Лабораторная работа №6: Конструкторы и расширение классов.

Цель: Познакомиться с конструктором как основным способом инициализации объектов. Научиться создавать несколько вариантов создания объекта (перегрузка), использовать первичный конструктор для лаконичного синтаксиса. Структурировать проект по пакетам. Освоить возвращаемые значения в функциях.

Шаг 1. Введение в конструкторы

В нашем классе **Quest** на данный момент инициализация переменных происходит с помощью метода **init**:

```
fun init(title: String, duration: Int, reward: Int, difficulty: String) {
    this.title = title
    this.duration = duration
    this.reward = reward
    this.difficulty = difficulty
}
```

Такой подход считается неудачным: любой другой разработчик, использующий ваш класс, может даже не догадываться, что ему нужно **вручную вызывать метод init**, иначе переменные останутся со значениями по умолчанию. Это нарушает один из принципов **чистого кода** — ясность и надёжность.

Kotlin предлагает встроенный механизм для инициализации — **конструкторы**. Они вызываются автоматически при создании объекта и позволяют сразу передать значения нужным свойствам:

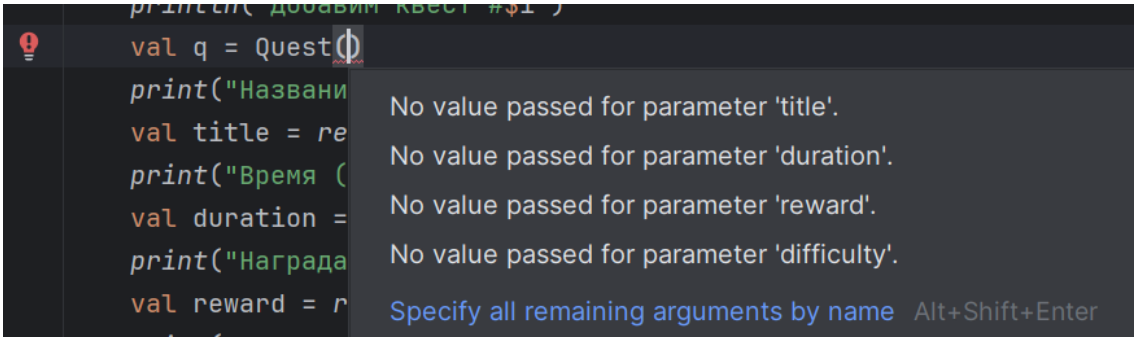
```
constructor (title: String, duration: Int, reward: Int, difficulty: String) {
    this.title = title
    this.duration = duration
    this.reward = reward
    this.difficulty = difficulty
}
```

Главное отличие от init

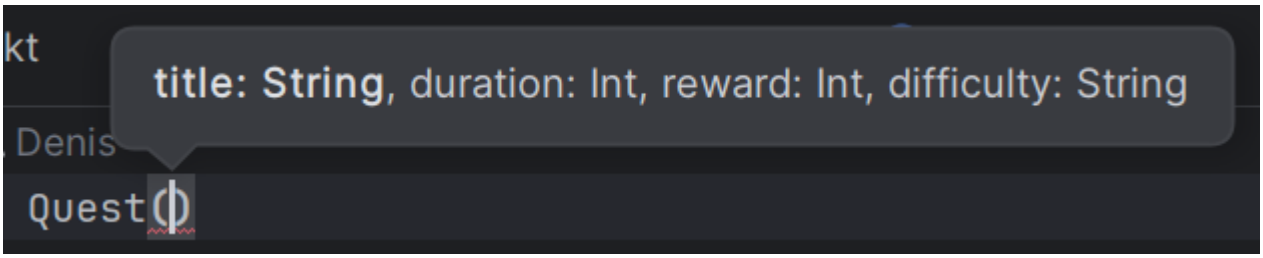
Методы можно **не вызывать** (как мы видели раньше), а вот конструктор будет вызван **обязательно**, иначе объект не создастся. Это делает поведение класса надёжным и предсказуемым.

Что происходит теперь?

Допустим, в другом файле вы хотите создать задание. Если вы забудете передать параметры, среда разработки покажет ошибку:



Наведите курсор внутрь скобок и нажмите **Ctrl+P**, чтобы увидеть список необходимых параметров.



Передадим в класс параметры и через метод **printInfo**() выведем информацию:

```
fun main() {  Denis *
    val quest = Quest( title = "Охота на гоблинов",  duration = 2,
      reward = 3400,  difficulty = "легкий")
    quest.printInfo()
}
```

Теперь нам **не нужен метод init**, и его можно смело удалить:

```
fun init(title: String, duration: Int, reward: Int, difficulty: String) {  Denis
    this.title = title
    this.duration = duration
    this.reward = reward
    this.difficulty = difficulty
}
```

Все нужные значения объект получает сразу, при создании. Это улучшает читаемость, предотвращает ошибки и делает ваш код соответствующим стандартам хорошей архитектуры.

