14.KOTLIN. Классы. Интерфейсы. Объекты

14.1 Интерфейсы

Интерфейсы в Kotlin напоминают Java 8: они могут содержать определения абстрактных и реализации конкретных методов (подобно методам по умолчанию в Java 8), но они не могут иметь состояний.

```
interface Drageable {
    fun drag()
}
```

Вместо ключевых слов extends и implements, используемых в Java, в языке Kotlin используется двоеточие после имени класса. Как в Java, класс может реализовать столько интерфейсов, сколько потребуется, но наследовать только один класс.

```
class Window : Drageable  {
    override fun drag()  = println("Drag")

}
```

Модификатор override используется для переопределения. *Применение модификатора override обязательно* (в Java нет).

Метод интерфейса может иметь **реализацию по умолчанию**. Пишем его как обычный метод.

```
interface Drageable {
    fun drag()
    fun logging() = println("it was draged")
}
```

Явная реализация унаследованных интерфейсов с реализацией по умолчанию. Если вы явно не реализуете метод logging , компилятор сообщит об ошибке

```
 interface Drageable {
    fun drag()
    fun logging() = println("it was draged")
}
interface Logable {
   fun logging() = println("it is logged")
}
class Window : Drageable, Logable  {
    override fun logging() {
        super<Drageable>.logging()
        super<Logable>.logging()
    }
    override fun drag()  = println("Drag")
}
```

если вы выбираете одну реализацию:

```
class Window : Drageable, Logable  {
    override fun logging()  =   super<Drageable>.logging()
    override fun drag()  = println("Drag")
}
```

Вызовы будут выглядеть так:

```
fun main(args: Array<String>) {
    val window = Window()
    window.logging()
    window.drag()

}
```

Интерфейсы в Kotlin могут включать **объявления абстрактных свойств**.

```
interface Human { val nickname: String }
```

Интерфейс не определяет, как будет доступно значение - как поле или через метод доступа. Поскольку сам интерфейс не имеет состояния, то только реализующие его классы смогут хранить соответствующее значение.

Кроме абстрактных свойств, интерфейс может содержать **свойства с методами чтения и записи** — при условии, что они не обращаются к полю в памяти (такое поле потребовало бы хранения состояния в интерфейсе, что невозможно).

```
interface Users {
    val email: String
    val nickname: String
        get() = email.substringBefore('@')
}
```

Этот интерфейс определяет абстрактное свойство email, а также свойство nickname с методом доступа. Первое свойство должно быть переопределено в подклассах, а второе может быть унаследовано.

В интерфейсах ключевые слова *final, open и abstract не используются*.

14.2 Наследование: модификаторы open, final и abstract

В Java все классы и методы открыты по умолчанию, а в Kotlin они *по умолчанию отмечены модификатором final*.

Если вы хотите разрешить наследовать свой класс, объявите его открытым, добавив модификатор **open**. Вам также понадобится добавить модификатор **open** ко всем свойствам и методам, которые могут быть переопределены.

```
open class List : Logable { // открытый класс - можно наследовать
    fun add (){} //закрытая функция - нельзя  переопределить
    open fun animate() {} //открытая функция -  можно  переопределить
    override fun logging() { // переопределение
        super.logging()
    }
}
```

Чтобы запретить переопределение вашей реализации в подклассах, добавьте в объявление переопределяющей версии модификатор **final**:

```
open class List : Logable {

   final override fun logging() {//запрет переопределения
        super.logging()
    }
}
```

Одно из основных преимуществ классов, закрытых по умолчанию: они позволяют выполнять автоматическое приведение типов в большем количестве сценариев.

В Kotlin класс можно объявить абстрактным, добавив ключевое слово **abstract**, и создать экземпляр такого класса будет невозможно. Абстрактные методы всегда открыты, поэтому не требуется явно использовать модификатор open.

```
abstract class Colorfull {
    abstract fun coclored()
    open fun stopColoring() {} //функции в абстрактных классах по умолчанию закрыты
    fun reDraw() {}
}
```

14.3 Модификаторы видимости

Используются те же ключевые слова: public, protected и private. Отличается лишь видимость по умолчанию: отсутствие модификатора в объявлении предполагает модификатор **public**.

Область видимости в Java по умолчанию ограничивается рамками пакета. В Kotlin такой модификатор видимости *отсутствует*: пакеты используются только для организации кода в пространстве имен, но не для управления видимостью. Kotlin предлагает модификатор видимости **internal**, обозначающий «видимость в границах модуля».

