Object-Oriented Programming

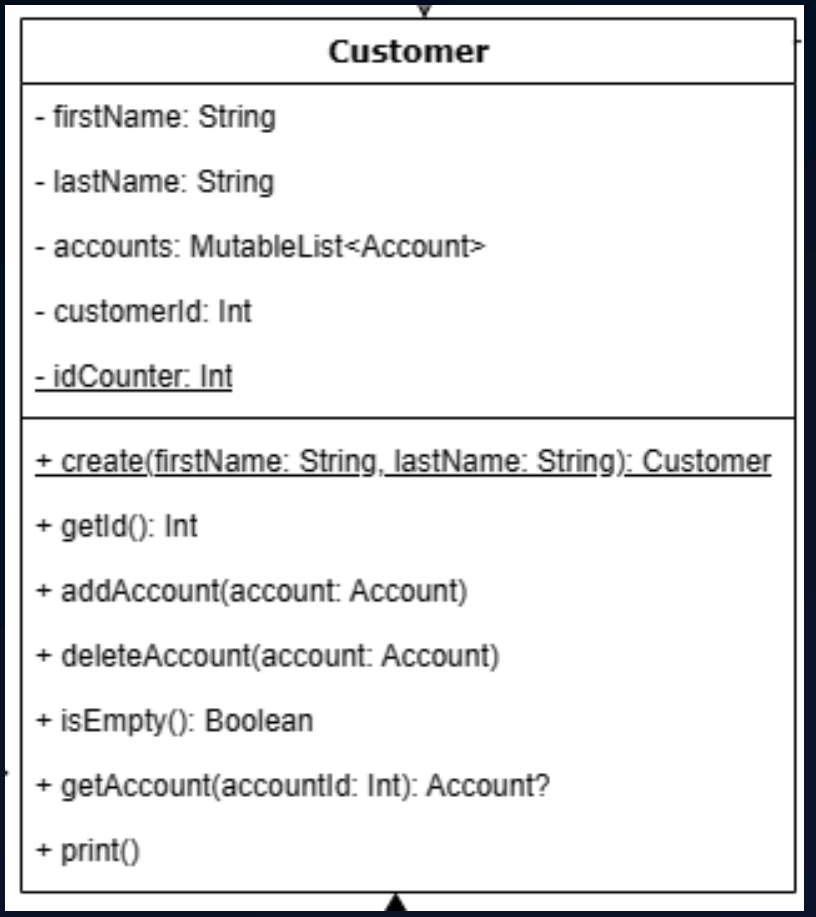
AN EXAMPLE IN KOTLIN
AND THE COMPARISON TO GO

Contents

- Structure and paradigms of OOP
- „CherryBanks“ – A Kotlin project
- Implementaion of OOP paradigms in Kotlin project
- Comparison to Go

