**SCALA INTERVIEW QUESTION**

1. What is a trait?

In Scala, a trait is a reusable component that defines a set of methods and fields that can be mixed into classes. Traits are like interfaces in other programming languages but provide additional functionality beyond simple method signatures.

2. Difference between trait and sealed trait?

In Scala, both traits and sealed traits are constructs that define reusable components and can be mixed into classes. However, they have some key differences in terms of inheritance and extensibility. Here are the main differences between traits and sealed traits:

**Traits**:

Traits can be inherited by any class or trait, even from outside the defining package.

Traits can be mixed into multiple classes and traits using the extends or with keywords.

Traits can have abstract and concrete methods, fields, and other members.

Classes mixing in traits can override methods defined in the trait.

Traits do not restrict the inheritance or extension of other traits or classes.

**Sealed Traits:**

Sealed traits are a specific type of trait that restricts the inheritance to a defined set of classes and traits within the same file or compilation unit.

Sealed traits can only be inherited by classes and traits within the same file or compilation unit where the sealed trait is defined.

Sealed traits are typically used to define closed hierarchies or sum types, where all possible subtypes are known and limited.

All direct subtypes of a sealed trait must be defined in the same file or compilation unit.

Sealed traits enable exhaustive pattern matching, where the compiler can check if all possible cases are covered in a pattern match expression.

3. What is an abstract class?

In object-oriented programming, an abstract class is a class that cannot be instantiated on its own and is meant to serve as a blueprint or template for other classes. It is designed to be extended by subclasses that provide concrete implementations for its abstract methods and may also inherit and override its non-abstract methods.

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

Java interfaces and Scala traits are both constructs used in object-oriented programming to define contracts and provide a way for classes to inherit and implement common behavior. However, there are several key differences between Java interfaces and Scala traits:

**Multiple Inheritance:** In Java, a class can implement multiple interfaces, allowing for multiple inheritance of behavior. In Scala, a class can extend only one class but can mix in multiple traits using the with keyword. This allows for a form of multiple inheritance in Scala, where a class can inherit and combine the behavior of multiple traits.

**Default Implementations**: In Java 8 and later versions, interfaces can have default method implementations. This allows interfaces to provide a default implementation for methods, reducing the need for implementing classes to override all methods. Scala traits, on the other hand, can have concrete methods with full implementation, similar to regular class methods. Traits in Scala can provide default behavior without the need for separate default methods.

**Fields:** Java interfaces traditionally only define method signatures, not fields. However, starting from Java 8, interfaces can also have static and default fields. Scala traits, on the other hand, can define fields that can be accessed and modified by classes mixing in the

**Constructor:** Java interfaces cannot have constructors, as they do not define any implementation. Scala traits can have constructors, allowing for initialization logic and parameters.

**Superclass Inheritance:** In Java, interfaces cannot extend classes or provide implementation inheritance. In Scala, traits can extend classes, allowing for inheritance of both behavior and implementation from a superclass.

**Mixing In:** In Java, class inheritance and interface implementation are separate concepts. A class can extend only one superclass but can implement multiple interfaces. In Scala, traits can be mixed in using the with keyword, allowing a class to inherit from a superclass and mix in multiple traits simultaneously.

**Abstract Members:** Java interfaces can define abstract methods, which must be implemented by classes that implement the interface. Scala traits can define abstract methods as well as concrete methods with an implementation. Classes mixing in traits in Scala are required to implement abstract methods but can also use the concrete methods provided by the trait.

5. What is a singleton

A singleton is a design pattern that restricts the instantiation of a class to a single instance and provides a global point of access to that instance. It ensures that only one instance of the class is created and shared throughout the application.

6. What is a higher order function?

A higher-order function is a function that can accept other functions as arguments and/or return a function as its result. This means that functions can be treated as values and manipulated just like any other data type.

7. What is a closure

A closure is a combination of a function and the environment in which it was created. It allows a function to access variables from its outer or surrounding scope even after the outer function has finished executing. In other words, a closure "closes" over the variables it references, preserving their values and allowing the function to access them later.

8. What is a companion object?

In Scala, a companion object is an object that has the same name as a class and is defined in the same file. The companion object and the class are considered to be companions, and they have special access to each other's private members. The companion object and class can refer to each other without the need for explicit import statements.

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

**Nil**: Nil is an empty list of any type. It represents the end of a list or the empty list itself. It is commonly used as a shorthand for creating an empty List in Scala.

**Null:** Null is a trait that represents the null reference. It is a subtype of all reference types (classes, traits, and their instances). It is typically used when interoperating with Java code that uses null as a value.

**null**: null is a literal that represents the absence of an object reference. It can be assigned to variables of reference types but not to variables of value types. In Scala, it is recommended to avoid using null as much as possible and instead use Option types (such as Some and None) to represent absence.

**Nothing:** Nothing is a bottom type in Scala, which means it is a subtype of all other types. It is used to indicate that a method does not normally complete its execution (e.g., throws an exception) or as a return type for methods that never return a value.

**None**: None is an object of the Option type and represents the absence of a value or a failure condition. It is used to handle optional values and is often used as an alternative to null. It is the counterpart to Some, which represents a value.