Ещё одно отличие: Kotlin позволяет приметить модификатор private к объявлениям верхнего уровня, в том числе к классам, функциям и свойствам. Такие объявления видны только в файле, где они определены.

Модификатор	Член класса	Объявление верхнего уровня
public (по умолчанию)	Доступен повсюду	Доступно повсюду
internal	Доступен только в модуле	Доступно в модуле
protected	Доступен в подклассах	–
private	Доступен в классе	Доступно в файле

В Kotlin член класса с модификатором protected доступен только в самом классе и его подклассах.

14.4 Внутренние и вложенные классы

И в Java, и в Kotlin можно объявить один класс внутри другого класса. Разница в том, что в Kotlin *вложенные классы не имеют доступа к экземпляру внешнего класса*, если не запросить его явно.

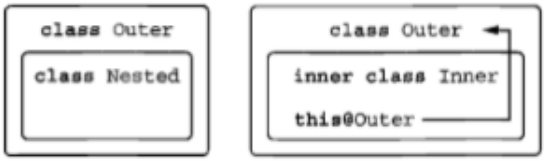
```
class Button : View {

    class ButtonState : State { /*...*/ }

}
```

В Kotlin вложенный класс без модификаторов — это полный аналог статического вложенного класса в Java. Чтобы превратить его во внутренний класс со ссылкой на внешний класс, нужно добавить модификатор **inner**.

Класс A, объявленный внутри другого класса B	В Java	В Kotlin
Вложенный класс (не содержит ссылки на внешний класс)	`static class A`	`class A`
Внутренний класс (содержит ссылку на внешний класс)	`class A`	`inner class A`



Синтаксис обращения к внешнему классу из внутреннего в Kotlin также отличается от Java. Чтобы получить доступ к классу Outer из класса Inner, нужно написать this@Outer.

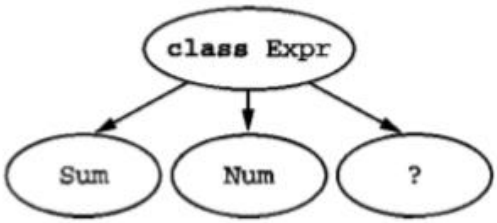
```
class Outer {
    inner class Inner {
        fun getOuterReference(): Outer = this@Outer
    }
}
```

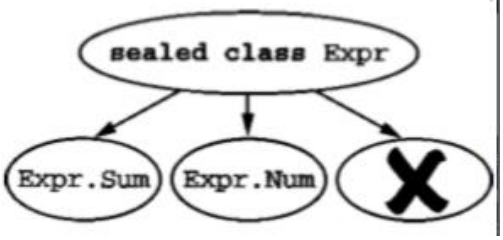
14.5 Запечатанные классы

Модификатор **sealed** в объявление суперкласса, и он ограничит возможность создания подклассов

```
sealed class Expr {     // запечатанный класс
    class Num(val value: Int) : Expr()
      //подклассы в виде вложенных классов
    class Sum(val left: Expr, val right: Expr) : Expr()
}
```

Модификатор sealed означает, что класс по умолчанию открыт, добавлять модификатор open не требуется.





Запечатанный класс не может иметь наследников, объявленных вне класса.

14.6 Конструкторы классов

Как известно, в Java-классе можно объявить один или несколько конструкторов. Kotlin поддерживает всё то же самое, кроме одного: он различает **основной (главный) конструктор** (который, как правило, является главным лаконичным средством инициализации класса и объявляется вне тела класса) и **вторичный (вспомогательный) конструктор** (который объявляется в теле класса). Kotlin также позволяет вставлять дополнительную логику инициализации в соответствующие блоки.

Объявим простой класс

```
class Student(val nickname: String)
```

Этот блок кода, окруженный круглыми скобками, называется **основным конструктором (primary constructor)**. Он преследует две цели: определение параметров конструктора и определение свойств, которые инициализируются этими параметрами.

Вот так выглядит явный код, который делает то же самое:

```
class Student constructor(_nickname: String) {
 // Основной конструктор
    val nickname: String // с одним параметром
    init {   // блок инициализации
          nickname = _nickname
    }
}
```

С ключевого слова **constructor** начинается объявление основного или вторичного конструктора. Ключевое слово **init** обозначает начало блока инициализации. Такие блоки содержат код инициализации, который выполняется при создании каждого экземпляра класса, и предназначены для использования вместе с первич-

ными конструкторами. Блоки инициализации приходится использовать, поскольку синтаксис первичного конструктора ограничен и он не может содержать кода инициализации.

