1.What is a trait ? when to use ?

A trait is like an interface with a partial implementation. In scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

Code Reusability: Traits allow you to define reusable pieces of code that can be mixed into multiple classes, promoting code reuse and avoiding duplication.

Composition over Inheritance: Scala encourages the use of traits as a way to compose classes with shared behavior, rather than relying heavily on class inheritance. This promotes a more flexible and modular design.

Enforcing Contracts: Traits can define abstract methods that concrete classes must implement, effectively serving as contracts that classes must adhere to.

Providing Default Implementations: Traits can also include concrete method implementations, providing default behavior that can be overridden by classes that mix in the trait.

Mixing in Functionality: Scala supports multiple traits to be mixed into a single class, allowing for the combination of different pieces of functionality.

2.Difference between trait and sealed trait?

In Scala, both traits and sealed traits are used to define abstractions and facilitate code organization, but they serve different purposes and have distinct characteristics:

Trait:

A trait is a reusable component that encapsulates method and field definitions.

Traits can have both abstract methods (methods without implementation) and concrete methods (methods with implementation).

Traits can be mixed into classes using the with keyword, allowing classes to inherit behavior from multiple traits.

Traits can be instantiated directly, but they cannot have constructor parameters.

Sealed Trait:

A sealed trait is a special type of trait that restricts the inheritance hierarchy within a specific scope.

Sealed traits are typically used in pattern matching expressions, where the compiler can ensure that all possible subclasses of the sealed trait are known.

Subclasses of a sealed trait must be declared in the same file or in files that are in the same compilation unit.

Sealed traits are often used in conjunction with pattern matching to ensure exhaustiveness checking, where the compiler verifies that all possible cases are covered.

Sealed traits cannot be directly instantiated or extended outside of the scope where they are defined.

3.What is an abstract class?

An abstract class, defined with the "abstract" keyword, is a blueprint for other classes to inherit from. It can contain both abstract methods (methods without implementation) and non-abstract methods (methods with implementation). Abstract classes are utilized to achieve abstraction, a process that involves hiding complex implementation details while exposing only the essential functionality to the user.

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

Java Interface:

Primarily used for defining types and contracts , Does not support default method implementations, Cannot contain fields or constructors , Cannot have concrete method implementations, Supports multiple inheritance with limitations , Cannot be mixed in with classes to add behavior , No constructor support

Scala Trait:

Used for defining types, contracts, and behavior, Supports default method implementations, Can contain fields and constructors, Can have concrete method implementations, Supports multiple inheritance with linearization, Can be mixed in with classes to add behavior, Can have constructors.

5.What is a singleton

Scala is more object oriented language than Java so, Scala does not contain any concept of static keyword. Instead of static keyword Scala has singleton object. A Singleton object is an object which defines a single object of a class. A singleton object provides an entry point to your program execution. If you do not create a singleton object in your program, then your code compile successfully but does not give output. So you required a singleton object to get the output of your program. A singleton object is created by using object keyword.

EX: // singleton object

object Main

{

def main(args: Array[String])

{........

}

}

6.What is a higher order function?

A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result. In other words, it treats functions as first-class citizens, allowing them to be passed around like any other value. Higher-order functions enable powerful programming paradigms such as functional programming and are commonly used for tasks like abstraction, composition, and encapsulation of behavior. Examples of higher-order functions include map, filter, and reduce in functional programming languages like Scala and JavaScript. ex: map,filter,reduce,foreach

7.What is a closure

Scala Closures are functions which uses one or more free variables and the return value of this function is dependent of these variable. 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.

8.What is a companion object? What rae the advantages ? example

Companion object is known as an object whose name is same as the name of the class. Or In other words, when an object and a class have the same name, then that object is known as the companion object and the class is known as companion class. A companion object is defined in the same source file in which the class is defined. A companion object is allowed to access both private methods and private fields of the class.

Advantages of companion objects:

Shared Access: Companion objects can access private methods and fields of their companion classes, facilitating shared access to class internals.

Static-like Behavior: Since companion objects are instantiated once per class, they provide a convenient way to implement static-like behavior in Scala.

Factory Methods: Companion objects can serve as factory methods for creating instances of their companion classes, providing a centralized place for object creation logic.

Name Consistency: Using companion objects with the same name as their companion classes can improve code readability and maintainability by keeping related code together.

example:

class ComapanionClass{

def hello(){

println("Hello, this is Companion Class.")

}

}

object CompanoinObject{

def main(args:Array[String]){

new ComapanionClass().hello()

println("And this is Companion Object.")

}

}

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

Nil: Empty list in Scala.

Null: Special value representing no object reference.

null: Keyword in Java and JVM languages representing no object reference.

Nothing: Type representing non-termination or no value.

None: Absence of value in Scala's Option type.

Unit: Type representing absence of a meaningful value.

10.What is pure function?

A pure function is a function that, given the same input, will always return the same output and has no side effects.

def add(x: Int, y: Int): Int = {

x + y

}

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

SBT (Scala Build Tool) is a popular build tool for Scala projects. It provides a simple and powerful way to build, test, and run Scala projects. SBT uses a declarative build definition in the form of a build.sbt file, where you can specify project dependencies, settings, and tasks.

SBT is a versatile tool that simplifies and automates various aspects of Scala project development, from dependency management to testing and deployment. Its ease of use and extensibility make it a preferred choice for many Scala developers and projects.

12.What is currying?

