Лабораторная работа №4: Функции

Цель:

1. **Освоить синтаксис функций**
   - Создание функций с параметрами и возвращаемыми значениями.
   - Управление областью видимости (top-level, локальные функции).
2. **Изучить механизмы для работы с функциями**
   - Перегрузка функций для обработки разных типов данных.
   - Использование значений по умолчанию и именованных аргументов.
3. **Понять продвинутые концепции**
   - Анонимные функции и лямбда-выражения.
   - Функции высшего порядка для реализации событий и колбэков.
   - Замыкания для сохранения состояния.
4. **Научиться применять функции в реальных сценариях**
   - Моделирование игровых процессов: атака, лечение, генерация событий.
   - Оптимизация кода через переиспользование функций.

Шаг 1. Основы функций

Одним из строительных блоков программы являются функции. Функция определяет некоторое действие. В Kotlin функция объявляется с помощью ключевого слова **fun**, после которого идет название функции. Затем после названия в скобках указывается список параметров. Если функция возвращает какое-либо значение, то после списка параметров через двоеточие можно указать тип возвращаемого значения. И далее в фигурных скобках идет тело функции:

```
fun имя_функции(параметры: Тип): Возвращаемый_тип {
    // Тело функции
}
```

- **fun** — ключевое слово для объявления функции.
- **имя_функции** — название, по которому функция вызывается.
- **параметры** — входные данные (не обязательны).
- **Возвращаемый_тип** — тип результата (если функция что-то возвращает).

Пример: Функция без параметров

Функция, которая выводит приветствие игрока:

```
fun main() {
    greetPlayer()  // Вызов функции
    greetPlayer()  // Вызов ещё раз
}

// Определение функции
fun greetPlayer() {  2 Usages
    println("Добро пожаловать в игру!")
}
```

Запустите программу.

Как это работает?

1. Функция **greetPlayer**() не принимает параметров и не возвращает значение.
2. Она просто выводит текст при каждом вызове.
3. Её можно вызывать многократно, избегая повторения кода.

Функции можно определять в файле вне других функций или классов, сами по себе, как например, определяется функция **main**. Такие функции еще называют функциями верхнего уровня (**top-level functions**).

Шаг 2. Передача параметров в функцию

Функции могут принимать данные для работы. Например, можно передать имя игрока или его уровень.

Пример: Функция с одним параметром

Функция, которая выводит сообщение в чат игры:

```
fun main() {
    showChatMessage( text = "Привет, рыцарь!")
    showChatMessage( text = "Враг приближается!")
}

fun showChatMessage(text: String) {  2 Usages
    println(text)
}
```

Как это работает?

- Функция **showChatMessage** принимает один параметр **text** типа **String**.
- При вызове функции в скобках передаётся конкретное сообщение.

Пример: Функция с несколькими параметрами

Функция, которая выводит данные игрока:

```
fun main() {
    showPlayerStats( name = "Aragorn",  health = 100)
    showPlayerStats( name = "Legolas",  health = 85)
}

fun showPlayerStats(name: String, health: Int) {  2 Usages
    println("Игрок: $name | Здоровье: $health HP")
}
```

Вывод:

```
Игрок: Aragorn | Здоровье: 100 HP
Игрок: Legolas | Здоровье: 85 HP
```

Как это работает?

- Функция принимает два параметра: **name** (строка) и **health** (число).
- При вызове значения передаются **по порядку**: сначала имя, потом здоровье.

Шаг 3. Необязательные параметры (значения по умолчанию)

Иногда не все параметры функции обязательны. Kotlin позволяет задать значения по умолчанию, сделав параметры необязательными.

Пример: Персонаж с необязательными параметрами

```
fun createCharacter(name: String, level: Int = 1, classType: String = "Новичок") {  3 Usages
    println("Имя: $name | Уровень: $level | Класс: $classType")
}

fun main() {
    createCharacter( name = "Арагорн",  level = 10,  classType = "Рыцарь") // Все параметры переданы
    createCharacter( name = "Леголас",  level = 5) // Пропущен classType (используется "Новичок")
    createCharacter( name = "Гэндальф")  // Только обязательный name
}
```

Вывод:

```
Имя: Арагорн | Уровень: 10 | Класс: Рыцарь
Имя: Леголас | Уровень: 5 | Класс: Новичок
Имя: Гэндальф | Уровень: 1 | Класс: Новичок
```

Правила:

- Если у параметра есть значение по умолчанию, он становится необязательным.
- Все параметры **после** необязательного тоже должны быть необязательными.

