Лабораторная работа №5: Введение в ООП

Цель: Познакомиться с основами объектно-ориентированного программирования на языке Kotlin: научиться создавать собственные классы и объекты, описывать их свойства и методы, передавать параметры в функции, использовать ссылку this и именованные аргументы. Научиться организовывать взаимодействие между объектами, проектировать начальные игровые модели — героев, врагов и задания. Получить навыки постепенного наращивания функционала через расширение классов и их поведения.

Шаг 1. Понятие классов и объектов

В любом файле объявим переменную:

```
val number = 0
```

Теперь укажем явно тип данных:

```
val number: Int = 0
```

Что такое **тип данных**? Это описание того, **что можно делать** с переменной. У разных типов — разные **свойства** и **методы**.

Попробуйте:

- Зажмите **Ctrl** и кликните по **Int** (или нажмите **Ctrl** + **B**).
- Вы перейдёте внутрь определения типа **Int**.

Внутри вы увидите объявление:

```
public class Int private constructor() : Number(), Comparable<Int>
    public companion object {
        A constant holding the minimum value an
        instance of Int can have.
        public const val MIN_VALUE: Int = -2147483648
```

А также разные свойства и методы, например:

```
public const val MIN_VALUE: Int = -2147483648
```

Вывод: **Int** — это **класс**, описывающий, как работает целое число. Когда мы пишем **val number: Int = 0**, мы создаём **объект класса Int** — реальную переменную с конкретным значением.

Что такое класс и объект?

Понятие	Объяснение
Класс	Шаблон, описание "что это такое", какие свойства и поведение есть у объектов
Объект	Конкретный экземпляр класса, с которым мы уже можем работать в коде

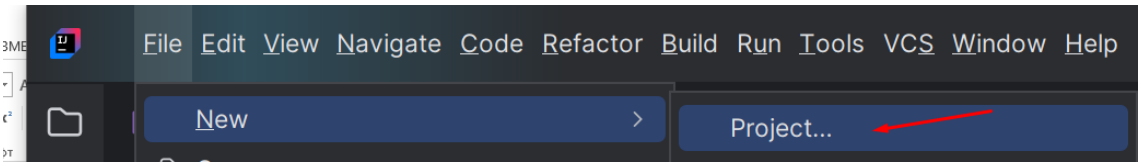
Примеры из жизни:

- **Класс**: Чашка — у неё есть свойства (цвет, объём) и поведение (налить, выпить).
- **Объект**: Моя синяя чашка на столе — конкретный экземпляр класса Чашка.

