Абстрактные классы и методы

Абстрактные классы - это классы, определенные с модификатором abstract. Отличительной особенностью абстрактных классов является то, что мы не можем создать объект подобного класса. Например, определим абстрактный класс Human:

```
abstract class Human(val name: String)
```

Абстрактный класс, как и обычный, может иметь свойства, функции, конструкторы, но создать его объект напрямую вызвав его конструктор мы не можем:

```
val kate: Human     // норм, просто определение переменной
val alice: Human = Human("Alice")   // ! ошибка, создать объект нельзя
```

Такой класс мы можем только унаследовать:

```
abstract class Human(val name: String){

    fun hello(){
        println("My name is $name")
    }
}
class Person(name: String): Human(name)
```

Стоит отметить, что в данном случае перед абстрактным классом не надо указывать аннотацию open, как при наследовании неабстрактных классов.

```
fun main(args: Array<String>) {

    val kate: Person = Person("Kate")
    val slim: Human = Person("Slim Shady")
    kate.hello()    // My name is Kate
    slim.hello()    // My name is Slim Shady
}
```

Абстрактные классы могут иметь абстрактные методы и свойства. Это такие функции и свойства, которые определяются с ключевым словом abstract. Абстрактные методы не содержат реализацию, то есть у них нет тела. А для абстрактных свойств не указывается значение. При этом абстрактные методы и свойства можно определить только в абстрактных классах:

```
abstract class Human(val name: String){

    abstract var age: Int
    abstract fun hello()
}
class Person(name: String): Human(name){

    override var age : Int = 1
    override fun hello(){
        println("My name is $name")
    }
}
```

Если класс наследуется от абстрактного класса, то он должен либо реализовать все его абстрактные методы и свойства, либо также быть абстрактным.

Так, в данном случае класс Person должен обязательно определить реализацию для функции hello() и свойства age. При этом, как и при переопределении обычных методов и свойств, применяется аннотация override.

Абстрактные свойства также можно реализовать в первичном конструкторе:

```
abstract class Human(val name: String){

    abstract var age: Int
    abstract fun hello()
}
class Person(name: String, override var age : Int): Human(name){
    override fun hello(){
        println("My name is $name")
    }
}
```

Зачем нужны абстрактные классы? Классы обычно отражают какие-то сущности реального мира. Но некоторые из этих сущностей представляют абстракцию, которая непосредственного воплощения не имеет. Например, возьмем систему геометрических фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами. В этом случае мы можем определить абстрактный класс фигуры и затем от него унаследовать все остальные классы фигур:

```
// абстрактный класс фигуры
abstract class Figure {
    // абстрактный метод для получения периметра
    abstract fun perimeter(): Float

    // абстрактный метод для получения площади
    abstract fun area(): Float
```

```
}
// производный класс прямоугольника
class Rectangle(val width: Float, val height: Float) : Figure()
{
    // переопределение получения периметра
    override fun perimeter(): Float{
        return width * 2 + height * 2;
    }
    // переопрелеление получения площади
    override fun area(): Float{
        return width * height;
    }
}
```

Интерфейсы

Интерфейсы представляют контракт, который должен реализовать класс. Интерфейсы могут содержать объявления свойств и функций, а также их реализацию по умолчанию.

Для определения интерфейса применяется ключевое слово interface. Например:

```
interface Movable{
    var speed: Int  // объявление свойства
    fun move()      // определение функции без реализации
    fun stop(){     // определение функции с реализацией по умолчанию
        println("Остановка")
    }
}
```

Например, в данном случае интерфейс Movable представляет функцонал транспортного средства. Он содержит две функции и одно свойство. Функция move() представляет абстрактный метод - она не имеет реализации. Вторая функция stop() имеет реализацию по умолчанию.

При определении свойств в интерфейсе им не присваиваются значения.

Мы не можем напрямую создать объект интерфейса, так как интерфейс не поддерживает конструкторы и просто представляет шаблон, которому класс должен соответствовать.

