Object-oriented programming - a comparison between Go and Scala

Doc Writer Tobias Djermester

2018-01-05

Chapter 1. Introduction

The following study compares the programming languages Go with Scala in the context of object-oriented programming. Both Go and Scala are considered as object-oriented languages. Object oriented languages are meant to follow the object-oriented programmming paradigm. Today, the object-oriented paradigm is one of the most important paradigm programming langueges are basing upon. Even though several programming languages are considered as multi-paradigm languages most of them involve object-oriented programming features. This study deals with the comparison of Go and Scala in terms of the most essential core features and concepts of object-oriented programming.

First of all the most importend differences and similarities in the typesystem of Go and Scala are discussed. The study continues with presenting the core concepts to accomplish encapsulation with both Scala and Go. Chapter 4 demonstrates optiones for implementing inheritance or inheritance like behaviour in both languages. The fifth chapter deals with the object-oriented concept of polymophism and demonstrates how to accomplish polymorphic behaviour in different situations. Following polymophism this paper illustrates the last but not least important object-oriented concept of abstraction and introduces the access specifications provided by Go and Scala in order to define access rights. This investigation closes with an introduction of object oriented concepts for concurrent programming.

Chapter 2. Type system

Go is a statically typed programming language in which type safety is accomplished by the compiler at compile time. Every variable in Golang owns an associated type. Go offers some primitive data types as well as derived and aggregated data types. Primitive data types in Golang comprise boolean, strings and numeric types like integer, float or complex.
Go doesn't provide any implicit type conversion. Therefore it is not possible to add, subtract, compare or perform any other kind of operation on two different types even if both of them are numeric. In order to perform operations on different types it is necessary to explicitly cast the variables to the target type first. In addition to primitive data types Go also provides derived data types including pointers, arrays, structures, unions, slices, interfaces, maps and channels.

In contrast to Go, Scala owns the same data types as Java, with the same memory footprint and precision. Compared to Go and Java, Scala does not have any primitive types. All data types, like Byte, Int or Char, available in Scala are objects. Therefore methods can be invoked directly on every data type.
Like Go, Scala is a statically typed language. So after defining a variables type it is static and thus can't be changed. Additionaly Scala provides various features to loose the strict type system without compromising type safety at compile time.
On the one hand both Scala and Go offer type inference also known as implicit typing. Type inference refers to the automatic detection of the data type of an expression. In other words this means Scala and Go automatically deduce the type of an expression without the need of mentioning it. This avoids verbose typing but still preserves the compile-time type safety.
Beyond type inference Scala also provides sub-typing. Sub-typing is a form of type polymorphism in which a subtype which is related to another datatype (supertype) will automaticly be converted upwards wherever necessary. Since every data type is an object and every object in Scala's type system hierarchy derives from another object, this is even possible with basic data types without the need of explecitely converting them. A simple example to demonstrate sub-typing in Scala is shown in Listing 1: Subtyping in Scala.

Listing 1: Subtyping in Scala

```
val list1 = List(10, 'a')
// -> List[Int] = List(10, 97)

val list2 = List(20.2, 10)
// -> List[Double] = List(20.2, 10.0)

val list3 = List("Hello", 10, true)
// -> List[Any] = List(Hello, 10, true)
```

When a heterogeneous list is constructed, the sub-typing converts the lower type into a higher type wherever necessary. The second list is a list of type `Double` since the common subtype is a double. Therefore the integer is implicitely converted to a double. If it doesn`t match, it continues with the next common supertype. All of these conversions can be translated to the type system hierarchy.
Another great feature offered by Scala to make the otherwise static type system from the developer view a bit more dynamic is the implicit convertion of data types. Scala allows to define methods which take a value of a certain type as a parameter and return a value of a different type as the

result. By declaring this method to be `implicit`, the compiler is informed to automatically use this conversion method in situations where a value of one type is needed but a value of another type is passed. An example which shows the opportunity of such features by adding a string with an integer is shown in Listing 2: Implicit convertion in Scala.