Теперь удалим **значения по умолчанию** у переменных в классе **Quest**:

```
class Quest {  1 Usage   Denis *
    var title: String = "Безымянное задание"
    var duration: Int = 1  3 Usages
    var reward: Int = 100  3 Usages
    var difficulty: String = "Лёгкий"  3 Usages
```

Ранее, если бы мы попытались так написать, компилятор выдал бы ошибку — он требовал установить начальные значения. Но теперь ошибок нет. Почему?

Это связано с тем, что мы добавили **собственный конструктор**, и он **обязателен для вызова** при создании объекта. Следовательно, все переменные инициализируются через него, и начальные значения внутри класса больше не нужны.

Кроме того, теперь можно заменить **var** на **val**, если вы не планируете менять свойства после создания объекта. В Kotlin принято **по возможности использовать val**, так как это делает объект **более безопасным и неизменяемым** (immutable):

```
class Quest {  1 Usage   De
    val title: String  3 Usa
    val duration: Int  3 Usa
    val reward: Int  3 Usage
    val difficulty: String
```

Теперь выполните **рефакторинг классов Hero и Enemy**:

- Удалите значения по умолчанию у переменных.
- Создайте **конструктор**, принимающий параметры.
- Используйте **val** вместо **var** там, где переменные не предполагается изменять.
- В классе **Person** измените способ создания объектов **Hero** и **Enemy**, чтобы передавать параметры сразу в конструктор:

```
val naruto = Hero( name = "Наруто",  gender = "мужской",  role = "Шиноби",  level = 5,
  element = "Ветер",  hp = 150,  mp = 200)
naruto.showStats()
val kakashi = Hero( name = "Какаши",  gender = "мужской",  role = "Шиноби",  level = 15,
  element = "Молния",  hp = 250,  mp = 300)
kakashi.showStats()
val orochimaru = Enemy( name = "Орочимару",  hp = 350,  element = "Ветер")
naruto.castSpellOn( enemy = orochimaru,  spellName = "Расенган",  damage = 30)
naruto.duel( opponent = kakashi)
```

Шаг 2. Перегрузка конструкторов

1. Создайте класс **Spell** со следующими характеристиками:
    - **Свойства**:
        - **name** (тип **String**) **—** название заклинания
        - **width (**тип **Int) —** ширина области эффекта

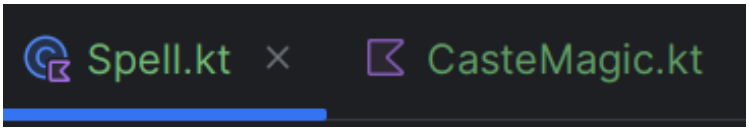
        - **height (**тип **Int) —** высота области эффекта
        - **symbol** (тип **String**) **—** символ, представляющий заклинание
    - **Конструктор**, инициализирующий все свойства
    - Метод **cast**(), который:
        - Выводит в консоль сообщение "Кастуем {name}!"
        - Отображает эффект заклинания в виде прямоугольника размером **width** × **height**, где каждый элемент заполнен символом **symbol**.
2. Создайте файл **CasteMagic**, в котором в функции **main**():
    - Создайте экземпляр заклинания **fireWall** с параметрами:
        - **name = "Огненная Стена"**
        - **width = 5**
        - **height = 3**
        - **symbol** = "\uD83D\uDD25" (эмодзи огня)
    - Вызовите метод **cast**() для этого заклинания

Создайте заклинание с другим символом (например, ❄ для ледяного щита) и вызовите его. Можете использовать следующие примеры кодировки заклинаний:

Название	**Юникод-кодировка**
Огненный шар	\uD83D\uDD25
Ледяная стрела	\u2744\uFE0F
Молния	\u26A1
Вихрь	\uD83C\uDF2A\uFE0F
Водяной удар	\uD83C\uDF0A
Искра	\u2728
Иллюзия (звёздочки)	*\uD83D\uDCAB*

Решение:

1. **Создаём два файла – Spell (тип класс) и CasteMagic (тип файл):**



Файл **Spell.kt** будет содержать описание класса заклинаний, а **CasteMagic.kt** — главный метод **main**(), в котором происходит выполнение программы.

2. Далее пишем класс **Spell**:

```
class Spell {  1 Usage   new *
    val name: String  3 Usages
    val width: Int  3 Usages
    val height: Int  3 Usages
    val symbol: String  3 Usages

    constructor(name: String, width: Int, height: Int, symbol: String) {
        this.name = name
        this.width = width
        this.height = height
        this.symbol = symbol
    }
```

Этот класс описывает заклинание. У него есть четыре свойства:

- **name** — название заклинания, например, "Огненная Стена"
- **width** и **height** — определяют размеры области, на которую воздействует заклинание
- **symbol** — визуальное отображение эффекта, например 🔥 для огня или ❄ для льда

Метод **cast**() отвечает за "применение" заклинания: он выводит сообщение о касте и визуально рисует область эффекта. **repeat**(**height**) и **repeat**(**width**) создают вложенные циклы, которые печатают нужное количество символов в форме прямоугольника.

```
fun cast() {  1 Usage   new *
    println("Кастуем $name!")
    repeat( times = height) {
        repeat( times = width) {
            print(symbol)
        }
        println()
    }
}
```