Определим два класса, которые применяют интерфейс:

```
class Car : Movable{

    override var speed = 60
    override fun move(){
        println("Машина едет со скоростью $speed км/ч")
    }
}
class Aircraft : Movable{
```

```
    override var speed = 600
    override fun move(){
        println("Самолет летит со скоростью $speed км/ч")
    }
    override fun stop(){
        println("Приземление")
    }
}
```

Для применения интерфейса после имени класса ставится двоеточие, за которым следует название интерфейса. При применении интерфейса класс должен реализовать все его абстрактные методы и свойства, а также может предоставить свою реализацию для тех свойств и методов, которые уже имеют реализацию по умолчанию. При реализации функций и свойств перед ними ставится ключевое слово override.

Так, класс Car представляет машину и применяет интерфейс Movable. Так как интерфейс содержит абстрактный метод move(), то класс Car обязательно должен его реализовать.

Тоже касается свойства speed - класс Car должен его определить. Здесь реализация свойства заключается в установке для него начального значения.

А вот функцию stop() класс Car может не реализовать, так как она уже содержит реализацию по умолчанию.

Класс Aircraft представляет самолет и тоже применяет интерфейс Movable. При этом класс Aircraft реализует обе функции интерфейса.

В последствии в программе мы можем рассматривать объекты классом Car и Aircraft как объекты Movable:

```
fun main() {

    val m1: Movable = Car()
    val m2: Movable = Aircraft()
    // val m3: Movable = Movable() напрямую объект интерфейса создать нельзя

    m1.move()
    m1.stop()
    m2.move()
    m2.stop()
}
```

Консольный вывод программы:

```
Машина едет со скоростью 60 км/ч
Останавливается
```

```
Самолет летит со скоростью 600 км/ч
Самолет приземляется
```

Реализация свойств

Рассмотрим еще пример. Определим интерфейс Info, который объявляет ряд свойств:

```
interface Info{
    val model: String
        get() = "Undefined"
    val number: String
}
```

Первое свойство имеет геттер, а это значит, что оно имеет реализацию по умолчанию. При применении интерфейса такое свойство необязательно реализовать. Второе свойство - number является абстрактным, оно не имеет ни геттера, ни сеттера, то есть не имеет реализации по умолчанию, поэтому классы его обязаны реализовать.

Для реализации интерфейса возьмем выше определенный класс Car:

```
class Car(override val model: String, override var number: String) : Movable,
Info{

    override var speed = 60
    override fun move(){
        println("Машина едет со скоростью $speed км/ч")
    }
}
```

Теперь класс Car применяет два интерфейса. Класс может применять несколько интерфейсов, в этом случае они указываются через запятую, и все эти интерфейсы класс должен реализовать. Класс Car реализует оба свойства. При этом при реализации свойств в классе необязательно указывать геттер или сеттер. Кроме того, можно реализовать свойства в первичном конструкторе, как это сделано в случае со свойствами model и number

Применение класса:

```
fun main() {

    val tesla: Car = Car("Tesla", "2345SDG")
    println(tesla.model)
    println(tesla.number)

    tesla.move()
    tesla.stop()
}
```

Правила переопределения

В Kotlin мы можем наследовать класс и применять интерфейсы. При этом мы можем одновременно и наследоваться от класса, и применять один или несколько интерфейсов. Однако что, если переопределяемая функция из базового класса имеет то же имя, что и функция из применяемого интерфейса:

```
open class Video {
    open fun play() { println("Play video") }
}

interface AudioPlayable {
    fun play() { println("Play audio") }
}

class MediaPlayer() : Video(), AudioPlayable {
    // Функцию play обязательно надо переопределить
    override fun play() {
        super<Video>.play()         // вызываем Video.play()
        super<AudioPlayable>.play() // вызываем AudioPlayable.play()
    }
}
```

Здесь класс Video и интерфейс AudioPlayable определяют функцию play. В этом случае класс MediaPlayer, который наследуется от Video и применяет интерфейс AudioPlayable, обязательно должен определить функцию с тем же именем, то есть play. С помощью конструкции super<имя_типа>.имя_функции можно обратиться к опредленной реализации либо из базового класса, либо из интерфейса.

Вложенные классы и интерфейсы

В Kotlin классы и интерфейсы могут быть определены в других классах и интерфейсах. Такие классы (вложенные классы или nested classes) обычно выполняют какую-то вспомогательную роль, а определение их внутри класса или интерфейса позволяет разместить их как можно ближе к тому месту, где они непосредственно используются.

