Лекция #9. Объектно-ориентированное программирование

Наследование

Наследование позволяет создавать классы, которые расширяют функциональность или изменяют поведение уже существующих классов. В отношении наследования выделяются два ключевых компонента. Прежде всего это базовый класс (класс-родитель, родительский класс, суперкласс), который определяет базовую функциональность. И производный класс (класс-наследник, подкласс), который наследует функциональность базового класса и может расширять или модифицировать ее.

Чтобы функциональность класса можно было унаследовать, необходимо определить для этого класса аннотацию **open**. По умолчанию без этой аннотации класс не может быть унаследован.

```
open class базовый_класс
class производный_класс: базовый_класс
```

Для установки наследования после названия производного класса идет двоеточие и затем указывает класс, от которого идет наследование.

Например:

```
open class Person{
    var name: String = "Undefined"
    fun printName(){
        println(name)
    }
}
class Employee: Person()
```

Например, в данном случае класс **Person** представляет человека, который имеет свойство **name** (имя человека) и метод **printName()** для вывода информации о человеке. Класс **Employee** представляет условного работника. Поскольку работник является человеком, то класс работника будет разделять общий функционал с классом человека. Поэтому вместо того, чтобы заново определять в классе **Employee** свойство **name**, лучше уснаследовать весь функционал класса **Person**. То есть в данном случае класс **Person** является базовым или суперклассом, а класс **Employee** - производным классом или классом-наследником.

Но стоит учитывать, что при наследовании производный класс должен вызывать первичный конструктор (а если такого нет, то конструктор по умолчанию) базового класса.

Здесь класс **Person** явным образом не определяет первичных конструкторов, поэтому в классе **Employee** надо вызывать конструктор по умолчанию для класса **Person**

Вызвать конструктор базового класса в производном классе можно двумя способами. Первый способ - после двоеточия сразу указать вызов конструктора базового класса:

```
1  class Employee: Person()
```

Здесь запись **Person()** как раз представляет вызов конструктора по умолчанию класса **Person**.

Второй способ вызвать конструктор базового класса – определить в производном классе вторичный конструктор и в нем вызвать конструктор базового класса с помощью ключевого слова **super**:

```
1  open class Person{
2      var name: String = "Undefined"
3      fun printName(){
4          println(name)
5      }
6  }
7  class Employee: Person{
8
9      constructor() : super(){
10
11     }
12 }
```

Здесь с помощью ключевого слова **constructor** в классе **Employee** определяется вторичный конструктор. А после списка его параметров после двоеточия идет обращение к конструктору базового класса: **constructor() :**

super(). То есть здесь вызов **super()** - это и есть вызов конструктора базового класса.

Вне зависимости какой способ будет выбран, далее мы сможем создавать объекты класса **Employee** и использовать для него уснаследованный от класса **Person** функционал:

```
fun main() {

    val bob: Employee = Employee()
    bob.name = "Bob"
    bob.printName()
}
open class Person{
    var name: String = "Undefined"
    fun printName(){
        println(name)
    }
}
class Employee: Person()
```

Наследование класса с первичным конструктором

Если базовый класс явным образом определяет конструктор (первичный или вторичный), то производный класс должен вызывать этот конструктор. Для вызова конструктора базового в производном применяются те ж способы.

Первый способ - вызвать конструктор после названия класса через двоеточие:

```
open class Person(val name: String){
    fun printName(){
        println(name)
    }
}
class Employee(empName: String): Person(empName)
```

В данном случае класс **Person** через конструктор устанавливает свойство **name**. Поэтому в классе **Employee** тоже определен конструктор, который принимает стороковое значение и передает его в конструктор **Person**.

Если производный класс не имеет явного первичного конструктора, тогда при вызове вторичного конструктора должен вызываться конструктор базового класса через ключевое слово **super**:

```
open class Person(val name: String){
    fun printName(){
        println(name)
    }
}
class Employee: Person{

    constructor(empName: String) : super(empName){}
}
```

Опять же, поскольку конструктор **Person** принимает один параметр, то в **super()** нам надо передать значение для этого параметра.