3. Пример использования метода cast(): В главной функции **main**() создаётся объект **fireWall** с заданными параметрами. Когда вызывается **fireWall.cast**(), консоль сначала показывает сообщение "**Кастуем Огненная Стена!**", а затем рисуется прямоугольник 5×3, заполненный символом 🔥. Это позволяет визуально представить действие магии:

```
fun main() {  new *
    val fireWall  = Spell( name = "Огненная Стена",  width = 5,
     height = 3,  symbol = "\uD83D\uDD25")
    fireWall.cast()
}
```

Перегрузка конструктора (overloading): Иногда ширина и высота области эффекта совпадают. Чтобы избежать дублирования кода и сделать создание таких заклинаний проще, можно добавить второй конструктор:

```
constructor(name: String, size: Int, symbol: String) {
    this.name = name
    this.width = size
    this.height = size
    this.symbol = symbol
}
```

Он устанавливает одинаковые значения для **ширины** и **высоты**. Это пример перегрузки — создания нескольких конструкторов с разным набором параметров. Позволяет гибко создавать объекты в зависимости от ситуации.

Создание заклинания с перегруженным конструктором: Теперь можно создать, например, ледяной щит:

```
val iceSheet = Spell( name = "Ледяной щит",  size = 4,  symbol = "\u2744\uFE0F")
iceSheet.cast()
```

Такой код легче читать и писать, особенно когда область квадратная. Вызов **iceSheet.cast()** нарисует квадрат 4×4, заполненный снежинками ❄.

Шаг 3. Конструктор по умолчанию

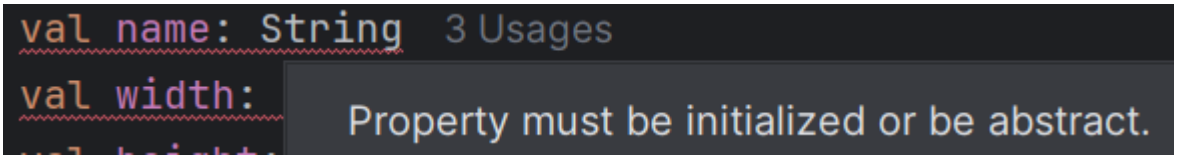
Когда мы создаём класс в Kotlin и **не указываем никакого конструктора**, компилятор автоматически создаёт **конструктор по умолчанию** — то есть пустой конструктор без параметров. Именно поэтому мы раньше могли писать просто **val quest = Quest()** и всё работало, если у всех свойств были значения по умолчанию.

Если мы сейчас создадим пустой конструктор:

```
constructor() {

}
```

То компилятор **подсветит красным переменные**, и появится ошибка - **Property must be initialized or be abstract**:



Это происходит потому, что переменные val обязательны к инициализации — либо при объявлении, либо в конструкторе. А мы здесь не задали **ни значений по умолчанию**, ни инициализацию через конструктор, ни сделали свойства **lateinit** (что для **val** недопустимо).

Мы можем создать **явный конструктор по умолчанию**, но при этом **обязательно установить значения вручную**, чтобы ошибка исчезла:

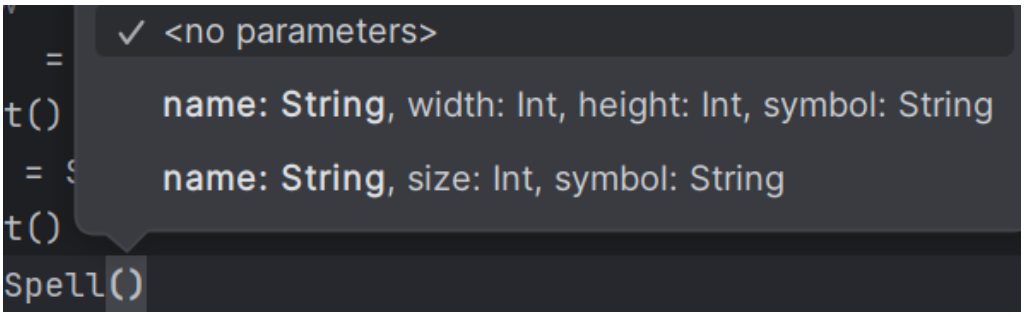
```
constructor() {  new *
    this.name = ""
    this.width = 0
    this.height = 0
    this.symbol = ""
}
```

Теперь всё работает корректно, и компилятор больше не ругается.

Теперь мы можем **создавать заклинания без параметров**, например:

```
val spell = Spell()
```

Также, если вы нажмёте внутри скобок при создании объекта **Ctrl+P** (или наведёте курсор), вы увидите, что у класса Spell теперь есть **несколько реализаций конструктора**. Это стало возможным благодаря **перегрузке конструкторов**:



- **Spell(name: String, width: Int, height: Int, symbol: String)**

- **Spell(name: String, size: Int, symbol: String)**
- **Spell() — наш новый конструктор по умолчанию**

Сделаем выводы:

- ✓ Если **все свойства класса проинициализированы при объявлении**, Kotlin автоматически создаёт **конструктор по умолчанию**.
- ✓ Если вы **хотите иметь несколько способов создания объектов** — используйте **перегрузку конструкторов**.
- ✓ При использовании **val** свойства обязаны быть инициализированы, иначе будет ошибка компиляции.

