Compare object oriented programming in Go with TypeScript

Ali Piriyaie

Table of Contents

Short introduction

The following pages compare the object-oriented programming between the programming languages Go and TypeScript. This paper begins with a comparison of the type system, followed by a comparison between the class models. With the help of these sections, basic knowledge is built up so that the concepts of inheritance and polymorphism can be explained and compared. At the end a summary is provided.

Let's start with Go's type system.

Type system in Go

In order to build up a basic knowledge and understanding of the coming sections, it is necessary to first consider the type system of Go more closely.

Basic types

Go already has built-in basic types. The basic data types are listed below:

- `string` → Built-in string data type
- `bool` → Build-in bool data type
- `int8`, `uint8`, `byte`, `int16`, `uint16`, `int32`, `rune`, `uint32`, `int64`, `unit64`, `int`, `uint`, `uinptr` → Build-in integer data types
- `float32`, `float64` → Build-in floating-point data types
- `complex64`, `complex128` → Build-in Complex data types

These basic types are not discussed any further, because the author assumes that the reader is familiar with these data types.

Composite types

In addition to the basic data types, Go also has composite data types. The composite data types are listed below:

- `pointer` → Very similar to C pointer
- `struct` → Also very similar to C struct
- `function` → Can be used as static functions or as methods
- `array`, `slice type`, `map types` → Container types
- `channel` → Channel types are used to synchronize data between goroutines
- `interface` → Interface types play a major role in polymorphism

The most important composite data types for this paper are briefly explained below. First, the composite data type pointer is explained, followed by the struct data type and then the interface data type. A corresponding basic knowledge of the function data type is assumed.

Pointer

Figure Simple example for Go pointer shows the use of Go pointer as an example. In line number 2 a new integer variable with the name 'num1' is defined and initialized. In the following line number 3 an integer pointer with the name 'ptrNum1' is also defined and initialized. ptrNum1 is a pointer to the variable num1. As value ptrNum1 contains the address of num1. With the *-operator the pointer can be dereferenced. If you then output the value of *ptrNum1 you get the actual value which the variable num1 contains. This is possible because ptrNum1 refers to num1.

Listing 1. Simple example for Go pointer

```
1 func main() {
2     num1 := 5
3     ptrNum1 := &num1 // memory address
4     fmt.Printf("Value of num1: %v \n", num1) // console output: 'Value of num1: 5'
5     fmt.Printf("Adress of num1: %v \n", ptrNum1)
6     // console output: 'Adress of num1: 0xc00000a090'
7     fmt.Printf("value of num1: %v \n", *ptrNum1)
8     // console output: 'Value of num1: 5'
9
10    *ptrNum1 = 15 // set value of num1 to 15
11    fmt.Printf("Value of num1: %v \n", *ptrNum1)
12    // console output: 'Value of num1: 15'
13 }
```

Struct

With reference to the heart of this essay, it should first be mentioned that there are no classes in the conventional sense in the programming language Go. Instead of classes Go offers the 'struct' composite data type. In this section the topic 'struct' shall be briefly addressed. How to use a 'struct' for object-oriented programming in Go is explained in more detail in section Class model in Go.

Figure Simple example for using Go structs shows a simple example of how to use structs in Go. The composite data type 'dog' is created in line 1 to 4. This contains two fields with primitive data types. The first field is 'name' and the second field is 'breed'. Both are of primitive data type 'string'.

In line 7 and in line 8 two variables of type 'dog' are defined and initialized. After initializing the variables, you can access the fields of the structs using the dot notation, as you can see in lines 10 to 13.

Listing 2. Simple example for using Go structs

```
1 type dog struct {
2     name string
3     breed string
4 }
5
6 func main() {
7     stewie := dog{name: "Stewie", breed: "Australian Shepherd X Swiss"}
8     jax := dog{"Jax", "Australian Shepherd"}
9
10    fmt.Println("First dog's name: " + stewie.name)
11    fmt.Println("Second dog's name: " + jax.name)
12    fmt.Println("First dog's breed: " + stewie.breed)
13    fmt.Println("Second dog's breed: " + jax.breed)
14 }
```

You can also use point notation on struct pointers. For this dereferencing of the pointer variables is not necessary. Furthermore, the composite data type struct can contain another composite data type as a field. This allows an arbitrarily deep encapsulation.

Interface

Figure Simple example for using Go interfaces represents a simple example of how to use Go interfaces. An interface with the name 'pet' is added to the previous example. This is done in lines 6 to 9.