In scala, method may have multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

In other words it is a technique of transforming a function that takes multiple arguments into a function that takes a single argument.

object MainObject {

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

a+b

}

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

var result = add(10)(10)

println("10 + 10 = "+result)

var addIt = add(10)\_

var result2 = addIt(3)

println("10 + 3 = "+result2)

}

}

13.Difference between currying and higher-order functions

Currying:

Is a technique used to transform a function with multiple arguments into a sequence of functions, each taking a single argument. It allows for partial function application and provides flexibility in function composition.

It is primarily applied to functions with multiple arguments to facilitate partial function application and create function chains.

Higher-Order functions:

Are functions that can take other functions as arguments or return functions as results. They enable powerful programming paradigms such as functional programming, where functions can be treated as first-class citizens.

Are used to abstract over actions, enabling code reuse and composability by passing functions as arguments or returning them as results.

14.Difference between var and val?

var:

A var is a mutable variable, meaning its value can be reassigned after it's been initialized.

You can modify the value of a var after it's been assigned.

Example: var x = 5, later x = 10 is valid.

val:

A val is an immutable variable, meaning its value cannot be reassigned once it's been initialized.

Once assigned, the value of a val cannot be changed.

Example: val y = 5, but y = 10 would result in a compilation error.

15.What is case class?

A case class in Scala is a special type of class that is primarily used for immutable data modeling. It is designed to be concise and convenient for defining classes whose main purpose is to hold data. Case classes automatically provide several features, such as:

-Automatic generation of getter methods for constructor parameters.

-Immutability of constructor parameters by default.

-Automatic generation of equals, hashCode, and toString methods based on constructor parameters.

-Support for pattern matching.

-A copy method for creating modified copies of instances.

16.Why/when to use case class? Example

Case classes are best used in Scala for defining immutable data structures and domain entities. They offer concise syntax, automatic methods for field access, equality comparison, and copying, making them ideal for modeling data-centric applications. Additionally, case classes integrate well with pattern matching, making them a powerful tool for data manipulation and transformation.

Example:

case class CaseClass(a:Int, b:Int)

object MainObject{

def main(args:Array[String]){

var c = CaseClass(10,10) // Creating object of case class

println("a = "+c.a) // Accessing elements of case class

println("b = "+c.b)

}

}

17.Difference between case class and normal class?

Case Class:

1. Case classes automatically generate methods for field access, equality comparison, hash code generation, and copying. This reduces the amount of boilerplate code needed for basic operations.

2. Case class constructor parameters are immutable by default. Once instantiated, their values cannot be changed.

3. Case classes are often used with pattern matching, a powerful feature in Scala for data extraction and manipulation. They integrate well with pattern matching syntax.

4. Case classes automatically implement equals and hashCode methods based on their constructor parameters, allowing for value-based equality comparison.

Normal Class:

1. Normal classes do not provide automatic methods for these operations. You have to manually implement methods for field access, equality comparison, and copying if needed.

2. In a normal class, you have to explicitly define immutability for fields by declaring them as val instead of var if you want them to be immutable.

3. While you can use normal classes with pattern matching, you have to manually implement pattern extraction methods or use extractors.

4. In a normal class, you have to manually implement equals and hashCode methods for value-based equality comparison.

18.Scala type hierarchy?

Scala's type hierarchy is structured as follows:

1. Any: The root type of Scala's type hierarchy. All other types in Scala are subclasses of Any.

AnyVal: The root type of Scala's value types (e.g., Int, Double, Boolean).

AnyRef: The root type of Scala's reference types (i.e., classes and traits).

Null: Subtype of all reference types; value is null.

Nothing: Subtype of all types; indicates "no value" or non-termination.

2. AnyVal: The root type of Scala's value types, extending Any.

Boolean: Boolean type with values true and false.

Byte: 8-bit signed integer type.

Char: 16-bit Unicode character type.

Short: 16-bit signed integer type.

Int: 32-bit signed integer type.

Long: 64-bit signed integer type.

Float: 32-bit floating-point type.

Double: 64-bit floating-point type.

3. Nothing: A subtype of every other type; represents "no value" or non-termination.

4. Null: A subtype of all reference types; value is null.

5. Unit: Scala's equivalent of void in Java; represents "no value" or side effect.

6. Option[T]: A generic type representing an optional value of type T, used to avoid null references.

Some[T]: Represents a value of type T.

None: Represents the absence of a value.

7. TupleN: A family of classes representing tuples of various sizes (e.g., Tuple2, Tuple3).

8. FunctionN: A family of traits representing functions of various arities (e.g., Function0, Function1, Function2).

19.What are partially applied functions?

Partially applied functions are functions that have been provided with fewer arguments than the number of arguments they expect. This results in a new function that takes the remaining arguments.

In other words, when you partially apply a function, you fix some parameters, creating a new function with the remaining parameters.

ex:

def add(x: Int, y: Int): Int = x + y

You can partially apply this function to create a new function that adds 2 to its argument:

val add2 = add(2, \_: Int)

Now, add2 is a new function that takes a single Int argument and adds 2 to it.

20.What is tail recursion.

Tail recursion is a special form of recursion in which the recursive call is the last operation performed by the function before returning its result. In other words, the recursive call (the "tail call") is in the tail position of the function's execution.

This has an important implication: in languages that support tail call optimization (TCO), such as Scala, the recursive call can be optimized by the compiler to reuse the current stack frame rather than creating a new one. This prevents stack overflow errors and improves the efficiency of recursive algorithms.

For a function to be tail recursive, it must meet two criteria:

1. The recursive call is the last operation performed by the function.

2. The result of the recursive call is returned directly, without further processing.