Лабораторная работа №7: Наследование и полиморфизм

Цель: Познакомиться с основами **наследования** в языке Kotlin и научиться - создавать базовые (родительские) и производные (наследники) классы; применять наследование для расширения логики игры; использовать переопределение методов (полиморфизм).

Шаг 1: Введение в наследование

Наследование — это основа ООП

Наследование позволяет создавать **новые классы** на основе уже существующих. Это помогает **повторно использовать код** и **организовать логику** по иерархии.

Пример в теории: Животные

- Есть базовый класс Животное, который описывает общие свойства: *дышит*, *спит*, *передвигается*.
- От него можно **унаследовать** более конкретные классы: Кошка, Собака, Птица.
- Каждое животное автоматически получает поведение базового класса, но также может иметь **свои уникальные особенности**: *мяукает*, *лает*, *летает*.

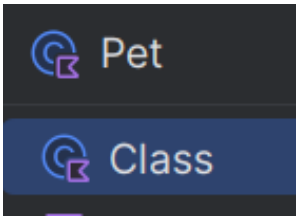
Что даёт наследование:

Преимущество	Описание
Повторное использование	Один раз описали — используем во всех потомках
Логика и структура	Код становится понятнее, легче поддерживать
Расширяемость	Легко добавлять новые типы животных без дублирования

Создайте пакет pets:

ПКМ по папке **src → New → Package → введите pets**.

Внутри создадим новый класс **Pet:**



Чтобы мы могли унаследоваться от нашего класса мы должны перед словом **class** ключевое слово **open**:

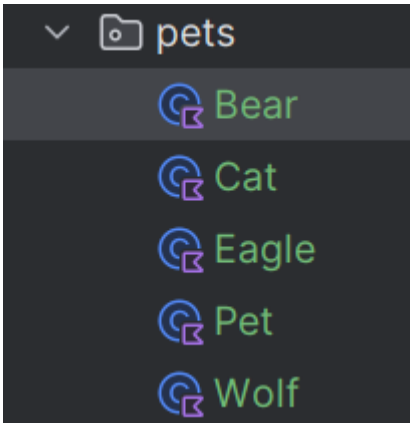
```
open class Pet {
```

Пока что не будем использовать конструкторы, добавим свойства со значениями по умолчанию:

```
val name: String = ""
val maxHealth: Int = 0
val speed: Int = 0
```

- **name** — кличка питомца
- **maxHealth** — максимальное здоровье
- **speed** — скорость

Далее:



Создайтете 4 класса в пакете **pets** которым присвойте свои свойства:

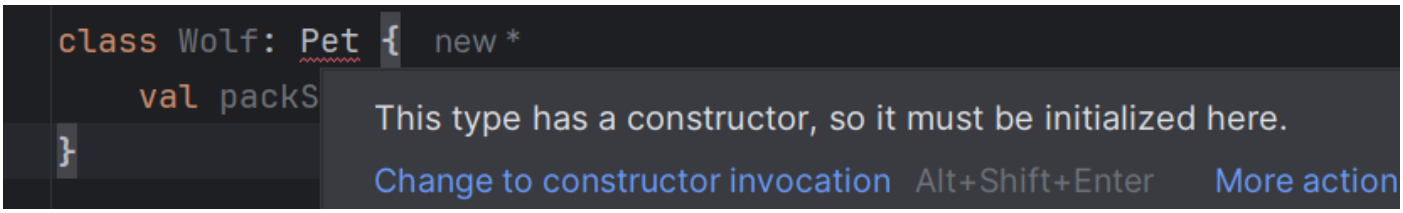
- **Wolf** – размер стаи (**packSize**)
- **Cat** - скрытность (**stealthLevel**)
- **Eagle** - высота полета (**flightHeight**)
- **Bear** - сила (**strength**)

Для наследования от класса **Pet**:

- После имени класса-наследника поставьте двоеточие
- Укажите Pet для вызова конструктора родителя

```
class Wolf: Pet
```

При появлении ошибки "This type has a constructor...":

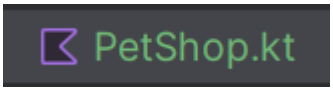


- Несмотря на то, что мы не объявляли конструктор явно, в Kotlin всегда существует конструктор по умолчанию
- Для его вызова добавьте пустые круглые скобки после Pet()

```
class Wolf: Pet()
```

Сделайте для остальных классов тоже самое наследование.

