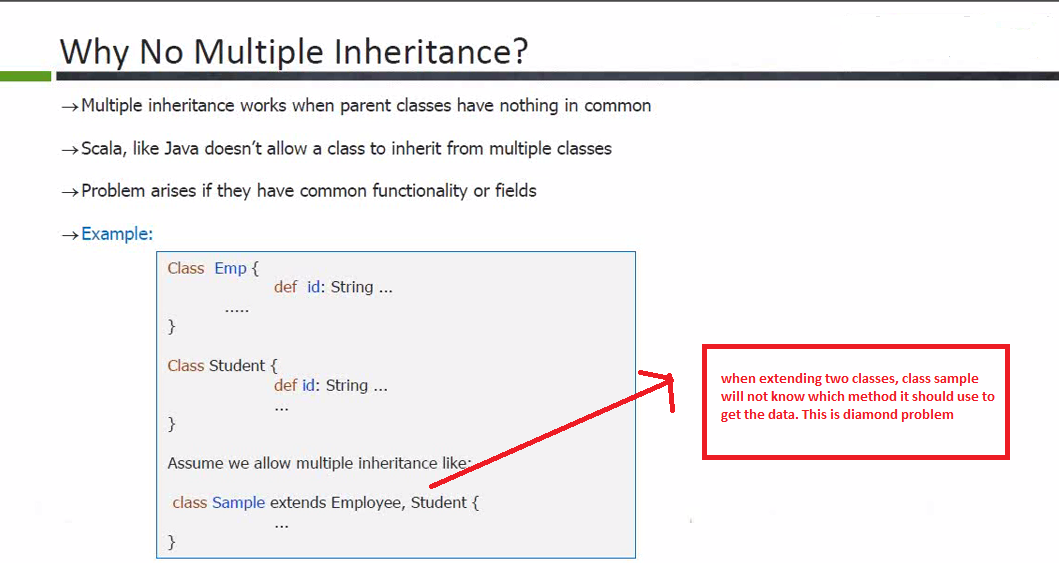
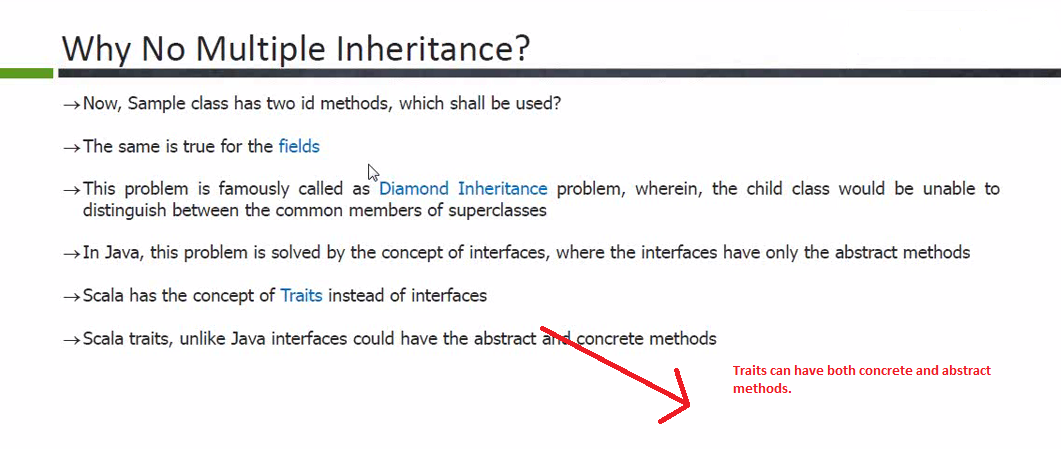
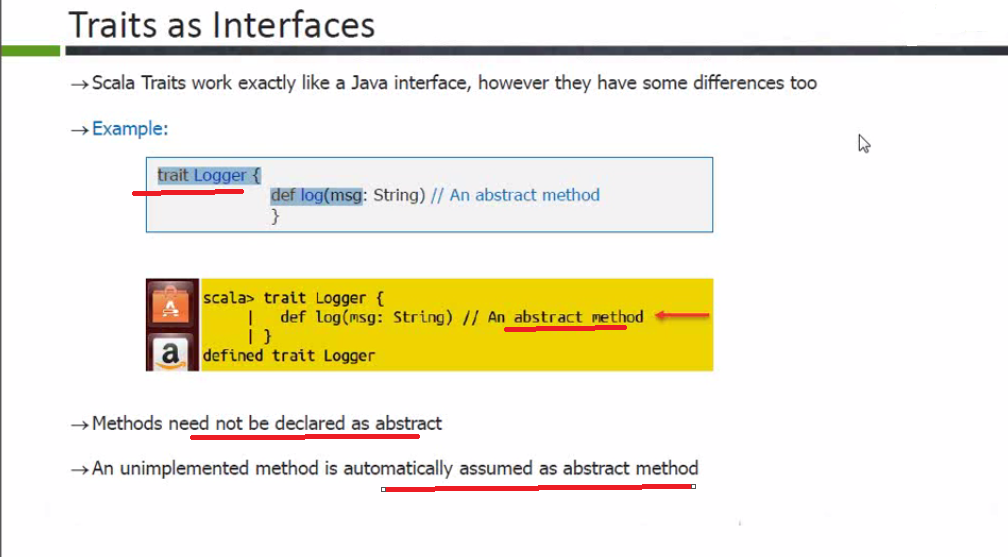
**No multiple Inheritance in Scala similar to java-------🡪**