Применение классов:

```
fun main() {

    val bob = Employee("Bob")
    bob.printName()
}

open class Person(val name: String){
    fun printName(){
        println(name)
    }
}
class Employee(empName: String): Person(empName)
```

Выше рассматривался случай, когда в базовом классе определен первичный конструктор.Но все то же действует и в том случае, если в базовом классе есть только вторичные конструкторы:

```
fun main() {

    val bob = Employee("Bob")
    bob.printName()
}

open class Person{

    val name: String
    constructor(userName: String){
        name = userName
    }
    fun printName(){
        println(name)
    }
}
class Employee(empName: String): Person(empName)
```

Расширение базового класса

Производный класс наследует функционал от базового класса, но также может определять и свой собственный функционал:

```
fun main() {

    val bob = Employee("Bob", "JetBrains")
    bob.printName()
    bob.printCompany()
}

open class Person(val name: String){
    fun printName(){
        println(name)
    }
}
class Employee(empName: String, val company: String): Person(empName){

    fun printCompany(){
        println(company)
    }
}
```

В данном случае класс **Employee** добаваляет к унаследованному функционалу свойство **company**, которое хранит компанию работника, и функцию **printCompany()**.

Стоит отметить, что в **Kotlin** мы можем унаследовать класс только от одного класса, множественное наследование не поддерживается.

Также, стоит отметить, что все классы по умолчанию наследуются от класса **Any**, даже если класс **Any** явным образом не указан в качестве базового. Поэтому любой класс уже по умолчанию будет иметь все свойства и функции, которые определены в классе **Any**. Поэтому все классы по умолчанию уже будут иметь такие функции как **equals**, **toString**, **hashcode**.

Модификаторы видимости

Все используемые типы, а также компоненты типов (классы, объекты, интерфейсы, конструкторы, функции, свойства) имеют определеннй уровень видимости, определяемый модификатором видимости (модификатором доступа). Модификатор видимости определяет, где те или иные типы и их компоненты доступны и где их можно использовать. В **Kotlin** есть следующие модификаторы видимости:

- **private**: классы, объекты, интерфейсы, а также функции и свойства, определенные вне класса, с этим модификатором видны только в том файле, в котором они определены. Члены класса с этим модификатором видны только в рамках своего класса
- **protected**: члены класса с этим модификатором видны в классе, в котором они определены, и в классах-наследниках
- **internal**: классы, объекты, интерфейсы, функции, свойства, конструкторы с этим модификатором видны в любой части модуля, в котором они определены. Модуль представляет набор файлов **Kotlin**, скомпилированных вместе в одну структурную единицу. Это может быть модуль **IntelliJ IDEA** или проект **Maven**
- **public**: классы, функции, свойства, объекты, интерфейсы с этим модификатором видны в любой части программы. (При этом если функции или классы с этим модификатором определены в другом пакете их все равно нужно импортировать)

Для установки уровня видимости модификатор ставится перед ключевыми словами **var/val/fun** в самом начале определения свойства или функции.

Если модификатор видимости явным образом не указан, то применяется модификатор **public**. То есть следующий класс:

```
class Person(){

    var name = "Undefined"
    var age = 18

    fun printPerson(){
        println("Name: $name  Age: $age")
    }
}
```

Будет эквивалентен следующему определению класса:

```
class Person(){

    public var name = "Undefined"
    public var age = 18

    public fun printPerson(){
        println("Name: $name  Age: $age")
    }
}
```

Если свойства объявляются через первичный конструктор и для них явным образом не указан модификатор видимости:

```
class Person(val name: String, val age: Int){
    public fun printPerson(){
        println("Name: $name  Age: $age")
    }
}
```

То также к таким свойствам автоматически применяется **public**:

```
class Person(public val name: String, public val age: Int){
    public fun printPerson(){
        println("Name: $name  Age: $age")
    }
}
```

Соответственно мы можем обращаться к подобным компонентам класса в любом месте программы:

```
fun main() {

    val tom = Person("Tom", 37)
    tom.printPerson()       // Name: Tom    Age: 37

    println(tom.name)
    println(tom.age)
}
```