Для работы с питомцами Создайтете новый файл **PetShop.kt** в нашем пакете **pets**:



Добавьте точку входа - функцию **main**():

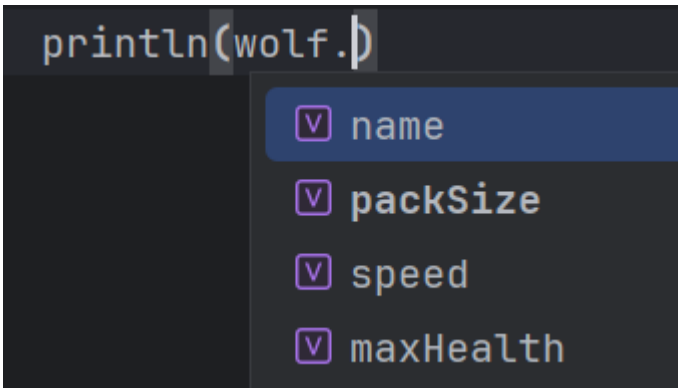
```
package pets

fun main() {

}
```

Внутри функции:

- Создайтете экземпляр класса **Wolf**
- Выведите значение его свойства **packSize**

```
val wolf = Wolf()
println(wolf.packSize)
```

- Обратите внимание, что доступны и свойства родительского класса (name, speed, maxHealth) благодаря механизму наследования



✓ Перенесите свойства в первичный конструктор:

1. Наведите курсор на класс
2. Нажмите **Alt+Enter**
3. Выберите "Move to constructor"

```
open class Pet(  4 Usages   4
    val speed: Int = 0,
    val name: String = "",
    val maxHealth: Int = 0
) {
}
```

✓ Удалите фигурные скобки, если класс пустой

✓ Уберите значения по умолчанию, чтобы они задавались при создании экземпляра

```
open class Pet(  4 Usages
    val speed: Int,
    val name: String,
    val maxHealth: Int
)
```

Измените классы-наследники, передавая конкретные значения в конструктор родителя, например:

```
class Wolf(name: String) : Pet(name = name, speed = 100, maxHealth = 250)
class Cat(name: String) : Pet(name = name, speed = 80, maxHealth = 150)
class Eagle(name: String) : Pet(name = name, speed = 250, maxHealth = 50)
class Bear(name: String) : Pet(name = name, speed = 50, maxHealth = 550)
```

Когда мы наследуем класс в Kotlin и передаём параметры его конструктора, **мы не указываем val или var в конструкторе подкласса**, потому что эти параметры **не становятся свойствами подкласса** — они просто передаются дальше в родительский класс.

В родительском классе **Pet**(...) параметры **speed**, **name**, **maxHealth** уже **объявлены как свойства** через **val**. В подклассе **Wolf**(...) мы лишь **передаём значения** в конструктор **Pet**(...), чтобы он их инициализировал. Указывать **val** повторно было бы попыткой создать **новые свойства в подклассе**, что здесь **не требуется**, потому что эти свойства уже есть в родителе.

Задание:

1. В классе **Pet** добавьте два новых метода:
    - **describe**() - выводит информацию о питомце (имя, здоровье, скорость)
    - **makeSound**() - выводит сообщение о том, что питомец подает голос
2. В файле **PetShop.kt** объявите список **pets**, содержащий:
    - Волка с именем "**Фенрир**"
    - Кота с именем "**Мурзик**"
    - Орла с именем "**Скайвинд**"
    - Медведя с именем "**Балу**"
3. Используя цикл **for**, для каждого питомца:
    - Вызовите метод **describe**()
    - Вызовите метод **makeSound**()

Решение:

Класс **Pet**:

```
fun describe() {  1 Usage   new *
    println("Питомец: $name, Здоровье: $maxHealth, Скорость: $speed")
}

fun makeSound() {  1 Usage   new *
    println("$name подаёт голос.")
}
```

Файл **PetShop.kt**:

```
val pets = listOf(
    Wolf( name = "Фенрир"),
    Cat( name = "Мурзик"),
    Eagle( name = "Скайвинд"),
    Bear( name = "Балу")
)

for (pet in pets) {
    pet.describe()
    pet.makeSound()
    println()
}
```

Шаг 2: Расширение иерархии классов

2.1. Расширяем пакет с заданиями

Создайте в пакете **world** базовый класс **Mission** со следующими характеристиками:

- Свойства:
  - **title: String** (название миссии)
  - **reward: Int** (вознаграждение)
- Класс должен быть открыт для наследования (**open**)

2. Рефакторинг существующих классов

Модифицируйте классы **Quest** и **Contract** так, чтобы они наследовались от **Mission**:

- Уберите дублирующиеся свойства (**title**, **reward**)
- В конструкторах передавайте эти параметры в родительский класс

3. Создание нового класса

Добавьте класс **SpecialOperation**, который:

- Наследуется от **Mission**
- Добавляет новые свойства:
  - **requiredClearance**: **Int** (уровень допуска)
  - **isCovert**: **Boolean** (секретность операции)
- Содержит метод **showReward**() для вывода дополнительной информации

4. Проверка работы

В файле **Guild.kt** в функции **main**():