**Traits can have both concrete and abstract methods.**





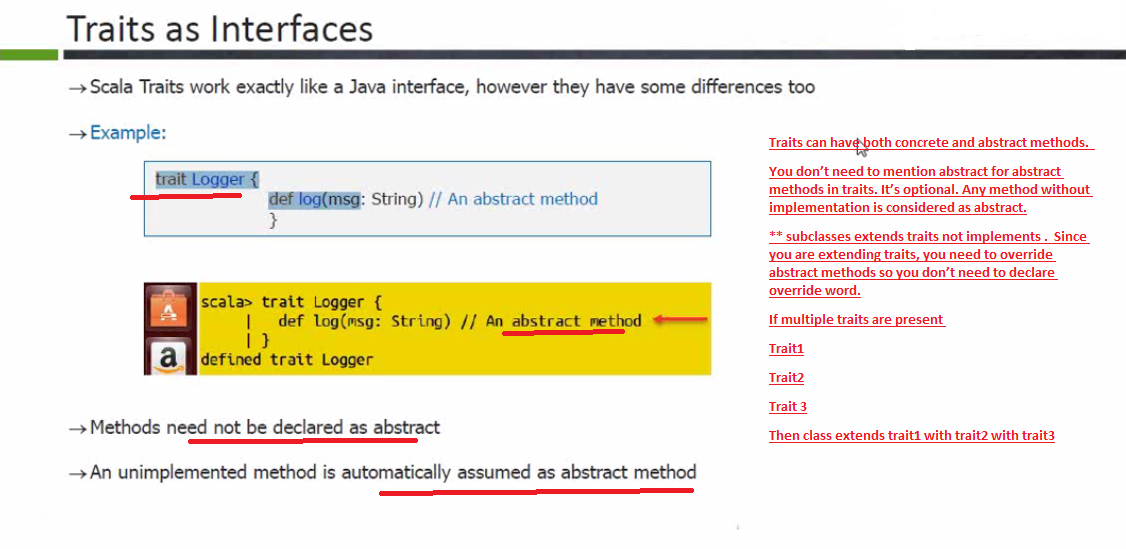
Traits can have both concrete and abstract methods.

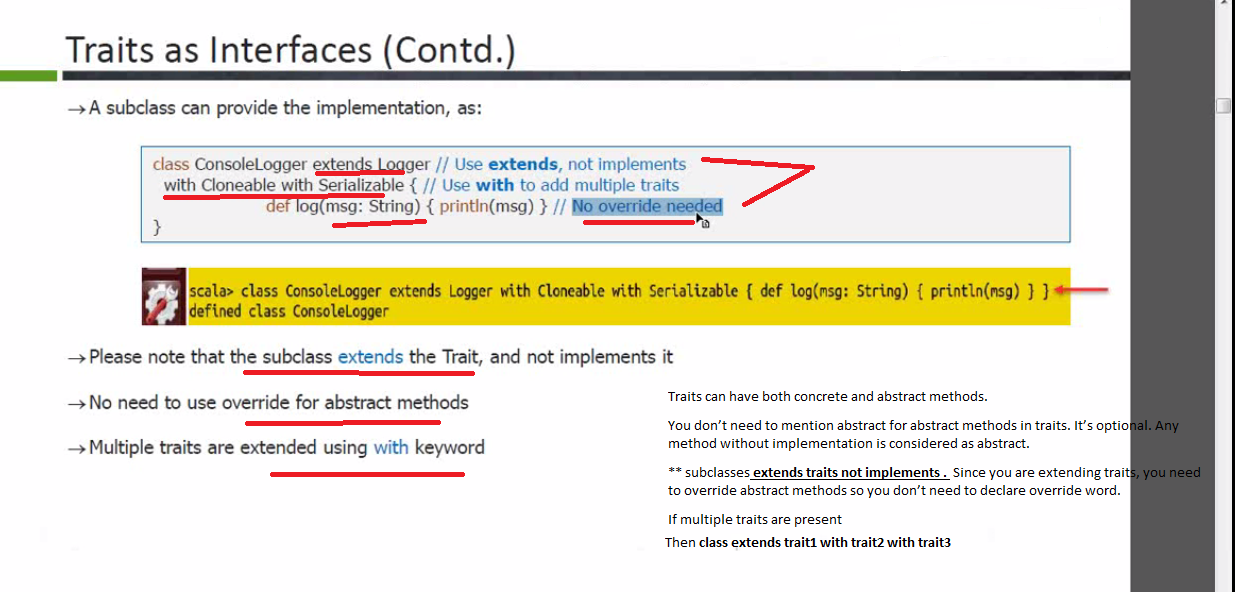
You don’t need to mention abstract for abstract methods in traits. It’s optional. Any method without implementation is considered as abstract.

\*\* subclasses **extends traits not implements .**  Since you are extending traits, you need to override abstract methods so you don’t need to declare override word.