В этом примере инициализирующий код можно совместить с объявлением свойства nickname, поэтому его не нужно помещать в блок инициализации. В отсутствие аннотаций и модификаторов видимости основного конструктора ключевое слово constructor также можно опустить:

```
class Student(_nickname: String) {
    val nickname = _nickname
}
```

Это ещё один способ объявления того же класса.

Обратите вниамание временным переменным, в том числе параметрам, которые используются только один раз, часто дают имена, начинающиеся с подчеркивания. Это показывает, что они одноразовые.

Если свойство инициализируется соответствующим параметром конструктора, код можно упростить, поставив ключевое слово val перед параметром. Такое определение параметра заменит объявление свойства в теле класса:

```
class Student(val nickname: String)
//означает, что для параметра должно быть создано соответствующее свойство
```

Все вышеперечисленные объявления класса эквивалентны, но последнее имеет самый лаконичный синтаксис. Параметрам конструктора и параметрам функций можно назначать значения по умолчанию.

Для каждого параметра конструктора вы указываете, будет ли он только для чтения или нет. Определяя параметры в конструкторе с помощью ключевых слов val и var, вы объявляете соответствующие свойства класса val или var, а также параметры, для которых конструктор будет ожидать аргументы. Вы также неявно присваиваете каждому свойству значение, которое передается в качестве аргумента.

Чтобы создать экземпляр класса, нужно вызвать конструктор напрямую, без ключевого слова new.

```
val alice = Student("Alesia")
```

Если все параметры конструктора будут иметь значения по умолчанию, компилятор сгенерирует дополнительный конструктор без параметров, использующий все значения по умолчанию.

Если вообще не объявить никакого конструктора, компилятор добавит конструктор по умолчанию, который ничего не делает:

```
open class Button //Будет сгенерирован конструктор  по
умолчанию без аргументов
```

Если вы захотите унаследовать класс Button в другом классе, не объявляя своих конструкторов, вы должны будете явно вызвать конструктор суперкласса, даже если тот не имеет параметров:

```
class RadioButton: Button()
```

Интерфейсы не имеют конструктора, поэтому при реализации интерфейса никогда не приходится добавлять круглые скобки после его имени в списке супертипов.

Конструкторы можно делать приватными

```
class Config private constructor() {}
```

Классы с несколькими конструкторами встречаются в Kotlin значительно реже, чем в Java. В большинстве ситуаций, когда в Java нужны перегруженные версии конструктора, в языке Котлин можно воспользоваться значениями параметров по умолчанию и синтаксисом именованных аргументов.

```
open class View {
    constructor(ctx: Context) {}  //  Вторичный конструктор
    constructor(ctx: Context, attr: AttributeSet) {}
 // Вторичный конструктор
}
```

Этот класс не имеет основного конструктора (это видно по отсутствию круглых скобок после имени в определении класса), зато у него два вторичных конструктора. **Вторичный конструктор** объявляется с помощью ключевого слова constructor. Вы можете объявить столько вторичных конструкторов, сколько нужно.

Как в Java, в Kotlin есть возможность вызвать один конструктор класса из другого (**делегирование**) с помощью ключевого слова this() .

```
class MyButton : Button {
    constructor(ctx: Context): this(ctx,  STYLE) {}
    // Делегирует выполнение другому конструктору
    constructor(ctx: Context, attr: AttributeSet):
super(ctx, attr) {}
}
```

Если класс не имеет основного конструктора, каждый вторичный конструктор должен либо инициализировать базовый класс, либо делегировать выполнение другому конструктору.

Во вторичном конструкторе в отличие от главного конструктора, нельзя объявить свойства. *Свойства класса можно объявить только в главном конструкторе или на уровне класса*.

Совместимость с Java - основная причина применения вторичных конструкторов.

Объявляя конструктор, также можно указать значения по умолчанию, которые получат параметры в отсутствие соответствующих им аргументов.

```
class Student(val nickname: String = "No name")
```

Kotlin позволяет использовать в вызове конструктора именованные аргументы, аналогичные именованным аргументам в вызовах функций.