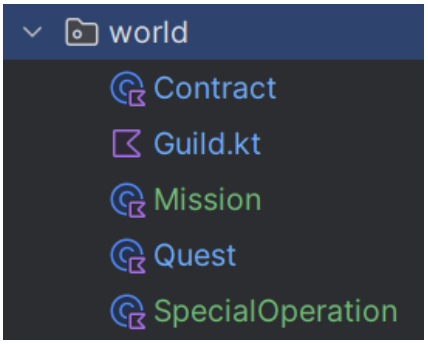
1. Создайтете по одному объекту каждого типа:
   - **Quest**
   - **Contract**
   - **SpecialOperation**
2. Выведите информацию о каждом объекте, демонстрируя:
   - Доступ к унаследованным свойствам
   - Уникальные свойства каждого класса
   - Работу метода **showReward**() для **SpecialOperation**

Пример вывода:

```
Информация о квесте:
Название: Поиск артефакта, Награда: 800

Информация о контракте:
Название: Защита каравана, Заказчик: Гильдия купцов

Информация о спецоперации:
Название: Операция 'Тень'
Требуемый допуск: 2
Режим секретности: Совершенно секретно
```

Решение:

В пакете **world** создаём два класса **Mission** и **SpecialOperation**:



Класс миссии:

```
open class Mission(  3 Usages   3 Inheritors   new *
    val title: String,  // Название миссии
    val reward: Int     // Награда
)
```

Класс специальной операции:

```
class SpecialOperation(  1 Usage   new *
    title: String,
    reward: Int,
    val requiredClearance: Int,
    val isCovert: Boolean
) : Mission(title, reward) {

    fun showReward() {  1 Usage   new *
        println("Требуемый допуск: $requiredClearance")
        println("Режим секретности: ${if (isCovert) "Совершенно
         секретно" else "Обычный"}")
    }
}
```

Изменения в классе с квестами:

```
class Quest(  3 Usages   Denis *
    title: String,
    val duration: Int,
    reward: Int,
    val difficulty: String
) : Mission(title, reward) {
```

Изменения в классе с контрактом:

```
class Contract(  1 Usage   Denis *
    title: String, // Берем из Mission
    val clientName: String,
    val taskDescription: String,
    reward: Int,
    val isUrgent: Boolean = false
) : Mission(title, reward) {
```

Вывод в **Guild.kt:**

```
val quest = Quest( title = "Поиск артефакта", duration = 3, reward = 800,
 difficulty = "Средний")
val contract = Contract( title = "Защита каравана",
 clientName = "Гильдия купцов",
    taskDescription = "Охрана груза", reward = 1200)
val specialOp = SpecialOperation( title = "Операция 'Тень'",
 reward = 2500, requiredClearance = 2, isCovert = true)

println("Информация о квесте:")
println("Название: ${quest.title}, Награда: ${quest.reward}")

println("\nИнформация о контракте:")
println("Название: ${contract.title}, Заказчик: ${contract
 .clientName}")

println("\nИнформация о спецоперации:")
println("Название: ${specialOp.title}")
specialOp.showReward()
```

2.2. Расширяем пакет с заклинаниями

Далее практика **наследования в системе заклинаний.**

Задание

1. Создание базового класса

Создайтете класс **MagicEffect** в пакете **magic** со свойствами:

- **name** - название эффекта

- **symbol** - символ для визуализации
- **duration** - длительность действия (секунды)

2. Рефакторинг класса Spell

Модифицируйте класс **Spell** так, чтобы:

- Он наследовался от **MagicEffect**
- Оставлял только уникальные свойства (**width**, **height**)

3. Создание нового класса

Добавьте класс **InstantSpell** для мгновенных заклинаний:

- Наследуется от **MagicEffect**
- Не имеет длительности (**duration** = 0)
- Добавляет свойство **power** (сила эффекта)

4. Проверка работы

В функции **main**() в файле **CastleMagic** продемонстрируйте:

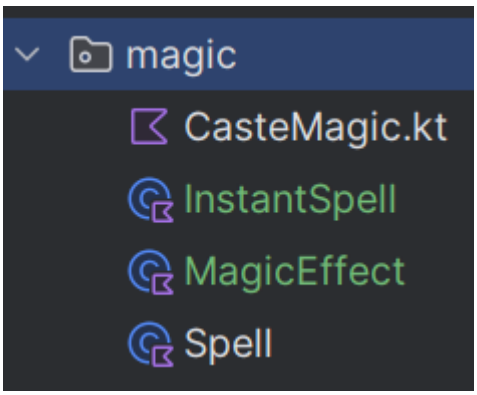
1. Работу оригинального класса **Spell**
2. Создание и применение **InstantSpell**

Пример **вывода**:

```
Заклинание: Огненный шар
Символ: 🔥
Длится: 10 сек

Мгновенное заклинание: Лечение
Символ: 💚
Сила: 5
Длится: 0 сек
```

Решение:

В пакете **magic** создаём два класса **MagicEffect** и **InstantSpell**:



Класс **MagicEffect:**

```
open class MagicEffect(
    val name: String,
    val symbol: String,
    val duration: Int
)
```

Добавляем наследование в класс **Spell**:

```
class Spell(  3 Usages   Denis *
    name: String,
    val width: Int = 0,
    val height: Int = 0,
    symbol: String
) : MagicEffect(name, symbol,  duration = 10) {
```