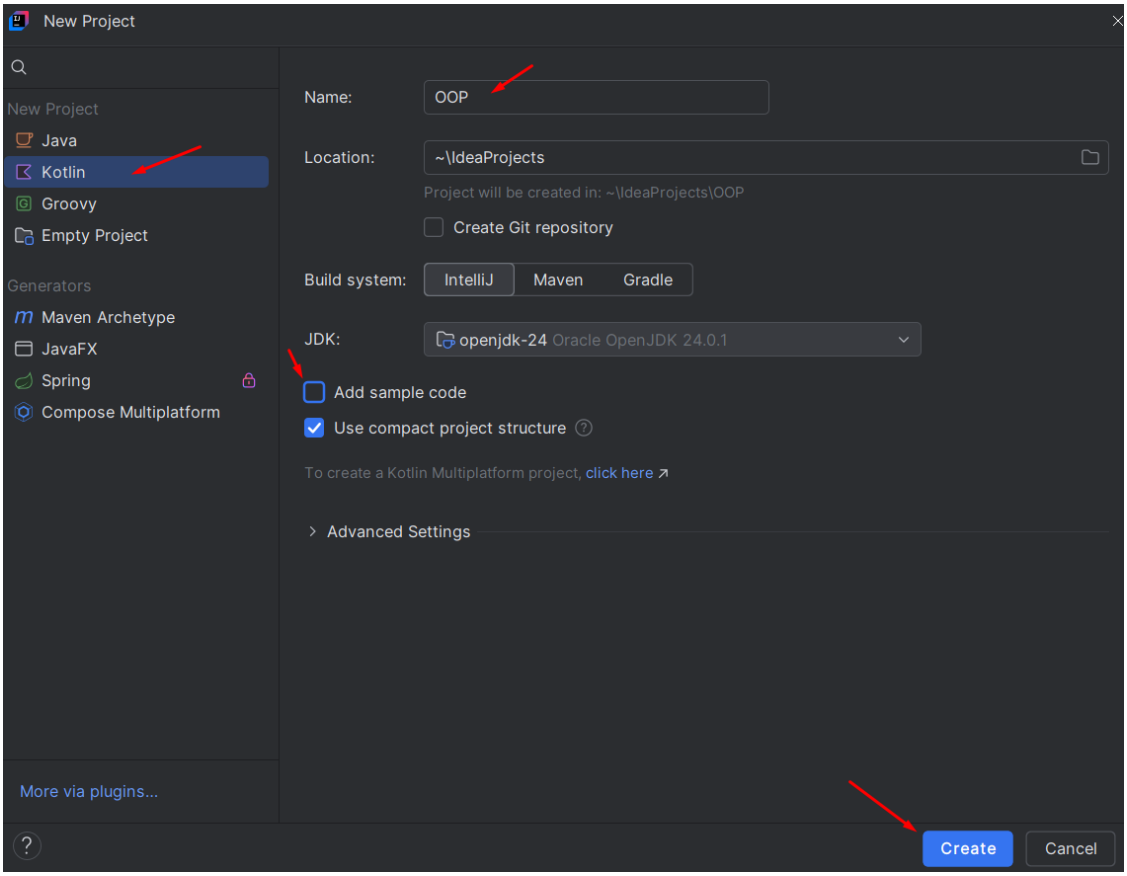
Шаг 2. Создание класса и объекта

Во всех предыдущих лабораторных работах мы использовали уже готовые типы данных, такие как Int, Boolean, String и другие. Теперь пришло время создать свой собственный тип данных.

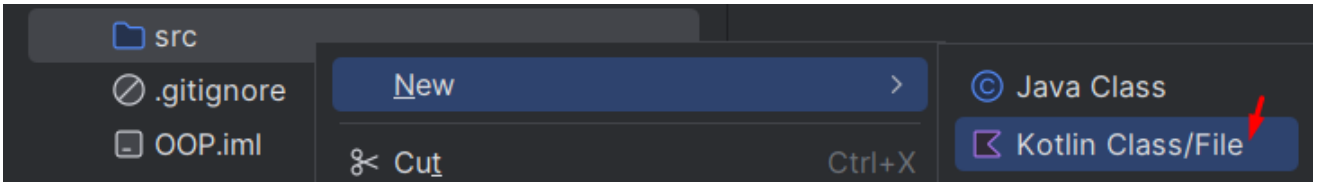
Создайте новый проект: **File → New → Project**



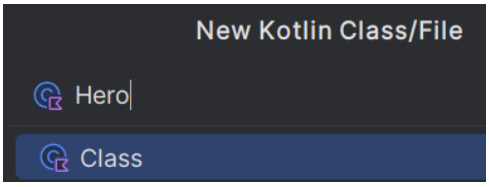
Укажите имя проекта - **OOP**, снимите галочку с **Add sample code** и нажмите **Create**.



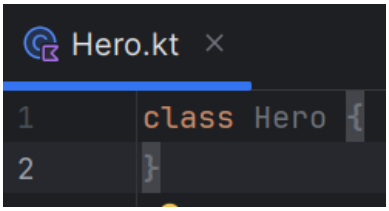
Затем кликните правой кнопкой мыши по папке **src** и выберите: **New → Kotlin Class/File**



Назовите новый класс **Hero**. По правилам имена классов пишутся с заглавной буквы, на английском, и при необходимости объединяются заглавными буквами (например, **MagicHero**).



У нас появился файл Hero.kt, в котором автоматически создана структура класса:

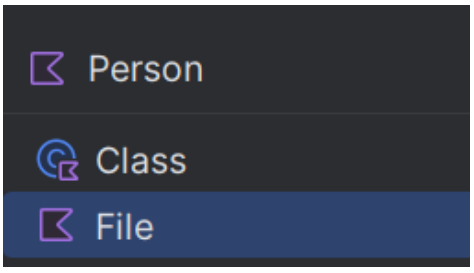


Внутри фигурных скобок — тело класса. Здесь мы объявим свойства нашего будущего героя:

```
class Hero {
    val name: String = "Неизвестный"
    val gender: String = "Не указан"
    val role: String = "Бродяга"
    val hp: Int = 100
    val mp: Int = 50
}
```

Мы пока используем значения по умолчанию, так как ещё не изучили конструкторы. К этому мы вернёмся позже.

Таким образом, мы описали **шаблон** (или тип данных), и теперь можем на его основе создавать **объекты** — конкретных героев.

