Scala

1. What is a trait ?

* Traits are a fundamental concept of object-oriented programming that provides a mechanism to reuse code
* Traits can contain both abstract and concrete methods, which allows for more flexible and modular code design.
* Multiple traits to be mixed in with a single class, which allows for more complex and powerful class hierarchies.

trait Print {  
 def print(): Unit = {  
 *println*("Printing...")  
 }  
}  
  
class MyClass extends Print {  
 override def print(): Unit = {  
 *println*("MyClass printing...")  
 }  
}  
object practice1 {  
 def main(args: Array[String]): Unit = {  
 val myClass = new MyClass()  
 myClass.print()   
 }  
}

1. Difference between trait and sealed trait?

Traits :

* Normal traits are open. Any number of classes can inherit from the trait.
* A normal traits hierarchy makes it easy to add additional sub-class

Sealed Traits:

* Sealed traits are closed. They only allow a fixed set of classes to inherit from them, and all inheriting classes must be defined together with the trait itself in the same file or REPL command.
* It is easy to add new methods, since a new method can simply pattern match on each sub-class and decide what it wants to do for each

1. What is an abstract class?

* Abstract class is constructed using the abstract keyword.
* It contains both abstract and non-abstract methods and cannot support multiple inheritances.
* A class can extend only one abstract class.

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

Java interface:

* limited form of multiple inheritance
* It can define abstract methods, which must be implemented by classes that implement the interface
* Leads diamond problem
* Java interfaces cannot have constructors

Scala trait:

* They are multiple inheritance friendly
* You can have abstract as well as non abstract methods
* It don't have diamond problem by having strict rules on evaluation order of mix-ins due to linearization.
* Scala traits can have constructors, allowing for initialization logic and parameters.

1. What is a singleton

* The singleton design pattern restricts the instantiation of a class to a single instance.
* The singleton class ensures that it’s only instantiated once, and can provide easy access to the single instance
* A singleton object provides an entry point to your program execution.
* Singletons are sometimes considered to be an alternative to global variables or static classes.

1. What is a higher order function?

* Higher order functions take other functions as parameters or return a function as a result.
* The functions that operate with another functions are known as Higher order Functions
* It is even utilized in minimizing redundant lines of code from a program.

1. What is a closure

* A closure is a function, whose return value depends on the value of one or more variables declared outside this function.
* The free variables are defined outside of the Closure Function and is not included as a parameter of this function.

1. What is a companion object?

* **Companion object** is known as an object whose name is same as the name of the class.
* A companion object and its class can access each other’s private members
* A companion object’s apply method lets you create new instances of a class without using the new keyword

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

**Nil** - Used for representing empty lists, or collections of zero length. For sets you can use Set.empty

**Null**– Its a trait that represents the null reference.

**null– It is an instance of Null.** It can be used as a replacement for all reference types

**Nothing -**is a Trait. Its a subtype of everything. But not superclass of anything. There are no instances of Nothing. It is like the leaf of a tree.

**None**– Used to represent a sensible return value. Just to avoid null pointer exception. Option has exactly 2 subclasses- Some and None. None signifies no result from the method.

**Unit**– Type of method that doesn’t return a value of anys sort.

1. 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

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

* SBT(Scala Build Tool) is a build tool that helps one manage their Scala project, which includes building, compiling, testing, as well as managing libraries and dependencies. It also is equipped with its own plugins which allow for integration of other features.
* SBT uses a declarative build configuration written in Scala, allowing developers to define their project's structure, dependencies, and tasks.

1. What is currying?

* **Currying**in Scala is a technique or a process of transforming a function. This function takes multiple arguments into a function that takes single argument.
* It is applied widely in multiple functional languages.
* It enables partial function application and provides a way to create new specialized functions by fixing some arguments of a function.

Eg: Def add(x:Int)=(y:Int)=>x+y;

Println(add(10)(20));

Val sum = add(40);

Println(sum(100));

1. Difference between currying and higher-order functions

A **higher-order function** is simply a function that takes a function as an argument or returns a function as a result, and it may or may not be curried. In general usage, someone referring to a higher-order function is usually talking about a function that takes another function as an argument.

A **curried function** returns a function as its result. A fully curried function is a one-argument function that either returns an ordinary result or returns a fully curried function. A curried function is necessarily a higher-order function, since it returns a function as its result.

1. Difference between var and val?

val: makes a variable immutable, Values can’t be changed once initialized

var: makes a variable mutable, Values can be changed after initialization

1. What is case class?

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching. It uses equal method to compare instance structurally. It does not use new keyword to instantiate object. All the parameters listed in the case class are public and immutable by default. Case classes are datacentric.

1. Why/when to use case class? Example

When you want structural equality. Use a case class, as it gives you proper hashCode and equals. For example, you want to be able to use them as keys into a Set or Map.

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

1. Difference between case class and normal class?

Case classes are specifically designed for modelling 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 behaviour.

1. Scala type hierarchy?

**Any**:  It is the root class in the type system. It is super class of all classes

**AnyVal**: It represents value classes. All value classes are predefined; they correspond to the primitive types. There are nine value classes in Scala : **Byte*,*Short, Char, Int, Long, Float, Double,Boolean**, and **Unit**.

**AnyRef**: It represents reference classes. All non-value types are defined as reference types. **String**

**Nothing and Null:**  
**Nothing** is a subclassify of all value types, it is also called the bottom type. Type nothing that has no value. We can use Nothing to signal non-termination such as a thrown exception, program exit, or an infinite loop.

**Null** is a subclassify of all reference types. The keyword literal null can identify a single value.

1. What are partially applied functions?

Partially applied functions are an expression in which we do not supply all the arguments needed by the function. Instead, we supply some of the needed arguments only. 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

1. What is tail recursion

There is no other operation to perform after executing the recursive function itself, the function can directly return the result of the recursive call. The compiler can easily optimize the tail-recursive function, as there isn’t any instruction left to be executed because the recursive call is the last statement. Hence, there is no need to save the stack frame of the function.