Добавляем новый класс **InstantSpell:**

```
class InstantSpell(  1 Usage   new *
    name: String,
    symbol: String,
    val power: Int
) : MagicEffect(name, symbol,  duration = 0)
```

Вывод в **CastleMagic.kt:**

```
val fireball = Spell( name = "Огненный шар",  width = 3,  height = 3,
  symbol = "\uD83D\uDD25")
val heal = InstantSpell( name = "Лечение",
  symbol = "\uD83D\uDC9A",  power = 5)

println("Заклинание: ${fireball.name}")
println("Символ: ${fireball.symbol}")
println("Длится: ${fireball.duration} сек")

println("\nМгновенное заклинание: ${heal.name}")
println("Символ: ${heal.symbol}")
println("Сила: ${heal.power}")
println("Длится: ${heal.duration} сек")
```

2.3. Расширяем пакет с персонажами

И последнее задание по наследованию **персонажей.**

1. В пакете **characters Создайтете базовый класс GameCharacter** с общими **свойствами**:

- **name** – имя персонажа
- **hp** – здоровье персонажа
- **element** – элемент персонажа

2. Модифицируйте существующие классы

Сделайте их наследниками **GameCharacter**:

2.1 Класс Enemy (враг)

2.2 Класс Hero (герой)

3. Создайтете два новых класса-наследника

3.1 Класс NPC (неигровой персонаж):

Добавляет **свойства**:

- **faction** - Добавляет фракционную принадлежность
- **hasQuest** - Содержит логику для выдачи квестов и торговли

Реализует **методы**:

- **giveQuest**() – выдача квестов
- **trade**() - торговли

3.2 Класс Boss (босс):

- Наследуется от **Enemy**

  Добавляет **свойства**:

- **phaseCount: Int** – система фаз боя; **isFinalBoss: Boolean** – логическую проверку

  Реализует методы:

- **startPhase**() – система фазы боя; **dropLoot**() – особая система лута

Пример реализации классов NPC и Boss:

```
Кузнец Ульфрик предлагает вам квест
Кузнец Ульфрик открыл торговлю

Алдуин активирует фазу 2!
Легендарные предметы
Алдуин получает 150 урона!
Осталось HP врага: 350
```

Решение:

Класс **GameCharacter:**

```
open class GameCharacter (  new *
    val name: String,
    var hp: Int,
    val element: String = "Нейтральный"
)
```

Модифицированный класс Enemy:

```
class Enemy(  6 Usages    Denis *
    name: String,
    hp: Int = 0,
    element: String = "Неизвестный"
) : GameCharacter(name, hp, element) {
```

Модифицированный класс Hero:

```
class Hero(  1 Usage    Denis *
    name: String,
    var gender: String = "Неизвестный",
    var role: String = "Неизвестный",
    var level: Int = 0,
    element: String = "Неизвестный",
    hp: Int = 0,
    var mp: Int = 0,
    val experience: Int = 0
) : GameCharacter(name, hp, element) {
```

Класс NPC (Неигровой персонаж):

```
class NPC(  new *
    name: String,
    hp: Int = 100,
    element: String = "Нейтральный",
    val faction: String = "Городские жители",
    val hasQuest: Boolean = false
) : GameCharacter(name, hp, element) {
```

Методы у класса **NPC**:

```
fun giveQuest(): String {  new *
    return if (hasQuest) {
        "$name предлагает вам квест"
    } else {
        "$name не имеет заданий для вас"
    }
}

fun trade() {  new *
    println("$name открыл торговлю")
}
```

Класс Boss (Уникальный враг):

```
class Boss(  new *
    name: String,
    hp: Int,
    element: String,
    val phaseCount: Int = 3,
    val isFinalBoss: Boolean = false
) : Enemy(name, hp, element) {
```

Не забываем сделать открытым класс **Enemy**:

```
open class Enemy(
```

Методы у класса **Boss** :

```
fun dropLoot(): String {  new *
    return if (isFinalBoss) {
        "Легендарные предметы"
    } else {
        "Эпические предметы"
    }
}
```

```
fun startPhase(phaseNumber: Int) {  new *
    require( value = phaseNumber in 1 ≤ .. ≤ phaseCount) { "Неверный номер фазы" }
    println("$name активирует фазу $phaseNumber!")
}
```

Вывод в **Person.kt:**

```
// Тестируем NPC
val blacksmith = NPC( name = "Кузнец Ульфрик", hasQuest = true)
println(blacksmith.giveQuest())
blacksmith.trade()
// Тестируем Boss
val dragon = Boss( name = "Алдуин",  hp = 500,  element = "Огонь", phaseCount = 4,
 isFinalBoss = true)
dragon.startPhase( phaseNumber = 2)
println(dragon.dropLoot())
dragon.takeDamage( amount = 150)
```

Шаг 3: Upcast, Downcast, Smartcast

Сейчас в пакете characters у нас с вами созданы классы – **GameCharacter**, **Hero**, **Enemy**, **Boss**, **NPC**.

