Object-Oriented Programming An example in Kotlin and the comparison to Go

Tobias Kirschner, Master Informatik, TH Rosenheim, 12.2022

Table of Contents

1 Introduction

In the course of the term paper a code project was created to demonstrate the basic aspects of Object-Oriented Programming (OOP). The project defines a banking system that manages bank customers and their bank accounts. The implementation was done in the programming language Kotlin which is well suited for OOP.

In the following, the most important and fundamental aspects of OOP are shown and explained on the basis of the implemented banking system. Followed by a comparison with the programming language Go it is analyzed how the treated aspect would be implemented in Go. Finally, possible advantages and disadvantages are discussed.

2 Structure of Object-Oriented Programming

In OOP the structure of a program is based on the concept of "objects", which contain data and code that manipulates that data. OOP has four main elements: classes, objects, attributes, and methods. [sb]

- Classes: A class is a blueprint or template for creating objects. It defines the attributes (data) and methods (code) that make up an object.
- Objects: An object is an instance of a class. It contains the actual data and code for a particular instance of the class.
- Attributes: Attributes are the data or state of an object. They are also known as instance variables or fields.
- Methods: Methods are the code or behavior of an object. They define the actions that an object can perform and the way it manipulates its data.

Referring to the bank system, classes are for example "Bank", "Customer" and "Account". Adding a customer to the bank creates an object of the "Customer" class. This customer is described by attributes, such as "customerId" or "accounts". Methods like "getId()" or "addAccount()" can describe certain actions of this object and can manipulate its data.

3 Principles of Object-Oriented Programming

OOP has several key aspects which are going to be explained by the given bank system example.

3.1 Abstraction

Abstraction is a paradigm of OOP that involves exposing only the necessary information to the user, while hiding unnecessary details. In general an abstract class specifies what its subclasses must be able to do. However, it does not necessarily prescribe how the methods must be implemented concretely. [sb]

3.1.1 Abstraction in Kotlin project

In the code project abstraction has been implemented as follows:

```
abstract class Account(var balance: Double) : Printable {
    protected var accountId: Int = 0

    init {
        accountId = idCounter
        idCounter++
    }

    companion object {
        var idCounter = 1
    }

    abstract fun deposit(amount: Double)
    abstract fun withdraw(amount: Double)

    fun getId(): Int {
        return this.accountId
    }
}
```

In Kotlin an abstract class is defined by the key word "abstract" in front of the "class" declaration. In this example, the "Account" class is an abstract class that defines the basic structure for an "Account" object. It has two abstract methods, "deposit" and "withdraw", which must be implemented by any subclass. The "CheckingAccount" and "SavingsAccount" classes are concrete subclasses of "Account" that implement the abstract methods to manipulate the balance of an account like shown in the following excerpts of the subclasses:

```
class CheckingAccount(balance: Double) : Account(balance) {

    override fun deposit(amount: Double) {
        balance += amount
    }

    override fun withdraw(amount: Double) {
        balance -= amount
    }
}
```

```
class SavingsAccount(balance: Double) : Account(balance) {

    override fun deposit(amount: Double) {
        balance += amount
    }

    override fun withdraw(amount: Double) {
        if (amount > balance) {
            throw IllegalArgumentException()
        }
        balance -= amount
    }
}
```

As shown an abstract class specifies what the subclasses must be able to do. However, it does not necessarily prescribe how the methods must be implemented concretely. With the key word "override" the compiler is informed that the concrete implementation of the method in the Subclass follows.

3.1.2 Comparison to Go

In Go, there is no concept of abstract classes. Go does not have inheritance or subclassing, and it does not have a way to declare abstract methods or abstract classes.

However, Go does have a way to define interfaces, which provide a way to specify a set of methods that a type must implement. An interface can be thought of as a kind of abstract type, in that it defines a set of methods that must be implemented, but does not provide any implementation for those methods. [ms]

Referring to the code project you could define an interface like this:

```
type Account interface {
    Deposit(amount float64)
    Withdraw(amount float64)
}
```

This interface defines two methods: "Deposit" and Withdraw. Any type that implements these methods can be used as an "Account". For example, you could define a "CheckingAccount" type like this:

```
type CheckingAccount struct {
    balance  float64
}

func (c *CheckingAccount) Deposit(amount float64) float64 {
    return balance += amount
}

func (c *CheckingAccount) Whithdraw(amount float64) float64 {
    return balance -= amount
}
```

As soon as the "CheckingAccount" type implements the "Deposit" and "Withdraw" methods it can be used as an "Account". You can use interfaces in Go to achieve similar behavior to abstract classes in other languages, but as shown the way they are implemented is quite different.