If multiple traits are present

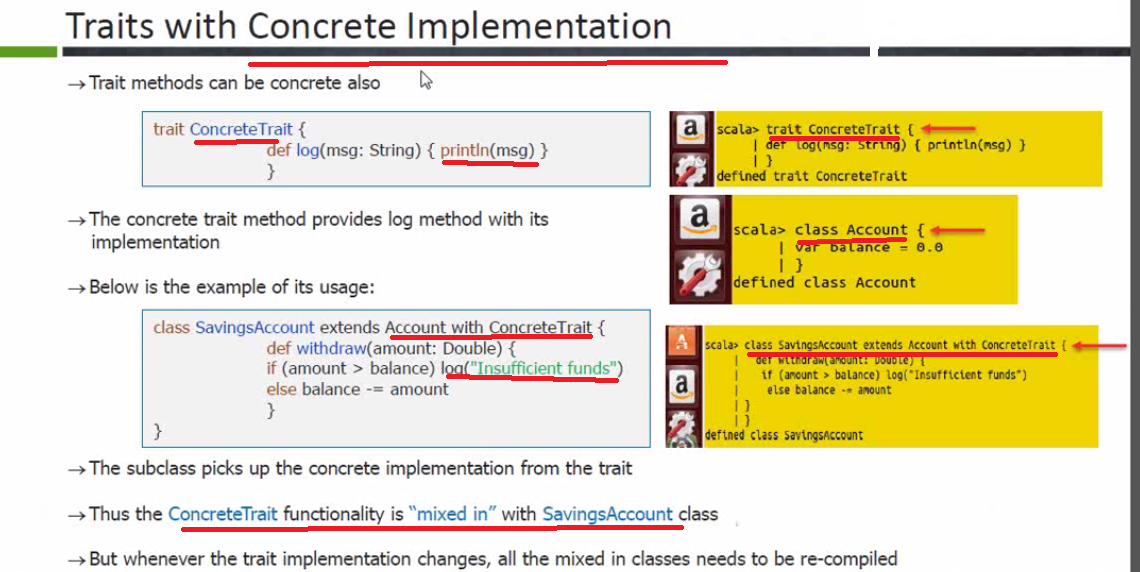
Trait1,Trait2,Trait 3 Then **class extends trait1 with trait2 with trait3. You can also import java packages**