Шаг 4. Вызов конструктора из конструктора (this(...))

В Kotlin можно **из одного конструктора вызвать другой**, чтобы избежать дублирования кода. Это особенно удобно, когда большинство параметров совпадают — тогда вся логика инициализации остаётся в одном месте.

Перепишем наш класс **Spell**, используя this(...):

```
// Основной конструктор, который инициализирует все свойства
constructor(name: String, width: Int, height: Int, symbol: String) {
    this.name = name
    this.width = width
    this.height = height
    this.symbol = symbol
}
```

```
// Конструктор для квадратных заклинаний (width = height)
constructor(name: String, size: Int, symbol: String) : this(name,
  width = size, height = size, symbol)  new *
```

```
// Конструктор по умолчанию
constructor() : this( name = "", width = 0, height = 0, symbol = "")
```

- Мы **избавились от повторяющейся инициализации**.
- Все остальные конструкторы теперь **делегируют** основной (главный) конструктор: **this**(...).
- Такой подход делает код **чище и проще для поддержки**.

Шаг 5. Первичный конструктор

Первичный конструктор — это способ описать параметры конструктора **сразу после имени класса**, не используя ключевое слово **constructor**.

Это наиболее **краткая и чистая** форма записи, при этом все свойства можно сразу объявить и инициализировать.

Перепишем класс **Spell** с использованием первичного конструктора:

```
class Spell(  5 Usages   new *
    val name: String,
    val width: Int,
    val height: Int,
    val symbol: String
) {

    // Конструктор для квадратных заклинаний (width = height)
    constructor(name: String, size: Int, symbol: String) :
     this(name,  width = size,  height = size, symbol)  new *

    // Конструктор по умолчанию
    constructor() : this( name = "",  width = 0,  height = 0,
      symbol = "")  new *
```

Преимущества первичного конструктора:

- **Код стал компактнее** — свойства объявлены и инициализированы сразу.
- Уменьшается дублирование.
- Легче читать и поддерживать.
- Можно комбинировать с вторичными конструкторами (как мы и сделали выше).

Мы можем указать **значения по умолчанию** для параметров конструктора. Это позволяет создавать объект **без передачи всех аргументов**, что избавляет нас от необходимости писать вторичный конструктор по умолчанию.

```
class Spell(  5 Usages   new *
    val name: String = "",
    val width: Int = 0,
    val height: Int = 0,
    val symbol: String = ""
) {
```

Таким образом, если мы не будем указывать какие-либо значения, то будем брать их из первичного конструктора.

Поэтому мы можем **удалить вторичный конструктор без параметров**, потому что он **больше не нужен**:

```
// Конструктор по умолчанию
constructor() : this( name = "", width = 0, height = 0, symbol = "")  new *
```

В IntelliJ IDE есть возможность быстрого преобразования конструктора в первичный. Давайте рассмотрим это на примере.

1. Создайте класс **Item,** который будет представлять информацию о товаре в магазине.

Требования:

- ✓ **Свойства:**
  - **name (String)** — название товара.
  - **price (Int)** — цена товара (в условных единицах).
- ✓ **Конструктор:**
  - Должен принимать **name** и **price** и инициализировать соответствующие свойства.
- ✓ **Метод displayInfo():**
  - Выводит информацию о товаре в формате:

Товар: [name], Цена: [price]

2. Создайте файл Shop.kt и в функции main():

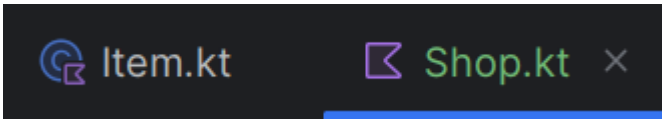
1. Создайте два объекта класса **Item**:
   - **sword**: "Меч", цена 45000.
   - **potion**: "Зелье маны", цена 350.
2. Вызовите метод **displayInfo()** для каждого товара.

Вывод:

```
Товар: Меч, Цена: 45000
Товар: Зелье маны, Цена: 350
```

Решение:

1. **Создаём два файла – Item (тип класс) и Shop(тип файл):**



Пишем класс **Item**:

```
class Item {  Denis
    val name: String
    val price: Int

    constructor(name: String, price: Int) {
        this.name = name
        this.price = price
    }

    fun displayInfo() {  Denis
        println("Товар: $name, Цена: $price")
    }
}
```

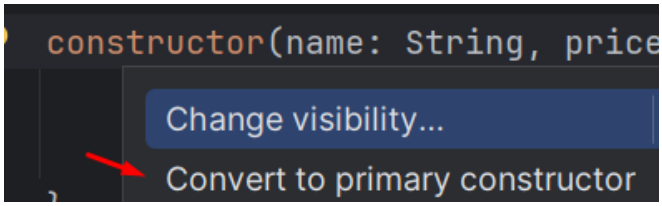
Реализуем в методе **main**():

```
fun main() {  new *
    val sword = Item( name = "Меч",  price = 45000)
    val potion = Item( name = "Зелье маны",  price = 350)
    sword.displayInfo()
    potion.displayInfo()
}
```

