Scala

Introduction

- Scala is a hybrid object-functional language: it supports both object-oriented and functional programming paradigms.
- Haskell and ML have heavily influenced the Scala design.
- Statically typed
- Scala compiles the code to java bytecode. Hence, works with any standard JVM

Datatypes

- There are no primitive datatypes in Scala
- All values are objects (i.e. All values are instance of a class)
- Superclass of all classes is: scala.Any
- AnyRef: Supertype of any reference type.
- Byte, Short, Int, Long, Char, String, Float, Double, Boolean.
- Unit: corresponds to no value

Classes

- Scala has a concept of Class arguments.
- Objects are instantiated using new keyword
- They can be parametrized with values and types.
- Access levels in Scala:
  - Members of classes or objects can be labeled with the access modifiers to affect the visibility of members.
    - Public is default access level (you can make members public in Scala by not explicitly specifying any access modifier)
    - Private
    - Protected

Singleton Objects

- Scala does not have the concept of static members.
- Instead, Scala has Singleton objects
- A singleton object definition looks like a class except for the class keyword, object keyword is used.
- You cannot instantiate a singleton object with the new keyword.
- The main method is put inside a Singleton Class.

Run Scala program

- To run a Scala program, you must supply the name of a standalone singleton object with a main method that takes one parameter, an Array[String], and has a result type of Unit.
- Any standalone object with a main method of the proper signature can be used as the entry point into an application

Values and Variable Declaration

- Syntax to initialize a variable in Scala is

```
val name:type = initialization
```

- Colon and type are optional (Because of type inference)
- The identifier declared with val cannot change value i.e. you cannot changed the binding to a val
- Variable on the other hand can change. Syntax:

```
var name:type = initialization
```

Functions in Scala

- Syntax:

```
      def functionName ([list of parameters]) : [return type] = {
              function body
              return [expr]
      }
```

- Anonymous functions syntax:

```
      ([list of parameters]) => {          function body}
```

Higher order functions

- Syntax:

```
    def functionName ([list of parameters]) : [return type] = {
            function body
            return [expr]
    }
```

- In the list of parameters, function as a parameter specifies its formal parameters and return type. If there are more than 1 parameter then enclose the parameters in a braces and separated by comma
- Example:

```
    def apply(f: Int => String, v: Int) = f(v)
```

Here f is a formal parameter that takes an Int argument and returns a string.

Traits

- Similar to interfaces in Java
- Traits can inherit from abstract or concrete classes or other traits.
- Traits are used to define object types by specifying the signature of the supported methods.
- A trait definition looks like a class, but uses the trait keyword instead of class.
- A class can only inherit from a single concrete or abstract base class, but it can combine as many traits as you want
- If there is no concrete or abstract base class, you still start with the extends keyword for the first trait, followed by the with keyword for the remaining traits.

Case classes

- Case classes are regular classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching.
- Similar to datatypes in ML.
- Case classes are Scala's way to allow pattern matching on objects without requiring a large amount of work.
- All you need to do is add a single case keyword to each class that you want to be pattern matchable.
- In declaration, each subclass has a case modifier.
- Classes with such a modifier are called *case classes*. Using the modifier makes the Scala compiler add some syntactic conveniences to your class.

Case classes

- For creating new case class instances, it is not required that the new keyword is used.
- All arguments in the parameter list of a case class implicitly get a val prefix, so they are maintained as fields.
- To match an expression or in other words to do pattern matching in Scala, “match” keyword is used.
- Intuitively, match corresponds to switch in Java, but is written after the selector expression:
  - selector match { alternatives }

Pattern matching

- Similar to ML, Scala has a built in pattern matching mechanism.
- Matches on any data are done with first-match policy.
- Syntax:
  - expr match {

```
    case expr1 => body
    case expr2 => body
}
```

Generic Classes

- Syntax

```
class className (list of class parameters) [Types]{
        class body
}
```

- Please note that the list of class parameters are optional
- Example:

```
class Queue [T] { }
```

Creating an instance of Queue:

```
val x=new Queue[Int]
```

Generic Classes (Variances)

- Just like in Java, default subtyping of generic types is invariant.
- Hence, Queue[S] is a subtype of Queue[T] only if S=T.
- As this is quite restrictive, Scala offers a mechanism (type parameter annotation) to control the subtyping behavior of generic types.
- There are 2 annotations that you can use:
  - +T
  - -T
- The annotation +T declares type T to be used in covariant positions.
- The annotation –T declares type T to be used in contravariant postions.

Covariant & Contravariant types

- For covariant type parameters we get a covariant subtype relationship regarding this type parameter.
  - Example:
    - Let Queue be defined as Queue[+S]
    - Queue[S] is a subtype of Queue[T], if S is a subtype of T.
- For contravariant type parameters we get a contravariant subtype relationship regarding this type parameter.
  - Example:
    - Let Queue be defined as Queue[-T]
    - Queue[S] is a subtype of Queue[T], if S is a supertype of T.

Upper Type Bounds

- Type parameters may be constrained by a type bound
- An upper type bound T <: A declares that type variable T refers to a subtype of type A.

Lower Type Bounds

- Lower type bounds declare a type to be a supertype of another type
- Term used to express lower type bounds is T >: A, expresses that the type parameter T or the abstract type T refer to a supertype of type A.