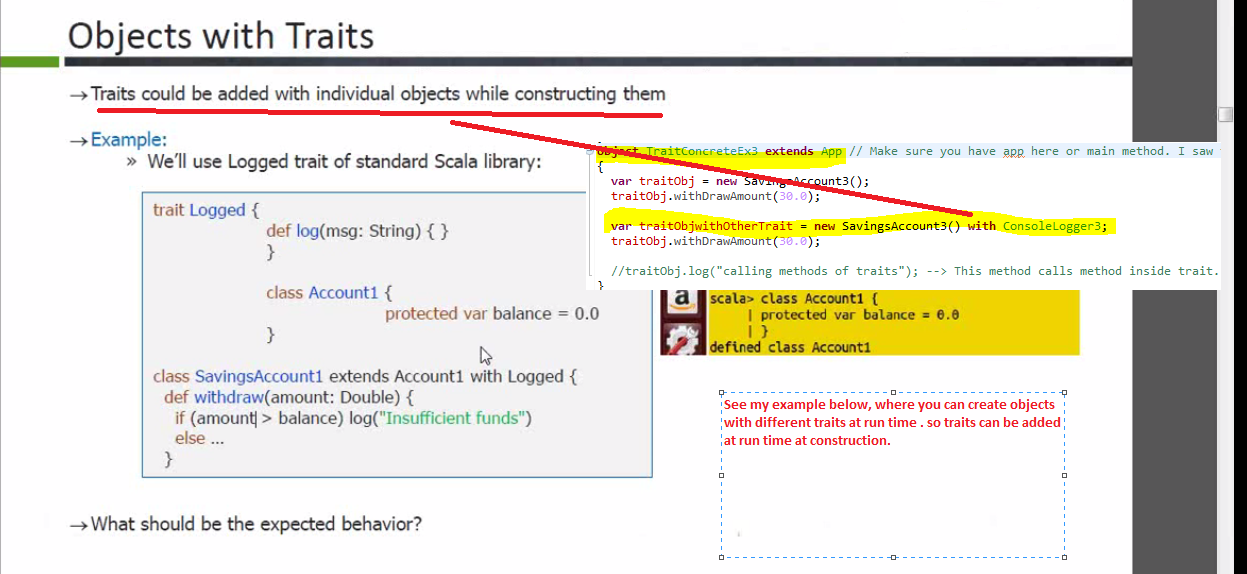


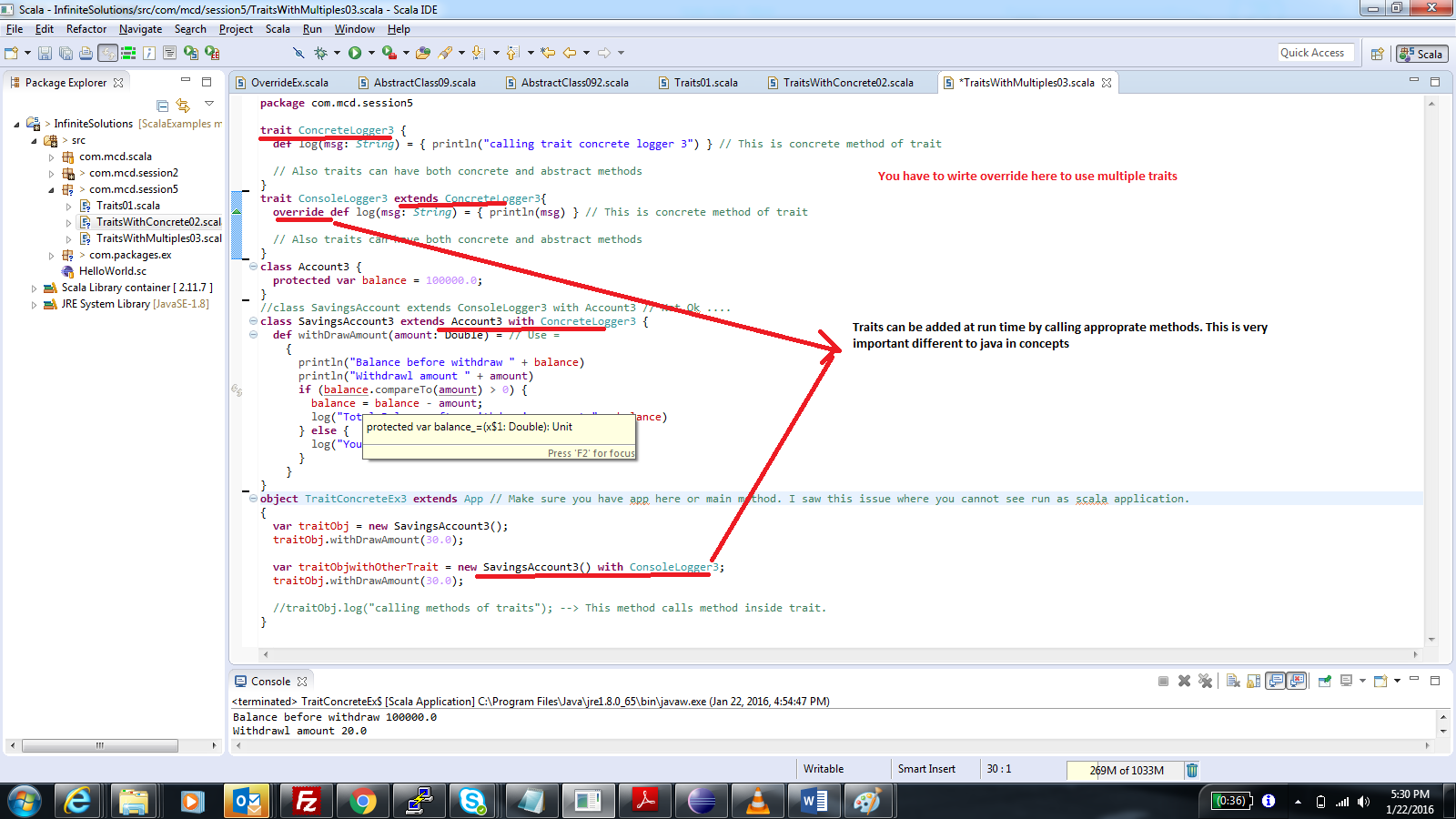
**Traits with Concrete methods.**





**Classes with Multiple Traits and how to invoke at object level.**





**-- Multiple Traits in Scala at object Creation ----🡪 ------🡪 ---------🡪**

**You can call multiple traits at object creation.**

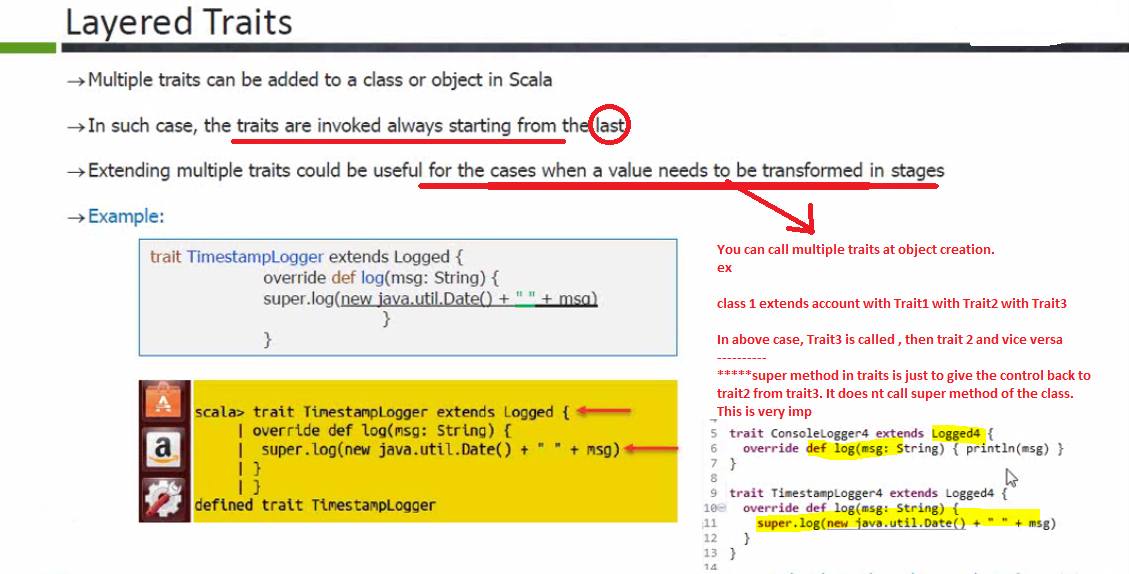
**ex**

**class 1 extends account with Trait1 with Trait2 with Trait3**

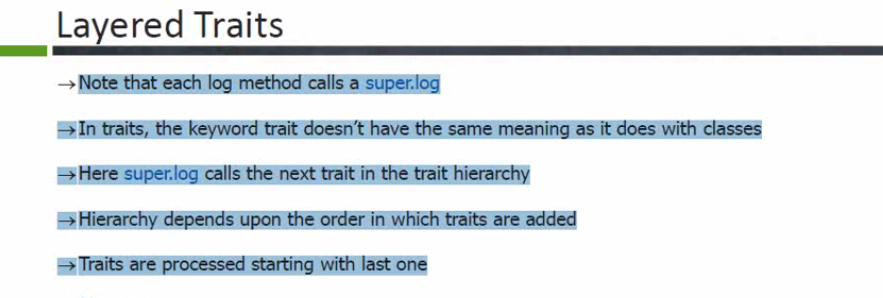
**In above case, Trait3 is called , then trait 2 and vice versa**

**----------**

**\*\*\*\*\*super method in traits is just to give the control back to trait2 from trait3. It does nt call super method of the class. This is very imp**



**\*\*\*\*\*\*In above case, it is not calling super constructor, it is just giving back control to the trait which is next from left.**