Например, в следующем случае определяется вложенный класс:

```
class Person{
    class Account(val username: String, val password: String){

        fun showDetails(){
            println("UserName: $username  Password: $password")
        }
    }
}
```

В данном случае класс Account является вложенным, а класс Person{ - внешним.

По умолчанию вложенные классы имеют модификатор видимости public, то есть они видимы в любой части программы. Но для обращения к вложенному классу надо использовать имя внешнего класса. Например, создание объекта вложенного класса:

```
fun main() {

    val userAcc = Person.Account("qwerty", "123456");
    userAcc.showDetails()
}
```

Если необходимо ограничить область применения вложенного класса только внешним классом, то следует определить вложенный класс с модификатором private:

```
class Person(username: String, password: String){

    private val account: Account = Account(username, password)

    private class Account(val username: String, val password: String)

    fun showAccountDetails(){
        println("UserName: ${account.username}  Password: $account.password")
    }
}
fun main() {

    val tom = Person("qwerty", "123456");
    tom.showAccountDetails()
}
```

Классы также могут содержать вложенные интерфейсы. Кроме того, интерфейсы тоже могут содержать вложенные классы и интерфейсы:

```
interface SomeInterface {
    class NestedClass
    interface NestedInterface
}

class SomeClass {
    class NestedClass
    interface NestedInterface
}
```

Внутренние (inner) классы

Стоит учитывать, что вложенный (nested) класс по умолчанию не имеет доступа к свойствам и функциям внешнего класса. Например, в следующем случае при попытке обратиться к свойству внешнего класса мы получим ошибку:

```
class BankAccount(private var sum: Int){

    fun display(){
        println("sum = $sum")
    }

    class Transaction{
        fun pay(s: Int){
            sum -= s
            display()
        }
    }
}
```

В данном случае у нас определен класс банковского счета BankAccount, который определяет свойство sum - сумма на счете и функцию display() для вывода информации о счете.

Кроме того, в классе BankAccount определен вложенный класс Transaction, который представляет операцию по счету. В данном случае класс Transaction определяет функцию pay() для оплаты со счета. Однако в нем мы не можем обратиться в свойствам и функциям внешнего класса BankAccount.

Чтобы вложенный класс мог иметь доступ к свойствам и функциям внешнего класса, необходимо определить вложенный класс с ключевым словом inner. Такой класс еще называют внутренним классом (inner class), чтобы отличать от обычных вложенных классов. Например:

```
fun main() {

    val acc = BankAccount(3400);
    acc.Transaction().pay(2500)
}
class BankAccount(private var sum: Int){

    fun display(){
        println("sum = $sum")
    }

    inner class Transaction{
        fun pay(s: Int){
            sum -= s
            display()
        }
    }
}
```

Теперь класс Transaction определен с ключевым словом inner, поэтому имеет полный доступ к свойствам и функциям внешнего класса BankAccount. Но теперь если мы хотим использовать объект подобного вложенного класса, то необходимо создать объект внешнего класса:

```
val acc = BankAccount(3400);
    acc.Transaction().pay(2500)
```

Совпадение имен

Но что если свойства и функции внутреннего класса называются также, как и свойства и функции внешнего класса? В этом случае внутренний класс может обратиться к свойствам и функциям внешнего через конструкцию this@название_класса.имя_свойства_или_функции:

```
class A{
    private val n: Int = 1
    inner class B{
        private val n: Int = 1
        fun action(){
            println(n)          // n из класса B
            println(this.n)     // n из класса B
            println(this@B.n)   // n из класса B
            println(this@A.n)   // n из класса A
        }
    }
}
```

Например, перепишем случай выше с классами Account и Transaction следующим образом:

```
fun main() {

    val acc = BankAccount(3400);
    acc.Transaction(2400).pay()
}
class BankAccount(private var sum: Int){

    fun display(){
        println("sum = $sum")
    }

    inner class Transaction(private var sum: Int){
        fun pay(){
            this@BankAccount.sum -= this@Transaction.sum
            display()
        }
    }
}
```

Data-классы

Иногда классы бывают необходимы только для хранения некоторых данных. В Kotlin такие классы называются data-классы. Они определяются с модификатором data:

```
data class Person(val name: String, val age: Int)
```

При компиляции такого класса компилятор автоматически добавляет в класс функции с определенной реализацией, которая учитывает свойства класса, которые определены в первичном конструкторе:

- equals(): сравнивает два объекта на равенство
- hashCode(): возвращает хеш-код объекта
- toString(): возвращает строковое представление объекта
- copy(): копирует данные объекта в другой объект

Например, возьмем функцию toString(), которая возвращает строковое представление объекта:

```
fun main() {

    val alice: Person = Person("Alice", 24)
    println(alice.toString())
}

class Person(val name: String, val age: Int)
```

Результатом программы будет следующий вывод:

```
Person@2a18f23c
```

По умолчанию строковое представление объекта нам практически ни о чем не говорит. Как правило, данная функция предназначена для вывода состояния объекта, но для этого ее надо переопределять. Однако теперь добавим модификатор data к определению класса:

```
data class Person(val name: String, val age: Int)
```

И результат будет отличаться:

```
Person(name=Alice, age=24)
```

То есть мы можем увидить, какие данные хранятся в объекте, какие они имеют значения. То же самое касается всех остальных функций. Таким образом, в случае с data-классами мы имеем готовую реализацию для этих функций. Их не надо вручную переопределять. Но вполне возможно нас может не устраивать эта реализация, тогда мы можем определить свою:

```
data class Person(val name: String, val age: Int){
    override fun toString(): String {
        return "Name: $name  Age: $age"
    }
}
```

В этом случае для функции toString() компилятор не будет определять реализацию.

Другим показательным примером является копирование данных:

```
fun main() {

    val alice: Person = Person("Alice", 24)
    val kate = alice.copy(name = "Kate")
    println(alice.toString())   // Person(name=Alice, age=24)
    println(kate.toString())    // Person(name=Kate, age=24)
}

data class Person(var name: String, var age: Int)
```

Опять же компилятор генерирует функцию копирования по умолчанию, которую мы можем использовать. Если мы хотим, чтобы некоторые данные у объкта отличались, то мы их можем указать в функции copy в виде именованных аргументов, как в случае со свойством name в примере выше.

При этом чтобы класс определить как data-класс, он должен соответствовать ряду условий:

- Первичный конструктор должен иметь как минимум один параметр
- Все параметры первичного конструктора должны предваряться ключевыми словами val или var, то есть определять свойства

Свойства, которые определяются вне первичного конструктора, не используются в функциях toString, equals и hashCode

- Класс не должен определяться с модификаторами open, abstract, sealed или inner.

Также стоит отметить, что несмотря на то, что мы можем определять свойства в первичном конструкторе и через val, и через var, например:

```
data class Person(var name: String, var age: Int)
```

Но вообще в ряде ситуаций рекомендуется определять свойства через val, то есть делать их неизменяемыми, поскольку на их основании вычисляет хеш-код, который используется в качестве ключа объекта в такой коллекции как HashMap.

Декомпозиция data-классов

Kotlin предоставляет для data-классов возможность декомпозиции на переменные:

```
fun main() {

    val alice: Person = Person("Alice", 24)

    val (username, userage) = alice
    println("Name: $username  Age: $userage") // Name: Alice  Age: 24
}

data class Person(var name: String, var age: Int)
```

Перечисления enums

Enums или перечисления представляют тип данных, который позволяет определить набор логически связанных констант. Для определения перечисления применяются ключевые слова enum class. Например, определим перечисление:

```
enum class Day{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Данное перечисление Day представляет день недели. Внутри перечисления определяются константы. В данном случае это названия семи дней недели. Константы определяются через запятую. Каждая константа фактически представляет объект данного перечисления.

```
fun main() {

    val day: Day = Day.FRIDAY
    println(day)            // FRIDAY
    println(Day.MONDAY)     // MONDAY
}
```

Классы перечислений как и обычные классы также могут иметь конструктор. Кроме того, для констант перечисления также может вызываться конструктор для их инициализации.

```

```

```
enum class Day(val value: Int){
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(100500)
}

fun main() {

    val day: Day = Day.FRIDAY
    println(day.value)        // 5
    println(Day.MONDAY.value) // 1
}
```

В примере выше у класса перечисления через конструктор определяется свойство value. Соответственно при определении констант перечисления необходимо каждую из этих констант инициализировать, передав значение для свойства value.

При этом перечисления - это не просто список значений. Они могут определять также свойства и функции. Но если класс перечисления содержит свойства или функции, то константы должны быть отделены точкой с запятой.

