Apache Spark and Scala

Module 3: Introducing Traits and OOPS in Scala

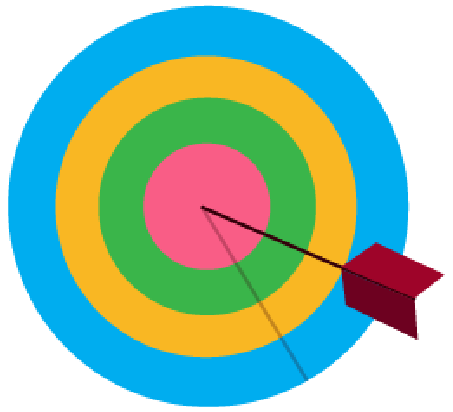
Course Topics

Module 1 Getting Started / Introduction to Scala	**Module 2** Scala – Essentials and Deep Dive	**Module 3** Introducing Traits and OOPS in Scala	**Module 4** Functional Programming in Scala
Module 5 Spark and Big Data	**Module 6** Advanced Spark Concepts	**Module 7** Understanding RDDs	**Module 8** Shark, SparkSQL and Project Discussion

Session Objectives

This session will help you to understand:

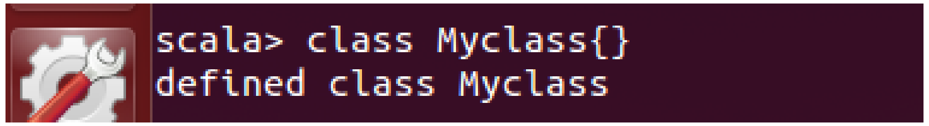
- Classes in Scala
- Properties with Getters and Setters
- Properties with only Getters
- Object-Private Fields
- Constructors
- Nested Classes
- Singletons
- Companion Objects
- Apply Method
- Packages
- Imports and Implicit Imports
- Extending a Class
- Overriding Methods
- Type Checking and Casting
- Super Construction
- Abstract Classes
- Introducing Traits in Scala



Classes in Scala

- Classes in Scala are static templates that can be instantiated into many objects at runtime
- A Class can contain information about:
  - Fields
  - Constructors
  - Methods
  - Superclasses (inheritance)
  - Interfaces implemented by the class, etc.

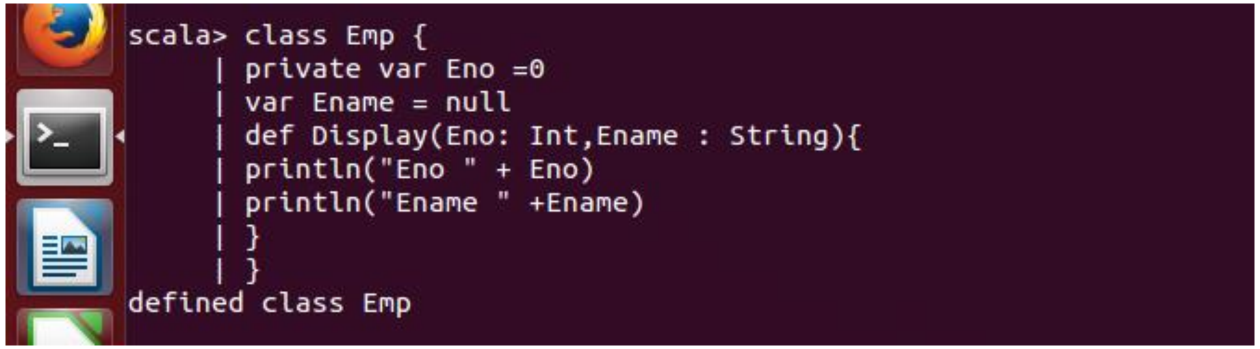
Simple class definition in Scala: `class Myclass{}`



```
scala> class Myclass{}
defined class Myclass
```

Classes in Scala (Cont'd)

- Class in Scala is very much similar to Java or C++
- Class contains fields and methods

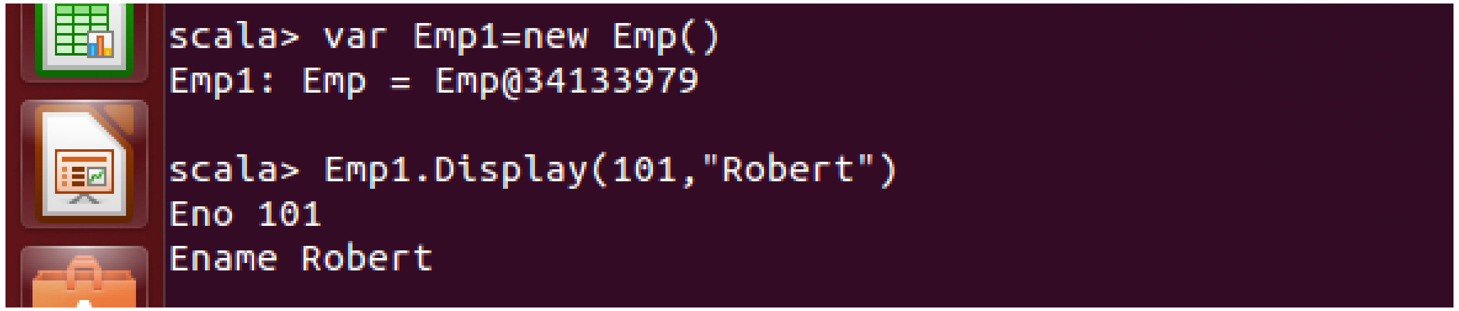


```
scala> class Emp {
     | private var Eno =0
     | var Ename = null
     | def Display(Eno: Int,Ename : String){
     | println("Eno " + Eno)
     | println("Ename " +Ename)
     | }
     | }
defined class Emp
```

- In Scala a class is NOT declared as public
- A source file can contain multiple classes
- All of the classes could be public

Classes in Scala (Cont'd)

- Previous class could be used in usual way



```
scala> var Emp1=new Emp()
Emp1: Emp = Emp@34133979

scala> Emp1.Display(101,"Robert")
Eno 101
Ename Robert
```

- Parameter less method could be called with or without parentheses
- Using any form is programmer's choice

However, as convention

- Use () for mutator method
- Use no parentheses for accessor method

Check your Understanding – 1

What is the output of the following code ?

```
 class add{
       var x:Int=10
       var y:Int=20
       def add(a:Int,b:Int)
{
    a=x+1
    println("Value of a after modification :"+a);
 }
}
Var p=new add()
p.add(5,10);
```

a. 5

b. 11

c. 16

d. Error

Check your Understanding – Solution

What is the output of the following code ?

```
  class add{
        var x:Int=10
        var y:Int=20
        def add(a:Int,b:Int)
 {
     a=x+1
     println(“Value of a after modification :”+a);
  }
 }
 Var p=new add()
 p.add(5,10);
```