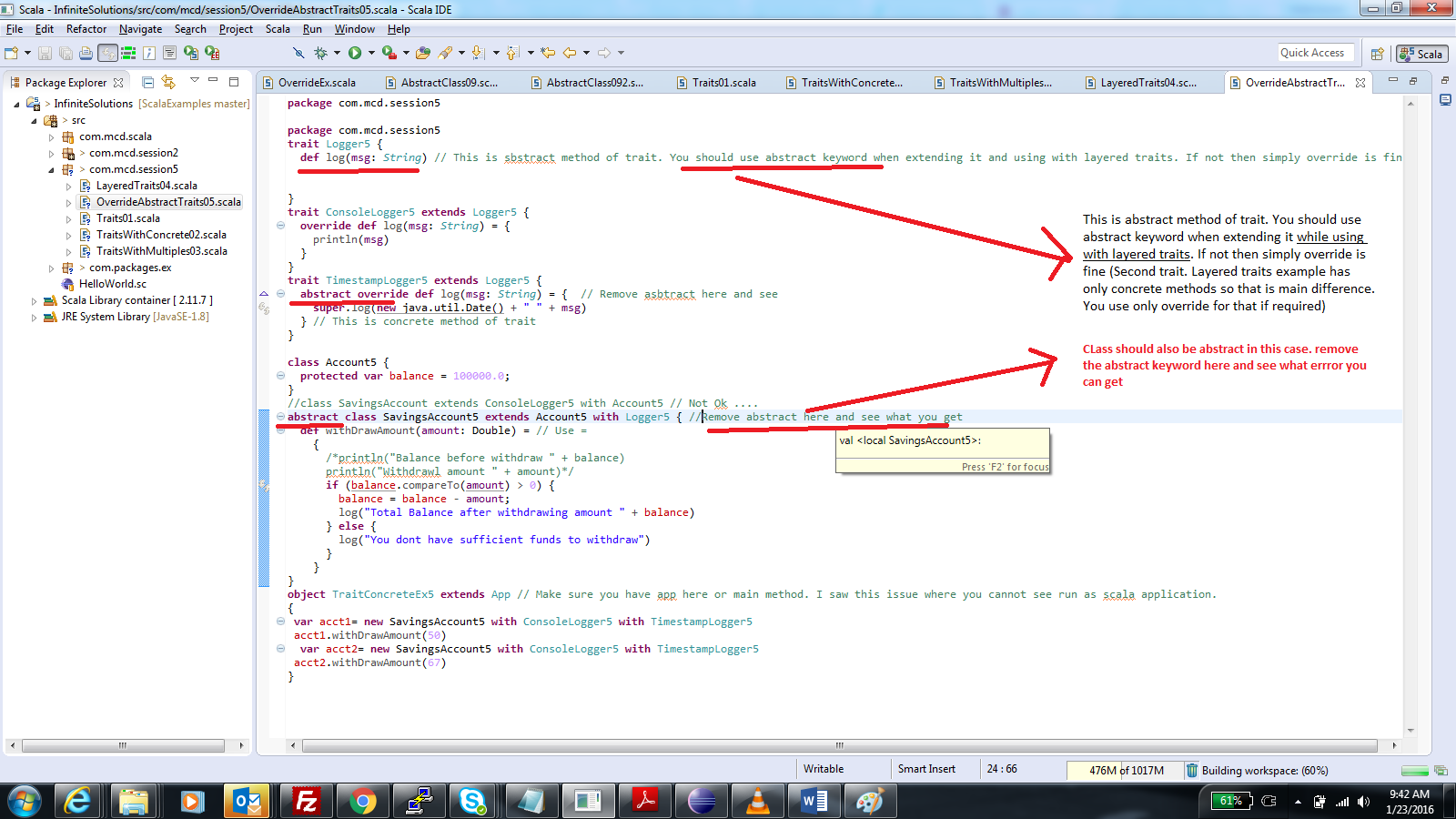
****

**In above case, first savingsaccount is called. While executing it sees log method, then it goes to short logger trait which is first on left. Takes the message from savings and then truncates it. After performing, it gives back control to next trait by super key word. This goes on until all traits are done.**