Давайте в файле **Person.kt** создадим по одному объекту каждого типа:

```
val enemy = Enemy( name = "Джин")
val hero = Hero( name = "Наруто")
val npc = NPC( name = "Торговец")
```

Если мы явно укажем тип Enemy для переменной enemy:

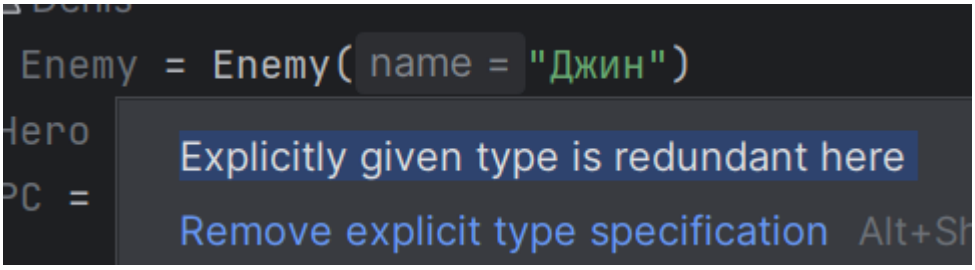
```
val enemy: Enemy = Enemy( name = "Джин")
```

— ошибки не возникнет, всё работает корректно.

То же самое можно сделать и для других объектов:

```
val hero: Hero = Hero( name = "Наруто")
val npc: NPC = NPC( name = "Торговец")
```

В таком случае среда разработки (IDE) может подсветить эти типы серым цветом — это означает, что **тип можно не указывать**, компилятор и так его выведет автоматически:



Теперь давайте укажем **общий тип GameCharacter** для всех объектов:

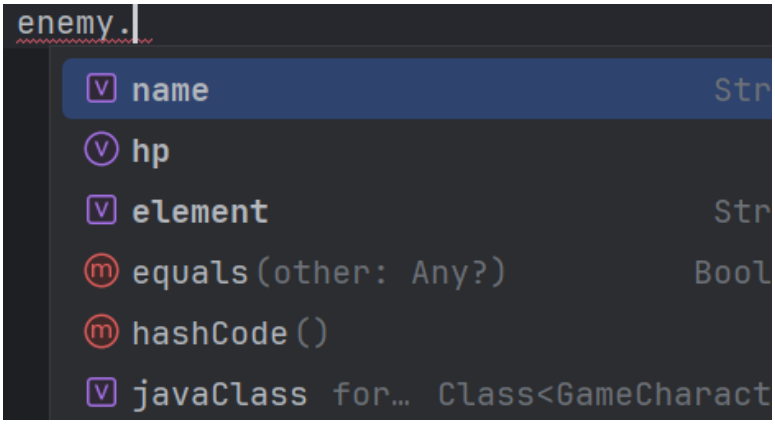
```
val enemy: GameCharacter = Enemy( name = "Джин")
val hero: GameCharacter = Hero( name = "Наруто")
val npc: GameCharacter = NPC( name = "Торговец")
```

Ошибок также не будет — ведь все эти классы **наследуются от GameCharacter**, значит, могут быть присвоены переменной такого типа. Это называется **Upcasting** — приведение объекта дочернего класса к родительскому типу.

Однако тут важно понимать одно отличие:

Если переменная имеет тип Enemy, то мы можем вызвать специфичные методы этого класса:

Но как только мы указали тип GameCharacter, эти методы **пропадают** из области видимости:



А как насчёт свойств?

Если у GameCharacter есть свойство name, то мы всё ещё можем его использовать:

```
println(enemy.name)
```

Это работает, потому что **name** — общее свойство, определённое в базовом классе **GameCharacter**.

Как снова вызвать методы Enemy?

Если вы точно знаете, что объект на самом деле является Enemy, вы можете привести его обратно:

```
enemy.takeDamage(20)
```

```
(enemy as Enemy).takeDamage( amount = 20)
```

Это называется **Downcasting** — приведение объекта от родительского типа обратно к дочернему. Только помните: **если объект на самом деле не является Enemy, то произойдёт ошибка**.

Вызовем дополнительные методы у enemy:

```
(enemy as Enemy).takeDamage( amount = 20)
enemy.takeDamage( amount = 20)
println(enemy.name)
```

Обратите внимание IDE внизу подсветила зелёным цветом. Если вы наведёте курсор, то IDE подсветит вызов зелёным с подсказкой: *"Smart cast to characters.Enemy"*.

Это означает, что компилятор автоматически понимает тип enemy после первого приведения. Дополнительное приведение не требуется:

```
enemy.takeDamage( amount = 20)
pr
    Smart cast to characters.Enemy
```

Шаг 4: Введение в полиморфизм

Полиморфизм — одно из ключевых понятий объектно-ориентированного программирования. Оно означает, что один и тот же метод может вести себя по-разному в зависимости от того, в каком классе он реализован. Это становится возможным благодаря **наследованию** и **переопределению** методов.