Private

Если же к свойствам и методам применяется модификатор private, то к ним нельзя будет обратиться извне - вне данного класса.

```
class Person(private val name:String, _age: Int){

    private val age = _age

    fun printPerson(){
        printName()
        printAge()
    }
    private fun printName(){
        println("Name: $name")
    }
    private fun printAge(){
        println("Age: $age")
    }
}
```

```
fun main() {

    val tom = Person("Tom", 37)
    tom.printPerson()

    // println(tom.name)   // Ошибка! - свойство name - private
    // tom.printAge()  // Ошибка! - функция printAge - private
}
```

В данном случае класс **Person** определяет два свойства **name** (имя человека) и **age** (возраст человека). Чтобы было более показательно, одно свойство определено через конструктор, а второе как переменная класса. И поскольку эти свойства определены с модификатором **private**, то мы можем к ним обращаться только внутри этого класса. Вне класса обращаться к ним нельзя.

Также в классе определены три функции **printPerson()**, **printAge()** и **printName()**. Последние две функции выводят значения свойств. А функция **printPerson** выводит информацию об объекте, вызывая предыдущие две функции.

Однако функции **printAge()** и **printName()** определены как приватные, поэтому их можно использовать только внутри класса.

Модификаторы конструкторов

Конструкторы как первичные, так и вторичные также могут иметь модификаторы. Модификатор указывается перед ключевым словом **constructor**. По умолчанию они имеют модификатор **public**. Если для первичного конструктора необходимо явным образом установить модификатор доступа, то конструктор определяется с помощью ключевого слова **constructor**:

```
fun main() {

    // val bob  = Person("Bob")    // Так нельзя - конструктор private
}
open class Person private constructor(val name:String){

     fun printPerson(){
        println("Name: $name")
    }
}
// class Employee(name:String) : Person(name)  // так нельзя - конструктор в Person private
```

Стоит отметить, что в данном случае, поскольку конструктор приватный мы не можем его использовать вне класса ни для создания объекта класса в функции **main**, ни при наследовании. Но мы можем использовать такой конструктор в других конструкторах внутри класса:

```
fun main() {

    val tom = Employee("Tom", 37)
    tom.printPerson()
}
open class Person private constructor(val name:String){

    var age: Int = 0
    protected constructor(_name:String, _age: Int): this(_name){    // вызываем приватный конструктор
        age = _age
    }
     fun printPerson(){
        println("Name: $name Age: $age")
    }
}
class Employee(name:String, age: Int) : Person(name, age)
```

Здесь вторичный конструктор класса **Person**, который имеет модификатор **protected** (то есть доступен в текущем классе и классах-наследниках) вызывает первичный конструктор класса Person, который имеет модификатор **private**.

Модификаторы объектов и типов верхнего уровня*

Классы, а также переменные и функции, которые определены вне других классов, также могут иметь модификаторы **public**, **private** и **internal**.

Допустим, у нас есть файл **base.kt**, который определяет одноименный пакет:

```
package base

private val privateVal = 3
val publicVal = 5

private class PrivateClass(val name: String)
class PublicClass(val name:String)

private fun privateFun(){
    println("privateFn")
    println(privateVal)
    val privateClass= PrivateClass("Tom")
}

fun publicFun(){
    println("publicFn")
    println(privateVal)
    val privateClass= PrivateClass("Tom")
}
```

Внутри данного файла мы можем использовать его приватные переменные, функции классы. Однако при подключении этого пакета в другие файлы, приватные переменные, функции и классы будут недоступны:

```
import base.*

fun main() {

    publicFun()
    val publicClass= PublicClass("Tom")
    println(publicVal)


    // privateFun()                          // функция недоступна
    // val privateClass= PrivateClass("Tom")    // класс недоступен
    // println(privateVal)      // переменная недоступна
}
```

Однако даже внутри одного файла есть ограничения на использование приватных классов:

```
package email

private class Message(val text: String)

fun send(message: Message, address : String){
    println("Message `${message.text}` has been sent to $address")
}
```

Здесь мы столкнемся с ошибкой, так как публичная функция не может принимать параметр приватного класса. И в данном случае нам надо либо сделать класс **Message** публичным, либо функцию **send** приватной.