Теперь поместите курсор на слово **constructor**(...):

```
constructor(name: String, price: Int) {
```

Нажмите **Alt + Enter** и выберите **"Convert to primary constructor":**

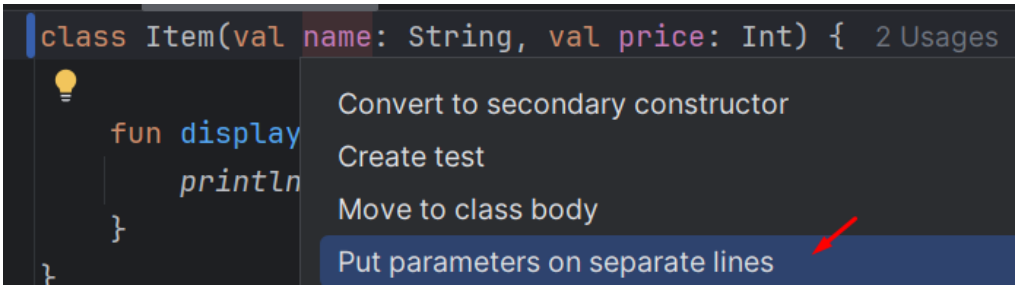


В итоге у нас получилось:

```
class Item(val name: String, val price: Int) {

    fun displayInfo() {  2 Usages   Denis
        println("Товар: $name, Цена: $price")
    }
}
```

Чтобы сделать запись свойств на разных строках нажмите внутри класс **Alt + Enter** и выберите **Put parameters on separate lines**:



Задание. Перепишите самостоятельно классы Quest, Hero и Enemy, используя первичный конструктор:

- Удалите вторичные конструкторы.
- Включите значения по умолчанию, если необходимо.
- Проверьте работу через **printInfo**() или **attack**().

Задание: Создайте класс Contract — «Контракт на выполнение задания»

Контракты в таверне могут отличаться от обычных квестов: это особые заказы с деталями, которые можно отправить в гильдию.

Требования:

1. Используйте **первичный конструктор**.
2. Укажите следующие свойства:
    - **clientName: String** — имя заказчика
    - **taskDescription: String** — описание задачи
    - **reward: Int** — награда
    - **isUrgent: Boolean = false** — срочный ли контракт (по умолчанию — нет)
3. Реализуйте метод **printContractInfo**(), который выводит полную информацию:

```
println("Заказчик: $clientName")
println("Задача: $taskDescription")
println("Награда: $reward монет")
println("Срочность: ${if (isUrgent) "Срочно!" else "Обычный контракт"}")
```

Пример использования:

```
Заказчик: Торговец Грам
Задача: Сопровождение каравана
Награда: 1500 монет
Срочность: Срочно!
```

```
Заказчик: Безымянный
Задача: Уничтожить крыс в подвале
Награда: 100 монет
Срочность: Обычный контракт
```

Решение:

```
class Contract(  1 Usage
    val clientName: String,
    val taskDescription: String,
    val reward: Int,
    val isUrgent: Boolean = false
) {
    fun printContractInfo() {  1 Usage
        println("Заказчик: $clientName")
        println("Задача: $taskDescription")
        println("Награда: $reward")
        println("Срочность: ${if (isUrgent) "Срочно!" else "Обычный
         контракт"}")
    }
}
```

Шаг 6. Возвращаемый тип функций

В Kotlin каждая функция может возвращать значение. Возвращаемый тип указывается **после списка параметров**, через двоеточие.

Пример. Простая функция с возвратом числа:

```
fun getReward(): Int {
    return 500
}
```

Также можно использовать сокращённый синтаксис:

```
fun getDifficulty(): String = "Лёгкий"
```

Рассмотрим на примере класса **Quest**. Добавим функцию **isHard**(), которая проверяет, сложный ли квест:

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

В файле **Guild** создадим квест:

```
val quest = Quest( title = "Побег из замка", duration = 5, reward = 700, difficulty = "Сложный")
println("Квест сложный? ${quest.isHard()}")
```

Другой пример в классе **Hero** создадим функцию **isAlive**(), которая проверяет, жив ли герой:

```
fun isAlive(): Boolean {
    return hp > 0
}
```

В файле **Person** создадим двух героев:

```
val hero1 = Hero(name = "Артур", hp = 100 )
println("Герой жив? ${hero1.isAlive()}")
val hero2 = Hero(name = "Артур", hp = 0 )
println("Герой жив? ${hero2.isAlive()}")
```

Также давайте посчитаем, сколько клеток занимает заклинание в классе **Spell**, для этого создадим функцию **area**():

```
fun area(): Int {  new *
    return width * height
}
```

И в файле **CasteMagic** проверим для наших заклинаний:

```
val fireWall  = Spell( name = "Огненная Стена", width = 5, height = 3, symbol = "\uD83D\uDD25")
fireWall.cast()
val iceSheet = Spell( name = "Ледяной щит", size = 4, symbol = "\u2744\uFE0F")
iceSheet.cast()
val spell = Spell()
spell.cast()

println(fireWall.area())
println(iceSheet.area())
println(spell.area())
```

Самостоятельная работа

1. В классе **Enemy** создайте функцию **isStrong**() — возвращает **true**, если здоровье выше **100**.
2. В классе **Spell** добавьте функцию **description**(): **String**, которая возвращает описание заклинания, например: "**Заклинание Огненная стена занимает область 5x3 и использует символ** 🔥"

Таким образом:

1. **return** завершает функцию и возвращает значение.
2. **Тип возвращаемого значения** должен совпадать с объявленным.
3. Если функция **ничего не возвращает**, можно:
    - Не указывать тип.
    - Указать: **Unit**.