Listing 3. Simple example for using Go interfaces

```
type dog struct {
    name string
    breed string
}

type pet interface {
    printName()
    printBreed()
}

func (d dog) printName(){
    fmt.Println("Dog's name: " + d.name)
}

func (d dog) printBreed(){
    fmt.Println("Dog's breed: " + d.breed)
}

func printDogInformation(p pet){
    p.printName()
    p.printBreed()
}

func main(){
    stewie := dog{name: "Stewie", breed: "Australian Shepherd X Swiss"}
    printDogInformation(stewie)
}
```

An interface is a defined composite data type of method declarations. To implement an interface in Go, it is sufficient to implement the methods declared in the interface on a struct. The function 'printDogInformation(p pet)' expects a struct which implements the interface 'pet'. Within the function the implemented methods are then called.

So you can say that an interface is a kind of 'promise' to the caller that the methods declared in the interface have been implemented.

We've now learned the basic primitive data types and composite data types in the Go programming language. Now let's look at the same concepts in TypeScript. The main differences and how the concepts are solved in TypeScript will be explained in section Type system in TypeScript.

Type system in TypeScript

TypeScript is a programming language developed by Microsoft that upgrades JavaScript for complex applications. It enables object-oriented programming in the conventional sense. Technically, TypeScript code is translated into JavaScript code using the TypeScript compiler.

In addition, TypeScript is a statically typed programming language. This means that the type of a

variable is known at translation time. In the following subsection we look at the basic data types.

Basic types

TypeScript also offers a number of primitive data types. These are:

- `boolean` → Also exists in Go. The difference here is that this data type in Go has the name `bool`.
- `number` → Compared to Go, TypeScript has only this data type for numbers. It is always a floating point number.
- `string` → As in Go, TypeScript has the data type `string`. In TypeScript, however, it is possible to use both single quotes and double quotes for surrounding strings.

Beside the data type `string` exists the data type template-`string`. This allows multiline texts and embedded expressions. To use tamplate-strings the string must be surrounded by backticks. Easy to pronounce and hard to find on the keyboard. Expressions must have this form: `${ expression }`. A simple example of a tamplate-string is shown in Figure Template-`string` example.

Listing 4. Template-`string` example

```
1 var x = 3.1415;
2 var tempString = `This is just a multi-line test.
3
4 The value of the variable x is: ${ x }`;
5
6 console.log(tempString);
```

In line one the variable 'x' of type number is declared and initialized with the value 3.1415. In line two the declaration and initialization of the template-`string` variables takes place. The value of the variable 'tempString' extends over several lines and contains the value of the variable 'x' as an expression.

To write multiline text with expressions in Go, there are similar constructs. However, these are not further explained here.

Even the basic data types that come with JavaScript can be used in TypeScript. These are additionally:

- `string` → As in TypeScript, JavaScript has the data type `string`.
- `number` → As in TypeScript, JavaScript has the data type `number`.
- `boolean` → As in TypeScript, JavaScript has the data type `boolean`.
- `null` → That's a null value.
- `undefined` → Indicates that no value has been assigned. There is something similar in Go called `nil`.

Next, the composite data types are described and compared with the composite data types in Go.

Composite types

In addition to the basic data types, TypeScript also has composite data types. These are:

- `array` → Container type - Similar to Go arrays
- `tuple` → This is a list of a finite number of objects that do not necessarily have to differ from each other. Tuples have a similarity with the datatype `struct` in Go.
- `enum` → It's an enumeration. We can achieve the same result in Go using the `const` datatype.
- `any` → You can use `any` if you don't know the type of variable when writing the code. It is similar to the assignment `:=` in Go.
- `object` → Represents a composite data type. `object` is similar to the datatype `struct` in Go.
- `function` → Also exists in Go.
- `interface` → A kind of contract or promise in the code. Details follow in section Interface.

In addition, the composite data types of JavaScript can be used in TypeScript. These are:

- `object` → As in TypeScript, JavaScript has the data type `object`.
- `function` → As in TypeScript, JavaScript has the data type `function`.
- `array` → As in TypeScript, JavaScript has the data type `array`.

Interface

Figure Interface example in TypeScript shows a simple example interface for illustration. Compared to Go, you can also specify attributes in an interface in TypeScript. In addition, the class that implements the interface must also implement the attributes and methods.

In Go, an interface is not implemented until all methods for the `struct` have been implemented. If the programmer does not implement all the necessary methods he receive an error message. However, the error is only output if the composite data type is assigned to a variable that expects the implementation of a particular interface. In TypeScript, the TypeScript compiler prevents compilation if the interface is not implemented correctly.