Structure of OOP

- Classes
- Objects
- Attributes
- Methods



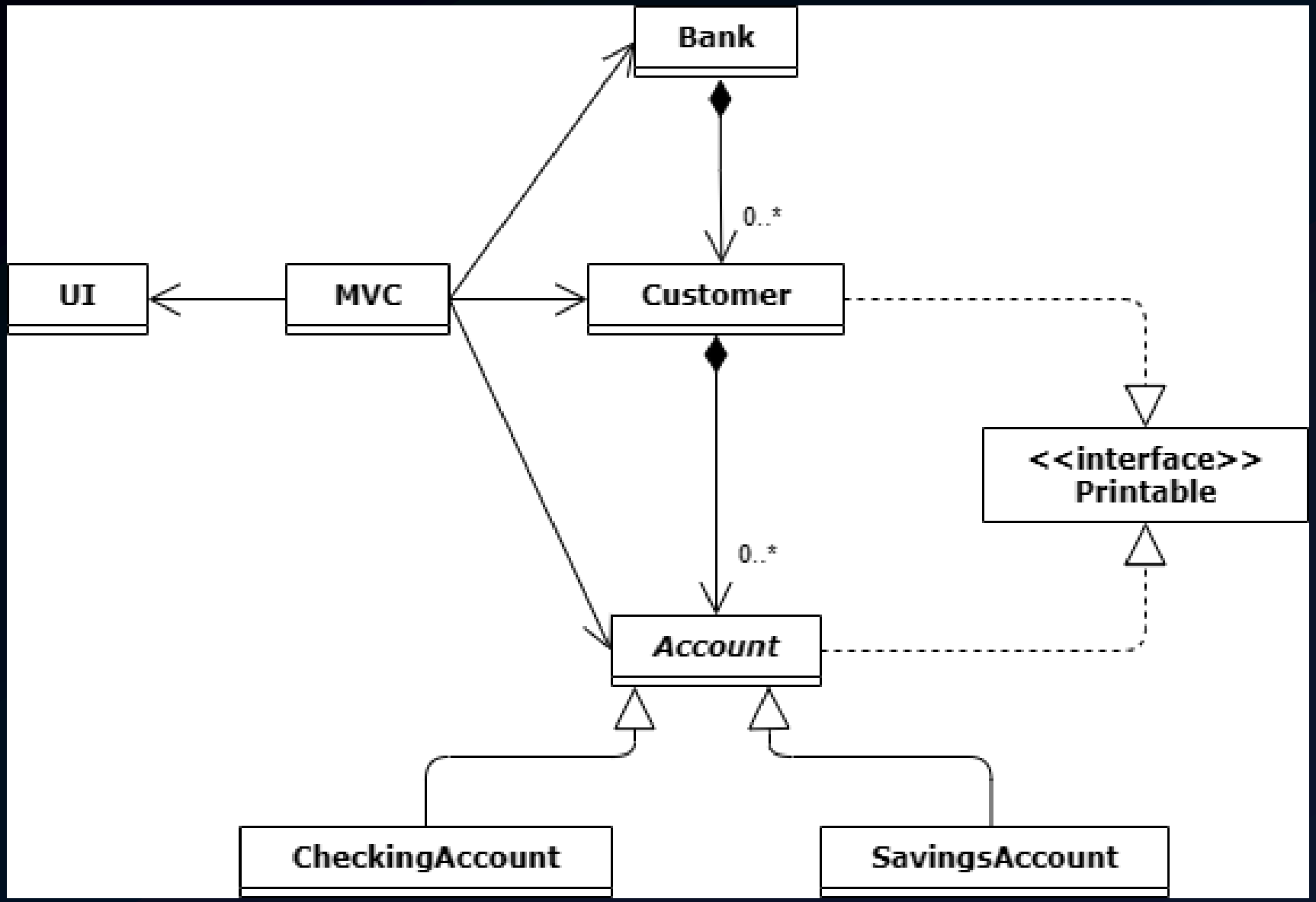
Paradigms of OOP

- Abstraction
- Inheritance
- Encapsulation
- Polymorphism

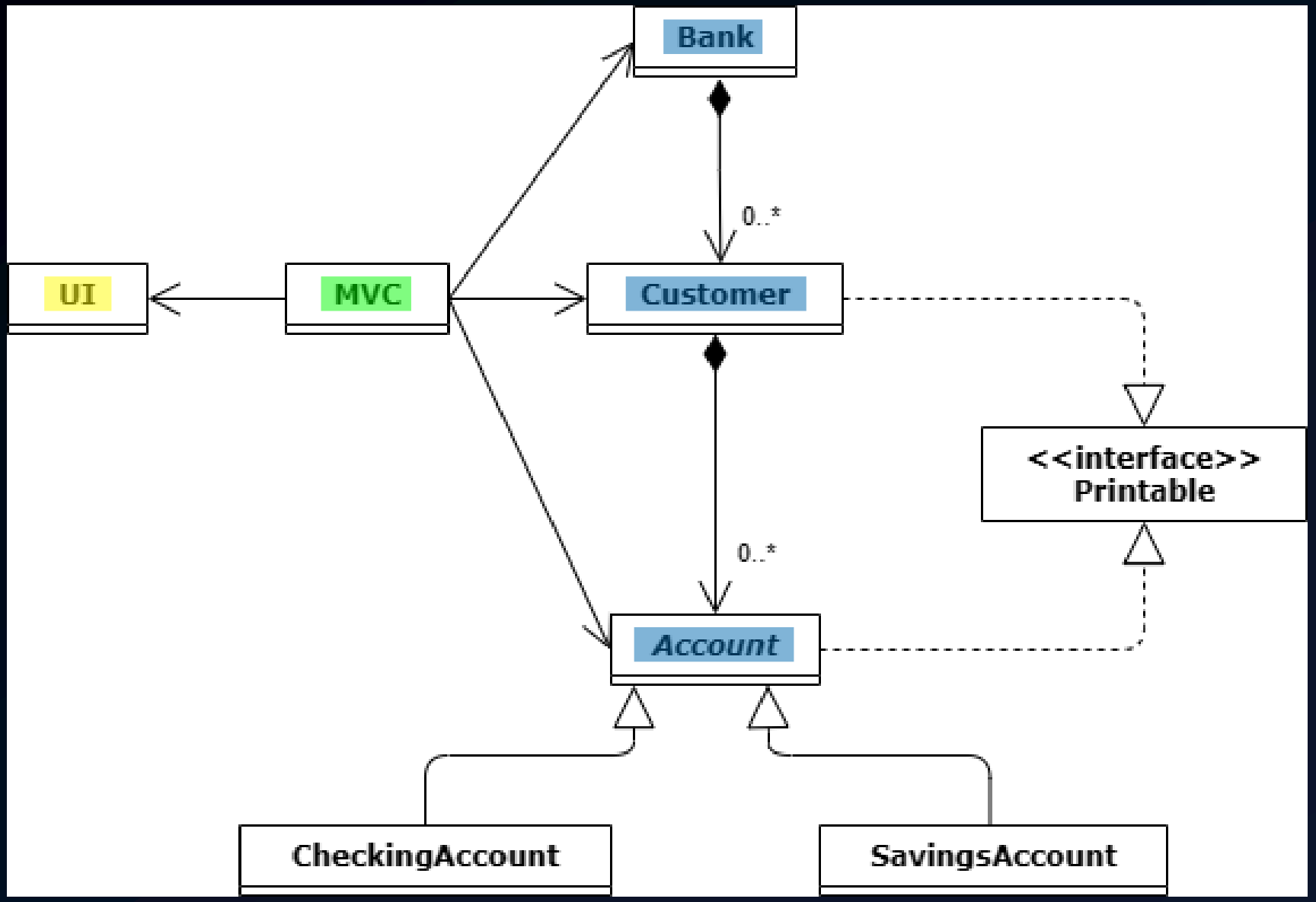
„CherryBanks“ – A Kotlin Project

- Bank account system
- Adding and deleting customers to/from bank
- Choosing customers and adding/deleting bank accounts
- Deposit and withdraw amounts to/from accounts
- Data collected from console input
- Actions displayed on console output

