advantages in situations such as defining a generic empty base class. Nil is such an example: Nil extends List[Nothing], thus allows it to be used with a List of any type, be it String, Int, or custom classes.

We can’t use Nothing as the return type of a function except in the case where the function throws an exception. This is because Nothing doesn’t have any value. One such use case is a function that logs an exception and throws another custom exception:

def logException(e:Exception):Nothing = { println("logging Exception: %s".format(e.getMessage)) throw new Exception("My New Exception") }

10.What is pure function?

Another feature that Scala offers to help you write functional code is the ability to write pure functions. A pure function can be defined like this:

A function f is pure if, given the same input x, it always returns the same output f(x)

The function’s output depends only on its input variables and its implementation

It only computes the output and does not modify the world around it

This implies:

It doesn’t modify its input parameters

It doesn’t mutate any hidden state

It doesn’t have any “back doors”: It doesn’t read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world

As a result of this definition, any time you call a pure function with the same input value(s), you’ll always get the same result. For example, you can call a double function an infinite number of times with the input value 2, and you’ll always get the result 4.

Examples of pure functions

Given that definition, as you can imagine, methods like these in the scala.math.\_ package are pure functions:

abs

ceil

max

These String methods are also pure functions:

isEmpty

length

substring

Most methods on the Scala collections classes also work as pure functions, including drop, filter, map, and many more.

In Scala, functions and methods are almost completely interchangeable, so even though we use the common industry term “pure function,” this term can be used to describe both functions and methods. If you’re interested in how methods can be used like functions, see the [Eta Expansion](https://docs.scala-lang.org/scala3/book/fun-eta-expansion.html) discussion.

11.What is SBT and how have you used it?

SBT is a build tool that helps one manage their Scala project, which includes building, compiling, testing, as well as managing libraries and dependencies. It also is equipped with its own plugins which allow for integration of other features.

sbt provides a wide range of features to make the process of building and managing Scala projects easy and efficient.[[5]](https://en.wikipedia.org/wiki/Sbt_(software)#cite_note-5) Some of the key features include:

Dependency management: Through its capacity to automatically download and handle project dependencies, sbt facilitates the usage of external libraries and frameworks.

[Incremental compilation](https://en.wikipedia.org/wiki/Incremental_compilation)':' sbt can recompile only parts of the code that have changed, resulting in significant time-saving during the development cycle.

Customizable build process: sbt is highly customizable, allowing developers to define custom build settings and configure the build process to align with the unique requirements of their projects.

[Plugin](https://en.wikipedia.org/wiki/Plug-in_(computing)) ecosystem: sbt boasts a plugin ecosystem that enhances its functionality, incorporating extra features such as code quality checks, deployment automation, and test coverage reports. The development and integration of new plugins is fully supported and encouraged.

[Continuous compilation and testing](https://en.wikipedia.org/wiki/Continuous_integration): With sbt, developers can set the system to automatically recompile and rerun tests whenever a source file is altered.

Multi-project builds: For developers working on multiple Scala projects, sbt offers helpful features to manage them within a single build.

[Interactive shell](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop): sbt comes equipped with an interactive shell, providing developers with a convenient method to execute tasks and commands throughout the development process.

Parallel task execution: sbt can execute tasks in parallel, speeding up build times for large projects.

Integration with IDEs: To optimize the development experience, sbt integrates smoothly with popular [Integrated Development Environments](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDEs), such as [IntelliJ IDEA](https://en.wikipedia.org/wiki/IntelliJ_IDEA) and [Visual Studio Code](https://en.wikipedia.org/wiki/Visual_Studio_Code).

12.What is currying?

Currying is a technique that transforms a function that takes multiple arguments into a sequence of functions that each take a single argument.

function areaCurried(width) {  
 return function (height) {  
 return width \* height;  
 };  
}

This function returns another function that takes the height parameter and returns the result. You can call this function by passing the width parameter first, and then passing the height parameter to the returned function. For example: areaCurried(10)(5); // returns 50 Alternatively, you can store the intermediate function in a variable and call it later. For example:

let areaWithWidth10 = areaCurried(10);  
areaWithWidth10(5); // returns 50  
areaWithWidth10(8); // returns 80

13.Difference between currying and higher-order functions

Currying is a specific technique where a function with multiple arguments is transformed into a sequence of functions each taking a single argument. - Higher-order functions, on the other hand, are functions that can take other functions as arguments or return them as results.

14.Difference between var and val?

The difference between val and var is that val makes a variable immutable — like final in Java — and var makes a variable mutable. Because val fields can't vary, some people refer to them as values rather than variables.

15.What is case class?