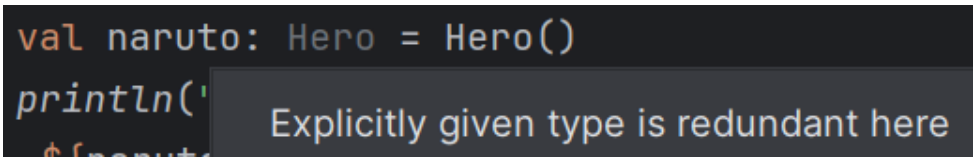


В нём добавляем точку входа в нашу программу **main**:

Для того чтобы создать объект, мы используем для указания типа данных наш класс и затем присваиваем его класс с круглыми скобками:

```
val naruto: Hero = Hero()
```

Тип **Hero** подсвечивается **серым**, потому что **Kotlin** умеет выводить тип автоматически. Мы можем его не указывать, и запись будет эквивалентной.



Выведем на консоль свойства нашего объекта. Для обращения к ним ставим точку, также обратите внимание на фигурные скобки:

```
val naruto: Hero = Hero()
println("Имя: ${naruto.name} \nПол: ${naruto.gender} \nКласс:
 ${naruto.role}\nЗдоровье: ${naruto.hp}\nМана: ${naruto.mp}")
```

Запускаем наш файл и видим вывод на консоль:

```
C:\Users\Leontev\
Имя: Неизвестный
Пол: Не указан
Класс: Бродяга
Здоровье: 100
Мана: 50
```

Сейчас у нас выводятся свойства по умолчанию, но что если бы мы хотели задать свои? Для этого через точечную нотацию вы также можете обращаться к ним и присвоить другие значения.

Если вы попытаетесь изменить свойства объекта:

```
val naruto: Hero = Hero()
naruto.name = "Наруто Узумаки"
```

Наводим на ошибку и видим описание **'val' cannot be reassigned**. Это потому что мы объявили свойства с ключевым словом val, а оно означает **нельзя менять значение после инициализации**.

Заменим **val** на **var** в файле **Hero.kt**, чтобы разрешить изменения:

```
class Hero {  2 Usages
    var name: String = "Неизвестный"  2 Usages
    var gender: String = "Не указан"  2 Usages
    var role: String = "Бродяга"  2 Usages
    var hp: Int = 100  2 Usages
    var mp: Int = 50  2 Usages
}
```

Теперь можно задать индивидуальные значения:

```
val naruto: Hero = Hero()
naruto.name = "Наруто Узумаки"
naruto.gender = "Мужской"
naruto.role = "Хокаге"
naruto.hp = 200
naruto.mp = 150
```

И если мы теперь выведем их на консоль, то увидим соответствующий результат:

```
C:\Users\Leontev\.jd
Имя: Наруто Узумаки
Пол: Мужской
Класс: Хокаге
Здоровье: 200
Мана: 150
```

Конечно же мы можем создавать и другие объекты. Так создадим ещё один объект - Саске Учиха:

```
val sasuke = Hero()
sasuke.name = "Саске Учиха"
sasuke.gender = "Мужской"
sasuke.role = "Шиноби-отступник"
sasuke.hp = 120
sasuke.mp = 180
println("Имя: ${sasuke.name} \nПол: ${sasuke.gender} \nКласс: ${sasuke
 .role}\nЗдоровье: ${sasuke.hp}\nМана: ${sasuke.mp}")
```

Для практики самостоятельно добавьте третьего героя — **Годжо Сатору**:

```
Имя: Годжо Сатору
Пол: Мужской
Класс: Маг Проклятий
Здоровье: 160
Мана: 300
```

В данный момент всё что мы делали это перезаписывали свойства. Давайте теперь сделаем возможность пользователю самому ввести информацию о герое:

```
val hero = Hero()
print("Введите имя героя: ")
hero.name = readln()
print("Введите пол героя: ")
hero.gender = readln()
print("Введите класс героя: ")
hero.role = readln()
print("Введите здоровье героя: ")
hero.hp = readln().toInt()
print("Введите ману героя: ")
hero.mp = readln().toInt()

println("Имя: ${hero.name} \nПол: ${hero.gender} \nКласс:
 ${hero.role}\nЗдоровье: ${hero.hp}\nМана: ${hero.mp}")
```

Теперь, когда мы научились создавать **класс**, создавать **объекты** и **менять их свойства**, давайте усилим наш класс ещё двумя дополнительными полями. Добавьте для вашего героя ещё два свойства самостоятельно:

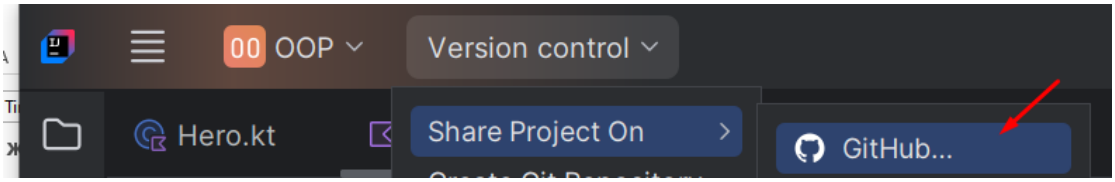
- уровень героя
- стихия героя (огонь, вода, воздух, земля и т.д.)