14.7. Реализация свойств интерфейсов

Рассмотрим пример

```
interface Human { val nickname: String }
```

Каждый из этих классов реализует абстрактное свойство интерфейса по-своему.

```
class PrivateUser(override val nickname: String) : Human
// Свойство основного конструктора
```

Можно использовать лаконичный синтаксис объявления свойства непосредственно в основном конструкторе. Ключевое слово override перед ним означает, что это свойство реализует абстрактное свойство интерфейса.

```
class SubscribingUser(val email: String) : Human {
    override val nickname: String
        get() = email.substringBefore('@')
    // метод чтения
}
```

В классе SubscribingUser свойство nickname реализуется с помощью метода доступа. У этого свойства отсутствует поле для хранения значения - есть только метод чтения, определяющий имя по адресу электронной почты.

```
class FacebookUser(val accountId: Int) : Human {
    override val nickname = getName(accountId) // Инициа-
лизация
}
```

В классе FacebookUser значение присваивают свойству nickname в коде инициализации. Здесь определено поле, которое хранит значение, вычисленное во время инициализации класса.

Доступ к полю из метода записи

```
class  SiteUser {
    val email: String = "tut@tut.by"
    var nickname: String = "No"
        get() = email.substringBefore('@')
        set(value: String) {
            field = value
        }
    }
```

В теле метода записи для доступа к значению поля используется специальный **идентификатор field**.

Способ обращения к свойству не зависит от наличия у него отдельного поля. *Компилятор сгенерирует такое поле для свойства*, если вы будете явно ссылаться на него или использовать реализацию методов доступа по умолчанию. Если вы определите собственные методы доступа, не использующие идентификатора field (в методе чтения, если свойство объявлено как val, или в обоих методах, если это изменяемое свойство), тогда отдельное поле не будет сгенерировано.

Изменяем значение свойства

```
val tempuser = SiteUser();
tempuser.nickname = "ID123"
```

Свойство объявленное как val доступно только для чтения и не может быть изменено даже методом записи. Это защищает значения val от изменений без вашего согласия. var-свойства имеют методы записи, — вне зависимости от того, объявляете вы для них свое поведение или нет.

По умолчанию методы доступа имеют ту же видимость, что и свойство. Но вы можете изменить её, добавив модификатор видимости перед ключевыми словами get и set.

```
class LengthCounter {
    var counter: Int = 0
        private set
}
```

Область видимости методов свойства не может быть шире области видимости самого свойства. Можно ограничить доступ к свойству, определив более узкую область видимости методов чтения, но нельзя присвоить методам свойства более широкую область видимости, чем объявлена для самого свойства.

Свойства должны инициализироваться при объявлении

Как в Java, все классы в Kotlin имеют несколько методов, которые можно переопределять: **equals, hashCode и toString**. В Kotlin оператор == представляет способ сравнения двух объектов по умолчанию: он сравнивает их, вызывая за кулисами метод equals.То есть, если вы переопределили метод equals в своем классе, можете сравнить экземпляры с помощью оператора ==.

14.8 Классы данных

В Kotlin вам не придется создавать toString, equals и hashCode. Добавьте в объявление класса модификатор data - и все необходимые методы появятся автоматически.

```
data class Client(val name: String, val postalCode: Int)
```

Чтобы ещё упростить использование классов данных в качестве неизменяемых объектов, компилятор Kotlin генерирует для них метод, который позволяет **копировать экземпляры**. Копия имеет собственный жизненный цикл и не влияет на код, ссылающийся на исходный экземпляр.

```
class Client(val name: String, val postalCode: Int) {
    fun copy(name: String = this.name,
             postalCode: Int = this.postalCode)
                    = Client(name, postalCode)
}
```

Еще одно преимущество классов данных — поддержка **автоматической деструктуризации данных**. Под внешней оболочкой деструктуризация опирается на функции с именами component1, component2 и т. д. Каждая функция предназначена для извлечения части данных, которую вы хотите вернуть.

```
data class Gamer (val experience: Int, val level:Int )
fun main(args: Array<String>) {
```

```
 val (exp, lev) = Gamer(12, 1)
}
```

Можно сделать любой класс деструктурированным, добавив функцию-оператор component

```
class PlayerGame(val experience: Int, val level:Int ){
    operator fun component1() = experience
    operator fun component2() = level
}

fun main(args: Array<String>) {

    val (experience, level) = PlayerGame(1250, 5)
}
```

Для классов данных существует ряд ограничений:

- должны иметь хотя бы один параметр в главном конструкторе;
- требуют, чтобы параметры главного конструктора объявлялись как var или val;
- не могут иметь модификаторы abstract, open, sealed, inner.