a. 5

b. 11

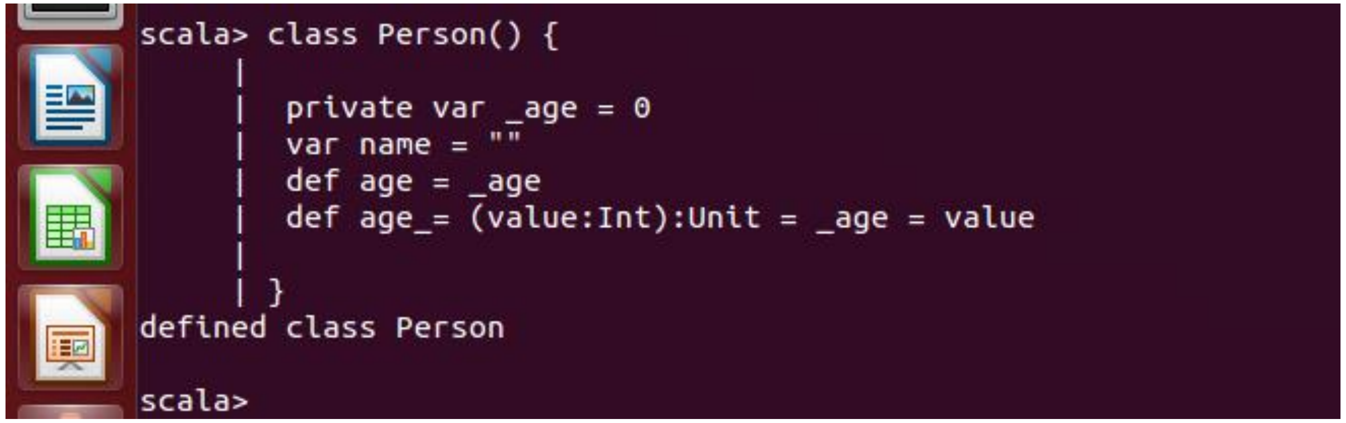
c. 16

[✓ d.] Error

Error

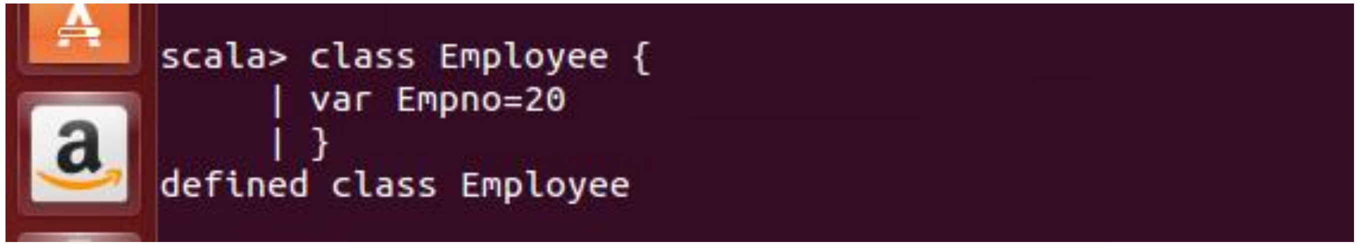
Properties with Getters and Setters

- Getters and Setters are better to expose class properties
- In Java, we typically keep the instance variables as private and expose the public getters and setters



```
scala> class Person() {
     |
     |   private var _age = 0
     |   var name = ""
     |   def age = _age
     |   def age_= (value:Int):Unit = _age = value
     |
     | }
defined class Person

scala>
```

- Scala provides the getters and setters for every field by default
- We define a public field

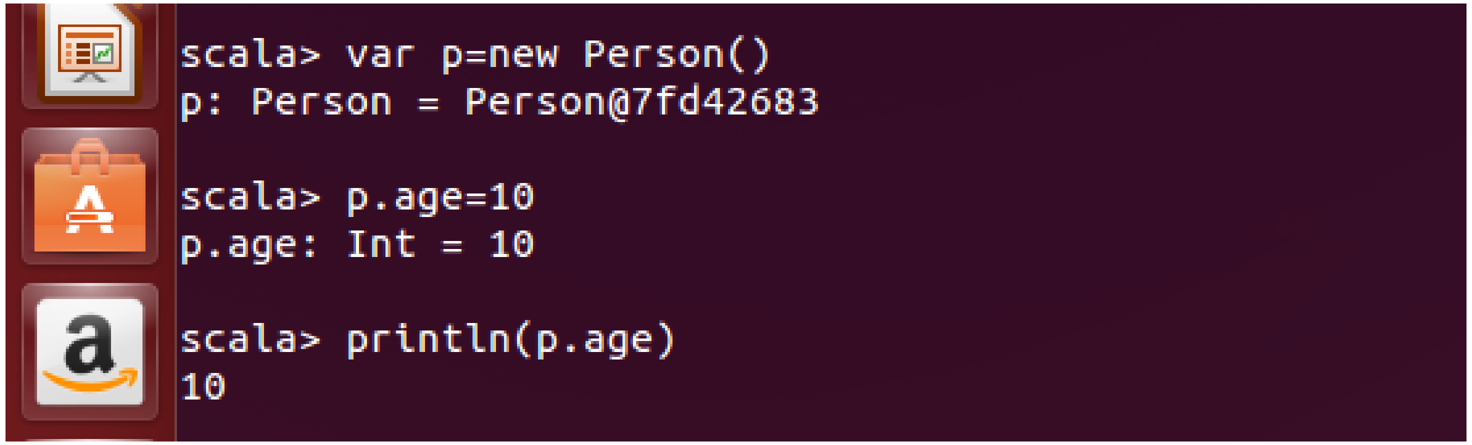


```
scala> class Employee {
     | var Empno=20
     | }
defined class Employee
```

Properties with Getters and Setters (Cont'd)

- Scala generate a class for the JVM with a private size variable and public getter and setter methods
- If the field is declared as private, the getters and setters would be private
- The getters and setter methods in previous case would be:

1. age and age_=
2. Example:

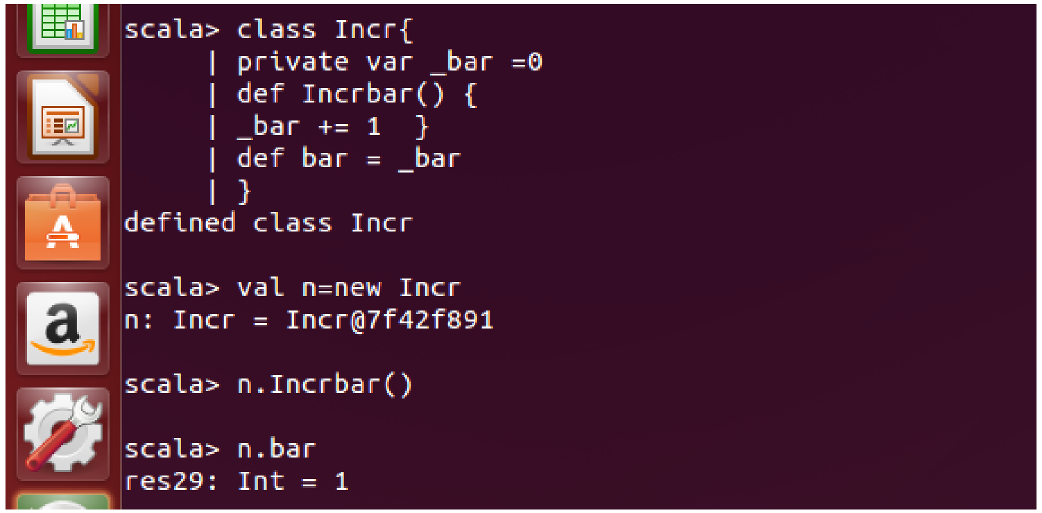


```
scala> var p=new Person()
p: Person = Person@7fd42683

scala> p.age=10
p.age: Int = 10

scala> println(p.age)
10
```

Properties with only Getters

- Sometimes we need read-only properties
- There are two possibilities:
  - The property value never changes
  - The value is changed indirectly
- For the first case, we declare the property as val. Scala treats it as final variable and thus generates only getter, no setter
- In second case, you need to declare the field as private and provide the getter, as explained below:

Semicolons are optional in Scala

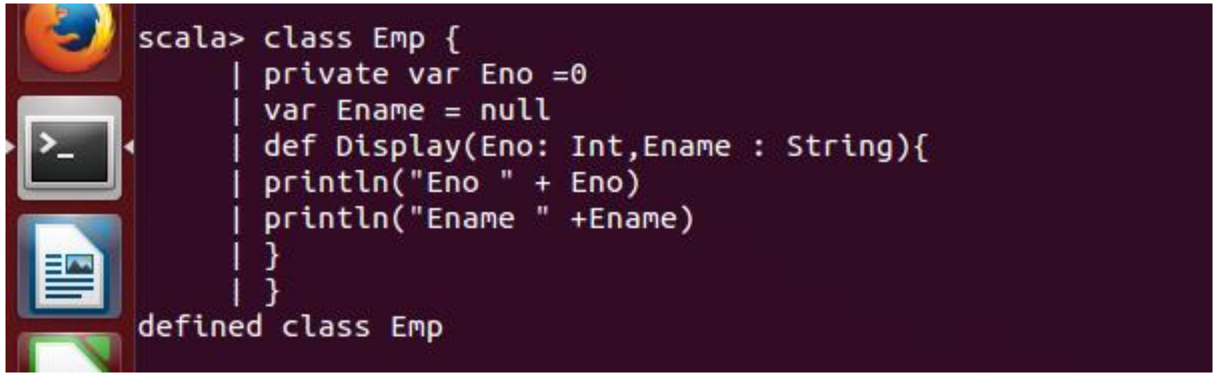


```
scala> class Incr{
     | private var _bar =0
     | def Incrbar() {
     | _bar += 1  }
     | def bar = _bar
     | }
defined class Incr

scala> val n=new Incr
n: Incr = Incr@7f42f891

scala> n.Incrbar()

scala> n.bar
res29: Int = 1
```

Object – Private Fields

In Scala (and other languages as well), a method can access the private fields of its class

Example:



```
scala> class Emp {
     | private var Eno =0
     | var Ename = null
     | def Display(Eno: Int,Ename : String){
     | println("Eno " + Eno)
     | println("Ename " +Ename)
     | }
     | }
defined class Emp
```