Пример вывода:

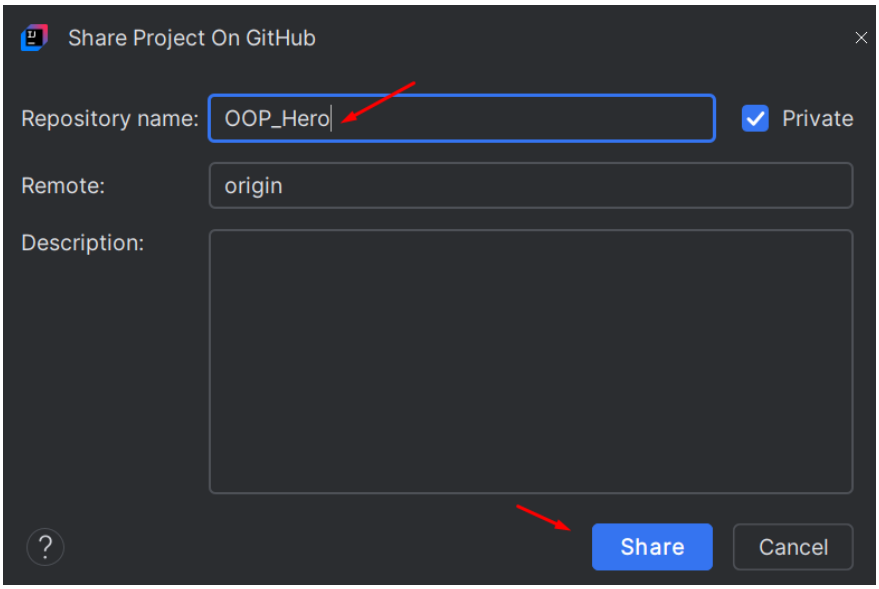
```
Имя: Артур
Пол: мужской
Класс: паладин
Здоровье: 350
Мана: 400
Уровень: 400
Элемент: 400
```

Шаг 3. Создание Git репозитория

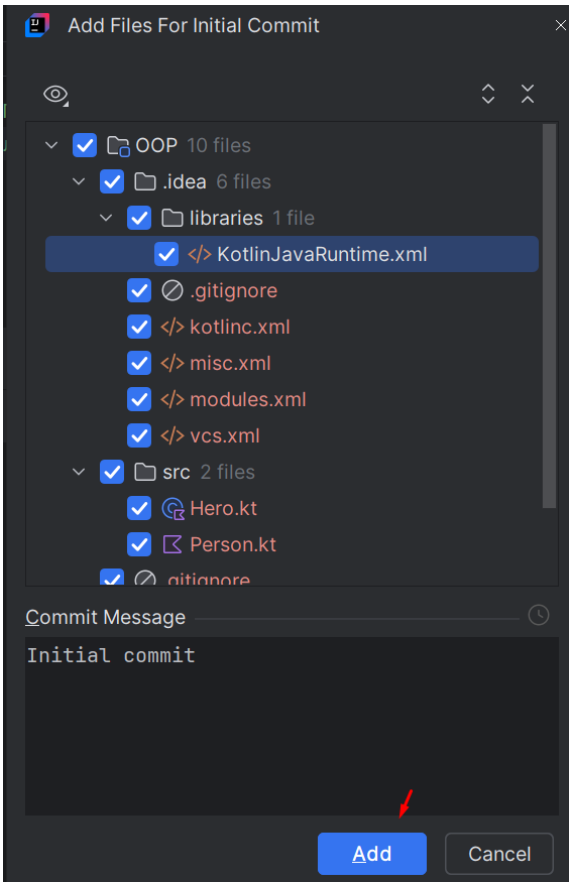
Нажмите на **Version control** и выберите пункт **Share Project On -> GitHub**:



Если вы не вошли в свою учетную запись, то войдите на данном шаге, затем нажмите **Share**:



Подтвердите добавление коммита:



Шаг 4. Методы внутри класса (функции как часть объекта)

Ранее мы писали функции отдельно:

```
fun hello() {
    println("Привет, герой!")
}
```

