1.What is a trait ? when to use ?

it can have abstract and non abstract methods. supports multiple inheritance using traits. no diamond problem - solved using traits – Linearization. add trait to an object instance

traits in Scala are a powerful tool for code organization, reuse, and composition, offering a flexible alternative to class inheritance and promoting modular, composable designs.

2.Difference between trait and sealed trait?

Both traits and sealed traits are used for defining abstractions and code reuse, traits are more general-purpose and can be mixed into classes freely, whereas sealed traits provide stricter control over inheritance and are often used in pattern matching scenarios to ensure completeness and safety.

3.What is an abstract class?

Abstract classes are useful for defining common behavior and attributes among multiple subclasses while enforcing a certain structure for those subclasses to follow. They allow for both code reusability and abstraction in the design of object-oriented systems.

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

Java interfaces are primarily focused on defining types and contracts and have limited capabilities compared to Scala traits, Scala traits offer more advanced features such as default method implementations, fields, constructors, and mixin composition, making them more powerful for defining types, contracts, and behavior in Scala applications.

5.What is a singleton

Scala doesn't use "static" like Java. It relies on singleton objects created with the object keyword. These objects serve as entry points to program execution and are essential for Scala programs to produce output.

6.What is a higher order function?

A higher-order function is a function that either takes other functions as arguments or returns a function as its result. ex: map,filter,reduce,foreach

7.What is a closure

In Scala, closures are functions that depend on variables defined outside their own scope, called free variables. These variables are not parameters of the function but are accessed from the enclosing context. The presence of free variables allows closures to encapsulate behavior dependent on external state, distinguishing them from regular functions.

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

Companion objects, sharing the same name as their classes, provide shared access to private members, static-like behavior, serve as factory methods, and ensure name consistency for related code.

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 represents an empty list, Null and null represent absence of a value, Nothing represents a type with no instances, None represents absence of a value in an Option, and Unit represents absence of a meaningful value.

10.What is pure function?

A pure function always produces the same output for a given input and has no side effects. It's deterministic and doesn't modify external state.

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

SBT (Scala Build Tool) is a popular build tool for Scala projects. It's used for dependency management, building, testing, running, and task customization. I've used it extensively in Scala projects to simplify and automate the build and deployment process.

12.What is currying?

In Scala, methods can have multiple parameter lists. When a method is called with fewer parameter lists, it creates a function that takes the missing parameter lists as arguments. This technique transforms a function with multiple arguments into a series of functions, each taking a single argument.

13.Difference between currying and higher-order functions

currying is a specific technique for transforming functions with multiple arguments, while higher-order functions are a broader concept enabling functions to be treated as values, passed around, and manipulated in various ways.

14.Difference between var and val?

var is for mutable variables, and val is for immutable variables. It's generally recommended to use val whenever possible to promote immutable data and avoid accidental mutations, as immutability leads to safer and more predictable code.

15.What is case class?

A case class in Scala is a concise way to define immutable data structures. It provides automatic methods for accessing fields, equality comparison, and copying, making it convenient for modeling data.

16.Why/when to use case class? Example

case classes are well-suited for scenarios where immutable data structures, value-based equality comparison, and pattern matching are required, making them a fundamental building block for writing idiomatic and functional Scala code.

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 classes provide a convenient way to define immutable data structures with automatic methods for common operations, making them ideal for modeling data-centric applications. Normal classes offer more flexibility and control but require more manual implementation for basic operations.

18.Scala type hierarchy?

Scala's type hierarchy is structured as follows:

1. Any: The root type, superclass of all types.

AnyVal: Superclass of Scala's built-in value types.

AnyRef: Superclass of reference types.

Null: Subtype of all reference types, represents a null value.

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

2. AnyVal: Superclass of Scala's built-in value types.

3. Nothing: Subtype of every other type, represents "no value" or non-termination.

4. Null: Subtype of all reference types, represents a null value.

5. Unit: Represents "no value" or side effect.

6. Option[T]: Represents an optional value of type T.

7. TupleN: Represents tuples of various sizes.

8. FunctionN: Represents functions of various arities.

19.What are partially applied functions?

Partially applied functions are useful for creating new functions from existing ones, especially in scenarios where you need to pass a function as an argument to another function or where you want to reuse a function with different parameters.

20.What is tail recursion.

Tail recursion is a special type of recursion where the recursive call is the last operation executed in a function. This allows the compiler to optimize memory usage by reusing stack frames, preventing stack overflow errors. It's commonly used in functional programming languages like Scala to write efficient recursive algorithms.