- We can declare the variables as object-private by private[this] qualifier
- Now the methods can only access the value field of current object
- For the class-private field, private getter and setter are generated
- For object-private field, NO getter and setter methods are generated

Summarizing Properties

- In Scala, the getters and setters are generated for each property
- For private properties, the getter and setter are private
- For a val, only getters are generated
- In Scala you can’t have a read-only property (i.e. only getter, no setter)
- No getters and setters are generated for object-private fields

Check your Understanding – 2

Which of the following statements are correct about getter() with properties?

a. The property value is changed indirectly

b. The property value is changed directly

c. The property value never changes

d. Option a and c

Check your Understanding – Solution

Which of the following statements are correct about getter() with properties?

a. The property value is changed indirectly

b. The property value is changed directly

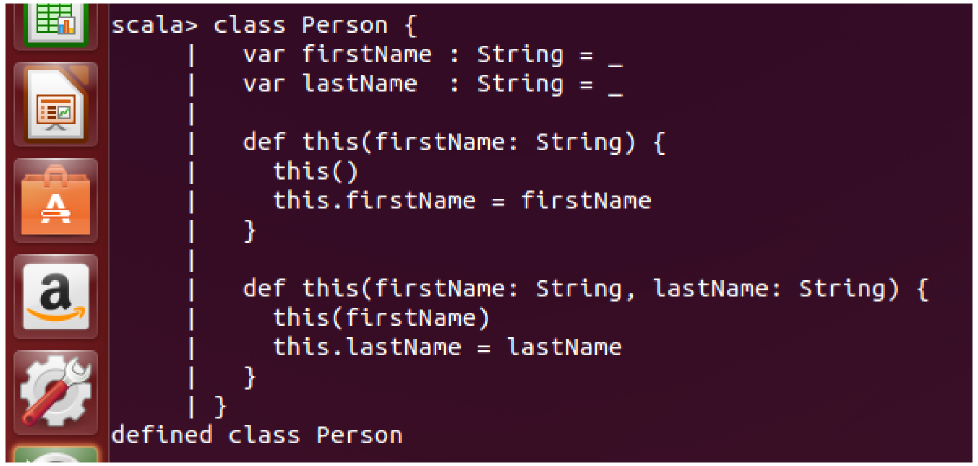
c. The property value never changes

[✓ d. Option a and c]

Option a and c

Constructor

- Constructors in Scala are a bit different than in Java
- Scala has 2 types of constructors
  - Primary Constructors
  - Auxiliary Constructors



```
scala> class Person {
     |   var firstName : String = _
     |   var lastName  : String = _
     |
     |   def this(firstName: String) {
     |     this()
     |     this.firstName = firstName
     |   }
     |
     |   def this(firstName: String, lastName: String) {
     |     this(firstName)
     |     this.lastName = lastName
     |   }
     | }
defined class Person
```

Auxiliary Constructor:

- The auxiliary constructors in Scala are called this. This is different from other languages, where constructors have the same name as the class
- Each auxiliary constructor must start with a call to either a previously defined auxiliary constructor or the primary constructor

Constructor (Cont'd)

Primary Constructor:

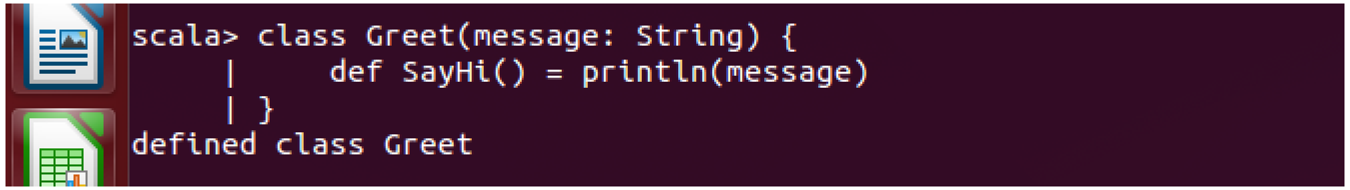
- Every class in Scala has a primary constructor
- Primary constructor isn't defined by this method

The parameters for primary constructor are placed immediately after the class name:

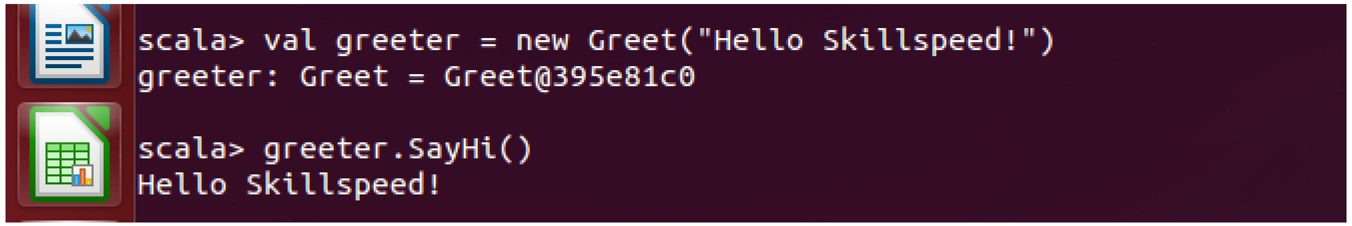
```
class Greet(message : String ) {
            // … code
           }
```

Constructor (Cont'd)

The primary constructor executes all the statements in the class definition, as explained below:



```
scala> class Greet(message: String) {
     |      def SayHi() = println(message)
     | }
defined class Greet
```



```
scala> val greeter = new Greet("Hello Skillspeed!")
greeter: Greet = Greet@395e81c0

scala> greeter.SayHi()
Hello Skillspeed!
```

The println statement is executed for every object creation

Check your Understanding – 3

Syntax of primary constructor is:

a.
```
Greet(message : String)
    {
       // …code
    }
```

b.
```
class Greet
{
     //…code
 }
```

c.
```
class Greet(message : String )
 {
           // … code
          }
```

d.
```
public Greet(message : String)
{
 //…code
}
```

Check your Understanding – Solution

Syntax of primary constructor is:

a.
```
Greet(message : String)
    {
      // …code
    }
```

b.
```
class Greet
 {
     //…code
  }
```

✓
```
class Greet(message : String )
  {
              // … code
             }
```

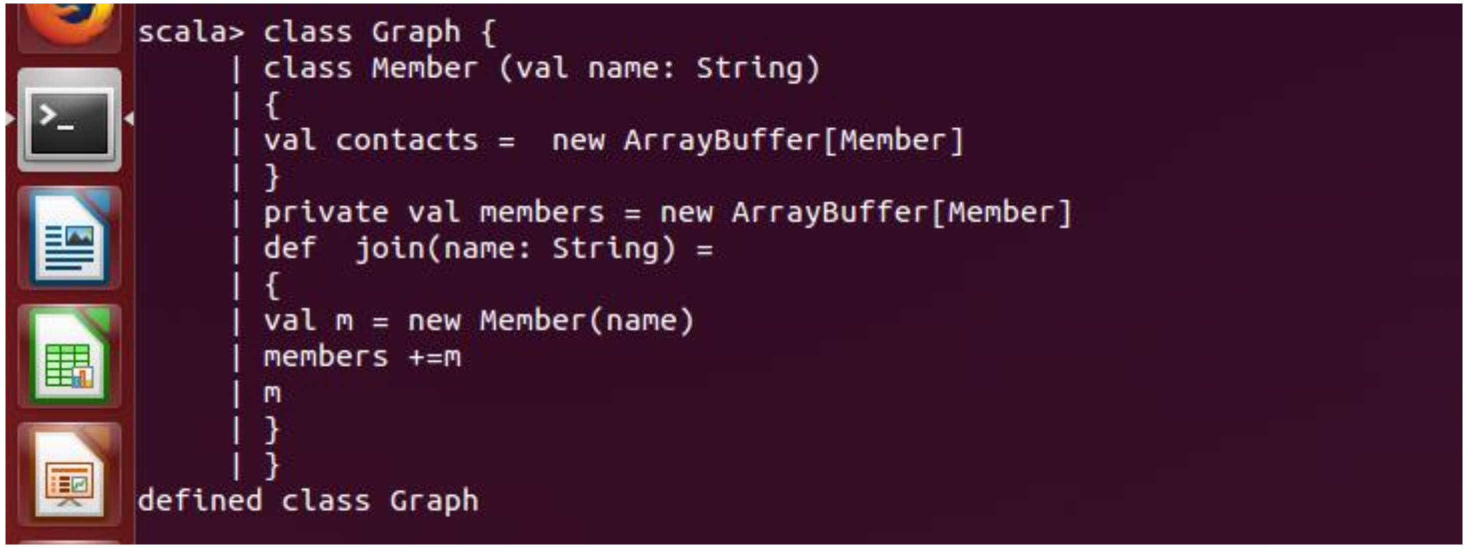
d.
```
public Greet(message : String)
{
 //…code
}
```

```
class Greet(message : String )
    {
              // … code
             }
```