Programm Structure



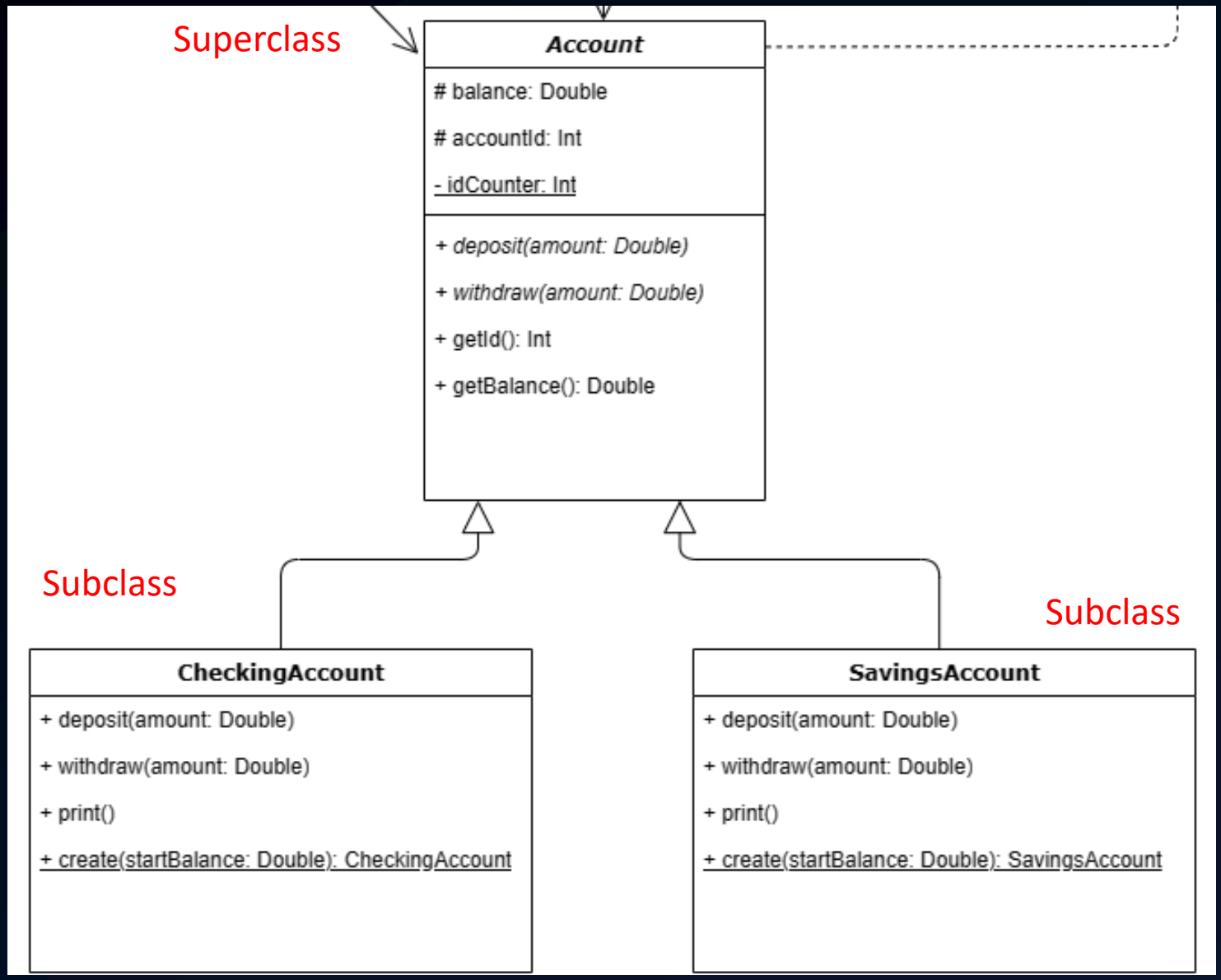
Programm Structure



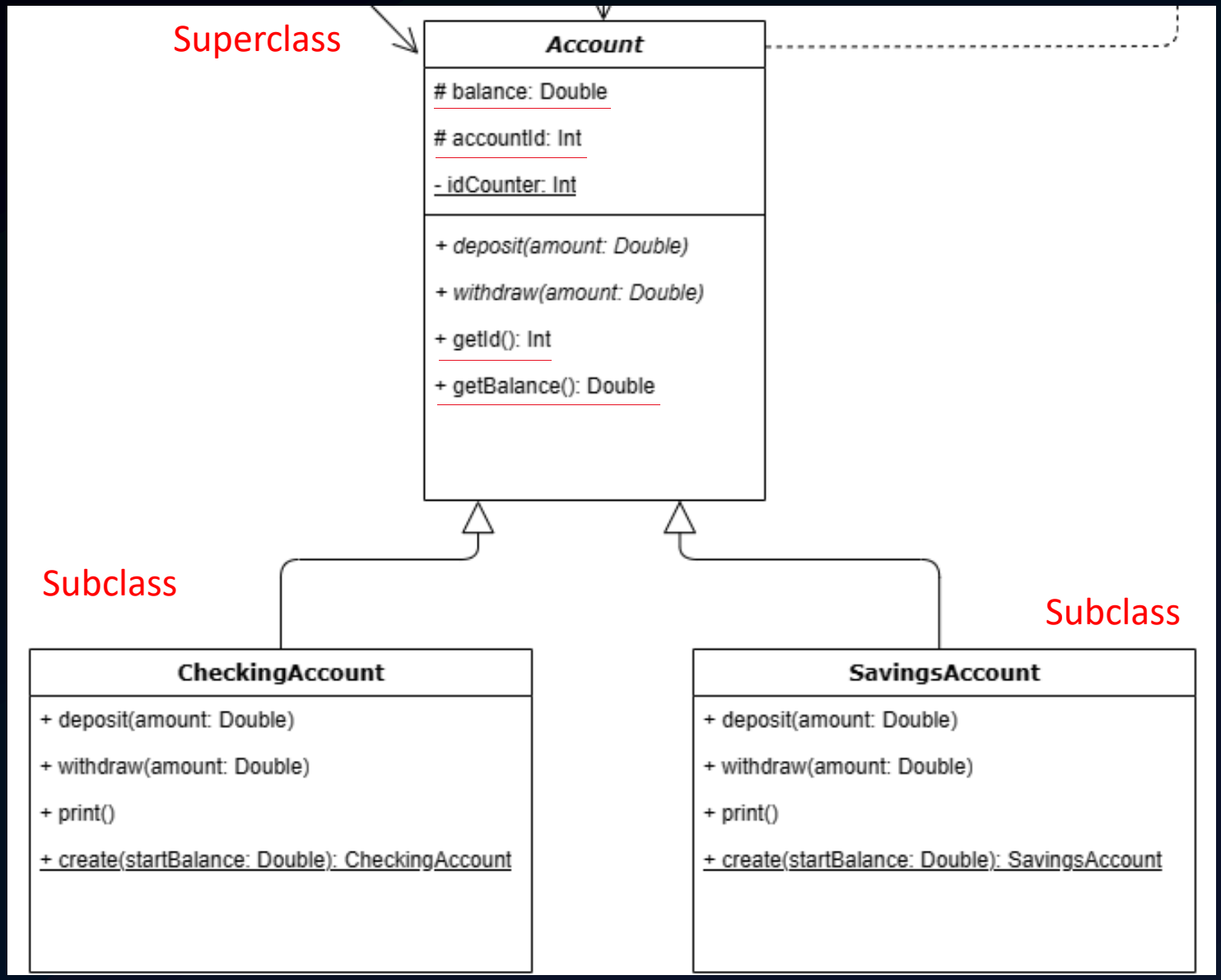
Inheritance

- Inheritance of properties and behaviors from another class
- Createing modified version of existing class
- Superclass, Subclass