Listing 2: Implicit convertion in Scala

```
implicit def str2int(str:String):Int = Integer.parseInt(str)

def addTwo(a:Int, b:Int) = a + b
addTwo("123", 456)
```

In contrast to Go, Scala has a rich set of collection libraries. Most collections contain linear sets of items. Collections may be strict or lazy. Lazy collections have elements that may not consume memory until they are accessed. In Scala lazy collection types are streams or ranges. Additionally, collection types may be mutable or immutable. In mutable collections the contents of the references can change. In immutable collections the object that a reference refers to is never changed. All in one the most commonly used collection types in Scala are Lists, Sets, Maps, Tuples and Options.

Chapter 3. Encapsulation

In terms of encapsulation and the class concept in general both languages Go and Scala differ greatly. Comparing Go with Scala in terms of encapsulation is similar to the comparison of a classic class concept with a class-free concept.

Although Go has types and methods and allows an object-oriented style of programming, there is no typical class concept. Go doesn't provide classes but it does provide structs. In Go methodes can be added on structs. This in combination with bundling the data by the stuct itself and allowing the associated methods to change their state, provides the behaviour akin to a class. Hence structs in Go can be an effective replacement for classes in Scala. Listing 3: Encapsulation in Go shows a typical struct similar to structs in other programming languages. In contrast to other programming languages the struct in Go owns a constructor represented by the `NewCircle()` function and also holds the additonal function `calculateArea()` by declaring `*circle` as the receiver.

Listing 3: Encapsulation in Go

```
type circle struct {
    radius  float64
}

func NewCircle(radius float64) *circle {
    c := new(circle)
    c.radius = radius
    return c
}

func (c *circle) calculateArea() float64 {
    area := math.Pi *  math.Pow(c.radius, 2)
    return area
}
```

Scala is purely object-oriented. In Scala everything is an object. As mentioned before conventional primitive types like integers or characters are objects in Scala. Furthermore static methods are missing. Therefore Scala is even more object-oriented than Java since there is no possibility of having static members. In addition to that Scala also provides first class functions which means that Scala treats functions as first class objects. This means that functions actually inherits from objects. This allows functions to be dynamically built, called and passed around as any other object. In Scala encapsulation is done by the help of classes.
Because of the missing possiblity of including static class members into classes, Scala provides singleton objects. Singleton objects are declared using the `object` keyword. When a singleton object is named the same as a class, it is called a companion object. Conversely, the class is the objects companion class. A companion class and object can access the private members of its companion. Companion objects can also be used to access functionality without instantiating an object with the `new` operator. Therefore companion objects are meant to be used for methods and values which are not specific to instances of the companion class. An example demonstrating the benefit of using companion objects is shown in Listing 4: Companion object in Scala.

Listing 4: Companion object in Scala

```
class Circle(var radius: Double) {
  import Circle._

  circlesCount += 1
  def area: Double = calculateArea(radius)
}

object Circle {

  private var circlesCount = 0
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
  def getNumberOfCircles: Int = circlesCount
}

object CompanionObject {

  def main(args: Array[String]) {

    val circle1 = new Circle(5.0)
    val circle2 = new Circle(5.0)
    circle2.area
    println(Circle.getNumberOfCircles)
  }
}
```

In terms of encapsulation, extending an already existing class with new functionality also plays a crucial role in modern-day programming languages.
In Go adding a method to a type can only be done in the same package. Moreover, methods in Go are more general than in Scala because they can be defined for any sort of data, even for primitiv types such as plain integers. As shown in Listing 5: Type extension in Go the type `Int` which has been assigned to the primitive type `int` has been extended by the new method `Add()`. Therefore `Int` has the same capability as `int` and additionally provides the `Add()` functionality.

Listing 5: Type extension in Go

```
  type Int int

  func (i Int) Add(j Int) Int {
      return i + j
  }

  func main() {
      i := Int(5)
      j := Int(6)
      fmt.Println(i.Add(j))
      fmt.Println(i.Add(j) + 12)
  }
```

Scala has managed to solve the problem of class extensions in a different but still type-safe way. In Scala you can not only add new methods to classes inside the same package, you can also extend existing classes imported from other packages or biblitheks. This can be done by means of implicit conversions. Listing 6: Class Extension in Scala demonstrates the extension of the already existing Scala class `String` by adding the new `printSelf()` function.