Nested Classes

- Scala allows inner classes to be defined, which is to say, classes that are declared inside another class
- Unlike Java, these classes are not scoped to the containing class, but to the containing object

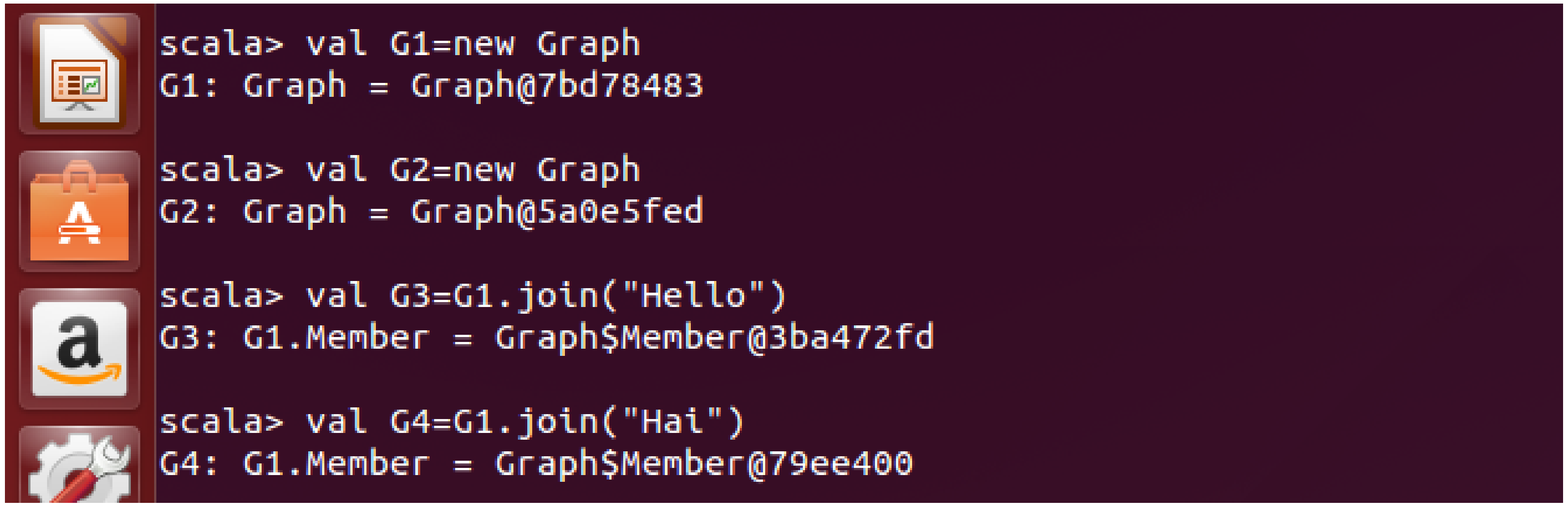
Example:



```
scala> class Graph {
     | class Member (val name: String)
     | {
     | val contacts =  new ArrayBuffer[Member]
     | }
     | private val members = new ArrayBuffer[Member]
     | def  join(name: String) =
     | {
     | val m = new Member(name)
     | members +=m
     | m
     | }
     | }
defined class Graph
```

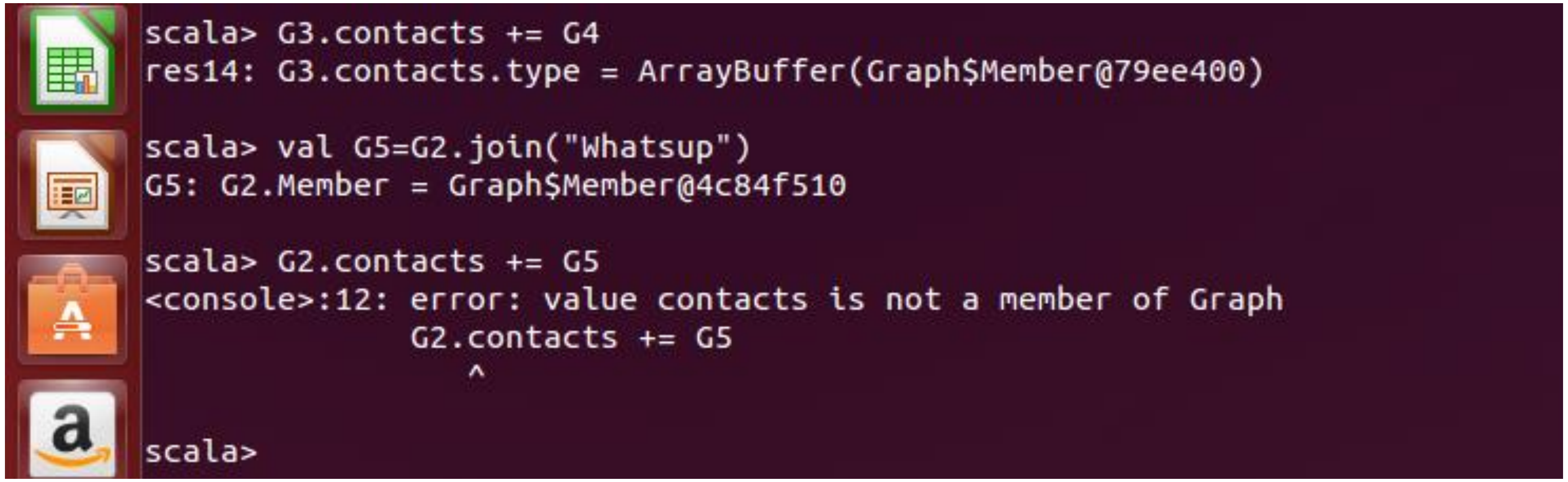
Nested Classes (Cont'd)

Accessing using Objects



```
scala> val G1=new Graph
G1: Graph = Graph@7bd78483

scala> val G2=new Graph
G2: Graph = Graph@5a0e5fed

scala> val G3=G1.join("Hello")
G3: G1.Member = Graph$Member@3ba472fd

scala> val G4=G1.join("Hai")
G4: G1.Member = Graph$Member@79ee400
```

Nested Classes (Cont’d)

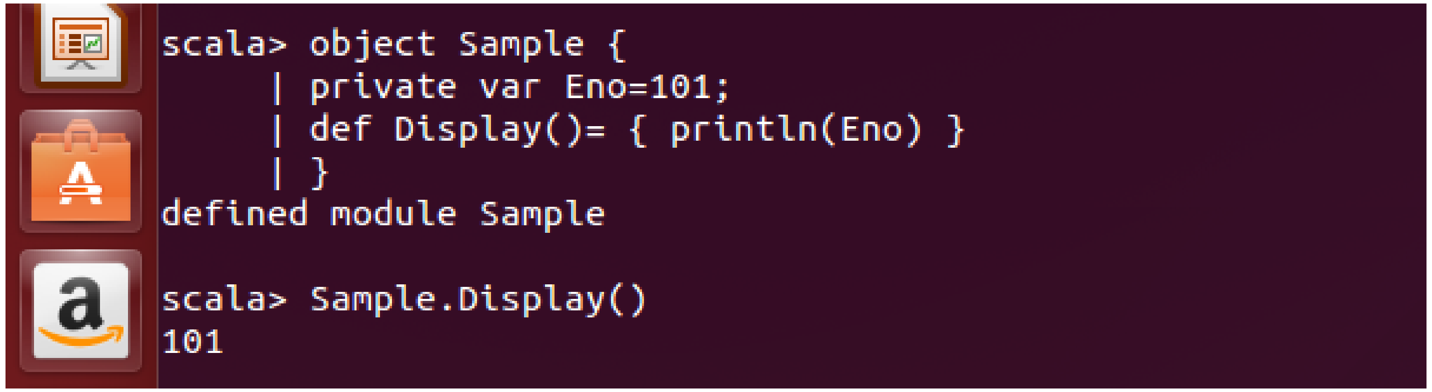


```
scala> G3.contacts += G4
res14: G3.contacts.type = ArrayBuffer(Graph$Member@79ee400)

scala> val G5=G2.join("Whatsup")
G5: G2.Member = Graph$Member@4c84f510

scala> G2.contacts += G5
<console>:12: error: value contacts is not a member of Graph
              G2.contacts += G5
                 ^

scala>
```