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching. It uses equal method to compare instance structurally. It does not use new keyword to instantiate object. All the parameters listed in the case class are public and immutable by default.

16.Why/when to use case class? Example

The toString, hashCode, and equals methods. When we use the case keyword, the Scala compiler adds these methods for free to avoid boilerplate code.

A companion object with apply and unapply methods, that's why there is no need to use new keywords to create instances of a Case classes copy method.

Easy to use in pattern matching. It is immutable because, by default, its constructor parameters are val.

In Scala, by default, case class and case objects are serializable.

17.Difference between case class and normal class?

[Class in Scala](https://www.includehelp.com/scala/classes-and-objects-in-scala.aspx) is a container for variables and related methods.

Syntax for creating a class in Scala:

class class\_name{  
 val fields;   
 def methods(parameters);   
 }

[Case Class in Scala](https://www.includehelp.com/scala/case-class-in-scala-how-to-create-a-case-object.aspx) is just like a common class in Scala used in modeling immutable class.

Syntax for creating case class in Scala:

case class class\_name (parameters)

Points of difference between Case Class and Class in Scala

Both have different syntax for defining. The case class is defined in a single statement with parameters ([syntax for defining case class](https://www.includehelp.com/scala/case-class-in-scala-how-to-create-a-case-object.aspx)) whereas the normal class is defined by defining method and fields ([syntax for defining class](https://www.includehelp.com/scala/classes-and-objects-in-scala.aspx)).

While creating objects of case class, new keyword is not used which is used to create instances of case class.

Regular classes do not support pattern matching whereas case class support pattern matching.

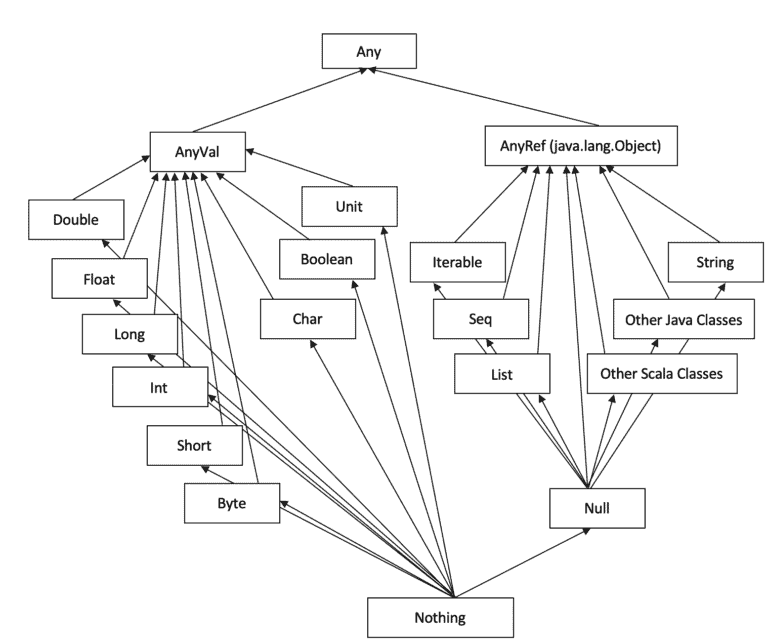
Regular classes can be inherited by another class whereas case classes cannot be inherited or extended.

Regular classes do not have any predefined method whereas case class has predefined hashcode and equals method.

The comparison of objects of class is done using reference comparison whereas the objects of case class compare the structure of objects.

18.Scala type hierarchy?

Scala is a statically typed language and provides an option for writing better and high-performance code. The JVM checks the type at compile-time and optimizes the code for better run-time performance. Scala doesn’t support primitive types like Java, and it has a very vast and diverse type-system. We’ll cover a few important high-level type classes in Scala, as shown in the diagram below:



19.What are partially applied functions?

The Partially applied functions are the functions which are not applied on all the arguments defined by the stated function i.e, while invoking a function, we can supply some of the arguments and the left arguments are supplied when required. we call a function we can pass less arguments in it and when we pass less arguments it does not throw an exception. these arguments which are not passed to function we use underscore( \_ ) as placeholder. Some important points:

Partially applied functions are helpful in minimizing a function which accepts many arguments to a function that accepts only some arguments.

It can replace any number of arguments defined by a function.

It is easier for users to utilize this method.

Syntax:

val multiply = (a: Int, b: Int, c: Int) => a \* b \* c  
  
// less arguments passed  
val f = multiply(1, 2, \_: Int)

20.What is tail recursion.

Tail recursion is a pattern of use that can be compiled or interpreted as iteration, avoiding the inefficiencies shown in the last section. A tail recursive function is one where every recursive call is the last thing done by the function before returning and thus produces the function's value.