Listing 5. Interface example in TypeScript

```
1 interface pet {
2     name: string;
3     breed: string;
4     printName(): any;
5     printBreed(): any;
6 }
```

In general, it can be said that the composite data types in TypeScript and Go are very similar to each other. There are different names like `enum` in TypeScript and `const` in Go. But the concepts are still very similar. The data type `pointer` from Go can also be programmed using 'getter' and 'setter' in TypeScript.

Now the most important points in the type system of Go and TypeScript have been described. The next section compares the class model in Go with the class model in TypeScript.

Class model in Go

In Go, the composite data type `struct` is used to define a class. The datatype `struct` has already been introduced in section Type system in Go. With a `struct` you can easily implement object-oriented paradigms like inheritance and polymorphism.

At the beginning we will describe how to add attributes to a class. The author then discusses the methods of a class. At the end of this section there is a description of how to work with instances of classes in Go. The example 'dog' already used in the previous sections will be further expanded here.

Attributes in classes

In Go, a class is defined by the data type `struct`. The fields within a `struct` are then the attributes of the class. These fields can be primitive data types as already described in Type system in Go or even of the data type `struct`. In our example 'dog' the class contains two attributes of type `string` and one attribute of type `float32`. Figure Class example in Go shows an example of a simple class in Go.

Listing 6. Class example in Go

```
1 type dog struct {
2     name string
3     breed string
4     weight float32
5 }
```

The attributes of the class 'dog' are declared in lines 1 to 5. You have the possibility to control the visibility of attributes. This is not done as usual with keywords like `private` and `public`, but with upper and lower case. If the first letter of the name of an attribute is capitalized, it will be accessible from the outside. If, however, the first letter of the name of an attribute is written in small letters, it will not be accessible from the outside. In our example, neither the object 'dog' nor the attributes of the object are accessible from outside. 'From the outside' in this case means that another Go file imports our example file and tries to access elements within the file via the point operator.

How to initialize, access and modify the attributes will be described in section Working with instances. The next section deals with methods of a class.

Methods in classes

Class methods are a very important part of object-oriented programming. Figure Method example in Go shows two methods in Go. The first method in lines 1 through 3 works on a copy of the instance. The second method in lines 5 to 11 works on the direct reference of the instance. The method header makes the difference. If the calling type, in this case our object 'dog', uses an star

operator ('*'), it is a direct reference to the calling instance. If the star operator is not used, a copy of the instance is passed.

Listing 7. Method example in Go

```
1 func (d dog) bark() {
2     fmt.Println("Dog " + d.name + " barks")
3 }
4
5 func (d *dog) addAge() {
6     d.age++
7     fmt.Printf(
8         "Hurray! It`s %s's birthday. He/She is %v years old now.",
9         d.name,
10        d.age)
11 }
```

The advantage is very easy to see here. In the case of the 'bark' method, the caller can be sure that the instance will not be changed by the called method. The second case 'addAge' should only be used if you really want to change the state of the instance.

Now methods of classes have been described. Let's look at how to work with instances in the next section.

Working with instances

To work with instances, they must first be created. In figure Example of how to work with instances in Go the example 'dog' was extended by a method named 'NewDog' (lines 9 to 16). Since the 'NewDog' method is capitalized, it is accessible to third parties from the outside. The task of this method is to create a new instance of the object 'dog', initialize the attributes and return them to the caller. In line 19 there is an exemplary call of the method 'NewDog'. Since we are in the same .go file as the object 'dog', the attributes can also be changed directly with the point operator. In line 33 the value of the attribute 'name' of the instance 'digga' is changed from 'Diga' to 'Digga' for presentation purposes.

Listing 8. Example of how to work with instances in Go

```
1 type dog struct {
2     name string
3     breed string
4     weight float32
5     age uint8
6 }
7
8 // NewDog returns an instance of a dog object
9 func NewDog(name string, breed string, weight float32, age uint8) *dog {
10     dog := new(dog)
11     dog.name = name
12     dog.breed = breed
13     dog.weight = weight
14     dog.age = age
15     return dog
16 }
17
18 func main() {
19     digga := NewDog("Diga", "German Shepherd", 44.6, 4)
20     stewie := dog{
21         name: "Stewie",
22         breed: "Australian Shepherd X Swiss Shepherd",
23         weight: 29.5,
24         age: 3
25     }
26     jax := dog{
27         "Jax",
28         "Australian Shepherd",
29         24.4,
30         1
31     }
32
33     digga.name = "Digga"
34 }
```

You can also create instances using the syntax as in lines 20 to 25. However, this is only possible if the struct-type is accessible. In this case it is accessible because we are in the same .go file as mentioned before. This syntax initializes the attributes within the curly brackets. The assignment is done using the names of the attributes. Alternatively, you can omit the names of the attributes, as shown in lines 26 to 31, if you respect the order of the attributes in the struct. However, this type of instantiation is not recommended, since it can become very confusing.