Listing 6: Class Extension in Scala

```
class MyString(str:String) {
  def printSelf() {
    println(str)
  }
}

implicit def str2myString(str:String) : MyString = new MyString(str)

"Hello".printSelf()
```

In this example, before the compiler throws an exception it checks if there is any chance of an implicit type conversion which might provide the `printSelf()` functionality. Although the `printSelf()` call makes you belief that the functionality is called right away from the class `String`, the class `String` has been implicitly converted to `MyString` when additional functionality was needed.

Chapter 4. Inheritance

A key feature supporting traditional object oriented design is inheritance. The main aspect of inheritance in object oriented languages is to eliminate redundant code.

In Go there is no type hierarchy because Go does not support inheritance. However there are ways to embed types in other types to provide something analogous but not identical to inheritance. Instead of inheritance Go offers composition. Although composition and delegation is possible in Scala, Go handles composition even simpler and therefore offers an effective alternative to inheritance in Scala.
In Go composition is done by embedding structs within others structs. In Listing 7: Multiple Composition in Go `circle` extends `geometricShape` by including `geometricShape` into `circle`. This will eliminate otherwise needed code in the `circle` structure by delegating to the `geometricShape` instance when needed. As shown afterwards Go offers the posibility of accessing the embedded fields and functions as if they were part of the outer struct. By this simple delegation Go's composition concept becomes syntactically similar efficient to the concept of inheritance in Scala. Additionaly, by including multiple types into one type Go offers the opportunity of multiple inheritance as shown by the second embedded struct `drawing` into `circle`.

Listing 7: Multiple Composition in Go

```
type geometricShape struct {
    id int
    description string
}

type drawing struct {
    color color.Color
    rotation float64
    description string
}

type circle struct {
    radius  float64
    geometricShape
    drawing
}

func main() {
    var c = NewCircle(15)
    c.color = color.Black
    c.id = 5
    c.geometricShape.description = "Geometric shape description"
    c.drawing.description = "Drawing description"
}
```

In terms of inheritance Scala provides a simple mechanism by using class-based extension. As demonstrated in Listing 8: Multiple inheritance in Scala Scala enables inheritance by extending a predefined base class with the `extend` keyword similar to other object oriented languages. In

contrast to Go, Scala doesn't allow multiple inheritance in the classical sense but allows the extension by multiple traits. In Scala traits encapsulate method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from at most one superclass, a class can extend any number of traits. In contrast to simple interfaces in Go, Scala also allows traits to be partially implemented. As shown in Listing 8: Multiple inheritance in Scala `Circle` extends one trait and one class. The difficulty which arises from multiple inheritance is the need to solve the diamond problem. The diamond problem refers to the inability to decide which implementation of the method to choose if multiple types with the same method declaration has been extended. Scala solves this problem by defining one main super type, whose code will be used whenever the diamond problem occurs. This main one is listed right behind the `extend` expression. Additional types have to be named after the `with` keyword.

Listing 8: Multiple inheritance in Scala

```
  class Circle(override val id: Int, override val description: String, var radius:
Double)
    extends GeometricShape(id, description) with Drawing {
  }

  class GeometricShape(val id: Int, val description: String) {
    def printDescription() {
      println(description)
    }
  }

  trait Drawing {
    var color: Color = Color.BLACK
    var rotation: Double = 0
    var description: String = "Default description"

    def printDescription() {
      println(description)
    }
  }

  object Main {
    def main(args: Array[String]) {
      val circle = new Circle(1, "description", 5.0)
      circle.color = Color.RED
      circle.printDescription()
    }
  }
```

Chapter 5. Polymorphism

In Go polymorphism is only supported via interfaces. Unlike traits in Scala interfaces in Go are implemented implicitly. A type implements an interface if it provides implementations for all the methods declared in the interface. In Go a variable of type interface can hold any value which implements the interface. The main goal of polymorphism is to run different implementations depending on which type was selected. Listing 9: Polymorphism in Go demonstrates both structs `circle` and `rectangle` indirectly implementing `geometricShapeInterface` by defining the `calculateArea()` method. As a result both types can be added to the same array and depending on the type the associated functionality is executed.
A second type of polymorphism in modern-day programming languages is the class-based polymorphism or subclassing in wich inheriting classes can be used instead of their base class. In Go this type isn't possible since Go doesn't offer inheritance.