```
enum class Day(val value: Int){
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6),
    SUNDAY(7);
    fun getDuration(day: Day): Int{
        return value - day.value;
    }
}

fun main() {

    val day1: Day = Day.FRIDAY
    val day2: Day = Day.MONDAY
    println(day1.getDuration(day2))        // 4
}
```

В данном случае в перечислении определена функция getDuration(), которая вычисляет разницу в днях между двумя днями недели.

Встроенные свойства и вспомогательные методы Все перечисления обладают двумя встроенными свойствами:

- name: возвращает название константы в виде строки
- ordinal: возвращает порядковый номер константы

```
enum class Day(val value: Int){
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
```

```
    THURSDAY(4), FRIDAY(5), SATURDAY(6),
    SUNDAY(7)
}

fun main() {

    val day1: Day = Day.FRIDAY
    println(day1.name)        // FRIDAY
    println(day1.ordinal)     // 4
}
```

Кроме того, в Kotlin нам доступны вспомогательные функции:

- valueOf(value: String): возвращает объект перечисления по названию константы
- values(): возвращает массив констант текущего перечисления

```
fun main() {

    for(day in Day.values())
        println(day)

    println(Day.valueOf("FRIDAY"))
}
```

Анонимные классы и реализация интерфейсов

Константы перечисления могут определять анонимные классы, которые могут иметь собственные методы и свойства или реализовать абстрактные методы класса перечисления:

```
enum class DayTime{
    DAY{
        override val startHour = 6
        override val endHour = 21
        override fun printName(){
            println("День")
        }
    },
    NIGHT{
        override val startHour = 22
        override val endHour = 5
        override fun printName(){
            println("Ночь")
        }
    };
    abstract fun printName()
    abstract val startHour: Int
    abstract val endHour: Int
}
```

```
fun main() {

    DayTime.DAY.printName()     // День
    DayTime.NIGHT.printName()   // Ночь

    println("Day from ${DayTime.DAY.startHour} to ${DayTime.DAY.endHour}")

}
```

В данном случае класс перечисления DayTime определяет абстрактный метод printName() и две переменных - startHour (начальный час) и endHour (конечный час). А константы определяют анонимные классы, которые реализуют эти свойства и функцию.

Также, классы перечислений могут применять интерфейсы. Для этого для каждой константы определяется анонимный класс, который содержат все реализуемые свойства и функции:

```
interface Printable{
    fun printName()
}
enum class DayTime: Printable{
    DAY{
        override fun printName(){
            println("День")
        }
    },
    NIGHT{
        override fun printName(){
            println("Ночь")
        }
    }
}

fun main() {

    DayTime.DAY.printName()     // День
    DayTime.NIGHT.printName()   // Ночь
}
```

Хранение состояния

Нередко перечисления применяются для хранения состояния в программе. И в зависимоси от этого состояния мы можем направить действие программы по определенному пути. Например, определим перечисление, которое представляет арифметические операции, и функцию, которая в зависимости от переданной операции выполняет то или иное действие:

```
fun main() {
```

```
    println(operate(5, 6, Operation.ADD))         // 11
    println(operate(5, 6, Operation.SUBTRACT))    // -1
    println(operate(5, 6, Operation.MULTIPLY))    // 30
}
enum class Operation{

    ADD, SUBTRACT, MULTIPLY
}
fun operate(n1: Int, n2: Int, op: Operation): Int{

    when(op){
        Operation.ADD -> return n1 + n2
        Operation.SUBTRACT -> return n1 - n2
        Operation.MULTIPLY -> return n1 *n2
    }
}
```

Функция operate() принимает два числа - операнды операции и тип операции в виде перечисления Operation. И в зависимоси от значения перечисления возвращает либо сумму, либо разность, либо произведение двух чисел.

Делегирование

Делегирование представляет паттерн объектно-ориентированного программирования, который позволяет одному объекту делегировать/перенаправить все запросы другому объекту. В определенной степени делегирование может выступать альтернативой наследованию. И преимуществом Kotlin в данном случае состоит в том, что Kotlin нативно поддерживает данный паттерн, предоставляя необходимый инструментарий.

Формальный синтаксис:

```
interface Base {
    fun someFun()
}

class BaseImpl() : Base {
    override fun someFun() { }
}

class Derived(someBase: Base) : Base by someBase
```

Есть некоторый интерфейс - Base, который определяет некоторый функционал. Есть его реализация в виде класса BaseImpl.

И есть еще один класс - Derived, который также применяет интерфейс Base. Причем после указания применяемого интерфейса идет ключевое слово by, а после него - объект, которому будут делегироваться вызовы.