Решение для **Enemy**:

```
fun isStrong(): Boolean {
    return hp > 100
}
```

Решение для **Spell**:

```
fun description(): String {  3 Usages   new *
    return "Заклинание $name занимает область ${width}x$height
      и использует символ $symbol"
}
```

Давайте создадим в классе **Hero** метод **canAcceptQuest**(), который проверяет, может ли герой принять квест в зависимости от его уровня и сложности квеста.

```
fun canAcceptQuest(quest: Quest): Boolean
 {  2 Usages   new *
    return when (quest.difficulty.lowercase()) {
        "лёгкий" -> level >= 1
        "средний" -> level >= 3
        "сложный" -> level >= 5
        else -> false
    }
}
```

- Проверяем сложность квеста (**difficulty**).
- Сравниваем уровень героя (**level**) с требуемым для каждой сложности:
    - Лёгкий: уровень ≥ 1
    - Средний: уровень ≥ 3
    - Сложный: уровень ≥ 5

- Возвращаем **true**, если герой соответствует требованиям, иначе **false**.

Модифицируем метод, чтобы он выводил сообщение о результате:

```
fun canAcceptQuest(quest: Quest): Boolean {  2 Usages   new *
    val canAccept = when (quest.difficulty.lowercase()) {
        "лёгкий" -> level >= 1
        "средний" -> level >= 3
        "сложный" -> level >= 5
        else -> false
    }

    if (canAccept) {
        println("$name может принять квест «${quest.title}»!")
    } else {
        println("$name не может принять квест «${quest.title}».
         Требуется более высокий уровень.")
    }

    return canAccept
}
```

Пример использования в файле Person. Создадим героя и квесты, проверим возможность их принятия:

```
val hero = Hero(name = "Артур", level = 4)  // Уровень 4
val easyQuest = Quest( title = "Сбор трав",  duration = 2,  reward = 100,
 difficulty = "Лёгкий")
val hardQuest = Quest( title = "Охота на дракона",  duration = 10,
 reward = 1000,  difficulty = "Сложный")

hero.canAcceptQuest(easyQuest)
hero.canAcceptQuest(hardQuest)
```

Далее попробуйте самостоятельно решить задачу. **Определение уровня угрозы врага (getThreatLevel())**

В классе **Enemy** создать метод, который анализирует здоровье (**hp**) врага и возвращает текстовое описание уровня угрозы:

- **"Низкий"** — если здоровье ≤ 50.
- **"Средний"** — если здоровье от 51 до 150.

- **"Высокий"** — если здоровье ≥ 151.

Решение. Если мы проанализируем с вами требования, то поймём, что нам нужно решить задачу используя три диапазона значений. Используем конструкцию **when** без параметра (в режиме условия):

```
fun getThreatLevel(): String {
    return when {
        hp <= 50 -> "Низкий"
        hp >= 150 -> "Средний"
        else -> "Высокий"
    }
}
```

Таким образом у нас будут проверяться условия сверху вниз, и если hp <= 50 — сразу возвращает "Низкий". Если hp > 50, проверяем, что hp <= 150 → "Средний". Всё остальное (hp > 150) → "Высокий".

Можно использовать диапазоны (**in**), но это менее эффективно для данной задачи:

```
return when (hp) {
    in 0 ≤ .. ≤ 50 -> "Низкий"
    in 51 ≤ .. ≤ 150 -> "Средний"
    else -> "Высокий"
}
```

Пример использования:

```
val enemy1 = Enemy( name = "Гоблин", hp = 30)
println(enemy1.getThreatLevel())  // "Низкий"

val enemy2 = Enemy( name = "Орк", hp = 100)
println(enemy2.getThreatLevel())  // "Средний"

val enemy3 = Enemy( name = "Дракон", hp = 200)
println(enemy3.getThreatLevel())  // "Высокий"
```

Теперь вопрос на засыпку, поставьте значения отрицательные у одного из врагов. Что мы увидим?

Ответ - если в **hp** передать отрицательное число, текущая реализация **getThreatLevel**() некорректно вернёт "Низкий":

```
val zombie = Enemy( name = "Зомби", hp = -100)
println(zombie.getThreatLevel()) // "Низкий" (но так быть не должно!)
```

Решение:

Добавить условие в начало функции:

```
if (hp < 0) {
    return "Некорректное здоровье"
}
```

Следующее **задание** – **Расчёт золота в час (goldPerHour())**

Создайте метод, который вычисляет, сколько золота (**reward**) игрок получает в час, учитывая длительность квеста (**duration**).