**Unit:** Unit is a type that represents the absence of a meaningful value. It is similar to void in languages like Java. Functions that do not return a value typically have a return type of Unit.

10. What is pure function?

In functional programming, a pure function is a function that, given the same inputs, always produces the same output and has no side effects. A pure function is deterministic and does not modify any external state or rely on external mutable state. It operates solely based on its input parameters and returns a result without causing any observable changes in the system.

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

SBT (Scala Build Tool) is a popular build tool for Scala projects. It is used to compile, test, run, and package Scala applications and libraries. SBT uses a declarative build configuration written in Scala, allowing developers to define their project's structure, dependencies, and tasks.

12. What is currying?

Currying is a technique in functional programming that allows transforming a function with multiple arguments into a sequence of functions, each taking a single argument. It enables partial function application and provides a way to create new specialized functions by fixing some arguments of a function.

In currying, a function that takes multiple arguments is transformed into a chain of functions, each accepting one argument and returning another function until all arguments are consumed, and finally returning the result. This process is called currying, named after mathematician Haskell Curry.

13. Difference between currying and higher-order functions

Currying is a technique to transform a function with multiple arguments into a sequence of functions, allowing partial function application and composability. Higher-order functions, on the other hand, treat functions as values, enabling function abstraction, composition, and encapsulation of behavior. Currying is a specific transformation applied to functions, while higher-order functions are a general concept that applies to any function that accepts or returns functions.

14. Difference between var and val?

**var** stands for "variable," and variables declared with var are mutable, meaning their values can be changed after initialization.

**val** stands for "value," and variables declared with val are immutable, meaning their values cannot be changed once initialized.

15. What is case class?

In Scala, a case class is a special class primarily used for modeling immutable data. It provides a concise way to define classes that are primarily used to hold and transfer data, often representing entities or value objects in the application domain. Case classes come with several additional features and behaviors compared to regular classes, making them suitable for common data-centric tasks.

Key characteristics and features of case classes:

**Automatic Property Accessors**: Case classes automatically generate getter methods for all constructor parameters, allowing easy access to the data they hold.

**Immutable by Default:** Case classes are immutable by default, meaning their properties cannot be modified once created. This helps ensure data integrity and facilitates functional programming principles.

**Structural Equality:** Case classes automatically implement structural equality (value equality) based on their constructor parameters. This means that instances of the same case class are considered equal if their properties have the same values.

**Pattern Matching Support:** Case classes work well with pattern matching, a powerful feature in Scala. They automatically generate an extractor pattern that allows decomposition of case class instances into their constituent parts.

**toString, hashCode, and equals:** Case classes provide default implementations for toString, hashCode, and equals methods based on their constructor parameters. These methods are automatically generated and can be customized if needed.

**Copying Instances:** Case classes provide a built-in copy method, which allows creating new instances based on an existing instance with modified properties. This method is useful when you need to make a copy of a case class instance with specific changes.

**Companion Object**: Case classes automatically generate a companion object, which contains factory methods for creating instances of the case class without using the new keyword. The companion object also provides useful utilities like pattern matching helpers and extractor methods.

16. Why/when to use case class? Example

Case classes are best suited for modeling immutable data structures, facilitating pattern matching, ensuring value equality, and providing concise syntax for data-centric classes. They offer convenience, readability, immutability, and seamless integration with Scala's features and libraries.

Case classes provide a concise syntax for defining data-centric classes. They eliminate boilerplate code by automatically generating common methods such as getters, toString, hashCode, equals, and a companion object. This saves development time and improves code readability.

17. Difference between case class and normal class?

**Case classes** are specifically designed for modeling immutable data structures and are well-suited for scenarios where value equality, pattern matching, and concise syntax are desired.

**Normal classes** provide more flexibility and control, supporting inheritance, mutable state, and customization of behavior.

18. Scala type hierarchy?

**Any:** The root of the Scala type hierarchy. It is the supertype of all other types in Scala.

**AnyVal:** The root of the hierarchy for value types (primitive types) in Scala.

**Unit**: Represents the absence of a meaningful value. It has only one value, ().

**Boolean**: Represents boolean values true and false.

**Byte**: 8-bit signed integer.

**Short**: 16-bit signed integer.

**Int**: 32-bit signed integer.

**Long**: 64-bit signed integer.

**Float**: 32-bit floating-point number.

**Double**: 64-bit floating-point number.

**Char**: 16-bit Unicode character.

**AnyRef**: The root of the hierarchy for reference types (non-primitive types) in Scala.

**String**: Represents a sequence of characters.

Classes and traits defined in the Scala standard library and user-defined classes and traits are also part of the AnyRef hierarchy.

19. What are partially applied functions?

Partially applied functions are functions that have been applied to some, but not all, of their arguments. Instead of providing all the arguments at once, you can create a new function by fixing some arguments of an existing function, leaving the remaining arguments to be provided later. This technique allows for the creation of specialized functions based on a more general function.

20. What is tail recursion

Tail recursion is a technique to optimize recursive functions by eliminating unnecessary stack frames and preventing stack overflow errors. In a tail-recursive function, the recursive call is the last operation performed in each recursive branch, allowing the compiler to optimize it into a loop-like structure.