3.2 Inheritance

In OOP inheritance is a mechanism that allows one class to inherit properties and behaviors from another class. Inheritance is a way to create a new class that is a modified version of an existing class. The new class is called the subclass or derived class, and the existing class is the superclass or base class. [wp]

3.2.1 Inheritance in Kotlin project

In the bank system the abstract class "Account" is also the superclass of "CheckingAccount" and "SavingsAccount" which inherit the "balance" property and the referring methods. In Kotlin a subclass is defined by adding the desired superclass with a colon behind the class declaration:

```
class CheckingAccount(balance: Double) : Account(balance) {
    [some code]
}
```

Thus, the "CheckingAccount" has access to the "balance" property which is defined in its superclass "Account" and the there declared methods "deposit" and "withdraw" as shown in topic 3.1.1.

3.2.2 Comparison to Go

Go does not have inheritance or subclassing, so it does not have a way to create a class that inherits properties and behaviors from another class. Go does not have a keyword for declaring classes, and it does not have a way to specify a superclass for a class.

Instead of inheritance Go provides other mechanisms for code reuse, such as type embedding. Type embedding is a technique for creating new types by combining other types. You can use it to create a new type that has the properties and behaviors of multiple other types. [ms]

```go
type Account struct {
    balance float64
}

func (a *Account) Withdraw(amount float64) error {
    a.balance -= amount
    return nil
}

type SavingsAccount struct {
    Account
    [other properties of "SavingsAccount"]
}

func (s *SavingsAccount) Withdraw(amount float64) error {
    if s.Account.balance < amount {
        return fmt.Errorf("Insufficient funds")
    }
    s.Account.balance -= amount
    return nil
}
```

In this example, the "Account" struct contains the fields and methods that define the behavior of an "Account". The "SavingsAccount" struct contains an embedded field of type "Account" and additionally other fields. The "SavingsAccount" struct also has its own implementation of the "Withdraw" method, which overrides the method inherited from the "Account" struct.

You can now create an instance of the "SavingsAccount" struct like this:

```go
s := SavingsAccount{
    Account: Account{
        balance: 50.0,
    },
    [additional fields],
}
```

You can then call the "Withdraw" method on the "SavingsAccount" instance:

```go
s.Withdraw(20.0)    // Output: 30.0
```

Using type embedding it is possible to achieve similar behaviors to inheritance in Go, but the way they are implemented is quite different from inheritance in other languages.

3.3 Encapsulation

Encapsulation is a fundamental concept in OOP. It refers to the bundling of data with the methods that operate on that data or the bundling of related properties and behaviors into a single unit or object. Encapsulation allows a class to hide its internal implementation details from other classes, so that the internal workings of the class are not exposed to the outside world. This promotes modularity and makes it easier to maintain and modify the code, because the internal implementation can be changed without affecting the rest of the system.

Encapsulation also helps to protect the integrity of the data stored in an object, by preventing other classes or code from directly accessing or modifying the data. Instead, other classes or code must use the methods provided by the object to access or modify the data, which gives the object control over how its data is used and manipulated. [sa]

3.3.1 Encapsulation in Kotlin project

In Kotlin, encapsulation can be achieved through the use of visibility modifiers, which control the visibility and accessibility of class members (i.e. properties and functions). The four visibility modifiers in Kotlin are "private", "protected", "internal", and "public". Adding no modifier in front of a field or a method makes it "public" by default. The following code excerpt shows one way to use encapsulation in Kotlin:

```
class Bank {
    private val customers: MutableList<Customer> = mutableListOf()  // --> private
field

    fun addCustomer(customer: Customer) {   // --> public function
        customers.add(customer)
    }
    [other code]
}
```

In this example the "Bank" class has a private list called "customers" which contains a banks customers. By using the "private" visibility modifier, the "customers" list is only visible and accessible within the "Bank" class. The class also has functions like "addCustomer". This function is defined as "public". Other classes and code can use the "Bank" object and call its function to add customers, but it cannot directly access or modify the "customers" property. This helps to ensure the integrity of the data and the proper functioning of the object.

3.3.2 Comparison to Go

In Go, encapsulation is achieved through the use of capitalization of names. If a name starts with a capital letter, it is considered public and can be accessed from outside the package it is defined in. If a name starts with a lowercase letter, it is considered private and can only be accessed within the package it is defined in. [ms] Here is an example of a simple struct in Go that demonstrates encapsulation:

```
type Bank struct {
    customers []*Customer
}

func (b *Bank) AddCustomer(c *Customer) {
    b.customers = append(b.customers, c)
}
```

The visibility in this Go example is the same as in the Kotlin code excerpt above. There is just a small difference in the implementation.