Listing 9: Polymorphism in Go

```
type geometricShapeInterface interface {
    calculateArea() float64
}

type circle struct {
    radius  float64
}

type rectangle struct {
    length float64
    width float64
}

func (c *circle) calculateArea() float64 {
    area := math.Pi *  math.Pow(c.radius, 2)
    return area
}

func (r *rectangle) calculateArea() float64 {
    area := r.length * r.width
    return area
}

func main() {
    circle := &circle{15}
    rectangle := &rectangle{10,15}
    shapes := []geometricShapeInterface{circle, rectangle}
    var totalArea float64 = 0
    for _, shape := range shapes {
        totalArea += shape.calculateArea()
    }
}
```

In Scala polymorphism can be implemented by using abstract data types. As mentioned before, traits are used to define object types by specifying the signature of the supported methods. In Listing 10: Polymorphism in Scala the trait `GeometricShapeInterface` has been implemented by the two classes `Circle` and `Rectangle`. This way both classes are also of type `GeometricShapeInterface` and can be used whenever a `GeometricShapeInterface` is required.
A second option is the usage of abstract classes, which in fact are very similar to Scala's traits. Both traits and abstract classes can't be instantiated since both contain abstract methods which have to be implented by a concrete type. In contrast to abstract classes traits doesn't allow constructors. So in order to create a base class that requires constructor arguments or if you need your code to be compatible with Java, since traits doesn't exist and aren't allowed in Java, you have to use abstract classes instead of traits.

Listing 10: Polymorphism in Scala

```
trait GeometricShapeInterface {
  def calculateArea(): Double
  def printDescription() {
    println("Default description")
  }
}

class Circle( var radius: Double)
  extends GeometricShapeInterface {
  def calculateArea(): Double = Pi * pow(radius, 2.0)
}

class Rectangle(var length: Double, var width: Double)
extends GeometricShapeInterface {
  def calculateArea(): Double = length * width
}

object Main {
  def main(args: Array[String]) {
    val circle = new Circle(10)
    val rectangle = new Rectangle(10,15)
    val shapes = Array(circle, rectangle)
    var totalArea = 0.0
    for ( shape <- shapes) {
      totalArea += shape.calculateArea()
    }
  }
}
```

Chapter 6. Abstraction

Abstraction is one of the key concepts of object-oriented programming languages, in order to handle complexity by hiding unnecessary functionality from the user. The concept of encapsulation is often used to hide the internal representation, or state, of an object from the outside.

Even though in Go encapsulation can be done via structs, abstraction and access specification is mainly realized by means of packages. In Go, the case of the first letter of variables, structs, fields, functions, etc. determine the access specification. In Go capitalized fields, methods and functions etc. are public and therefore exported by the package. All other fields are local to the package and not exported. There's no protected because there is no inheritance. In Go, when an identifier is exported by a package, then this means that the identifier can be directly accessed by any other package in the code base. When an identifier is unexported by a package, it can't be directly accessed by any other package. Listing 11: Access specification in Go and Listing 12: Access specification in Go (Part 2) demonstrates the characteristics of Go's access specifications.

Listing 11: Access specification in Go

```
package shapes

type geometricShape struct {
    Id int
    Description string
}

type Circle struct {
    geometricShape
    Radius  float64
}
```

Listing 12: Access specification in Go (Part 2)

```
package main

import (
    "./shapes"
)

func main() {

    circle := shapes.Circle{
        Radius: 10,
    }
    circle.Id = 5
    circle.Description = "description of circle"

    println(circle.Description)
}
```

Directly creating the `geometricShape` struct inside the `circle` struct is not possible because `geometricShape` is private to other packages. Nevertheless, directly accessing the exported fields of the `geometricShape` is possible even though they came from an unexported type.