Рассмотрим это на примере наших классов питомцев.

В файле **PetShop.kt** в списке указываем явно класс:

```
val pets = listOf<Pet>(
```

Всё работает, ошибок нет — потому что все объекты в списке являются наследниками **Pet**.

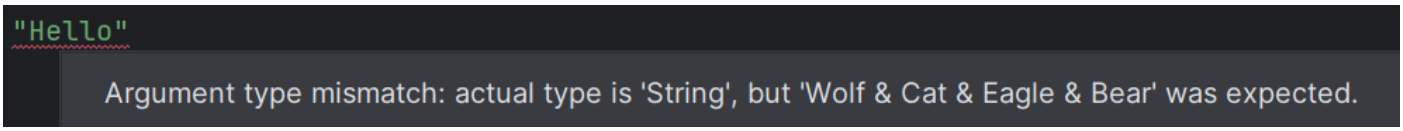
А теперь передадим обычную строку текста:

```
val pets = listOf<Pet>(
    Wolf( name = "Фенрир"),
    Cat( name = "Мурзик"),
    Eagle( name = "Скайвинд"),
    Bear( name = "Балу"),
    "Hello"
)
```

Теперь появляется ошибка:

```
  Bear( name = "Балу"),
  "Pets"
```

Наводим и видим описание:



Это логично — в список List<Pet> нельзя передать объект, не являющийся Pet.

А теперь давайте удалим явно указываемый нами тип данных **Pet** у списка:

```
val pets = listOf(
```

Теперь ошибок нет. **Почему**?

Когда мы **не указываем явно тип**, Kotlin автоматически находит **общий родитель всех объектов в списке**, и в этом случае им становится Any — базовый класс всех классов в Kotlin. Следовательно, мы можем явным способом указать тип данных:

```
val pets = listOf<Any>(
```

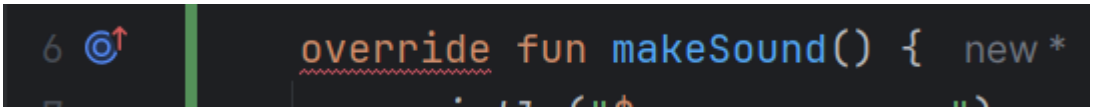
Предположим, что мы хотим, чтобы каждый питомец издавал свой уникальный звук. Мы могли бы добавить новые методы в каждом классе, но правильнее — **переопределить метод makeSound()**.

Попробуем сначала просто написать одноимённый метод в классе Cat:

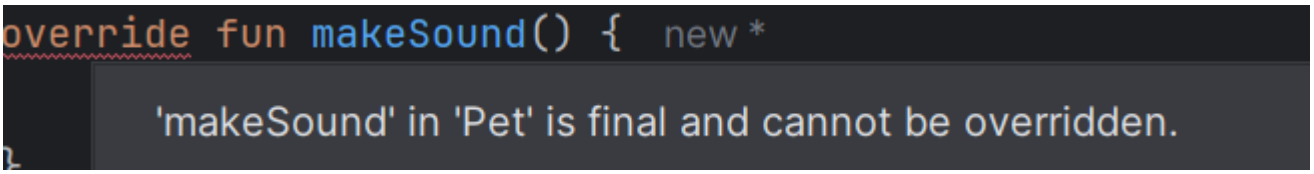
```
fun makeSound() {  new *
    println("$name мяукает.")
}
```

Возникает **ошибка**. Почему?

Метод **makeSound**() уже есть в родительском классе Pet. Мы не можем просто так его "переписать". Но, что если мы не хотим добавлять новый метод, а хотим изменить поведение уже существующего? В программировании это называется переопределение метода. Для этого нужно добавить ключевое слово **override**. Добавьте у обратите внимание, что слева у нас появилась стрелка, которая ведёт к тому методу, который мы хотим переопределить:



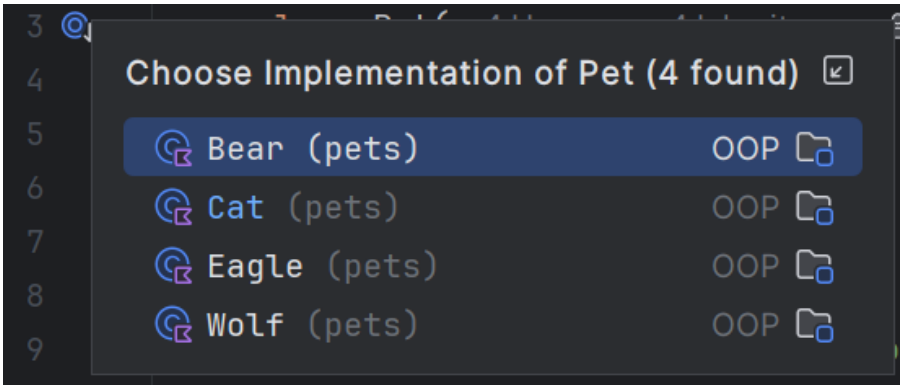
Кликнув на неё мы попадём в родительский класс **Pet**. Давайте вернёмся и посмотрим почему у нас выводится ошибка:



в **Kotlin** по умолчанию все методы имеют модификатор **final**, то есть **не могут быть переопределены**. Для решения проблемы в родительском классе нужно пометить метод **open**.