3.4 Polymorphism

Polymorphism refers to the ability of different objects to respond to the same message or method call in different ways. This allows for a flexible and extensible design where objects can be treated uniformly even if they have different underlying implementations or behaviors. [sb]

There are two main types of polymorphism in OOP:

- Dynamic polymorphism, also known as runtime polymorphism or late binding, occurs when a method call is resolved at runtime based on the actual type of the object being called. This allows for a single method to have multiple implementations depending on the type of the object.
- Static polymorphism, also known as compile-time polymorphism or early binding, occurs when a method call is resolved at compile-time based on the static type of the object being called. This allows overloading a method, i.e. having multiple methods with the same name but different signatures or parameter lists. [cw]

3.4.1 Polymorphism in Kotlin project

In the bank system code project polymorphism is supported by an interface called "Printable". It defines an abstract "print()" method. The "Customer" and "Account" classes implement this interface. While "Customer" directly contains a concrete implementation of the method, the interface is inherited by the abstract class "Account" to its subclasses "SavingsAccount" and "CheckingAccount", which can now provide their own implementations of "print()". In kotlin an interface is defined by the key word "interface" in front of the interface name:

```
interface Printable {
    fun print()
}
```

Classes can implement the interface by adding a colon followed by the interface name behind the class name:

```
class Customer(private val firstName: String, private val lastName: String) :
Printable {
    private val accounts: MutableList<Account> = mutableListOf()

    override fun print() {
        println("Customer [ID: $customerId]:\n$firstName $lastName\n")
        accounts.forEach { it.print() }
    }
    [some code]
}
```

As shown in the code excerpt the "print()" implementation of "Customer" calls the "print()" method of "Account". Account also implements the "Printable" interface which makes it possible to print every account in the "accounts" list no matter how the concrete "print()" implementation of an "SavingsAccount" or "CheckingAccount" looks like.

3.4.2 Comparison to Go

Go also supports polymorphism for example by using interfaces. In Go an interface is defined by adding the key word "interface" behind the interface name. An interface in Go is a set of methods that a type must implement in order to implement the interface. This allows defining a common set of behaviors that can be shared among different types. [ms] Referring to the bank system polymorphism supported by an interface in Go could look like this:

```
type Printer interface {
    Print() string
}

type Customer struct{}

func (c Customer) Print() string {
    [implementation of the Customer Print() function]
}

type CheckingAccount struct{}

func (c CheckingAccount) Print() string {
    [implementation of the CheckingAccount Print() function]
}

type SavingsAccount struct{}

func (s SavingsAccount) Print() string {
    [implementation of the SavingsAccount Print() function]
}
```

Here, the "Printer" interface defines a "Print()" method, which the "Customer", "CheckingAccount" and "SavingsAccount" types implement. This allows to use a "Printer" variable to refer to either a "Customer", "CheckingAccount" or a "SavingsAccount" object, and call the "Print()" method on it. Using the interface could look like this:

```
func main() {
    var printer Printer

    customer := Customer{}
    printer = customer
    fmt.Println(printer.Print()) // Output: [output of the Customer Print() function]

    checkingAccount := CheckingAccount{}
    printer = checkingAccount
    fmt.Println(printer.Print()) // Output: [output of the CheckingAccount Print()
function]

    savingsAccount := SavingsAccount{}
    printer = savingsAccount
    fmt.Println(printer.Print()) // Output: [output of the SavingsAccount Print()
function]
}
```

This code creates a variable called "printer" of type "Printer", and then assigns it a reference to each of the three types that implement "Printer". When the "Print()" method is called on "printer", the version of the method defined in the actual type of the object being referred to is called (either "Customer.Print()", "CheckingAccount.Print()", or "SavingsAccount.Print()"). This is an example of polymorphism because the "printer" variable can take on multiple forms (i.e., refer to objects of different types) and exhibit different behaviors (i.e., call different versions of the "Print()" method) depending on the actual type of the object it refers to.

4 Conclusion

With the code project in Kotlin all important concepts of OOP could be demonstrated. Kotlin supports all principles of OOP while in Go sometimes a few detours have to be made to achieve similar behavior. Go supports encapsulation via an OOP style syntax. Polymorphism in Go only can be achieved by using interfaces. In Go there is no concept of abstract classes like in Kotlin. However, in Go also interfaces can be used to achieve similar behavior of abstraction in other languages. At least Go does not support inheritance, but it provides other mechanisms such as type embedding which makes reuse of code possible. Overall, Kotlin and Go are both Object-Oriented languages, but they have some differences in their approach to OOP. Both languages offer a powerful and flexible way to structure and organize code and which one to choose will depend on the needs and preferences.

References

- [sb] Stefik, M., Bobrow, D. G. (1985): Object-Oriented Programming: Themes and Variations. AI Magazine, Vol. 6, No. 4.
- [wp] Wegner, P. (1990): Concepts and Paradigms of Object-Oriented Programming. OOPSLA-89.
- [sa] Snyder, A. (1986): Encapsulation and Inheritance in Object-Oriented Programming Languages. OOPSLA-86.
- [cw] Cardelli, L., Wegner, P. (1985): On Understanding Types, Data Abstraction, and Poymorphism. Computing Surveys, Vol. 17, No. 4.
- [ms] Macke, S. (2022): Go Programming - OOP: Concepts of Programming Languages [Lecture slides]. GitHub. https://s-macke.github.io/concepts-of-programming-languages/docs/03-Go-Programming-OOP.html#1