Если вашему классу не требуются функции toString, copy, equals или hashCode, тогда его объявление как класса данных не даст никаких преимуществ. Если вам нужна своя реализация equals, использующая только определенные свойства для сравнения, а не все, классы данных вам не подойдут, потому что они включают все свойства в автоматически сгенерированную функцию equals.

14.9 Блок инициализации

Блок инициализации — это способ настроить переменные или значения, а также произвести их проверку, то есть убедиться, что конструктору переданы допустимые аргументы. Код в блоке инициализации выполняется сразу после создания экземпляра класса.

```
class Player(_name: String,
             var healthPoints: Int = 100,
             var level: Int = 1) {
    var name = _name
        get() = field.capitalize()
        private set(value) {
```

```
            field = value.trim()
        }
    init {
  require(healthPoints > 0, { "healthPoints must be
greater than zero." })
  require(name.isNotBlank(), { "Player must have a name."
})
    }
    constructor(name: String, _level:Int) : this(name) {
        if (name.toLowerCase() == "kar")
            healthPoints = 40
        level = _level;
    }
}
```

Если хотя бы одно из условий не выполнится, будет возбуждено исключение IllegalArgumentException

Одно и то же свойство может использоваться в нескольких инициализациях, поэтому порядок их выполнения очень важен.

Инициализация выполняется в следующем порядке:

1. Создаются и инициализируются свойства, встроенные в объявление главного конструктора.

2. Выполняются операции присваивания на уровне класса.

3. Выполняется блок init, присваивающий значения свойствам и вызывающий функции.

4. Выполняется тело вспомогательного конструктора, в котором присваиваются значения свойствам и вызываются функции.

Порядок выполнения блока init (пункт 3) и операций присваивания на уровне класса (пункт 2) зависит от порядка, в котором они указаны. Если блок init поместить после операций присваивания значений свойствам на уровне класса, он выполнится после них.

Поздняя инициализация

Класс Activity в Android представляет экран приложения. Вы не контролируете момент, когда именно будет вызван конструктор Activity. Зато известно, что самая ранняя точка выполнения — это функция с именем onCreate.

Это та ситуация, когда важное значение приобретает поздняя инициализация, и это нарушение правил инициализации компилятора Kotlin.

В любое **объявление var-свойства можно добавить ключевое слово lateinit.** Тогда компилятор Kotlin позволит отложить инициализацию свойства до того момента, когда такая возможность появится.

```
class Player {
    lateinit var alignment: String
    fun determineFate() {
        alignment = "Good"
    }
    fun proclaimFate() {
        if (::alignment.isInitialized) println(alignment)
    }
}
```

Если переменная с поздней инициализацией получит начальное значение до первого обращения к ней, проблем не будет. Но если сослаться на такое свойство до его инициализации, вас ждет исключение UninitializedPropertyAccessException. Эту проблему можно решить, использовав тип с поддержкой null, но тогда вам придется обрабатывать возможное значение null по всему коду.

Ключевое слово **lateinit** действует как негласный договор: «Я обязуюсь инициализировать эту переменную до первой попытки обратиться к ней». Kotlin предоставляет инструмент для проверки факта инициализации таких переменных: метод isInitialized, показанный в примере выше. Вы можете вызывать isInitialized каждый раз, когда есть сомнения, что переменная lateinit инициализирована, чтобы избежать UninitializedPropertyAccess Exception. Тем не менее isInitialized следует использовать экономно.

Отложенная инициализация

Можно задерживать инициализацию переменной до первого обращения к ней. Эта идея известна как отложенная инициализация. Отложенная инициализация реализована в Kotlin через механизм делегатов.

Стандартная библиотека Kotlin включает несколько готовых делегатов, например **lazy**. Делегат lazy принимает лямбда-выражение с кодом, который вы бы хотели выполнить для инициализации свойства.

```
class Team(val name: String){
    var hometown =  lazy { selectHometown() }
    private fun selectHometown() = File("towns.txt")
                                   .readText()
                                   .split("\n")
                                   .shuffled()
                                   .first()
}
```

Свойство hometown остается неинициализированным до первого обращения к нему. В этот момент выполнится код лямбды в lazy. Код выполнится только один раз, при первом обращении к делегированному свойству. Все дальнейшие обращения к этому свойству будут использовать ранее полученный результат вместо повторных вычислений.