Now the class model has been described in Go. It was shown how to create objects in Go, methods are defined for these objects, and instances of these objects are used. In the next section, let's look at the class model in TypeScript and compare it to the class model in Go.

Class model in TypeScript

In TypeScript you can use classes in the 'conventional' sense, as you know it from Java or C#, for example. A `struct` like in Go is not necessary. In this section we look at how to implement classes in TypeScript, define attributes and methods on classes, and compare them with the class model in Go. At the end, working with TypeScript classes is considered and compared with Go.

Here we take over our example 'dog' which has already been used several times.

Attributes in classes

Figure Example of a class in TypeScript represents the class 'dog' written in TypeScript. To define a class in TypeScript, first use the keyword `class` followed by the class name. The class name should be capitalized. Within the class block you can then define the attributes. Compared to Go, the visibility is not case sensitive, but you can use the keyword `private`, `public` or `protected`. In our example all attributes are `private` and therefore not accessible from the outside for third parties.

Listing 9. Example of a class in TypeScript

```
1 class Dog {
2     private name: string;
3     private breed: string;
4     private weight: number;
5     private age: number;
6 }
```

Already here you can see clear differences in visibility and in the keywords in a comparison with Go. In the next section we will look at how methods for a class are implemented in TypeScript and compare them with Go.

Methods in classes

Figure Example of class methods in TypeScript displays the methods 'bark' and 'addAge' already presented in Go now in TypeScript. Unlike Go, in TypeScript you can implement methods directly within the class. You can also use the 'this' operator to access the current instance and initialize, read, or change attribute values. For visibility you have to use the keywords `private`, `public` and `protected` just like with the attributes in TypeScript and you don’t have to be case-sensitive.

Listing 10. Example of class methods in TypeScript

```
class Dog {
    // Attributes

    public bark(){
        console.log("Dog " + this.name + " barks");
    }

    public addAge(){
        this.age++;
        console.log(
            "Hurray! It`s " +
            this.name +
            "'s birthday. He/She is " +
            this.age +
            " years old now.");
    }
}
```

In TypeScript you don't have to specify the return type of a method, because TypeScript recognizes it from the return value itself. In comparison, you have to specify the return type in Go if a method has a return value.

Now it was described how the class model looks in TypeScript, how to define attributes for classes and how class methods look in TypeScript. In the next section, let's look at how to create and work with instances in TypeScript.

Working with instances

In this section, we will look at how to generate and edit instances in TypeScript.

In TypeScript there is a special method called 'constructor' (see figure Example of a cunstructor in TypeScript) besides the 'normal' methods. This method is called automatically when creating an instance. There is no such method in Go. You can create a constructor in Go itself, but it's up to the developer. It does not require a fixed name like TypeScript, such as 'constructor'.

Listing 11. Example of a cunstructor in TypeScript

```
constructor(name: string, breed: string, weight: number, age: number) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.age = age;
}
```

Figure Example of how to work with instances in TypeScript shows in line 1 the creation of an instance of type 'Dog'. The 'constructor' method of the 'Dog' class is called implicitly in this line. In line 2, the 'bark' method is called using the point operator.

Listing 12. Example of how to work with instances in TypeScript

```
1 var jax = new Dog("Jax", "Australian Shepherd", 24.6, 1);
2 jax.bark();
```

In this example you can see that the constructor of the 'Dog' class is implicitly called when creating an instance with the 'new' operator. In Go you have to call a self-developed 'NewYourClassName' method, which represents the constructor, directly.

Now the class model in TypeScript was compared with the class model in Go. Attributes within classes and methods of classes were described and also compared with the implementation in Go. Next, we take a closer look at inheritance in Go and TypeScript.

Inheritance in Go

In this section the inheritance in Go shall now be shown. The example 'dog' from the previous sections will still be used and extended.

Figure Example of inheritance in Go shows an example of inheritance in Go. In Go, inheritance takes place by specifying the name of the superclass or superstructs in the subclass as an attribute. Here the superclass 'animal' is defined in lines 1 to 4. The structure 'dog' inherits all attributes and methods of 'animal'. This happens in line 7.