Но теперь мы работаем с **объектами**, и логично, чтобы их действия описывались **внутри самого класса**. Например, герой может:

- Представиться
- Атаковать
- Восстановить ману

Добавим функцию в класс **Hero**:

1. Функция приветствия:

```
fun sayHello() {  new *
    println("Я – $name, мой путь только начинается!")
}
```

2. Функцию выводящую информацию о герое:

```
fun showStats() {  new *
    println("Имя: $name | Класс: $role | HP: $hp | MP: $mp |
     Level: $level | Element: $element")
}
```

3. Функция медитации, которая восстанавливает ману:

```
fun meditate() {  new *
    println("$name медитирует...")
    mp += 20
    println("Мана восстановлена! Текущая мана: $mp")
}
```

4. Функцию получения урона:

```
fun takeDamage() {  new *
    println("$name получает урон!")
    hp -= 10
    println("Осталось здоровья: $hp")
}
```

Теперь в main() мы можем у нашего героя и вызывать его методы:

```
hero.sayHello()
hero.showStats()
hero.meditate()
hero.takeDamage()
```

После запустите программу, создайте нового героя, и посмотрите на вывод.

Теперь самостоятельно добавьте два метода:

- **метод castSpell()** — при вызове у героя уменьшается **mp** на **10**, и в консоли пишется, что герой применяет магию. Вызовите метод 10 раз. Когда **мана** станет равно **0**, запретить использование магии.
- **метод heal()** — восстанавливает **hp** на **10**, но тратит **10 mp**. Если не хватает маны, выведите сообщение об ошибке (Недостаточно маны! У вас только XX). Если здоровье полное, выведите сообщение - Ваше здоровье уже полное!

Пример вывода:

```
Лечусь! Восстановлено здоровье, потрачено 10 маны.
Здоровье: 60 | Мана: 0
Недостаточно маны! У вас только 0.
```

Шаг 5. Функции с параметрами и взаимодействие объектов

Создадим в нашем классе простую функцию:

```
fun greet(name: String) {  2 U
    println("Привет, $name!")
}
```

- **name** — это **параметр** функции.
- Он указывается в круглых скобках при **объявлении**.
- Когда вы вызываете эту функцию, ты передаёте туда **аргумент**:

В методе **main** вызовите её:

```
fun main() {  Denis *
    val hero = Hero()
    hero.greet( name = hero.name)
    hero.greet( name = "Наруто")
```

В первом случае мы использовали **встроенное свойство объекта** hero.name, чтобы передать его имя в метод greet(). Во втором — передали **строку напрямую как аргумент** "Наруто", независимо от имени, сохранённого в объекте.

То есть:

- **hero.greet(hero.name)** — использует текущее имя из самого героя
- **hero.greet("Наруто")** — игнорирует внутреннее имя и передаёт внешнее значение

Мы можем добавить **функцию с параметром**, например, чтобы герой получал урон в разном размере:

```
fun takeDamage(amount: Int) {  Denis *
    println("$name получает $amount урона!")
    hp -= amount
    if (hp < 0) hp = 0
    println("Осталось здоровья: $hp")
}
```

Теперь мы можем передать в неё любой урон:

```
fun main() {  Denis *
    val hero = Hero()
    hero.takeDamage( amount = 30)
    hero.takeDamage( amount = 20)
```

Напишите метод **die(),** при котором если у персонажа становится 0 здоровья, то выводится сообщение – «**Герой умер**» и вызовите его в методе **takeDamage()**.

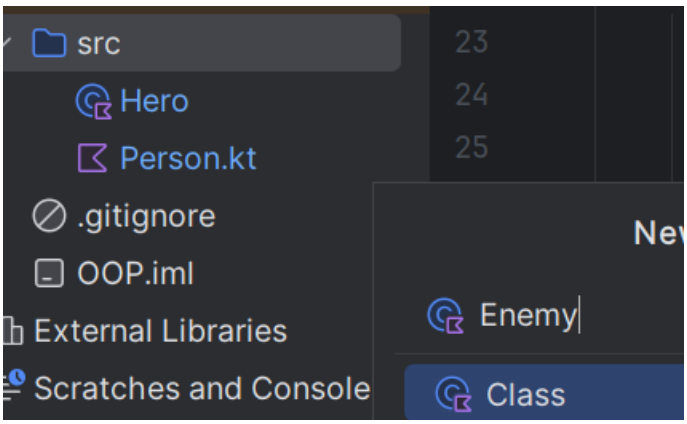
Создайте класс **Enemy**, который представляет противника в игре.

