**Scala Interview Questions**

1.What is a trait?

A trait in Scala is a reusable code component that can contain both abstract and concrete methods. It's similar to an interface in Java but can also provide method implementations. Traits are used for code reuse and composition in Scala.  
  
2. What's the difference between a trait and a sealed trait?

A trait is a reusable code component in Scala, while a sealed trait is a trait with the "sealed" keyword. A sealed trait restricts inheritance to classes and objects defined in the same file, typically used in algebraic data types. This ensures that all subtypes are known and exhaustive for pattern matching.  
  
3. What is an abstract class?

An abstract class in Scala is a class that cannot be instantiated on its own and is meant to be subclassed. It can contain both abstract (unimplemented) and concrete (implemented) methods. Abstract classes provide a common structure and interface for related classes in an inheritance hierarchy.  
  
4. What is the difference between a Java interface and a Scala trait?

In **Java**, an interface is a contract that defines a set of abstract methods that implementing classes must provide. Interfaces do not allow concrete method implementations.

In contrast, a **Scala** trait is similar to an interface but can contain both abstract and concrete methods. Traits offer more flexibility, as they support code reuse through method implementation and can be mixed into classes.  
  
5. What is a singleton?

A singleton, in the context of design patterns, is a pattern that ensures a class has only one instance and provides a global point of access to that instance. In Scala, you can create singletons using the "object" keyword, which defines a single, lazily initialized instance. This object can contain methods and fields, and it's commonly used to implement utility classes and manage shared resources.  
  
6. 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 a result. In other words, it treats functions as first-class citizens. Higher-order functions are often used in functional programming to facilitate code abstraction, composition, and reusability.  
  
7. What is a closure?

 A closure is a function bundled together with the lexical environment in which it was declared. This means the function "closes over" its surrounding variables, allowing it to access and manipulate those variables even after the outer function has finished executing. Closures are commonly used in functional programming and are crucial for maintaining state in functional contexts.  
8. What is a companion object?

In Scala, a companion object is an object with the same name as a class. It's used to group related methods and fields that are not specific to instances of the class but pertain to the class itself. Companion objects serve as a replacement for static methods and fields in Java and provide a way to define class-level behaviour and shared state for Scala classes. They often work together with class instances and are a powerful feature of Scala's object-oriented model.  
  
9. Nil vs Null vs null vs Nothing vs None vs Unit   
Certainly, let's simplify it:

1. Nil:

- Represents an empty list.

- Example: `val myList = Nil`

2. Nul:

- A type that denotes the absence of a reference value.

3. null

- A reference to nothing or no value (similar to `null` in Java).

- Example: `val myVar = null`

4. Nothing

- A type that indicates a value will never be returned (often because an exception will be thrown).

5. None

- Represents the absence of a value in a context.

- Example: `val myOption: Option[String] = None`

6.Unit

- A type that signifies no meaningful value is returned (like `void` in Java).

- Example: `def myFunction(): Unit = println("Hello")`

In summary:

- `Nil`, `null`, and `None` are used to represent the absence or emptiness of a value.

- `Null` and `Nothing` are types used in type hierarchies to denote nullability and non-returning methods, respectively.

- `Unit` is used to show that a method doesn’t return a meaningful value.

10. What is a pure function?

A pure function is a function that, given the same input, always produces the same output and has no side effects. It doesn't modify any external state or variables and doesn't rely on external state. Pure functions are a fundamental concept in functional programming, and they are predictable, testable, and easy to reason about.

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

SBT (Simple Build Tool) is a popular build tool for Scala and Java projects. It's used for building, compiling, testing, and packaging projects. SBT uses a declarative build configuration defined in Scala and is known for its ability to manage project dependencies, incremental compilation, and efficient build processes. I have experience using SBT to build and manage Scala projects, including setting up build configurations, managing dependencies, and automating project tasks.  
  
12. What is currying?

Currying is a technique in functional programming where a function that takes multiple arguments is transformed into a series of functions, each taking a single argument. Currying allows you to partially apply functions, creating new functions with reduced arity. It's a way to achieve greater flexibility and composability in functional code.  
  
13. Difference between currying and higher-order functions

* Currying is a technique for transforming a multi-argument function into a series of single-argument functions. It's primarily about how a function is structured.
* Higher-order functions are functions that take other functions as arguments or return functions as results. It's about how functions are used in a program, emphasizing the passing of functions as values.

14. Difference between var and val:

* val is used to declare an immutable variable. Once a value is assigned to it, it cannot be changed. It's similar to a constant.
* var is used to declare a mutable variable. You can change the value assigned to a `var` after it's initially set.

15. What is a case class?

In Scala, a case class is a special class primarily used for modeling immutable data. Case classes come with built-in features like automatic getter methods, equality checks, pattern matching support, and a `copy` method for creating modified copies of instances. They are commonly used to represent structured data, such as records or data transfer objects. Case classes are often preferred for their concise and convenient syntax for working with data.  
  
16. Why/when to use a case class? Example

Case classes are typically used for representing structured, immutable data. You should consider using case classes when you need:

* **Immutability**: Case class instances cannot be modified, ensuring data integrity.
* **Pattern** **matching**: They are well-suited for pattern matching, making code more concise.
* **Automatic** **methods**: Case classes automatically generate methods for equality, hashing, and copying.

   Example:  
  
   case class Point(x: Int, y: Int)  
   val p1 = Point(1, 2)  
   val p2 = Point(1, 2)  
   println(p1 == p2) // true (equality is automatically provided)

17. Difference between case class and normal class:

* Case classes are designed for immutable data and come with several built-in methods (like `equals`, `hashCode`, `toString`, and `copy`) and support for pattern matching. Normal classes do not provide these features by default
* Case class constructor parameters are public by default, whereas in normal classes, they can be made private.
* Case classes are typically used for simple, data-centric structures, while normal classes are more versatile and can represent various object-oriented constructs.

18. Scala type hierarchy:

The Scala type hierarchy is rooted in the `Any` class, which has two direct subclasses: `AnyVal` (for value types like integers and booleans) and `AnyRef` (for reference types, similar to Java objects). All other classes in Scala, including user-defined classes, extend either `AnyVal` or `AnyRef`

19. What are partially applied functions?

Partially applied functions are a feature in Scala where you provide some, but not all, of the arguments to a function, creating a new function with fewer parameters. This can be useful for creating specialized functions from more general ones or for delaying the application of certain arguments. It allows you to create functions with specific configurations or defaults.  
  
20. What is tail recursion?

Tail recursion is a specific form of recursion where the recursive function calls itself as its last action, with the result of that call immediately returned. This optimization allows the compiler to optimize the recursion, avoiding a buildup of stack frames and preventing stack overflow errors. Tail recursion is often used in functional programming for solving problems that can be naturally expressed through recursion, such as factorial calculations and list processing algorithms.  
  
21.     Null, null, Nil, Nothing, None, and Unit