Scala Interview Questions:

1. What is a trait ?

In Scala, a trait is a reusable component that defines methods, fields, and state. It can be mixed into classes to provide shared behaviour. Traits promote code reuse, support multiple inheritance, and enable the creation of modular components in Scala code.

1. Difference between trait and sealed trait?

Traits in Scala are reusable components that define methods, fields, and state and can be mixed into multiple classes. Sealed traits, on the other hand, are a subtype of traits that restrict inheritance to a closed set of subclasses within the same file. Sealed traits work well with pattern matching to ensure exhaustive handling of cases.

1. What is an abstract class?

An abstract class in programming serves as a blueprint for other classes. It cannot be directly instantiated and contains abstract methods, i.e. method declarations without implementations, that must be implemented by its subclasses. Abstract classes allow for the definition of common characteristics and methods that subclasses can inherit and implement.

1. What is the difference between an java interface and a Scala trait?

Java interfaces and Scala traits have some similarities but also key differences as follows:

1. Method Signatures:

- Java Interfaces: Define method signatures without implementations (prior to Java 8).

- Scala Traits: Can define method signatures, provide default implementations, and include concrete methods.

2. Multiple Inheritance:

- Java Interfaces: Allow multiple interface implementation.

- Scala Traits: Also support multiple inheritance, allowing classes to extend multiple traits.

3. Fields:

- Java Interfaces: Cannot define fields.

- Scala Traits: Can define fields in addition to methods.

4. Mixin Composition:

- Java Interfaces: Primarily used for contracts, not behaviour composition.

- Scala Traits: Enable code reuse and behaviour composition through mixin, providing default implementations and fields.

5. Exhaustiveness Checks:

- Java Interfaces: No exhaustiveness checks during pattern matching.

- Scala Traits: Sealed traits allow exhaustiveness checks, ensuring all cases are handled.

Scala traits offer more flexibility and capabilities compared to Java interfaces. They support method implementations, field definitions, and mixin composition, whereas Java interfaces focus mainly on contracts and multiple interface implementation.

1. What is a singleton?

A singleton is a design pattern that ensures only one instance of a class is created and provides a global point of access to that instance. It is commonly used for managing shared resources or creating a centralised, stateful object. Singleton classes have a static method to access the single instance, and the instance is created only when it is first requested.

1. What is a higher order function?

A higher-order function is a function that takes one or more functions as arguments or returns a function as its result. It allows functions to be treated as values and enables powerful functional programming techniques such as abstraction, modularity, and composition.

1. What is a closure?

A closure is a function that retains access to variables from its outer scope even after the outer function has finished executing. It allows functions to "remember" and access variables that are no longer in scope, enabling encapsulation and data privacy.

1. What is a companion object?

A companion object in Scala is an object with the same name as a class and is defined in the same file. It can access private members of the class and provides a way to define static-like methods or factory methods associated with the class. Companion objects enhance code organisation and allow convenient collaboration between the object and class.

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

Nil, Null, null, Nothing, None, and Unit are distinct concepts in Scala with different meanings:

1. Nil: Nil is an empty list of a specific type. It is used to represent an empty collection and is commonly used with lists in Scala.

2. Null: Null is a reference type that represents the absence of a value or a reference to no object. It is a subtype of all reference types and can be assigned to any reference variable.

3. null: null is a literal representing a null reference. It is used to indicate the absence of an object or the lack of a value for a reference type.

4. Nothing: Nothing is a bottom type in Scala that is a subtype of all types. It is used to represent a computation that does not result in a value or a method that never returns normally.

5. None: None is an object representing the absence of a value in Scala's Option type. It is used to indicate that a value is missing or that an operation did not produce a result.

6. Unit: Unit is a type in Scala that represents the absence of a meaningful value. It is similar to void in some other programming languages. Functions that do not return a value typically have a return type of Unit.

1. What is pure function?

A pure function is a function that consistently produces the same output for the same input and has no side effects. It does not modify external state and relies solely on its input parameters to generate a result. Pure functions are predictable, testable, and reusable.

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

SBT (Scala Build Tool) is a build tool used for Scala projects. It handles compilation, testing, dependency management, and packaging. SBT uses a “build.sbt” file to define project settings and dependencies. It provides an interactive shell for executing commands and supports plugins for extending its functionality. SBT simplifies the build process and helps manage Scala projects effectively.

SBT is commonly used for:

1. Compiling and running Scala projects.

2. Managing dependencies for Scala projects.

3. Running tests using popular testing frameworks.

4. Packaging projects into deployable artifacts like JAR files.

5. Automating tasks with custom-defined tasks.

6. Integrating with Continuous Integration (CI) systems for build and test automation.

1. What is currying?

Currying is a technique in functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking one argument. It allows partial application of arguments and creating specialised functions. Currying facilitates function composition and code reuse.

1. Difference between currying and higher-order functions?

Currying is a technique that transforms a function with multiple arguments into a sequence of functions, while higher-order functions are functions that can accept or return other functions. Currying focuses on function transformation, while higher-order functions operate on functions themselves.

1. Difference between var and val?

“var” is used to declare mutable variables whose values can be changed, while “val” is used to declare immutable variables whose values cannot be reassigned once assigned.

1. What is case class?

A case class in Scala is a lightweight, immutable data class that provides automatic implementations for methods such as: “toString”, “equals” and “hashCode”. Case classes are commonly used for modelling immutable data structures and work well with pattern matching and structural equality.

1. Why/when to use case class? Example

Case classes are used in Scala to model immutable data structures and entities. Case classes are commonly used for pattern matching, structural equality, and copying with modified field values.

Case classes in Scala have a variety of use cases:

- Data modelling and representing entities.

- Pattern matching and handling different cases.

- Managing immutable state and ensuring thread-safety.

- Message passing in actor-based systems.

- Creating value objects with equality and hashing semantics.

- Serialisation and deserialisation of objects.

- Defining structured API responses.

- Facilitating testing with immutability and automatic equality checks.

1. Difference between case class and normal class?

As aforementioned, case classes in Scala provide automatic generation of common methods like “toString”, “equals”, “hashCode” and “copy”. They also promote immutability by default, work seamlessly with pattern matching, and offer structural sharing with the “copy” method. On the other hand, normal classes require manual implementation of these methods and provide more flexibility in terms of mutability and inheritance.

1. Scala type hierarchy?

Scala's type hierarchy includes “Any” at the root, with two main branches: “AnyVal” for value types and “AnyRef” for reference types. All classes in Scala inherit from either “AnyVal” or “AnyRef”, making it a unified hierarchy. “AnyRef” is equivalent to “java.lang.Object” in Java and represents reference types, while “AnyVal” represents value types.

1. What are partially applied functions?

Partially applied functions are created by fixing some arguments of a function and creating a new function with the remaining arguments. It allows you to specialise functions based on a base function by providing only a subset of the required arguments. They offer flexibility and code reuse.

1. What is tail recursion?

Tail recursion is a recursive technique where the recursive call is the last operation performed in a function. It allows for optimised execution by eliminating the need for additional stack frames, preventing stack overflow errors. Tail recursion is achieved when the result of the recursive call is directly returned without any further processing.