Требования:

1. **Название класса**: Enemy
2. **Свойства (поля)**:
    - **name**: строка, по умолчанию "Враг"
    - **hp**: целое число, по умолчанию 50 — количество здоровья
3. **Метод**: **takeDamage(amount: Int)**
    - Принимает значение урона
    - Выводит сообщение о получении урона
    - Уменьшает **hp** на **amount**
    - Если hp становится меньше нуля — устанавливает его в **0**
    - Выводит оставшееся здоровье

Постарайтесь выполнить самостоятельно, прежде чем смотреть решение.

В папке **src** создаём новый класс **Enemy:**



Внутри класса объявляем поля имени противника и его здоровья:

```
class Enemy {  new *
    var name: String = "Враг"
    var hp: Int = 50
}
```

Далее согласно условию пишем метод **takeDamage():**

```
fun takeDamage(amount: Int) {  new *
    println("$name получает $amount урона!")
    hp -= amount
    if (hp < 0) hp = 0
    println("Осталось HP врага: $hp")
}
```

В методах можно передавать параметры любого типа, включая **собственные классы**. Это даёт возможность одному объекту воздействовать на другой.

Теперь в классе **Hero** добавим функцию с параметром типа **Enemy**:

```
fun attack(enemy: Enemy, damage: Int) {  new *
    println("$name атакует врага ${enemy.name}!")
    enemy.takeDamage( amount = damage)
}
```

Внутри функции мы можем вызвать enemy.takeDamage(...) — это значит, что **один объект воздействует на другой**.

Пример использования в **main**:

```
val naruto = Hero()
naruto.name = "Наруто"
naruto.role = "Шиноби"
naruto.element = "Ветер"

val orochimaru = Enemy()
orochimaru.name = "Орочимару"
naruto.showStats()
println("Наруто увидел врага ${orochimaru.name}! Его здоровье - ${orochimaru
 .hp}")
naruto.attack( enemy = orochimaru,  damage = 25)
```

Запустите на выполнение программу и увидите, что в результате:

- Герой Наруто атакует противника Орочимару
- Урон передаётся через метод takeDamage
- Выводится информация о действии и оставшемся здоровье врага

Решим следующие задачи:

1. Добавим в класс **Hero** функцию **castSpellOn**(), чтобы магия тоже наносила урон. В качестве параметров передадим ей:
    - **enemy: Enemy**
    - **spellName: String**
    - **damage: Int**

2. Добавим в **Enemy** свойство **element: String** и сделаем проверку: если элемент героя совпадает с элементом врага — урон уменьшается.

3. Создадим двух героев и сделаем функцию **duel(opponent: Hero)**, где они наносят урон друг другу по очереди.

Первая задача. В классе **Hero** создаём метод **castSpellOn**() с параметрами:

```
fun castSpellOn(enemy: Enemy, spellName: String, damage: Int)
```

- **enemy: Enemy** — объект врага, по которому будет нанесён магический урон.
- **spellName: String** — название заклинания, чтобы красиво вывести в консоль.
- **damage: Int** — базовое значение урона, которое может быть изменено по условиям.

Выводим действие заклинания:

```
println("$name использует заклинание '$spellName' против ${enemy.name}!")
```

Далее создадим блок проверки:

```
if (element == enemy.element) {
    println("Стихии совпадают! Урон снижен.")
    val reducedDamage = damage / 2
    enemy.takeDamage( amount = reducedDamage)
} else {
    enemy.takeDamage( amount = damage)
}
```

- Проверка стихии (условие if)

Если у героя и врага одинаковая стихия (element), магия менее эффективна.

Пример: герой с элементом "огонь" применяет магию против врага, у которого тоже "огонь" → урон снижается.

- Вычисляем уменьшенный урон. Если стихии совпадают, урон делится на два и применяется.
- Иначе — обычный урон (блок else). Если стихии разные, урон проходит полностью.

Тратим ману. Магия расходует 10 единиц маны. Если мана ушла в минус, устанавливаем её в 0:

```
mp -= 10
if (mp < 0) mp = 0
```

Выводим остаток маны:

```
println("Осталось маны: $mp")
```

Вторая задача. Решается довольно просто. Добавляем в Enemy новый элемент. И ставим по умолчанию элемент, например, огонь:

```
class Enemy {  3 Usages
    var name: String = "Враг"  4 Usages
    var hp: Int = 50  4 Usages
    var element: String = "Огонь"  2 Usages
```

Третья задача. В классе **Hero** создаём метод **duel(opponent: Hero)**, который имитирует пошаговое сражение между двумя героями:

```
fun duel(opponent: Hero) {  1 Usage   Denis
    println("Дуэль между $name и ${opponent.name} начинается!")

    println("$name атакует первым!")
    opponent.takeDamage( amount = 15)

    println("${opponent.name} отвечает!")
    this.takeDamage( amount = 15)

    println("Дуэль завершена.")
    println("Состояние героев:")
    this.showStats()
    opponent.showStats()
}
```

- **opponent: Hero** — параметр типа Hero, позволяет одному объекту взаимодействовать с другим.
- **opponent.takeDamage**(...) — герой атакует своего оппонента.
- **this.takeDamage**(...) — ответный удар.
- **showStats**() — метод, который выводит текущее состояние (должен быть реализован заранее).