Inheritance



Inheritance



Inheritance

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

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

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

    override fun print() {
        println(this::class.simpleName + "[ID: $accountId]")
        println("Balance: $balance\n")
    }
```

Inheritance in Go

- No inheritance
- No keyword for classes/superclasses/subclasses
- ➢ Solution: Type embedding

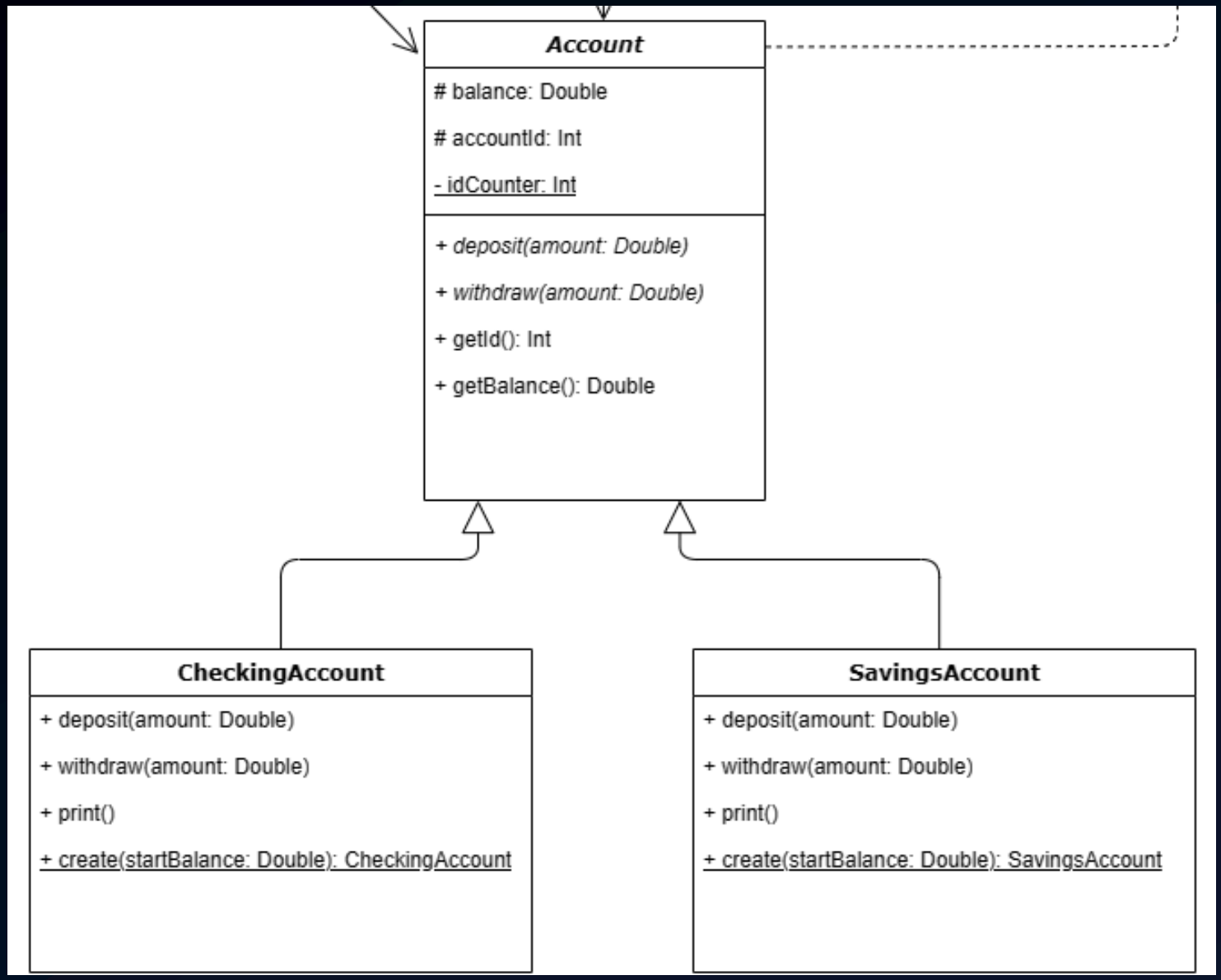
Inheritance in Go

```
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
}
```

Abstraction

- Exposing only necessary information to the user
- Specifies what subclasses must be able to do
- Does not necessarily prescribe how methods must be implemented

Abstraction



Abstraction

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

    init {
        accountId = idCounter
        idCounter++
    }

    companion object {
        private var idCounter = 1
    }

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

Abstraction

```
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
    }
```

class Customer:

```
fun getAccount(accountId: Int): Account? {
    return accounts.find { it.getId() == accountId }
}
```

Abstraction in Go

- No concept of abstract classes or methods
- No subclassing

➢ Solution: Interfaces

➢ Set of methods that a type must implement

Abstraction in Go

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

```
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
}
```

Encapsulation

- Bundling of data
- Hiding a classes internal impementation details
- Prevents other classes/code from direct access to the data
- Promotes modularity
- ➢ Makes it easier to maintain and modify code

Encapsulation

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

    fun addCustomer(customer: Customer) {
        customers.add(customer)
    }
```

Encapsulation in Go

- Achieved through capitalization of names
- Capital letter: public (access from outside the package)
- Lowercase letter: private (access only within package)