**Similar it is case for second one. First timestamplogger is called in this case.**

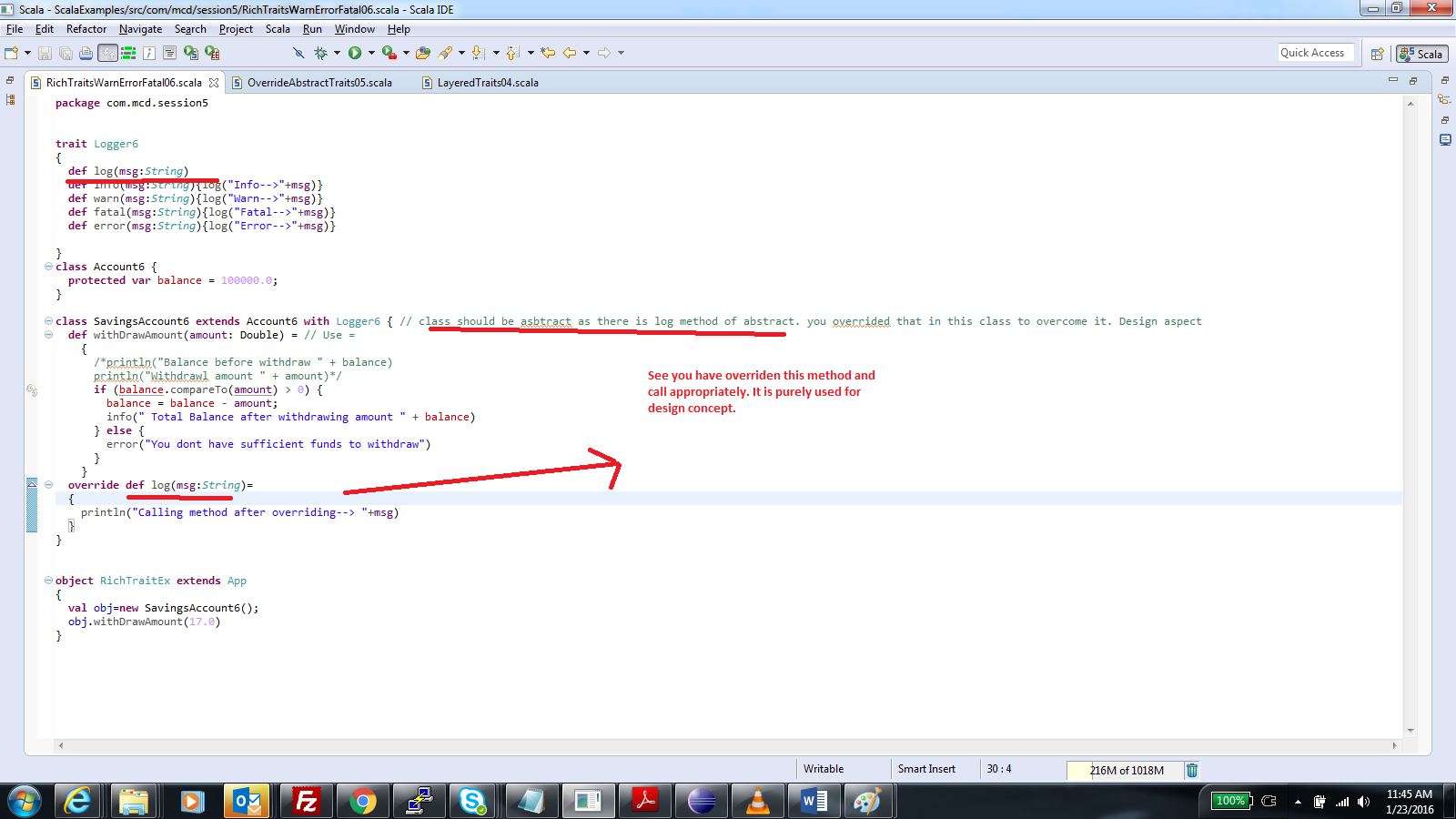
**----------------------🡪Override abstract methods in Traits --🡪**

This is abstract method of trait. You should use abstract keyword when extending it while using with layered traits. If not then simply override is fine (Second trait. Layered traits example has only concrete methods so that is main difference. You use only override for that if required)



**----🡪 Rich Traits in Scala ------🡪**

* No explanation needed. Better not explain as I am still working on this.
* In this example, see you have implemented log method inside the class.
* Class should be abstract as there is log method of abstract. you overridden that in this class to overcome it. Design aspect.



**-------------------🡪Abstract and concrete Fields in Traits-------------------------------🡪**

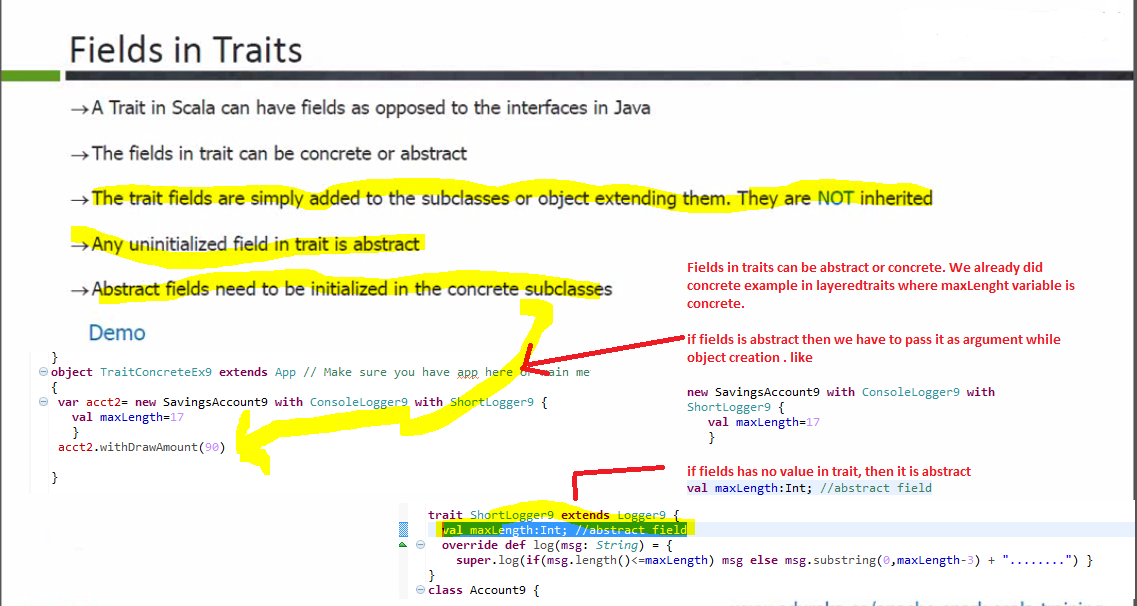
**Fields in traits can be abstract or concrete. We already did concrete example in layeredtraits where maxLenght variable is concrete.**