Решение.

Проанализируем требования.

Формула для расчёта:

$$\text{Золото в час} = \frac{\text{Награда}}{\text{Длительность}}$$

Особые случаи: Если длительность = 0, возвращаем 0 (деление на ноль).

Реализуем функцию:

```
fun goldPerHour(): Int {  new *
    if (duration == 0) return 0
    return reward / duration
}
```

Пояснение:

- Проверяем, что **duration** не равен нулю.
- Делим **reward** на **duration** (целочисленное деление, так как тип Int).

Пример использования:

```
val quest1 = Quest( title = "Охота", duration = 2, reward = 300, difficulty = "Средний")
println(quest1.goldPerHour())  // 150

val quest2 = Quest( title = "Рыбалка", duration = 0, reward = 500, difficulty = "Лёгкий")
println(quest2.goldPerHour())  // 0
```

Также мы можем добавить проверку, что **duration** и **reward** неотрицательные:

```
fun goldPerHour(): Int {  2 Usages   new *
    require( value = duration >= 0 && reward >= 0) { "Длительность и
     награда не могут быть отрицательными!" }
    if (duration == 0) return 0
    return reward / duration
}
```

Шаг 7: Организация кода с помощью пакетов

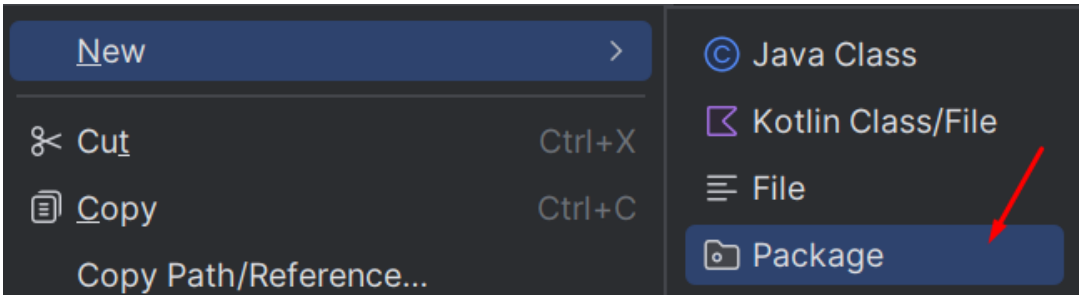
Как вы уже заметили, наш проект разрастается: у нас появились много классов (**Hero, Enemy, Spell, Item, Quest, Shop, Contract, Guild,** и др.) и их число будет расти.

Чтобы **структурировать код**, в **Kotlin** принято использовать **пакеты**.

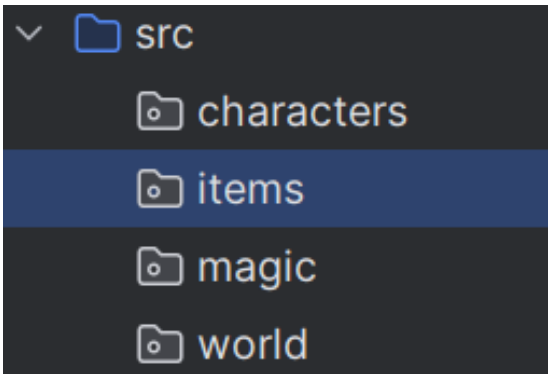
📁 Создание пакета

Чтобы создать пакет:

1. ПКМ по папке **src → New → Package**



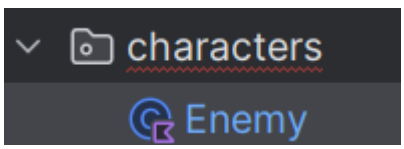
2. Создайте пакеты в папке **src - characters; magic**; **world**; **items:**



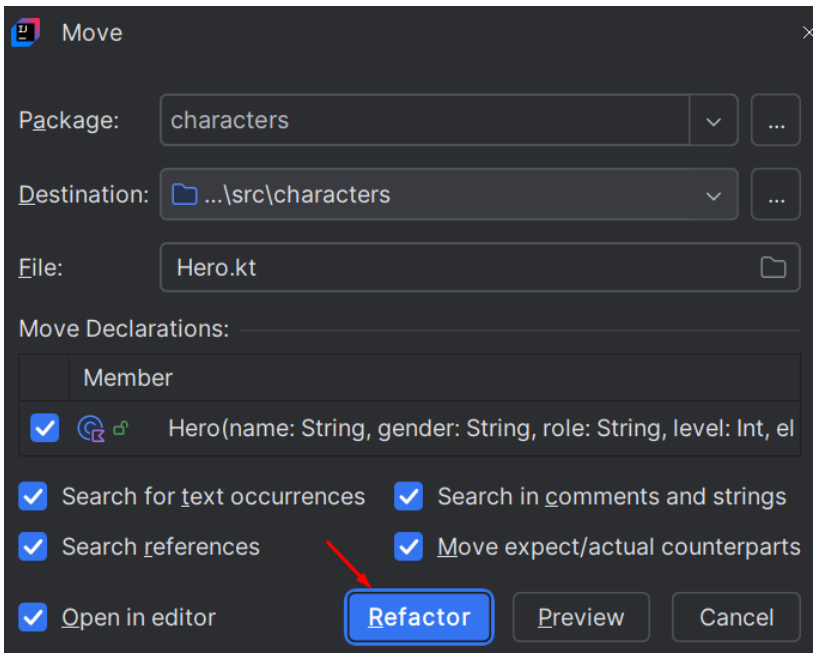
Правила именования пакетов: с маленькой буквой, на английском языке, желательно одним слово, если не получается, то новое слово начинается с заглавной буквы.