Listing 13. Example of inheritance in Go

```
 1 type animal struct {
 2     kind string
 3     gender string
 4 }
 5
 6 type dog struct {
 7     animal
 8     name string
 9     breed string
10     weight float32
11     age uint8
12 }
```

If you want to create a new instance of the struct 'dog' you have to instantiate the superclass, too. This could look like in the figure Example of creating an instance which inherits from a superclass.

Listing 14. Example of creating an instance which inherits from a superclass

```
1 jax := dog{
2     animal {
3         "Dog",
4         "male"
5     },
6     "Jax",
7     "Australian Shepherd",
8     24.4,
9     1
10 }
```

There are other ways to create instances. These are not all listed here, as this would go beyond the scope of this work.

When accessing the inherited attributes or methods, you can simply use the point operator.

Now it has been described how the inheritance in Go looks like. Next, let's look at what inheritance looks like in TypeScript.

Inheritance in TypeScript

This section describes the inheritance of TypeScript and compares it with the inheritance in Go.

Figure Example of inheritance in TypeScript shows an example of inheritance in TypeScript. In TypeScript, the subclass must use the keyword `extends` followed by the class that is inherited. In the example, this is done in line 11. All methods and attributes in the superclass are then inherited by the subclass. The same is true in Go, but there with structs and not with classes. The most important difference is the constructor in TypeScript. If a class inherits from a superclass, it must then call the constructor of the superclass in its own constructor. This happens in our example in line 22.

Listing 15. Example of inheritance in TypeScript

```
class Animal {
    private kind: string;
    private gender: string;

    constructor(kind: string, gender: string) {
        this.kind = kind;
        this.gender = gender;
    }
}

class Dog extends Animal {
    private name: string;
    private breed: string;
    private weight: number;
    private age: number;

    constructor(gender: string,
        name: string,
        breed: string,
        weight: number,
        age: number) {
            super("Dog", gender);
            this.name = name;
            this.breed = breed;
            this.weight = weight;
            this.age = age;
    }
}
```

In comnparison Go does request an instance of the superclass when instantiating the subclass. You can also program a constructor yourself in Go and create the instance of the superclass or superstructure in this constructor. So it's somehow similar to a certain extent and still not similar.

Now that the class concept has been presented, the next thing we look at is how polymorphism works in Go and compare it with the TypeScript implementation.

Polymorphism in Go

This section describes how the concept of polymorphism is implemented in Go. The previous example 'dog' is also used and extended in this example. Important here is an understanding of Go interfaces, which has already been explained in section Interface.

Polymorphism stands for versatility and is an important concept of object-oriented programming. This assigns more than one type to an identifier.

Figure Example for polymorphism in Go shows as an example the implementation of polymorphism in Go. Here the interface 'moveable' is defined in lines 1 to 3. The implementation of

this interface requires the implementation of the 'walk' method. As already explained in Interface, a struct implements an interface as soon as all methods from the interface have been implemented on the struct. The implementation then takes place implicitly. In lines 10 to 12 the method 'walk' is implemented for the struct 'animal' and in lines 22 to 24 for the struct 'dog'. In lines 26 to 28 a function is defined, which requires a data type, which implements the 'movable' interface. Within the function 'foo' the method 'walk' can then be accessed, because the implementation of an interface in Go is similar to a promise that the methods listed in the interface will be implemented on the struct. Since the method 'foo' here assumes both the data type 'animal' and the data type 'dog', this is polymorphism.

Listing 16. Example for polymorphism in Go

```
type movable interface {
    walk()
}

type animal struct {
    kind string
    gender string
}

func (a animal) walk(){
    fmt.Println("Animal walks.");
}

type dog struct {
    animal
    name string
    breed string
    weight float32
    age uint8
}

func (d dog) walk(){
    fmt.Printf("Dog %s walks. \n", d.name)
}

func foo(m movable){
    m.walk()
}

func main(){
    digga := dog{
        animal: animal{kind: "Dog", gender: "male"},
        name: "Digga",
        breed: "German Shepherd",
        weight: 44.3,
        age: 4}
    elefant := animal{kind: "Elefant", gender: "female"}

    foo(digga)
    foo(elefant)
}
```

Now polymorphism was presented and explained in Go using a clear example. Let's look at polymorphism in TypeScript and compare it to the implementation in Go.