Пример использования в **main**:

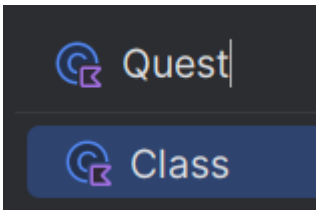
```
fun main() {  Denis
    val naruto = Hero()
    naruto.name = "Наруто"
    naruto.role = "Шиноби"
    naruto.element = "Ветер"
    naruto.mp = 100

    val kakashi = Hero()
    kakashi.name = "Какаши"
    kakashi.role = "Шиноби"
    kakashi.element = "Молния"
    kakashi.mp = 100

    val orochimaru = Enemy()
    orochimaru.name = "Орочимару"
    orochimaru.element = "Ветер"

    naruto.castSpellOn( enemy = orochimaru,  spellName = "Расенган",  damage = 30)
    naruto.duel( opponent = kakashi)
```

Шаг 6. Именованные параметры, this и init

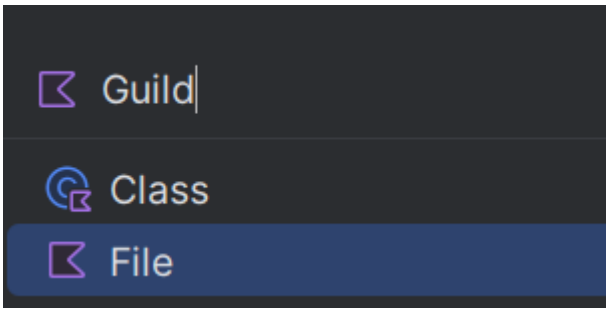
Создайте класс **Quest**, содержащий следующие свойства:

- **title** - Название задания
- **duration** - Время выполнения (в часах)
- **reward** - Награда за выполнение (монеты)
- **difficulty** - Уровень сложности

В папке **src** создаём новый класс **Quest:**



Также для выводи создадим новый файл **Guild**:



В нём объявим точку входа в программу метод **main**:

```
fun main() {

}
```

Теперь возвращаемся к нашему классу **Quest**. Согласно условию создаём свойства, и присваиваем им значения по умолчанию:

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

Далее в **Guild** создадим экземпляр нашего класса:

```
val quest = Quest()
```

Далее запросим ввод пользователя и сохраним в переменные:

```
print("Введите название квеста: ")
val title = readln()
print("Введите время выполнения (в часах): ")
val duration = readln().toInt()
print("Введите награду (в монетах): ")
val reward = readln().toInt()
print("Введите уровень сложности: ")
val difficulty = readln()
```

- **title**: название задания
- **duration**: продолжительность в часах
- **reward**: количество монет за выполнение
- **difficulty**: уровень сложности, например, "лёгкий", "средний", "тяжёлый"

Затем присваиваем значения объекту:

```
quest.title = title
quest.duration = duration
quest.reward = reward
quest.difficulty = difficulty
```

В данном коде мы передаём полученные данные в поля объекта **quest**.

И после выводим итоговую информацию:

```
println("Название квеста: ${quest.title} Время выполнения: ${quest.duration}
  Награда: ${quest.reward} золотых Уровень сложности: ${quest.difficulty}")
```

Запустим программу и проверим что она корректно работает.

Всё работает корректно, но такой способ подходит, если у нас **только один квест**. Но если будет несколько заданий или нужно часто выводить данные — писать такую длинную строку каждый раз неудобно и нарушает читаемость.