Encapsulation in Go

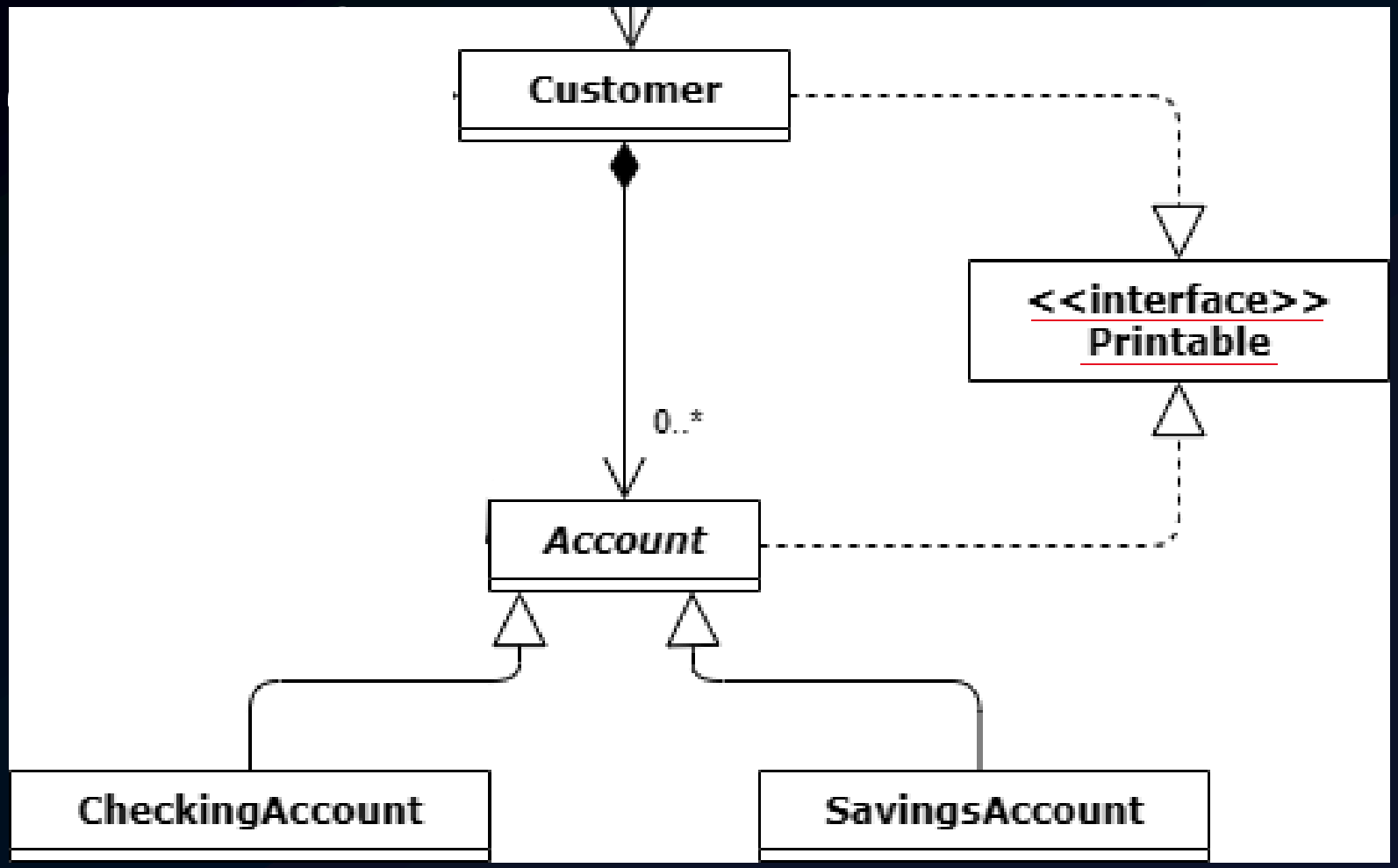
```
type Bank struct {
    customers []*Customer
}

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

Polymorphism

- Different objects respond to the same method call in different ways
- Objects can be treated uniformly despite different implementation

➢ Flexible and extensible code design

Polymorphism



Polymorphism

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

```
class Customer(private val firstName: String, private val lastName: String) : Printable {
    private val accounts: MutableList<Account> = mutableListOf()
    private var customerId: Int = 0
```

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

Polymorphism in Go

- Also supported through interfaces
- Set of methods type must implement (to implement Interface)
- ➢ Sharing common set of behaviors among different types

Polymorphism in Go

```
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]
}
```

Polymorphism in Go

```
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]
}
```

Conclusion

- Kotlin supports all important principles of OOP
- Go allows OOP with some detours
- ✓ Banking system successfully implemented
- ✓ Lessons learned

Thank you!

TOBIAS KIRSCHNER
MASTER INFORMATIK
TH ROSENHEIM
16.01.2023