# Trait Linearization in Scala

Last Updated : 06 Jun, 2019

**Scala Linearization** is a deterministic process which comes into play when an object of a class is created which is defined using inheritance of different traits and classes. linearization helps to resolve the diamond problem which occurs when a class or trait inherits a same property from 2 different concrete classes or traits.

**Syntax :**

```
trait C{}
trait B{}
class A{}
object a_obj= new class A extends B with C
```

The linearization will look like as follows :-

```
C-> AnyRef-> Any
B-> AnyRef-> Any
A-> AnyRef-> Any
a_obj-> A-> C-> B-> AnyRef-> Any
```

Here **Any** is the superclass of all classes, also called the top class. It defines certain universal methods such as equals, hashCode, and toString. **AnyRef** represents reference classes. All non-value types are defined as reference types. AnyRef corresponds to java.lang.object . Every Scala trait and class implicitly extend these Scala objects at the end of linearization hierarchy.

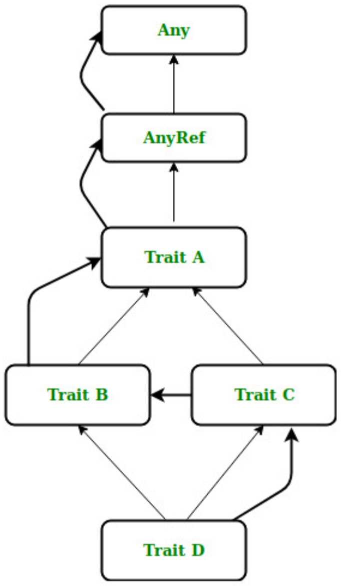
```
// Scala program defining trait A
```

```
// defining trait B inheriting A
trait B extends A
{
    override def name: String ="class b"
}

// defining trait C inheriting A
trait C extends A
{
    override def name: String ="class c"
}

// defining class D inheriting B and C both
class D extends B with C
{
    override def name: String = super.name
}

// Creating object
object GFG
{
        var class_d = new D

        // whose property will be inherited
        println(class_d.name)
    }
}
```

**Output :**

```
class c
```

**Linearization** for class D follows dark bold arrow. **Inheritance** for class D follows light arrow.



*Trait linearization and inheritance diagram*

As we can see in above diagram linearization will not be as same as inherited structure. Scala traits/classes are dynamically placed in linear order in which will the linearization will be applied as below.

```
D-> C-> B-> A-> AnyRef-> Any
```

**Following rules are followed for the determining the linearization:**

1. Take the first **extended** trait/class and write its complete inherited hierarchy in vertical form, store this hierarchy as X.
2. Take the next trait/class after the **with** clause, write its complete hierarchy and

traits/classes to the front of the hierarchy X.
3. Go to step 2 and repeat the process, until no trait/class is left out.
4. Place the class itself in front of hierarchy as head for which the hierarchy is being written.

Let's understand some examples.

**Example :**

```
// Scala program for linearization
// defining old_car class
class old_Car
{
    def method: String= "old car "
}

// defining new_Car_Designs trait
trait new_Car_Designs extends old_Car
{
    override def method: String ="Designing-> "+ super.method
}

// defining new_Car_Part trait
trait new_Car_Part extends old_Car
{
    override def method: String = "Add new part-> "+ super.method
}

// defining new_Car_Paint trait
trait new_Car_Paint extends old_Car
{
    override def method: String = "Repainting-> "+ super.method
}

// defining new_Car class
class new_Car extends new_Car_Paint with
new_Car_Part with new_Car_Designs
{
    override def method: String = "new car-> "+ super.method
}

// Creating object
object geekforgeeks
```

```
    {
        // new_Car object
        var car1 = new new_Car
        println(car1.method)
    }
}
```

**Output :**

```
new car-> Designing-> Add new part-> Repainting-> old car
```

**Example :**

```
// Scala program for trait linearization
// defining classes and traits
class flavour
{
    def make (flavour: String): Unit =
    {
        println(flavour)
    }
}

// defining texture trait
trait texture extends flavour
{
    abstract override def make (flavour : String)
    {
        super.make(flavour + "texture ")
    }
}

// defining cream trait
trait cream extends texture
{
    abstract override def make (flavour : String)
    {
        super.make(flavour + "with cream ")
    }
}

// defining jelly trait
trait jelly extends texture
```

```
        super.make(flavour + "with jelly ")
    }
}
// defining cone trait
trait cone extends flavour
{
    abstract override def make (flavour : String)
    {
        super.make(flavour + "in cone ")
    }
}

// creating new ice-cream flovours
// with above traits and classes
// inheriting different traits and classes
class Myflavour extends flavour with jelly
{
    override def make (flavour : String)
    {
        super.make(flavour)
    }
}
class Myflavour2 extends flavour with cream with cone
{
    override def make (flavour : String)
    {
        super.make(flavour)
    }
}

// Creating object
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        // creating new objects
        var icecream1 = new Myflavour
        var icecream2 = new Myflavour2 with jelly
        println(icecream1.make("chocolate "))
        println(icecream2.make("vanilla "))
    }
}
```

Output :

```
chocolate with jelly texture
()
vanilla with jelly in cone with cream texture
()
```

**Important Points About Linearization**

- Scala solves ambiguity of traits/classes by linearization process.
- Scala uses linearization whenever a new class has been instantiated. Taking all the traits/classes and forming a linear order which points to corresponding super classes/traits thus **super** method knows its parent method.
- These super method calling is done in a stackable manner.
- Linearization may or not be the same as the inherited mixins as they are written.
- We cannot explicitly add a class to inheritance when it is already been implicitly inherited in a linearization otherwise it will result in error as **inheritance twice**.
- No trait/class is ever repeated in linearization.

Previous
Scala | Trait Mixins

Next
Scala Lists

01 **Scala Trait Traversable | Set-1**
19, Mar 19

02 **Scala Trait Traversable | Set-2**
25, Mar 19

03 **Scala Trait Traversable | Set-3**
25, Mar 19

04 **Scala Trait Traversable | Set-4**
25, Mar 19

05 **Scala | Trait App**
12, Apr 19

06 **Scala | Trait Mixins**
15, Apr 19

07 **Scala | Sealed Trait**
24, Apr 19

08 **Scala Tutorial – Learn Scala with Step By Step Guide**
25, Nov 19

**Article Contributed By :**

Patabhu
@Patabhu

**Vote for difficulty**

Easy Normal Medium Hard Expert

**Article Tags :** Picked, Scala

Improve Article Report Issue