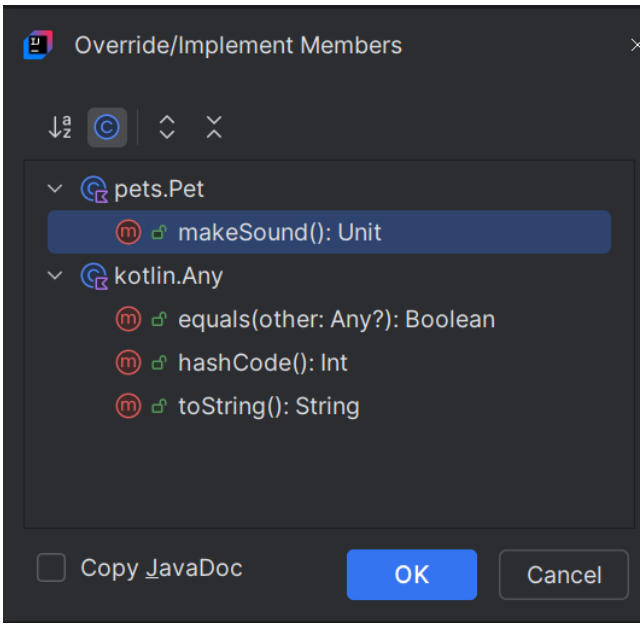
После этого в IDE появится **стрелочка**, указывающая, что метод переопределён. Кликнув на неё, можно перейти к методу в базовом классе Pet.

Например, если мы нажмём на неё у класса **Pet**, то увидим в каких классах он используется:



Переопределение на практике используется довольно часто, поэтому для него есть удобные сочетания клавиш. Давайте перейдём в класс **Wolf.** Внутри класса нажмём сочетание клавиш **Ctrl+O:**

У нас появится окно, которое предложит переопределить различные методы:



Обратите внимание, что дополнительно к методу **makeSound**() который мы открыли, есть также встроенные 3 метода – **equals**(), **hashCode**(), **toString**(). Это подтверждает, что у нас есть общий класс **Any**.

Выбираем наш метод **makeSound**(). Удалим пока что строчку **super**.**makeSound**():

```
override fun makeSound() {
    super.makeSound()
}
```

И добавим вывод что волк у нас воет:

```
println("$name воет.")
```

Задание.

1. Переопределите **makeSound**() для всех питомцев:
    - **Eagle**: "**кричит**"
    - **Bear**: "**рычит**"
2. Переопределите метод **describe**() для добавления уникальных характеристик:

Например, у Wolf можно добавить "опасный и быстрый" или указать размер стаи, у **Cat** — "тихий и ловкий".

Пример вывода:

```
Питомец: Фенрир, Здоровье: 250, Скорость: 100
У Фенрир размер стаи - 5
Фенрир воет.

Питомец: Мурзик, Здоровье: 150, Скорость: 80
Наш Мурзик очень скрытный (90 скрытность!) и тихий!
Мурзик мяукает.

Питомец: Скайвинд, Здоровье: 50, Скорость: 250
Скайвинд своим острым взглядом видит всё далеко
Скайвинд кричит.

Питомец: Балу, Здоровье: 550, Скорость: 50
У Балу силушка богатырская - 350
Балу рычит.
```

Задание: Расширение класса Mission и переопределение в наследниках

1. В родительском классе **Mission** добавить:
   - Метод **describe**(), который выводит информацию о миссии.
   - Метод **isHighReward**(), возвращающий **true**, если награда больше **1000**.
2. В каждом из наследников (**Quest**, **SpecialOperation**, **Contract**) переопределить метод **describe**() с учётом специфики класса (например, длительность квеста, заказчик контракта и т.д.).
3. В **main**() создать список из миссий разных типов и вызвать для каждой **describe**() и **isHighReward**().

Пример **вывода**:

```
Квест 'Охота на монстров' на 5 часов, сложность: Средний, награда: 600 золотых
Высокая награда? Нет

Спецоперация 'Ночной рейд'. Уровень допуска: 2, режим: Секретно, награда: 1500
Высокая награда? Да

Контракт от Гильдия Торговцев: Доставить груз через лес (СРОЧНО), награда: 800
Высокая награда? Нет
```

Решение:

Добавляем в класс **Mission** два метода:

```
open fun describe() {  new *
    println("Миссия: $title, Награда: $reward монет")
}

fun isHighReward(): Boolean {  new *
    return reward > 1000
}
```