```
class Derived(someBase: Base) : Base by someBase
```

То есть в данной схеме класс Derived будет делегировать вызовы объекту someBase, который представляет интерфейс Base и передается через первичный конструктор. При этом Derived может не реализовать интерфейс Base или реализовать неполностью - какие-то отдельные свойства и функции.

Например, рассмотрим следующие классы:

```
interface Messenger{
    fun send(message: String)
}
class InstantMessenger(val programName: String) : Messenger{

    override fun send(message: String){
        println("Message `$message` has been sent")
    }
}
class SmartPhone(val name: String, m: Messenger): Messenger by  m
```

Здесь определен интерфейс Messenger, который представляет условно программу для отправки сообщений. Для условной отправки сообщений определена функция send().

Также есть класс InstantMessenger - программа мгновенных сообщений или проще говоря мессенджер, который применяет интерфейс Messenger, реализуя его функцию send()

Далее определен класс SmartPhone, который представляет смартфон и также применяет интерфейс Messenger, но не реализует его. Вместо этого он принимает через первичный конструктор объект Messenger и делегирует ему обращение к функции send().

Применим классы:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    pixel.send("Hello Kotlin")
    pixel.send("Learn Kotlin on Metanit.com")
}
```

Здесь создан объект pixel, который представляет класс SmartPhone. Поскольку SmartPhone применяет интерфейс Messenger, то мы можем вызвать у объекта pixel функцию send() для отправки условного сообщения. Однако сам класс SmartPhone НЕ реализует функцию send - само выполнение этой функции делегируется объекту telegram, который в реальности выполняет отправку сообщения. Соответственно при выполнении программы мы увидим следующий консольный вывод:

```
Message `Hello Kotlin` has been sent
Message `Learn Kotlin on Metanit.com` has been sent
```

Множественное делегирование

Подобным образом один объект может делегировать выполнение различных функций разным объектам. Например:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val photoCamera = PhotoCamera()
    val pixel = SmartPhone("Pixel 5", telegram, photoCamera)
    pixel.send("Hello Kotlin")
    pixel.takePhoto()
}

interface Messenger{
    fun send(message: String)
}
class InstantMessenger(val programName: String) : Messenger{
    override fun send(message: String) = println("Send message: `$message`")
}
interface PhotoDevice{
    fun takePhoto()
}
class PhotoCamera: PhotoDevice{
    override fun takePhoto() = println("Take a photo")
}
class SmartPhone(val name: String, m: Messenger, p: PhotoDevice)
    : Messenger by  m, PhotoDevice by p
```

Здесь класс SmartPhone также реализует интерфейс PhotoDevice, который предоставляет функцию takePhoto() для съемки фото. Но выполнение этой функции он делегирует параметру p, который представляет интерфейс PhotoDevice и в роли которого выступает объект PhotoCamera.

Переопределение функций

Класс может переопределять часть функций интерфейса, в этом случае выполнение этих функций не делегируется. Например:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    pixel.sendTextMessage()
    pixel.sendVideoMessage()
}
```

```
interface Messenger{
    fun sendTextMessage()
    fun sendVideoMessage()
}
class InstantMessenger(val programName: String) : Messenger{
    override fun sendTextMessage() = println("Send text message")
    override fun sendVideoMessage() = println("Send video message")
}
class SmartPhone(val name: String, m: Messenger) : Messenger by  m{
    override fun sendTextMessage() = println("Send sms")
}
```

В данном случае класс SmartPhone реализует функцию sendTextMessage(), поэтому ее выполнение не делегируется. Консольный вывод программы:

```
Send sms
Send video message
```