Singletons

- Scala doesn’t have the concept of static methods or fields
- Instead a Scala class can have what is called a singleton object, or sometime a companion object
- A singleton object definition looks like a class definition, except instead of the keyword class you use the keyword object
- An object defines a single instance of a class

Example:



```
scala> object Sample {
     | private var Eno=101;
     | def Display()= { println(Eno) }
     | }
defined module Sample

scala> Sample.Display()
101
```

Singletons (Cont'd)

- If we need new eno number, we can call sample.Display()
- Constructor of Singleton Object is executed when the object is first used
- An object has all the features of a class
- There is only one exception: Parameters can't be provided to the constructor

Singletons (Cont’d)

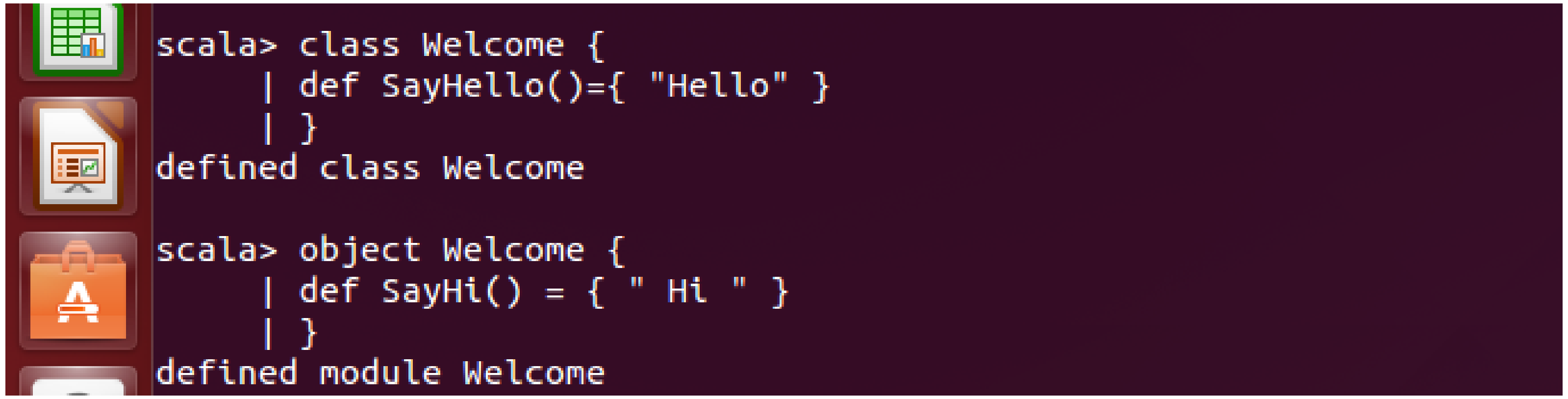
Singletons can be used in Scala as:

- When a singleton instance is required for co-ordinating a service
- When a single immutable instance could be shared for efficiency purposes
- When an immutable instance is required for utility functions or constants

Companion Objects

- When a singleton object is named the same as a class, it is called a companion object. A companion object must be defined inside the same source file as the class
- In many programming languages, we typically have both instance methods and static methods in same class

Here is an example:



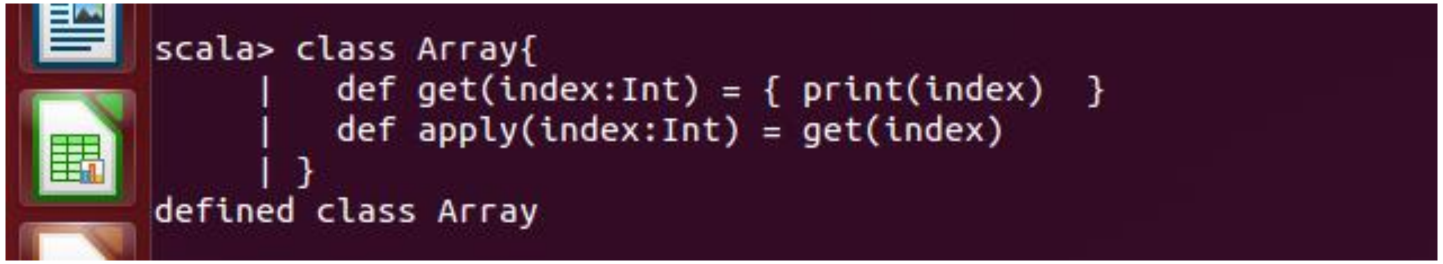
```
scala> class Welcome {
     | def SayHello()={ "Hello" }
     | }
defined class Welcome

scala> object Welcome {
     | def SayHi() = { " Hi " }
     | }
defined module Welcome
```

- The class and it's companion objects need to be in same source file
- The class and it's companion object can access each other's private features
- The companion object of the class is accessible, but NOT in scope

Apply Method

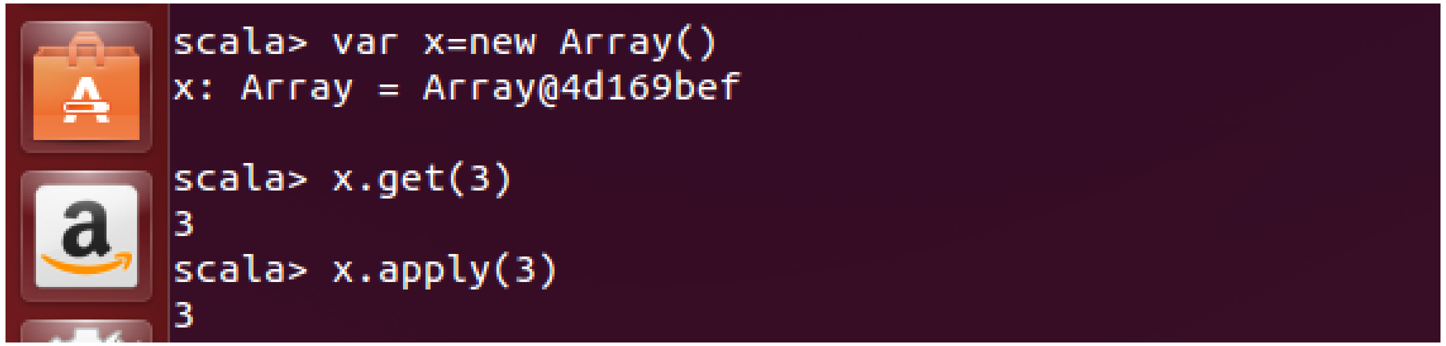
- Scala objects typically have an apply method
- The general form of Apply method is: object(arg1)
- This is same as object.apply(arg1)

Example:



```
scala> class Array{
     |   def get(index:Int) = { print(index)  }
     |   def apply(index:Int) = get(index)
     | }
defined class Array
```

Now the new Array object can be created as follows:



```
scala> var x=new Array()
x: Array = Array@4d169bef

scala> x.get(3)
3
scala> x.apply(3)
3
```

Check your Understanding – 4

Companion singletons provide an equivalent to Java's static methods

a. True

b. False

Check your Understanding – Solution

Companion singletons provide an equivalent to Java's static methods

[✓ a. True]

b. False

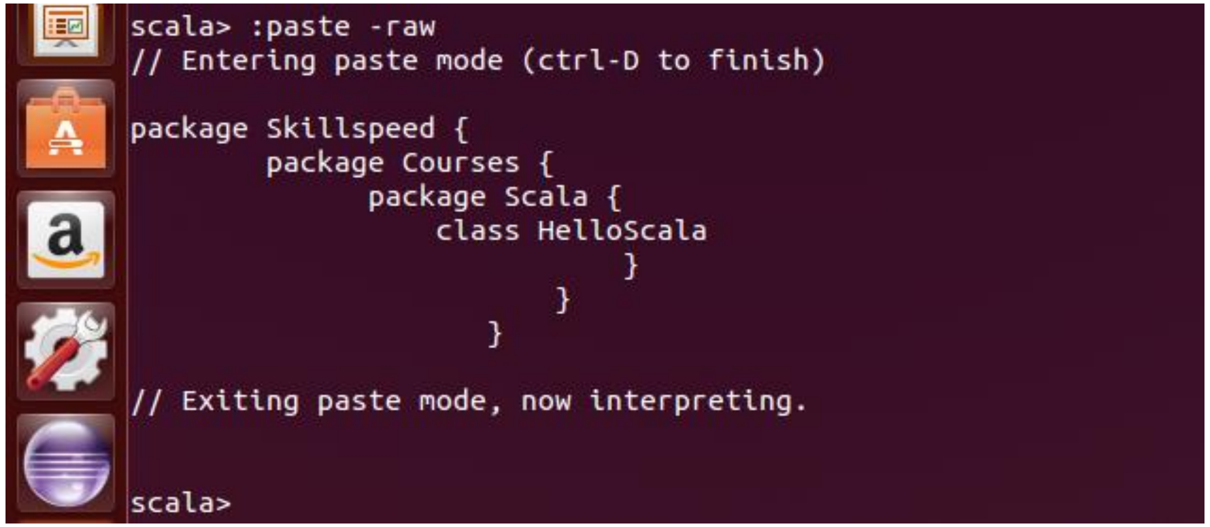
True

Packages

- A Package is a special object which defines a set of member classes, objects and packages
- In Scala, packages serve the same purpose as in Java: to manage the names in a large program
- To add the items to a package, they can be included in package statements