14.10 Наследование классов

Помним, чтобы класс можно было унаследовать, его надо отметить ключевым словом open.

```
open class  SiteUser (email: String) {
    val email: String = "tut@tut.by"
    var nickname: String = "No"
        get() = email.substringBefore('@')
        set(value: String) {
            field = value
        }
    }
class AdminSiteUser : SiteUser ("user@bstu.by")
```

Вызов конструктора указывает на то, какой родительский конструктор нужно вызвать. Ключевым словом open нужно отметить не только класс, но также функцию, чтобы ее можно было переопределить.

Переопределим protected свйоство

```
open class  SiteUser (email: String) {
    protected open val group = 1;
    }

class AdminSiteUser : SiteUser ("user@bstu.by"){
    override val group = super.group +5
}
```

14.12 Проверка типа

```
fun main(args: Array<String>) {
var admin = AdminSiteUser()
var className = when(admin) {
    is SiteUser -> "SiteUser"
    is AdminSiteUser -> "AdminSiteUser"
    else -> throw IllegalArgumentException()
}
```

```
print(className)
}
```

Вернет SiteUser.

Первое условие в выражении when определяется как истинное. Второе условие тоже истинное. Но для вас это уже неважно, так как первое условие уже удовлетворено и второе проверяться просто не будет.

14.13 Ключевое слово object

Ключевое слово object используется в языке Kotlin в разных случаях, которые объединены общей идеей: это ключевое слово одновременно *объявляет класс и создает его экземпляр* (другими словами, объект).

Рассмотрим различные ситуации, когда оно используется:

1)объявление объекта как способ реализации шаблона «Одиночка»;

2) реализация объекта-компаньона, содержащего лишь фабричные методы, а также методы, связанные с классом, но не требующие обращения к его экземпляру. К членам такого объекта можно обращаться просто по имени класса;

3) запись объекта-выражения, которое используется вместо анонимного внутреннего класса Java.

Singtone

Объявление объекта начинается с ключевого слова object и фактически определяет класс с переменной этого класса в одном выражении. По аналогии с классом объявление объекта может содержать определения свойств, методов, блоков инициализации и т. д. Единственное, что не допускается, - конструкторы, основные или вторичные.

```
object Payroll {
    val allEmployees = arrayListOf<Client>()
    fun calculateSalary() {
        for (human in allEmployees) {
            //...
        }
    }
}
```

Как и обычные переменные, объявления объектов позволяют вызывать методы и обращаться к свойствам с помощью имени объекта слева от символа .:

```
fun main(args: Array<String>) {
    Payroll.allEmployees.add(Client("Pet", 220050))
    Payroll.calculateSalary()
```

```
}
```

Объекты также можно объявлять в классах. Такие объекты существуют в единственном числе - у вас не будет отдельного объекта для каждого экземпляра содержащего его класса.

```
data class Card(val name: String) {
    object NameComparator : Comparator<Card> {
        override fun compare(p1: Card, p2: Card): Int =
            p1.name.compareTo(p2.name)
    }
}
```

Объекты –компаньоны

Классы в Kotlin *не могут иметь статических членов*; ключевое слово static, имеющееся в Java, не является частью языка Kotlin. В качестве замены Kotlin используются функции уровня пакета (которые во многих ситуациях могут заменить статические методы Java) и объявления объектов (которые заменяют статические методы Java в других случаях, наряду со статическими полями).

Один из объектов в классе можно отметить специальным ключевым словом: companion. Сделав это, вы сможете обращаться к методам и свойствам такого объекта непосредственно через имя содержащего его класса, без явного указания имени объекта. В итоге синтаксис выглядит почти так же, как вызов статического метода в Java.

```
class Some {
    companion object {
        fun bar() {
            println("Companion object called")
        }
    }
}
fun main(args: Array<String>) {
    Some.bar();
}
```

Объекты-компаньоны не могут переопределяться в подклассах.

Но может иметь имя, реализовать интерфейс или обладать функциями-расширениями и свойствами-расширениями.

Объект-компаньон класса компилируется так же, как обычный объект: статическое поле класса ссылается на собственный экземпляр.