**if fields is abstract then we have to pass it as argument while object creation . like**

**new** SavingsAccount9 **with** ConsoleLogger9 **with** ShortLogger9 {

**val** maxLength=17

}

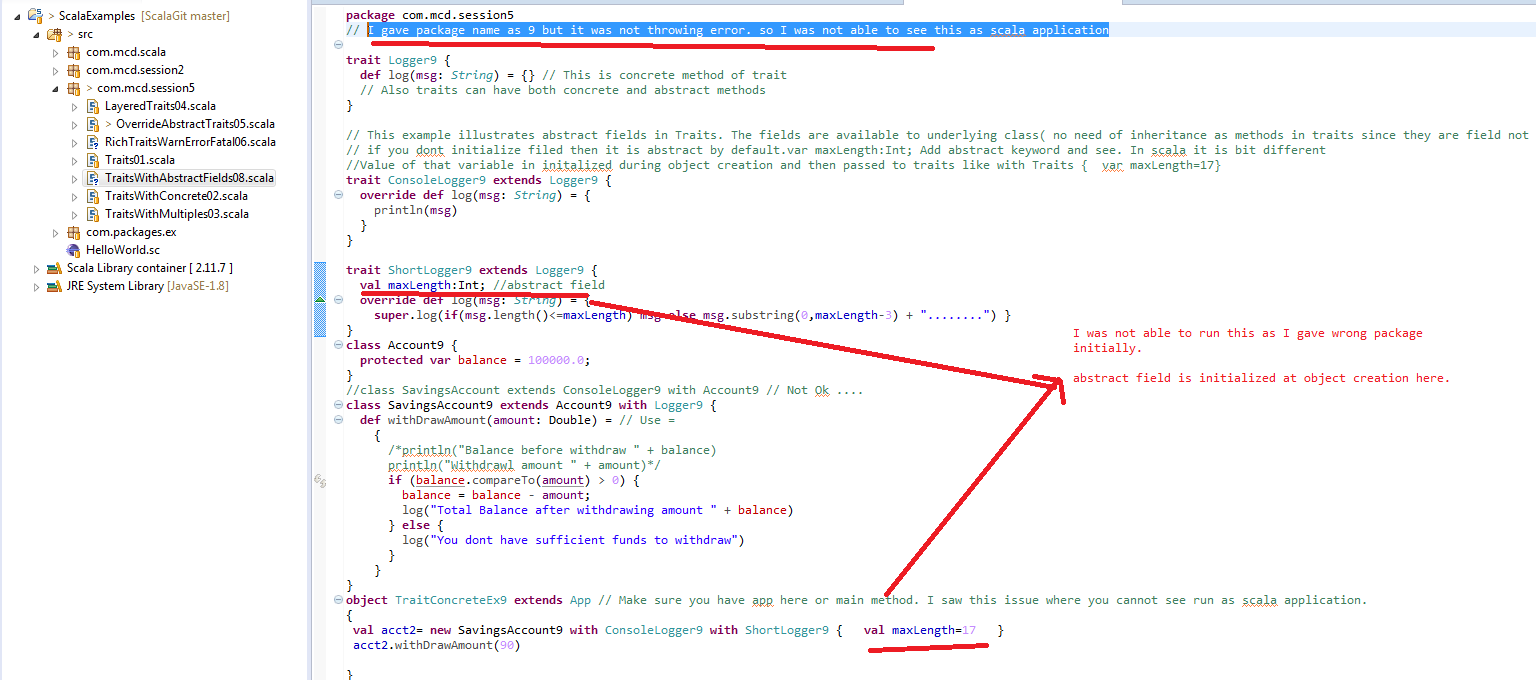


// This example illustrates abstract fields in Traits. The fields are available to underlying class( no need of inheritance as methods in traits since they are field not methods).

// if you dont initialize filed then it is abstract by default.var maxLength:Int; Add abstract keyword and see. In scala it is bit different

//Value of that variable in initalized during object creation and then passed to traits like with Traits { var maxLength=17}

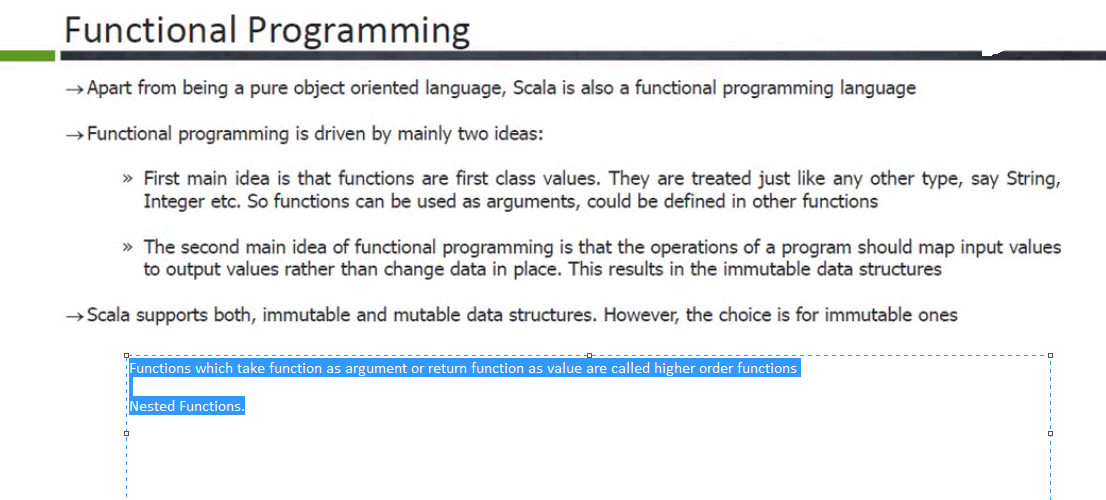
I gave package name as 9 but it was not throwing error. so I was not able to see this as scala application