Решение: добавим метод **printInfo**() внутри класса. Если мы раньше могли обращаться к объекту, то внутри класса для этого есть ключевое слово **this.** В Kotlin означает **"этот объект"** — то есть текущий экземпляр класса, в котором находится метод.

```
fun printInfo() {  new *
    println("Название квеста: ${this.title} Время выполнения: ${this.duration}
     Награда: ${this.reward} золотых Уровень сложности: ${this.difficulty}")
}
```

В нашем примере:

- **this.title** означает "взять поле **title** из текущего объекта **Quest**"
- мы могли бы написать просто **title**, но **this.title** помогает явно указать, что это **свойство объекта**, а не переменная из main() или другого места

Когда использовать this:

Ситуация	**Пример**
Внутри класса, чтобы обратиться к его полям	this.hp или this.title
Когда имена переменных совпадают с параметрами	this.name = name в конструкторе
Чтобы подчеркнуть, что обращаемся к текущему объекту	при взаимодействии с другими объектами

В файле **Guild.kt** вызовите метод **printInfo()** у нашего квеста:

```
quest.printInfo()
```

Далее обратите внимание, когда мы заполняли объект **quest** вручную:

```
quest.title = title
quest.duration = duration
quest.reward = reward
quest.difficulty = difficulty
```

Это работало, но при большом количестве объектов становится неудобно: нужно повторять строки присваивания каждый раз, легко допустить ошибку, и код теряет читаемость.

Решение — создаём в классе Quest отдельную функцию init()

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

Метод init(...) принимает нужные параметры и **сразу заполняет все поля объекта**

Мы используем ключевое слово this, чтобы сказать: “обращаемся к свойствам текущего объекта”

Теперь используем в main:

```
quest.init(title, duration, reward, difficulty)
```

Только одна строка вместо четырёх, а код выглядит аккуратно и профессионально.

Когда мы вызываем метод так:

```
quest.init( title = "Охота на тролля",  duration = 3,  reward = 500,  difficulty = "Средний")
```

все параметры передаются **по порядку**, и Kotlin считает:

1. "Охота на тролля" — title
2. 3 — duration
3. 500 — reward
4. "Средний" — difficulty

При этом компилятор автоматически добавляет аргументы.

Но если перепутать местами reward и duration, то получится неправильное поведение:

```
quest.init( title = "Охота на тролля",  duration = 3,  reward = 500,  difficulty = "Средний")
quest.init( title = "Охота на тролля",  duration = 500,  reward = 3,  difficulty = "Средний")
```

Теперь у нас 500 часов, и награда 3 монеты.

Решение — использовать именованные параметры

Kotlin позволяет **явно указать, какой аргумент к какому параметру относится**, независимо от порядка:

```
quest.init(
    title = "Охота на тролля",
    reward = 500,
    duration = 3,
    difficulty = "Средний"
)
```

Теперь мы не зависим от порядка — код стал более читаемым и защищённым от ошибок.

Шаг 7. Список квестов

Пусть пользователь сам добавит несколько квестов, которые будут храниться в списке:

```
val quests = mutableListOf<Quest>()
```

Это **список объектов типа Quest**, который можно дополнять, изменять и перебирать. Пока что она пустая.

Далее создадим цикл for и три раза повторим процесс создания и добавления квеста:

```
for (i in 1 ≤ .. ≤ 3) {

}
```

Внутри цикла запрашивается ввод от пользователя (название, время, награда, сложность) и создаётся объект Quest:

```
println("Добавим квест #$i")
val q = Quest()
print("Название: ")
val title = readln()
print("Время (ч): ")
val duration = readln().toInt()
print("Награда: ")
val reward = readln().toInt()
print("Сложность: ")
val difficulty = readln()
```

После заполняется через метод init(...) и добавляется в коллекцию с помощью add:

```
q.init(title, duration, reward, difficulty)
quests.add(q)
println()
```

На этом цикл завершается. Далее просмотрим все элементы коллекции:

```
}

println("Все доступные квесты:")
for (q in quests) {
    q.printInfo()
}
```

Мы используем цикл for по коллекции quests, и каждый элемент (q) — это объект Quest. Для каждого объекта вызывается метод printInfo(), который выводит параметры квеста красиво и структурно.

Выведи только те квесты, где сложность, например, "Сложный":

Самостоятельные задания

Задание 1. Создай класс Weapon

Класс Weapon должен содержать:

Свойство	Тип	По умолчанию
name	String	"Оружие"
damage	Int	10
durability	Int	100
type	String	"Обычное"

Методы:

- **showInfo**() — выводит полную информацию
- **use**() — уменьшает прочность (**durability**) на 10. Если <= 0, выводит: "Оружие сломано!"
- **upgrade(bonus: Int)** — увеличивает урон на **bonus**

Задание 2. Класс GuildHero — гильдия героев

Свойство	Тип
name	String
members	MutableList<Hero>

Методы:

- **addMember(hero: Hero**) — добавляет героя в список
- **showMembers()** — выводит информацию обо всех героях
- **averageLevel()** — выводит средний уровень гильдии