In Scala abstraction and access specification can not just be done on package level but on class level as well. By initialising a type, variable or function etc. as `private` it only can be accessed within the associated class. The access from outside is denied independent of it's enclosing package. In addition to that, top-level classes in Scala can either be defined as `public`, which means exported from the package or `private`. By default top-level classes are only visible for members in the enclosing package. In contrast to Go, in Scala the direct access to puplic class members is denied if the class itself is declared as `private`.

Chapter 7. Object Orientation in Concurrent Programming

In terms of concurrency, Scala provides several strategies for implementing a concurrent system. Apart from thread-based concurrency, derived from Java, Scala offers actor-based concurrency. Regarding to object orientation, the Akka toolkit provides actor functionality by implementing the `Actor` trait within your scala project. As shown in Listing 13: Actor in Scala the concrete actor class `CircleCalculator` has to implement the `receive` functionality in order to specify the concrete behaviour depending on the received message. After processing the message, actors can reply by returning an answer to the sender. In this example every message has been sent asynchronously by using the `!` operator.

Listing 13: Actor in Scala

```
1   case class CalculateArea(radius: Double)
2
3   class CircleCalculator extends Actor {
4
5     def receive= {
6       case CalculateArea(radius) => sender ! Pi * pow(radius, 2.0)
7   }
8   }
9
10  def main(args: Array[String]) {
11
12    val system = ActorSystem("CalculatorActorSystem")
13    val myCircleCalculator = system.actorOf(Props[CircleCalculator], name =
"myCircleCalculator")
14
15    implicit val timeout = Timeout(10 seconds)
16    val future = myCircleCalculator ? CalculateArea(19)
17    val result = Await.result(future, timeout.duration).asInstanceOf[Double]
18    println(result)
19  }
```

With Go similar functionalities can be realized via goroutines and channels. Goroutines are used to run code concurrently with other functions, while channels enables communication between them.

Chapter 8. Conclusion

Based on the findings Go as well as Scala serve several options to accomplish object-oriented programming.

Go isn't a pure object oriented language. Go combines several programming paradigms and is in consequence less object-oriented compared to Scala. Although Go does have encapsulation and type member functions it lacks inheritance and therefore traditional polymorphism, subclassing and type hierarchy. Since there is no type hierarchy, Go provides general and syntacticaly easy to use interfaces. This in combination with the universal concept of defining methods for any sort of type, not beeing restricted to structs, attests Go a more lightweight and general purpose coding style compared to Scala.

In terms of object-orientation Scala on the other hand can be used as an even better object oriented language than Java since everything is an object, even primitive types and functions (first-class functions). In addition to that all static members have been removed. In contrast Scala provides singleton and companion objects to compensate the missing static class members and functions. With Scala and its extensive libraries, there is always more than one way to accomblish a single task due to its diverse object-oriented constructs ranging from trivial classes over traits, objects and abstract classes to extensive collections like lists, maps, tuples and many more.

All in all Go and Scala are mostly compareable in terms of their ability to manage object oriented tasks. The difference lies in the way of accomplishing them. While Scala serves several options in complying with single tasks in Go object oriented functionality is meant to be accomblished mostly in a particular way.

References

1. Docs Scala-Lang. Documentation. https://docs.scala-lang.org/. Retrieved 2018-12-18.
2. A. Alvin. Hello Scala. http://hello-scala.com/. Retrieved 2018-12-20.
3. Nhan. The awesomeness of Scala is implicit. http://technically.us/code/x/the-awesomeness-of-scala-is-implicit/, 2007. Retrieved 2018-12-20.
4. J. Eichar. Companion Object. http://daily-scala.blogspot.com/2009/09/companion-object.html, 2009. Retrieved 2018-12-27.
5. A. Alvin. Simple Scala Akka Actor example. https://alvinalexander.com/scala/simple-scala-akka-actor-examples-hello-world-actors, 2017. Retrieved 2018-01-02.
6. Golang.org. Effective Go. https://golang.org/doc/effective_go.html. Retrieved 2018-12-23.
7. N. Ramanathan. Golangbot. https://golangbot.com/. Retrieved 2018-12-27.