Example:

```
package Skillspeed {
         package Courses{
            package  Scala{
              class HelloScala
                    }} }
```



```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed {
        package Courses {
              package Scala {
                  class HelloScala
                         }
                     }
                 }

// Exiting paste mode, now interpreting.


scala>
```

Packages (Cont'd)

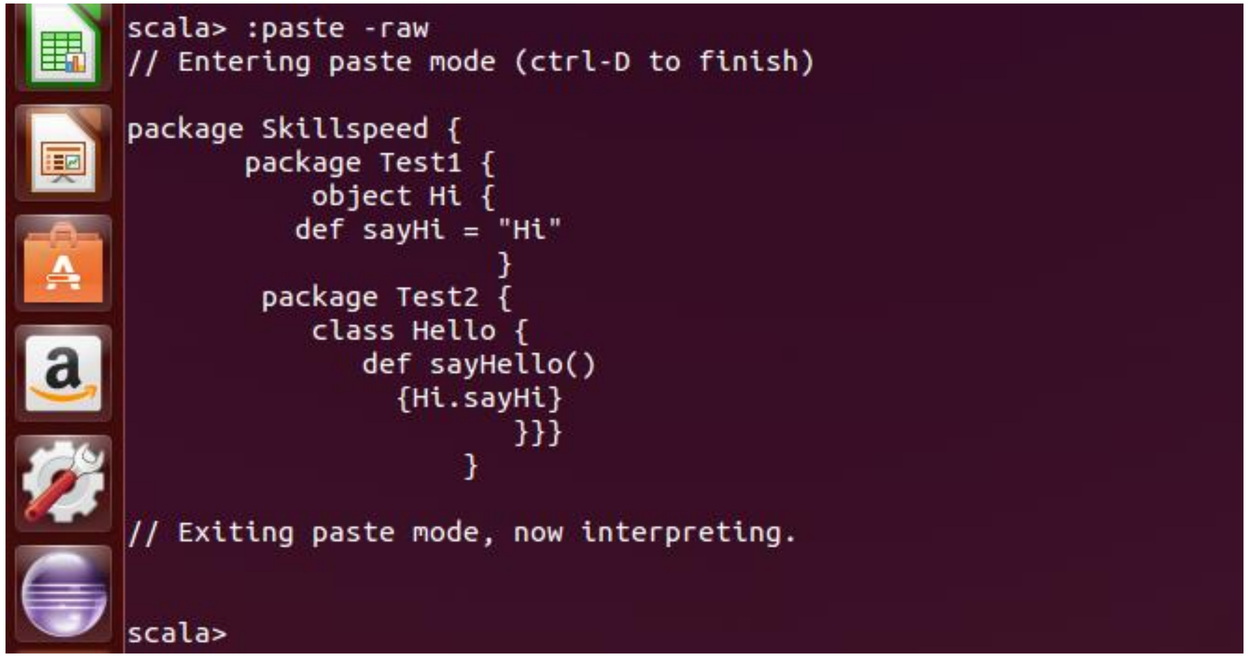
- The class HelloScala can be accessed from anywhere as Skillspeed.Courses.Scala.HelloScala
- Unlike, classes, a package can be defined in multiple files
- Conversely, a single file can have more than one package

Packages (Cont'd)

Scope Rules:

- Scope rules for packages in Scala are more consistent than Java
- Scala packages just like all other scopes
- Member names could be accessed from enclosing scope, i.e.

```
package Skillspeed {
       package Test1 {
           object Hi {
          def sayHi = "Hi" }
        package Test2 {
           class Hello {
              def sayHello()
                {Hi.sayHi} }}}}
```



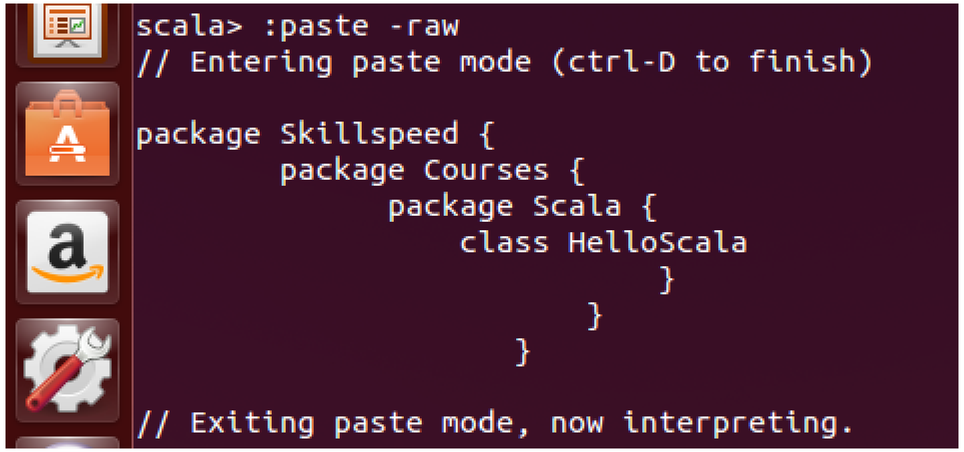
```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed {
       package Test1 {
           object Hi {
          def sayHi = "Hi"
                 }
        package Test2 {
           class Hello {
              def sayHello()
                {Hi.sayHi}
                       }}}
              }

// Exiting paste mode, now interpreting.


scala>
```

Packages (Cont'd)

Top of File Notation:

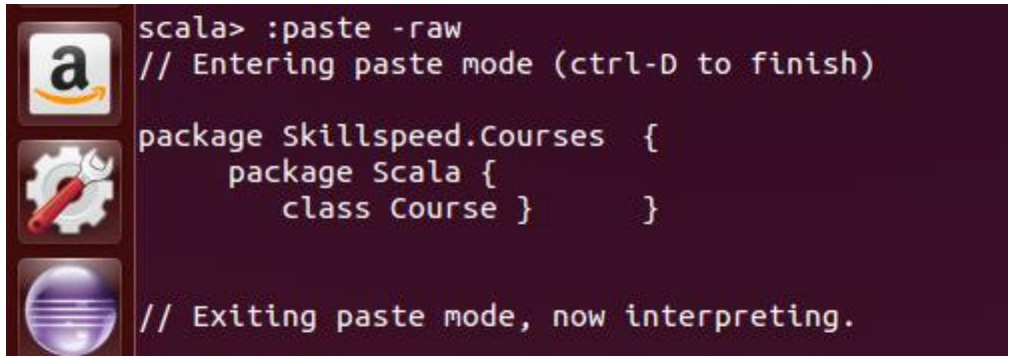
Instead of nested notation, we could have the package notation at the top of file also

Example:



```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed {
        package Courses {
              package Scala {
                  class HelloScala
                             }
                         }
                     }

// Exiting paste mode, now interpreting.
```

Is equivalent to



```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed.Courses  {
     package Scala {
        class Course }       }


// Exiting paste mode, now interpreting.
```

Note: In the above example, everything belongs to package Skillspeed.Courses.Scala, but the package Skillspeed.Courses.Scala is also opened up, so it's contents could also be referred

Packages (Cont'd)

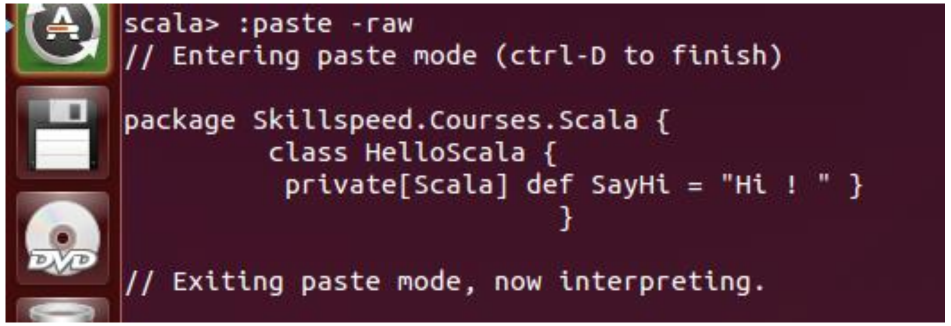
skillspeed
for the serious learner

Package Visibility

- In Java, we typically control the access of the class members by public, private or protected
- In Scala, the same effect could be achieved through qualifiers