Но вам может понадобиться работать с Java-кодом, который требует, чтобы методы вашего класса были статическими. Для этого добавьте перед соответствующим методом аннотацию @JvmStatic. Если понадобится объявить статическое поле, используйте аннотацию @JvmField перед свойством верхнего уровня или свойством, объявленным в объекте. Эти аннотации нужны только для совместимости и, строго говоря, не являются частью ядра языка.

```
class Some {
    companion object {
        @JvmStatic
        fun bar() {
            println("Companion object called")
        }
        @JvmField
        val name = "Companion"
    }
}
```

Как любые другие объявления объектов, объекты-компаньоны могут реализовать интерфейсы.

Объекты-выражения

Ключевое слово object можно использовать не только для объявления именованных объектов-одиночек, но и для создания анонимных объектов. Анонимные объекты заменяют анонимные внутренние классы в lava.

Реализация обработчика событий с помощью анонимного объекта

```
window.addMouseListener(
object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {

    }
    ... )

    override fun mouseEntered(e: MouseEvent) {
        // ...
        }
}
)
```

Синтаксис ничем не отличается от объявления объекта, за исключением указания его имени (здесь оно отсутствует). Объект-выражение объявляет класс и создает экземпляр этого класса, но не присваивает имени ни классу, ни экземпляру. Как правило, в этом нет необходимости, поскольку объект используется в качестве параметра вызова функции.

Если объекту потребуется дать имя, его можно сохранить в переменной:

```
val listener = object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) (...)
    override fun mouseEntered(e: MouseEvent) (...)
}
```

В отличие от анонимных внутренних классов Java, которые могут наследовать только один класс или реализовать только один интерфейс, анонимный объект Kotlin может реализовывать несколько интерфейсов или вовсе ни одного.

Как анонимные классы в Java, код в объекте-выражении может обращаться к переменным в функциях, где он был создан. Но, в отличие от Java, это не ограничено переменными с модификатором final.

14.14 Перегрузка операторов

Как известно, некоторые особенности языка Java тесно связаны с определенными классами в стандартной библиотеке. Например, объекты, реализующие интерфейс java. lang. Iterable, можно использовать в циклах for, а объекты, реализующие интерфейс j ava. lang. AutoCloseable, можно использовать в инструкциях try-with-resources. В Kotlin есть похожие особенности: некоторые конструкции языка вызывают функции, определяемые в вашем коде. Но в Kotlin они связаны не с определенными типами, а с функциями, имеющими специальные имена. Такой подход в Kotlin называется **соглашениями**.

Множество интерфейсов, реализованных Java-классами, фиксировано, и Kotlin не может изменять существующих классов, чтобы добавить в них реализацию дополнительных интерфейсов. С другой стороны, благодаря механизму функций-расширений есть возможность добавлять новые методы в классы. Вы можете определить любой метод, описываемый соглашениями, как расширение и тем самым адаптировать любой существующий Java-класс без изменения его кода.

Перегрузка арифметических операторов

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}
```

```
fun main(){
    println ((Point(10, 20) + Point(30, 40)).toString())
}
    // Point(x=40, y=60)
```

Объявить оператор можно не только как функцию-член, но также как функцию-расширение.

```
operator fun Point.div(other: Point): Point {
    return Point(x /other.x, y / other.y)
}
```

Kotlin ограничивает набор операторов, доступных для перегрузки. Каждому такому оператору соответствует имя функции, которую нужно определить в своем классе.

Выражение	Имя функции
a * b	times
a / b	div
a % b	mod
a + b	plus
a - b	minus

Операторы для любых типов следуют тем же правилам приоритета, что и для стандартных числовых типов. Например, в выражении a + b * c умножение всегда будет выполняться перед сложением, даже если это ваши собственные версии операторов. Операторы *, / и % имеют одинаковый приоритет, который выше приоритета операторов + и -.

Перегруженные операторы Kotlin вполне доступны в Java-коде: так как каждый перегруженный оператор определяется как функция, их можно вызывать как обычные функции, используя полные имена.

Определяя оператор, *необязательно использовать одинаковые типы* для операндов.

Тип значения, возвращаемого функцией-оператором, также *может отличаться* от типов операндов.

В Kotlin *отсутствуют любые поразрядные* (битовые) операторы для стандартных числовых типов - и вследствие этого отсутствует возможность определять их для своих типов. Для этих целей используются обычные функции, поддерживающие инфиксный синтаксис вызова.

Составные операторы

Обычно, когда определяется оператор plus, Kotlin начинает поддерживать не только операцию +, но и +=.

Если определить функцию с именем plusAssign, возвращающую значение типа Unit, Kotlin будет вызывать её, встретив оператор +=. Другие составные бинарные операторы выглядят аналогично: minusAssign, timesAssign и т. д.

Перегрузка унарных операторов

Процедура перегрузки унарного оператора ничем не отличается от описанной выше: объявляется функция (член или расширение) с предопределенным именем, которая затем отмечается модификатором operator.