**------------------------🡪Scala if Functional Language:-----------------🡪**

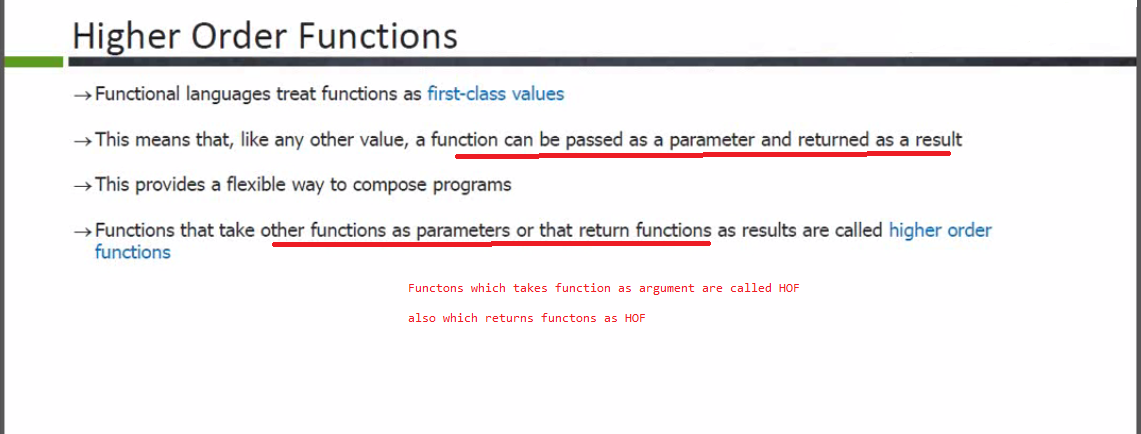
* Functions which take function as argument
* return function as value
* Nested Functions.

All these are features of functional programming which are available in scala that is reason it is **also called** as FP language (also OO programming)

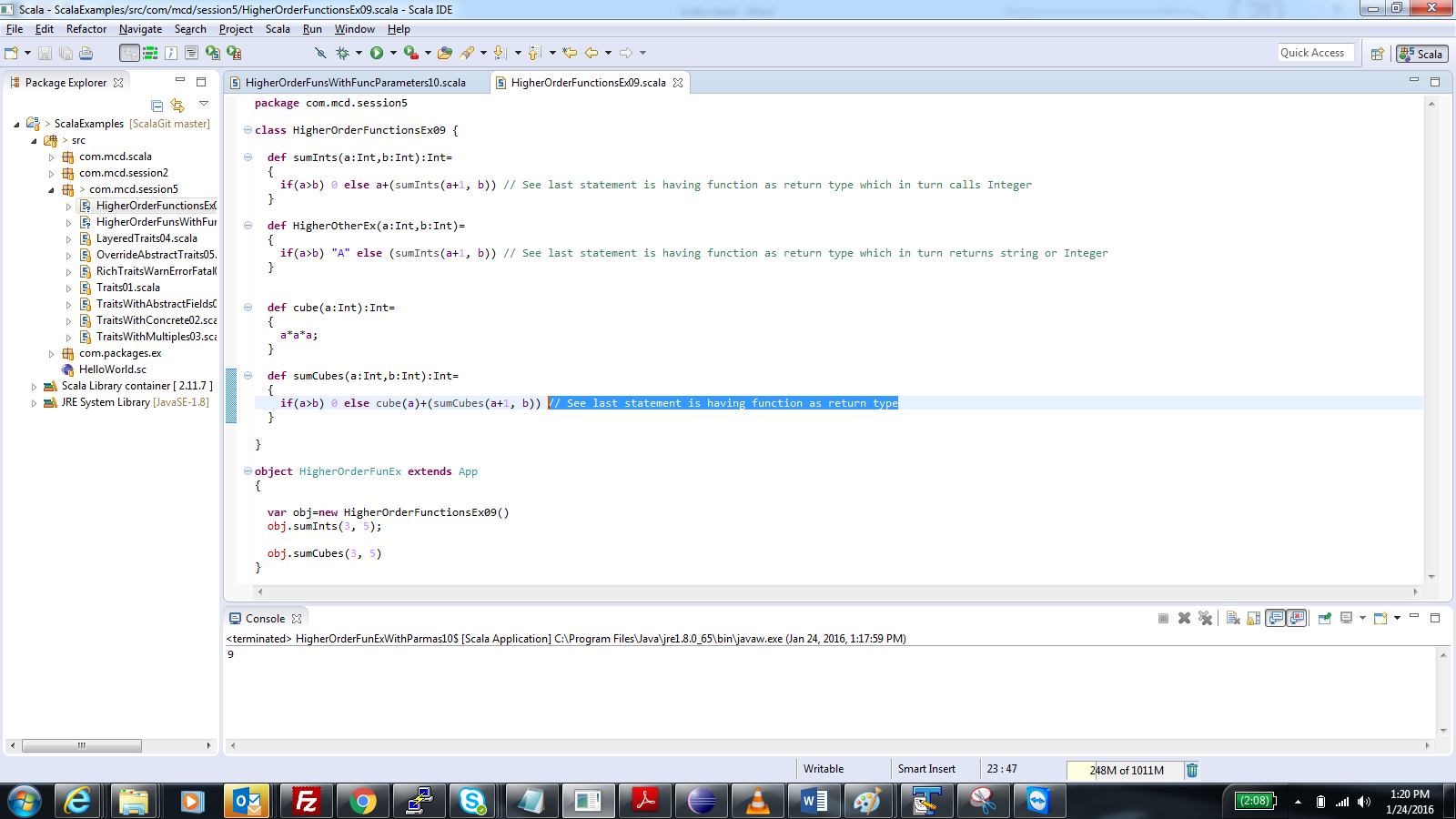


**-------------------------------🡪Higher Order Functions in Scala-----------------🡪**

Functions which take function as argument or return function as value are called higher order functions. Examples flatToMap

****

* // See last statement is having function as return type which in turn calls Integer
* // See last statement is having function as return type which in turn returns string or Integer
* // See last statement is having function as return type



See this function taking as other fnc as parameter. Return type mandatory because facorial is recursive so return type mandatory for recursive fns

here f(a) is function of scala which in turns calls app function