Вносим изменения в класс **Quest**:

```
override fun describe() {  new *
    println("Квест '$title' на $duration часов, сложность: $difficulty,
     награда: $reward золотых")
}

fun goldPerHour(): Int {  Denis *
    require( value = duration >= 0) { "Длительность не может быть
     отрицательной!" }
    return if (duration == 0) 0 else reward / duration
}

fun isHard(): Boolean {  new *
    return difficulty.lowercase() == "сложный"
}
```

Вносим изменения в класс **SpecialOperation:**

```
override fun describe() {  new *
    println("Спецоперация '$title'. Уровень допуска: $requiredClearance,
     режим: ${if (isCovert) "Секретно" else "Открыто"}, награда: $reward")
}
```

Вносим изменения в класс **Contract:**

```
override fun describe() {  new *
    println("Контракт от $clientName: $taskDescription (${if (isUrgent)
     "СРОЧНО" else "Обычный"}), награда: $reward")
}
```

Выводим в файле **Guild.kt**:

```
val missions = listOf<Mission>(
    Quest( title = "Охота на монстров",  duration = 3,  reward = 600,  difficulty = "Средний"),
    SpecialOperation( title = "Ночной рейд",  reward = 1500, requiredClearance = 2,
     isCovert = true),
    Contract( title = "Сопровождение каравана",  clientName = "Гильдия Торговцев",
     taskDescription = "Доставить груз через лес",  reward = 800, isUrgent = true)
)

for (mission in missions) {
    mission.describe()
    println("Высокая награда? ${if (mission.isHighReward()) "Да" else "Нет"}")
    println()
}
```

Самостоятельные задания

Задание 1: Средневековый транспорт

Создайте **новый пакет** под названием **transport**, в котором реализуешь систему передвижения в RPG-мире.

Структура:

1. Создайте **родительский класс Transport** — базовый вид транспорта в игре.
2. Создайте **3–4 дочерних класса**, которые будут наследоваться от него.
3. Реализуйте переопределение методов (**override**) и использование списка объектов с базовым типом.
4. Сделайте вызов всех методов для каждого объекта в main().

Родительский класс Transport

Свойства:

- **name** — название транспорта
- **speed** — скорость
- **capacity** — сколько персонажей можно перевозить

Методы:

- **describe**() — выводит краткую информацию о транспорте
- **move**() — выводит, как этот транспорт двигается
- **specialAbility**() — виртуальный (открытый) метод, который будет переопределяться каждым дочерним классом по-своему

Дочерние классы (варианты):

1. Horse *(Лошадь)*

- Указывает, что это скакун
- В **move**() выводится: "скачет по равнинам"
- В **specialAbility**() выводится: "может ускориться в бою"

2. Dragon *(Дракон)*

- Вывод в **move**(): "летит высоко над землёй"
- В **specialAbility**(): "дышит огнём и перевозит героя по воздуху"

3. Boat *(Лодка)*

- В **move**(): "плывёт по воде"
- В **specialAbility**(): "может перевозить группу по рекам и озёрам"

4. Teleport *(Телепорт)*

- В **move**(): "мгновенно перемещается в другое место"
- В **specialAbility**(): "нужна мана для активации, мгновенное перемещение"

В main() реализуйте:

- Создание списка из разных видов транспорта, но с типом **Transport**
- Вызов для каждого: **describe**(), **move**(), **specialAbility**()
- Демонстрация **полиморфизма** — каждый объект по-разному реализует поведение общего метода

Задание 2: Система магических артефактов

1. Создайте новый **пакет artifact**

2. Базовый класс MagicItem

Должен содержать:

- **Свойства:**
  - **name** - название артефакта
  - **power** - сила артефакта (число)
  - **rarity** - редкость (Common/Rare/Epic)
- **Методы:**
  - **describe**() - выводит базовую информацию (название, силу, редкость)
  - **use**() - возвращает строку с эффектом применения

3. Создайте 3 класса-наследника

1. Weapon (магическое оружие)

- **Доп. свойства:**

- **damageType** - тип урона (огонь/лед/тьма и т.д.)
- **isTwoHanded** - требует две руки?

- **Переопределить методы:**
  - **describe**(): добавить информацию о типе урона
  - **use**(): вернуть строку "Наносит [тип урона] урон силой [power]"

2. Potion (зелье)

- **Доп. свойства:**
  - **effectDuration** - длительность эффекта (в секундах)
  - **isConsumable** - одноразовое?
- **Переопределить методы:**
  - **describe**(): указать длительность эффекта
  - **use**(): вернуть разные строки для одноразовых/многоразовых зелий

3. Relic (реликвия)

- **Доп. свойства:**
  - **origin** - происхождение (эльфийское/демоническое и т.д.)
  - **charges** - количество зарядов
- **Переопределить методы:**
  - **describe**(): добавить информацию о происхождении
  - **use**(): уменьшать заряды на 1 при использовании

4. Демонстрация работы

Создайте файл **ArtifactDemo.kt** где:

1. Создайте по одному объекту каждого типа
2. Поместите их в список **List<MagicItem>**
3. Для каждого элемента вызовите:
   - **describe**()
   - **use**()
4. Выведите результаты в консоль