```
operator fun Point.unaryMinus(): Point {
    return Point(-x, -y)
}
```

Выражение	Имя функции
+a	unaryPlus
-a	unaryMinus
!a	not
++a, a++	inc
--a, a--	dec

Когда определяются функции inc и dec для перегрузки операторов инкремента и декремента, компилятор автоматически поддерживает ту же семантику пред- и постинкремента, что и для обычных числовых типов.

```
operator fun BigDecimal.inc() = this + BigDecimal.ONE

fun main(){
     println ((Point(10, 20) + Point(30, 40)).toString())

    var bd = BigDecimal.ZERO
    println(bd++) // 0
    println(++bd) //2
```

Перегрузка операторов сравнения

В языке Kotlin оператор == транслируется в вызов метода equals. Это лишь одно из применений принципа соглашений.

a == b	a?.equals(b) ?: b === null
a != b	!(a?.equals(b) ?: b === null)

Операторы отношений

Kotlin поддерживает е интерфейс Comparable. Но метод compareTo этого интерфейса может вызываться по соглашениям и используется операторами сравнения (<, >, <= и >=), которые автоматически транслируются в вызовы compareTo. Значение, возвращаемое методом compareTo, должно иметь тип Int. Выражение p1 < p2 эквивалентно выражению p1.compareTo(p2) < 0. Другие операторы сравнения работают аналогично.

```
class Person( val firstName: String, val lastName: String
) : Comparable<Person> {
    override fun compareTo(other: Person): Int {
        return compareValuesBy(this, other,
            Person::lastName, Person::firstName)
    }
}
 fun  main (){
     val p1 = Person("Alice", "Smith")
     val p2 = Person("Bob", "Johnson")
     println(p1 < p2) // false
 }
```

Чтобы упростить реализацию compareTo, использовали функцию *compareValuesBy из стандартной библиотеки* Kotlin. Она принимает список функций обратного вызова, результаты которых подлежат сравнению. Каждая из этих функций по очереди вызывается для обоих объектов, сравниваются полученные значения, и возвращается результат. Если значения первой пары отличаются, возвращается результат их сравнения. Если они равны, вызывается следующая функция, и сравниваются её результаты для каждого из объектов - или, если не осталось других функций, возвращается 0. В качестве функций обратного вызова можно передавать лямбда-выражения или, как в данном примере, ссылки на свойства.

Соглашения для коллекций и диапазонов

К наиболее распространенным операциям для работы с коллекциями относятся операции извлечения и изменения значений элементов по их индексам, а

также операция проверки вхождения в коллекцию. Все такие операции поддерживают синтаксис операторов: чтобы прочитать или изменить значение элемента по индексу, используется синтаксис a[n] (называется оператором индекса).

a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

Для проверки вхождения элемента в коллекцию или в диапазон, а также для итераций по коллекциям можно использовать оператор in. Эти операции можно добавить в свои классы, действующие подобно коллекциям.

a..b	a.rangeTo(b)
a in b	b.contains(a)
a !in b	!b.contains(a)

Делегирование свойств

```
class Foo { var p: String by Delegate() }

class Delegate {
    operator fun getValue(...) { }
    operator fun setValue(...) {}
}
```

Свойство p делегирует логику своих методов доступа другому объекту: в данном случае новому экземпляру класса Delegate. Экземпляр создается выражением, следующим за ключевым словом by, и может быть чем угодно, удовлетворяющим требованиям соглашения для делегирования свойств.

Компилятор создаст скрытое вспомогательное свойство, инициализированное экземпляром объекта-делегата, которому делегируется логика работы свойства p.

В соответствии с соглашением класс Delegate должен иметь методы getValue и setValue (последний требуется только для изменяемых свойств). Как обычно, они могут быть членами или расширениями.

Свойство p можно использовать как обычно, но за кулисами операции с ним будут вызывать методы вспомогательного свойства типа Delegate. Это используется для поддержки отложенной инициализации в библиотеке.

Делегирование свойств дает возможность повторно использовать логику хранения значений свойств, инициализации, чтения и изменения. Это очень мощный механизм для разработки фреймворков.