Polymorphism in TypeScript

In this section, the previous example from Go will now also be implemented in TypeScript. It can be confirmed at the beginning that the example of polymorphism in Go can also be implemented in TypeScript without any problems.

Let us now look at Figure Example for polymorphism in TypeScript. The interface 'movable' is defined in lines 38 to 40. Then the classes 'Dog' and 'Animal' implement the 'movable' interface. This is a promise that the 'walk' method has been implemented. The function 'foo' relies on this promise, which only accepts arguments as parameters, which implement the 'movable' interface. Then the function 'foo' calls the 'walk' method. The test calls of the function 'foo' take place in lines 56 and 57.

Apart from the already known differences to inheritance and interfaces in Go, there are hardly any other differences in the concept of polymorphism.

Listing 17. Example for polymorphism in TypeScript

```
1  class Animal implements moveable {
2      private kind: string;
3      private gender: string;
4
5      public walk(): any{
6          console.log("Animal walks.");
7      }
8
9      constructor(kind: string, gender: string) {
10         this.kind = kind;
11         this.gender = gender;
12     }
13 }
14
15 class Dog extends Animal implements moveable {
16     private name: string;
17     private breed: string;
18     private weight: number;
19     private age: number;
20
21     public walk(): any{
22         console.log("Dog " + this.name + " walks.");
23     }
24
25     constructor(gender: string,
26         name: string,
27         breed: string,
28         weight: number,
29         age: number) {
30             super("Dog", gender);
31             this.name = name;
32             this.breed = breed;
33             this.weight = weight;
```

```
            this.age = age;
    }
}

interface moveable {
    walk(): any;
}

function foo(m: moveable){
    m.walk();
}

var digga = new Dog(
        "male",
        "Digga",
        "German Shepherd",
        44.3,
        4
    );

var elefant = new Animal("Elefant", "female");

foo(digga);
foo(elefant);
```

Now the object-oriented concepts of Go have been compared to those in TypeScript. The next section provides a summary and the author's own opinion.

Summary

After a short introduction the type system of Go was described and a distinction was made between basic data types and composite data types. Subsequently, the type system of TypeScript was described and compared with the type system of Go. It was shown that TypeScript has less basic data types, especially the numeric values. The type system of JavaScript was also explained, because TypeScript is translated into JavaScript code by using the TypeScript compiler. In the following section, the Go class model was described, followed directly by the TypeScript class model. These two were then compared. Using the knowledge of these chapters, it was then possible to compare the inheritance of Go and TypeScript with each other, and to explain and compare the concept of polymorphism in both Go and TypeScript.

If you ask me as a programmer with C# as my favorite language, which programming language would I choose if I had to choose between TypeScript or Go? My choice would have clearly fallen on TypeScript. In my opinion the implementation of object-oriented programming using structs is very nice, and the way in which inheritance and the use of interfaces work is impressive. But for me, the 'classic' object-oriented programming with real classes is just a tick more beautiful, as it was implemented in TypeScript. In spite of all this I would like to say that the developers of Go have done a very good job and I am very happy to have dealt with this topic.

Many thanks to Mr. Weigend who introduced me to the programming language Go last semester.

Ali Piriyaie Master Student in Computer Science at the University of Applied Sciences in Rosenheim.

Reference

Internetsites:

- Overview of Go Type System - access on: 02.12.2018 - https://go101.org/article/type-system-overview.html
- Go by Example: Structs - access on: 13.12.2018 - https://gobyexample.com/structs
- Go by Example: Interfaces - access on: 16.12.2018 - https://gobyexample.com/interfaces
- Drei Gründe warum TypeScript das bessere JavaScript ist - access on: 16.12.2018 - https://www.communardo.de/techblog/drei-gruende-warum-typescript-das-bessere-javascript-ist/
- TypeScript hat gute Chancen auf eine Verbreitung auch außerhalb des Microsoft-Kosmos - access on: 16.12.2018 - https://entwickler.de/online/typescript-hat-gute-chancen-auf-eine-verbreitung-auch-ausserhalb-des-microsoft-kosmos-157424.html
- Basic Types TypeScript - access on: 18.12.2018 - https://www.typescriptlang.org/docs/handbook/basic-types.html
- Tuple - Wikipedia - access on: 22.12.2018 - https://de.wikipedia.org/wiki/Tupel
- Private and Public Visibility with Go Packages - Will Haley - access on: 25.12.2018 - https://willhaley.com/blog/private-and-public-visibility-with-go-packages/
- Constructors - Go Language Patterns - access on: 04.01.2019 - http://www.golangpatterns.info/object-oriented/constructors