Example:

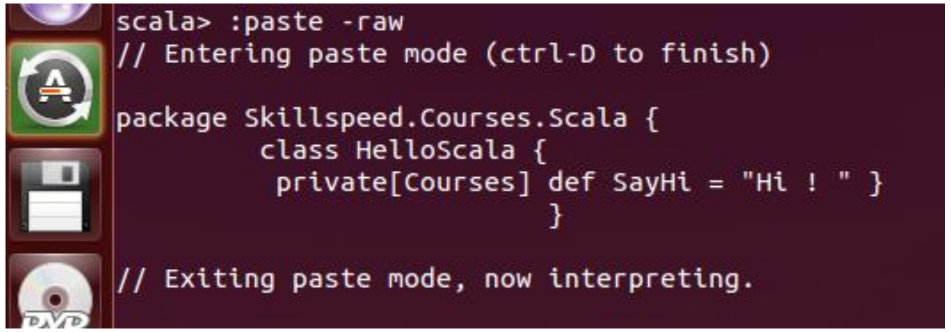
```
package Skillspeed.Courses.Scala {
               class HelloScala {
        private[Scala] def sayHi = "Hi !"
                 } }
```



```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed.Courses.Scala {
         class HelloScala {
          private[Scala] def SayHi = "Hi ! " }
                              }

// Exiting paste mode, now interpreting.
```

The visibility could also be extended to enclosing package:

```
package com.spark.skillspeed
               class Courses {
        private[spark] def sayHi = "Hi !"
}
```



```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package Skillspeed.Courses.Scala {
         class HelloScala {
          private[Courses] def SayHi = "Hi ! " }
                              }

// Exiting paste mode, now interpreting.
```

Imports and Implicit Imports

- Packages/ classes can be imported in Scala
- Import serve the same purpose as in Java:
  To use short names instead of long ones
- All the members of a package can be imported as:
  - import java.awt._
  - Note that "_" is used instead of "*"
- In Scala, imports can be anywhere, instead of being at the top of the file, unlike Java
- We can use selectors to import only few members of a package like:
  - import java.awt.{Color, Font}
- Every Scala program implicitly starts with:
  - import java.lang._
  - import scala._
  - import Predef._

Check your Understanding – 5

import p1.p2.z means:

a. All members of p1 are members of z
b. All members of p2 are members of z
c. The member z of p2, itself member of p1
d. All of the above

Check your Understanding – Solution

import p1.p2.z means:

a. All members of p1 are members of z

b. All members of p2 are members of z

[✓ c.] The member z of p2, itself member of p1

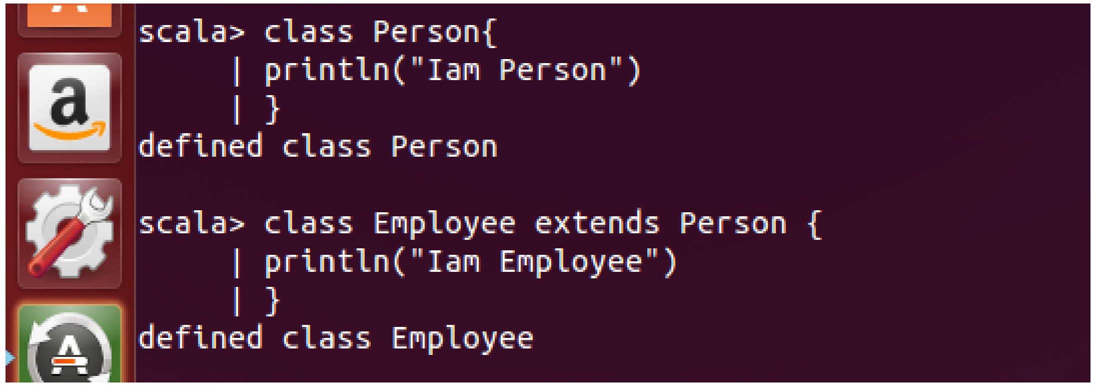
d. All of the above

The member z of p2, itself member of p1

Extending a Class

Just like Java, classes can be extended using extends keyword

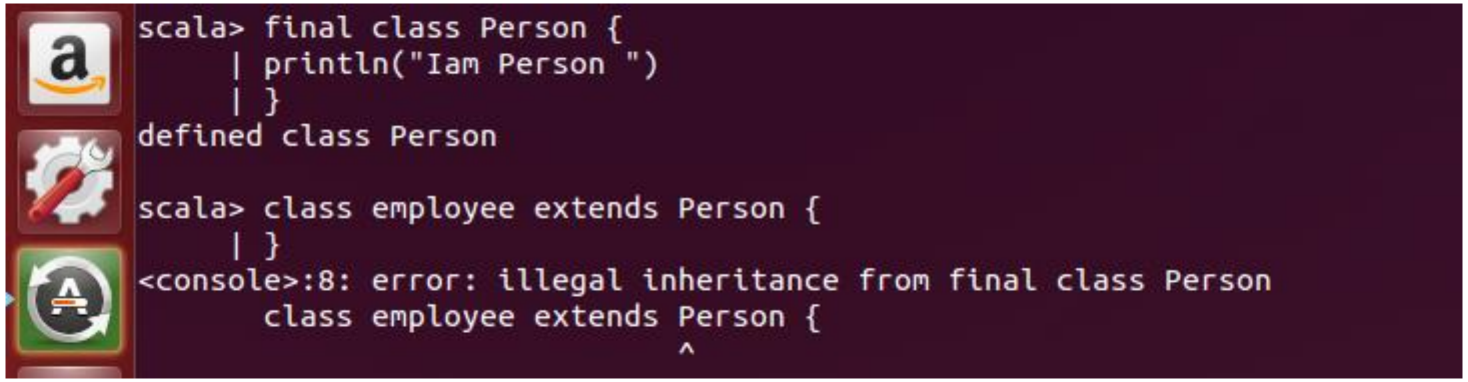
```
 class Employee extends
Person{
    //code
     }
```



```
scala> class Person{
     | println("Iam Person")
     | }
defined class Person

scala> class Employee extends Person {
     | println("Iam Employee")
     | }
defined class Employee
```

Extending a Class (Cont'd)

- Just like Java, new methods and fields can be introduced or superclass methods or fields could be overridden in subclasses
- A class can be declared as final to avoid it being extended
- Unlike Java, individual field or method could also be marked as final to avoid them being overridden



```
scala> final class Person {
     | println("Iam Person ")
     | }
defined class Person

scala> class employee extends Person {
     | }
<console>:8: error: illegal inheritance from final class Person
       class employee extends Person {
                              ^
```

Overriding Methods

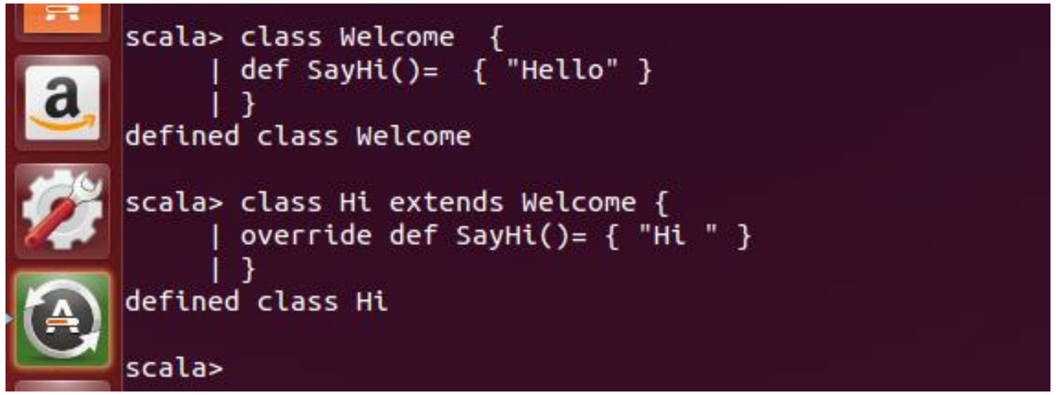
When overriding a method we should use the override modifier