3. Перенесите файлы в соответствующие пакеты (перетаскиванием в IDE).

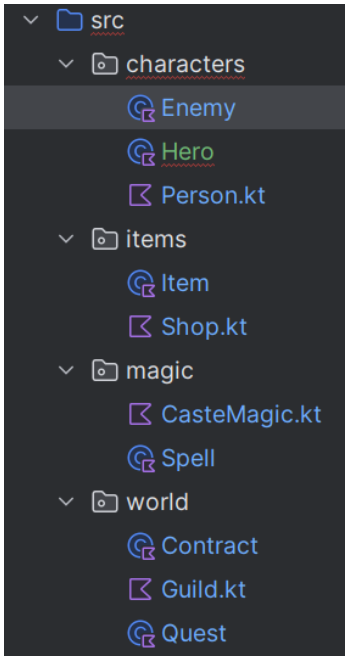
Переносим файл Enemy в пакет characters обычным перетаскиванием:



При переносе соглашаемся на изменения:



Перенесите следующим образом:

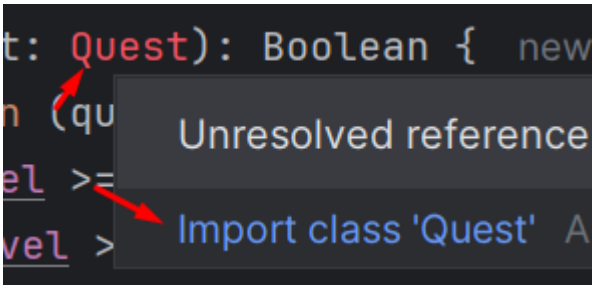


- **characters** — персонажи (Hero, Enemy, Person)
- **magic** — заклинания и магические действия (Spell, CasteMagic)
- **world** — мир, квесты, гильдии (Quest, Guild, Contract)
- **items** — предметы, магазины (Item, Shop)

Обратите внимание класс Hero Класс **Hero** подсвечивается красным — это указывает на ошибку компиляции. При наведении курсора на подчёркнутый класс **Quest** IDE показывает сообщение: *"Unresolved reference: Quest"* (ссылка на класс не распознана).

Причина проблемы - Класс **Quest** находится в другом пакете (**world**), но не импортирован в текущий файл. В Kotlin/Java для использования классов из других пакетов требуется явное указание импорта.

IDE автоматически предлагает исправить ошибку (лампочка или сочетание клавиш **Alt+Enter**). Выбираем вариант *"Import class"* → **import world.Quest**.



В начале файла автоматически добавляются строки:

```
package characters
import world.Quest
```

- **package characters** - указывает принадлежность класса **Hero** к пакету **characters**.
- **import world.Quest** - даёт доступ к классу **Quest** из пакета **world** без указания полного пути **(world.Quest)**.

Самостоятельные задания

Задание 1: Создание класса Item (предмет) с несколькими конструкторами

Описание: Создайте класс Item (предмет), который имеет следующие свойства:

- **name (String)** — название предмета
- **type (String)** — тип ("оружие", "броня", "зелье" и т.д.)
- **value (Int)** — стоимость

Требования:

- Реализуйте **основной конструктор**, принимающий все параметры.
- Реализуйте **вторичный конструктор**, принимающий только name и type, а value устанавливает в 0.
- Добавьте метод printInfo(), который выводит информацию о предмете.

Задание 2: Использование первичного конструктора в классе Location

Описание: Создайте класс Location, представляющий игровую локацию. Свойства:

- **name (String)**
- **dangerLevel (String)**

- **requiredLevel (Int)**

  Требования:

- Используйте **первичный конструктор** с параметрами по умолчанию.
- Добавьте метод isDangerous(): Boolean, возвращающий true, если dangerLevel == "Высокий".

  ### Задание 3: Проверка доступности заклинания героем

  Описание: Дополните класс Hero методом:

  fun canCast(spell: Spell): Boolean

- Герой может применить заклинание, если:
  - Длина имени заклинания ≤ уровень героя * 2
  - Ширина и высота заклинания ≤ 5

  Проверьте это условие и верните true или false.

  ### Задание 4: Калькуляция силы врага

  Описание: Добавьте в класс **Enemy** метод:

  fun calculatePower(): Int

- Формула: **power = health * aggressionLevel**, где **aggressionLevel** (передайте как дополнительное свойство или установите по умолчанию = 2).
- Верните это значение.