Делегирование свойств

По аналогии с функциями объект может делегировать обращение к свойствам:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    println(pixel.programName)  // Telegram
}
interface Messenger{
    val programName: String
}
class InstantMessenger(override val programName: String) : Messenger
class SmartPhone(val name: String, m: Messenger) : Messenger by  m
```

Здесь при интерфейс Messenger определяет свойство programName - название программы отправки. Класс SmartPhone не реализует это свойство, поэтому обращение к этому свойству делегируется объекту m.

Если бы класс SmartPhone сам реализовал это свойство, то делегирования бы не было:

```
fun main() {
    val telegram = InstantMessenger("Telegram")
    val pixel = SmartPhone("Pixel 5", telegram)
    println(pixel.programName)  // Default Messenger
}
interface Messenger{
    val programName: String
```

```
}
class InstantMessenger(override val programName: String) : Messenger
class SmartPhone(val name: String, m: Messenger) : Messenger by  m{
    override val programName = "Default Messenger"
}
```

Анонимные классы и объекты

Иногда возникает необходимость создать объект некоторого класса, который больше нигде в программе не используется. То есть класс необходим только для создания только одного объекта. В этом случае мы, конечно, можем, как и обычно, определить класс и затем создать объект этого класса. Но Kotlin для таких ситуаций предоставлять возможность определить объект анонимного класса.

Анонимные классы не используют ключевое слово class для определения. Они не имеют имени, но как и обычные классы могут наслдовать другие классы или применять интерфейсы. Объекты анонимных классов называют анонимыми объктами.

Для определения анонимного объекта применяется ключевое слово object:

```
fun main() {

    val tom = object {
        val name = "Tom"
        var age = 37
        fun sayHello(){
            println("Hi, my name is $name")
        }
    }
    println("Name: ${tom.name}  Age: ${tom.age}")
    tom.sayHello()
}
```

После ключевого слова object идет блок кода в фигурных скобках, в которые помещается определение объекта. Как и в обычном классе, анонимный объект может содержать свойства, функции. И далее по имени переменной мы можем обращаться к свойствам и функциям этого объекта.

Наследование анонимных объектом

При наследовании после слова object через двоеточия указывается имя наследуемого класса или его первичный конструктор:

```
fun main() {

    val tom = object : Person("Tom"){

        val company = "JetBrains"
        override fun sayHello(){
```

```
            println("Hi, my name is $name. I work in $company")
        }
    }

    tom.sayHello()  // Hi, my name is Tom. I work in JetBrains
}
open class Person(val name: String){
    open fun sayHello(){
        println("Hi, my name is $name")
    }
}
```

Здесь класс анонимного объекта наследует класс Person и переопределяет его функцию sayHello().

Анонимный объект как аргумент функции

Анонимный объект может передаваться в качестве аргумента в вызов функции:

```
fun main() {
    hello(
        object : Person("Sam"){
            val company = "JetBrains"
            override fun sayHello(){
                println("Hi, my name is $name. I work in $company")
            }
    })
}
fun hello(person: Person){
    person.sayHello()
}
open class Person(val name: String){
    open fun sayHello() = println("Hi, my name is $name")
}
```

Здесь поскольку класс анонимного объекта наследуется от класса Person, мы можем передавать этот анонимный объект параметру функции, который имеет тип Person.

Анонимный объект как результат функции

Функция может возвращать анонимный объект:

```
fun main() {
    val tom = createPerson("Tom", "JetBrains")
    tom.sayHello()
}
private fun createPerson(_name: String, _company: String) = object{
    val name = _name
    val company = _company
```

```
    fun sayHello() = println("Hi, my name is $name. I work in $company")
}
```

Однако тут есть нюансы. Чтобы мы могли обращаться к свойствам и функциям анонимного объекта, функция, которая возвращает этот объект, должна быть приватной, как в примере выше.

Если функция имеет модификатор public или private inline, то в этом случае свойства и функции анонимного класса (за исключением унаследованных) недоступны:

```
fun main() {
    val tom = createPerson("Tom", "JetBrains")
    println(tom.name)   // норм - свойство name унаследовано от Person
    println(tom.company)    // ! Ошибка - свойство недоступно
}
private inline fun createPerson(_name: String, _comp: String) = object:
Person(_name){
    val company = _comp
}

open class Person(val name: String)
```

В данном случае функция createPerson() имеет модификатор private inline, поэтому у анонимного объекта будут доступны только унаследованные свойства и функции от класса Person, но собственные свойства и функции будут не доступны.