- The override modifier is useful in following scenarios:

  - When name of the method being overridden is misspelled
  - When a wrong parameter type is provided
  - When a new method is introduced in superclass which clashes with a subclass method

- Invoking superclass method is same as in Java, by super keyword:
- The following code sample shows how to override the toString() method

```
Ex: public class Employee extends Person {
    ……
     override def toString=super.toString + “Hi …! “
}
```



```
scala> class Welcome  {
     | def SayHi()=  { "Hello" }
     | }
defined class Welcome

scala> class Hi extends Welcome {
     | override def SayHi()= { "Hi " }
     | }
defined class Hi

scala>
```

Type Checking and Casting

- isInstanceOf method is used to decide whether an object belongs to a class
- asInstanceOf method is used to convert a reference to a subclass reference

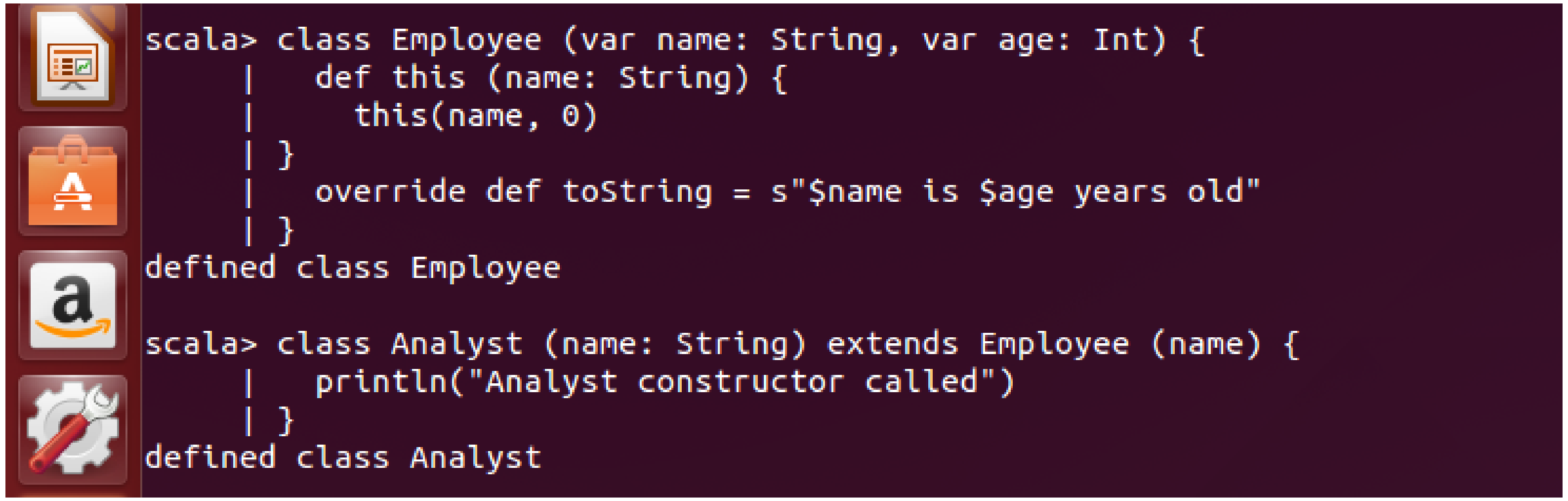
```
if (a.isInstanceOf(Person )) {
    val b = a.asInstanceOf(Person)     // b has the type Person
        …
        }
```

- classOf method is used to determine the class of a given reference

```
if(a.getClass == classOf[Person]) {
                     ……
                    }
```

Superclass Construction

- All the classes have a primary constructor and many auxiliary constructors, and all the auxiliary constructor should either call a primary constructor or previous auxiliary constructor
- It means an auxiliary constructor can never invoke a superclass constructor directly

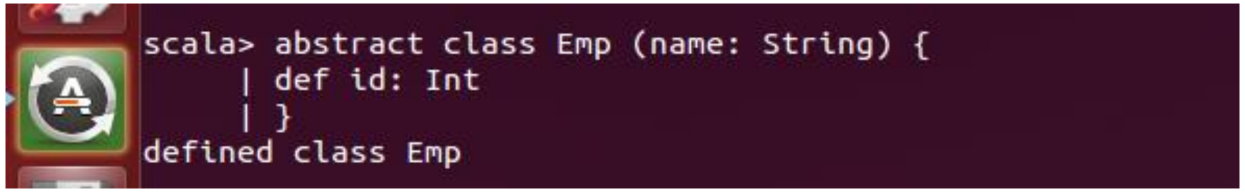


```
scala> class Employee (var name: String, var age: Int) {
     |   def this (name: String) {
     |     this(name, 0)
     | }
     |   override def toString = s"$name is $age years old"
     | }
defined class Employee

scala> class Analyst (name: String) extends Employee (name) {
     |   println("Analyst constructor called")
     | }
defined class Analyst
```

- Putting the class and constructor together makes a very short code in Scala
- The Emp class has three parameters, out of which, two are passed to its Superclass Person

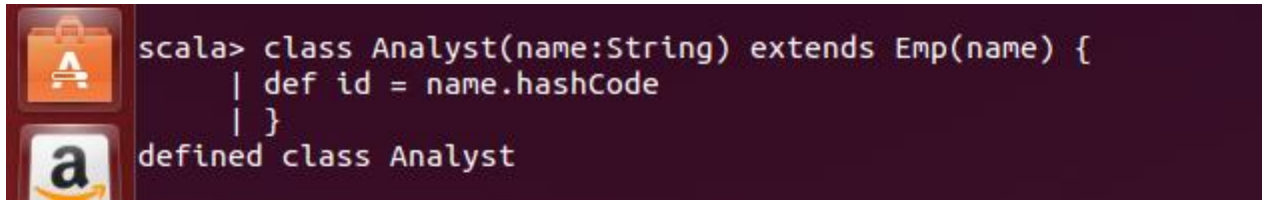
Abstract Classes

- Just like Java, you can use the abstract keyword for a class, which can't be instantiated:



```
scala> abstract class Emp (name: String) {
     | def id: Int
     | }
defined class Emp
```

- Here we declared a method to generate id, but didn't provide implementation
- Each concrete subclass of Emp should provide the implementation of id
- In a subclass, we don't need to specify override while defining an abstract superclass method:



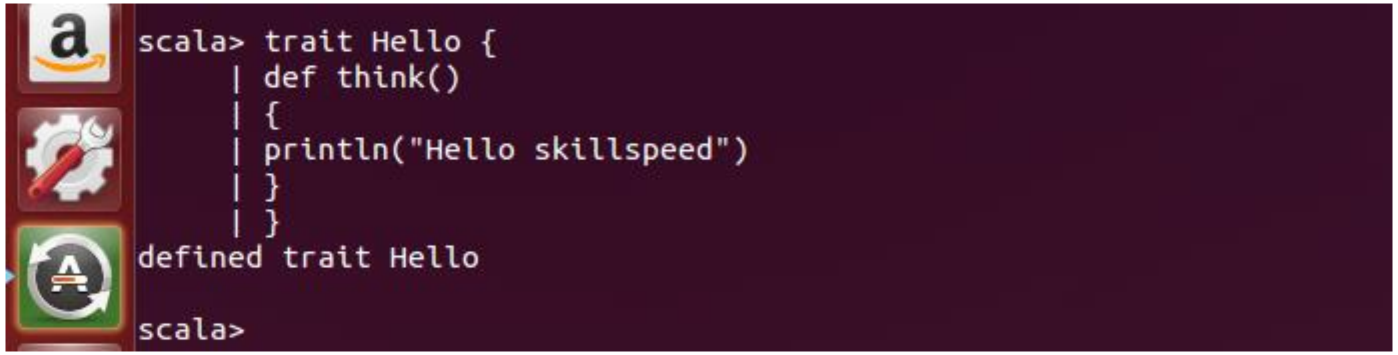
```
scala> class Analyst(name:String) extends Emp(name) {
     | def id = name.hashCode
     | }
defined class Analyst
```

Introducing Traits in Scala

- Traits encapsulate methods and fields definitions we can reuse by mixing them into classes we define
- Unlike classes inheritance that allow each class to inherit one class only, a class can mix in any number of traits

Trait Definition:

The syntax is the same syntax we use when defining a class. The only difference is using the trait keyword instead of class



```
scala> trait Hello {
     | def think()
     | {
     | println("Hello skillspeed")
     | }
     | }
defined trait Hello

scala>
```

Questions

skillspeed
for the serious learner

thank you!