Шаг 4. Именованные аргументы

Kotlin позволяет передавать аргументы в любом порядке, указывая их имена.

Пример: Создание предмета с именованными аргументами

```
fun createItem(name: String, price: Int, isMagic: Boolean = false) {  3 Usages
    println("Предмет: $name | Цена: $price${if (isMagic) " (магический)" else ""}")
}

fun main() {
    createItem( name = "Меч",  price = 50)
    createItem(price = 200, name = "Зелье")
    createItem( name = "Посох", isMagic = true, price = 150)  // Порядок не важен!
}
```

Вывод:

```
Предмет: Меч | Цена: 50
Предмет: Зелье | Цена: 200
Предмет: Посох | Цена: 150 (магический)
```

Правила:

- Если хотя бы один аргумент передан по имени, остальные после него **тоже должны быть именованными**.
- Полезно для функций с множеством необязательных параметров.

Шаг 5. Изменение параметров

Параметры функций в Kotlin — это **val** (неизменяемые). Попытка изменить их приведёт к ошибке:

```
fun buffAttack(attack: Int) {
    attack = attack * 2  // Ошибка: Val cannot be reassigned!
}
```

Но можно изменять свойства объектов:

```
fun upgradeWeapon(weapon: MutableList<String>) {
    weapon.add("Улучшение +1")  // Так можно!
}

fun main() {
    val sword = mutableListOf("Меч")
    upgradeWeapon( weapon = sword)
    println(sword)  // [Меч, Улучшение +1]
}
```

Итог:

- **Необязательные параметры** упрощают вызов функций.
- **Именованные аргументы** делают код читаемым.
- Параметры **нельзя перезаписать**, но можно менять содержимое объектов.

Шаг 6. Возвращение результата (return)

Функции могут возвращать результат. Для этого:

1. Укажите **тип возвращаемого значения** после списка параметров.
2. Используйте ключевое слово **return**.

Пример: Функция, вычисляющая урон

```
fun calculateDamage(baseDamage: Int, multiplier: Double): Int {
    return (baseDamage * multiplier).toInt()
}

fun main() {
    val damage = calculateDamage( baseDamage = 50,  multiplier = 1.5)
    println("Нанесён урон: $damage HP")  // Нанесён урон: 75 HP
}
```

Как это работает?

- Функция **calculateDamage** принимает **baseDamage** и **multiplier**, возвращает **Int**.
- **return** передаёт результат в место вызова функции.

Шаг 7. Функции без возвращаемого значения (Unit)

Если функция **не возвращает** результат, её тип — **Unit** (аналог **void** в Java). Его можно не указывать.

Пример: Функция с Unit

```
fun showWarning(message: String): Unit {
    println("ВНИМАНИЕ: $message")
}

fun main() {
    showWarning("Низкий уровень здоровья!")
}
```

Важно:

- **Unit** используется, например, для функций, которые только выводят данные или изменяют состояние системы.

Шаг 8. Однострочные функции (=)

Если функция состоит из **одного выражения**, можно использовать сокращённый синтаксис.

Пример: Сокращённая функция

```
fun isPlayerAlive(health: Int): Boolean = health > 0

fun main() {
    println(isPlayerAlive( health = 10))  // true
    println(isPlayerAlive( health = 0))   // false
}
```

Где полезно?

- Для простых проверок или вычислений.

Шаг 9. Перегрузка функций (Function Overloading)

Что такое перегрузка функций?

Перегрузка позволяет создавать **несколько функций с одним именем**, но **разными параметрами**. Kotlin различает их по:

- **Количеству параметров**
- **Типам параметров**
- **Порядку параметров**

Например, определим функцию **sum**() пятью перегруженными версиями:

```
fun sum(a: Int, b: Int) : Int{  new *
    return a + b
}
fun sum(a: Double, b: Double) : Double{  new *
    return a + b
}
fun sum(a: Int, b: Int, c: Int) : Int{  new *
    return a + b + c
}
fun sum(a: Int, b: Double) : Double{  new *
    return a + b
}
fun sum(a: Double, b: Int) : Double{  new *
    return a + b
}
```

Каждая из версий отличается либо по типу, либо количеству, либо по порядку параметров. При вызове функции **sum** компилятор в зависимости от типа и количества параметров сможет выбрать для выполнения нужную версию:

```
fun main() {  new *
    val a = sum( a = 1,  b = 2)
    val b = sum( a = 1.5,  b = 2.5)
    val c = sum( a = 1,  b = 2,  c = 3)
    val d = sum( a = 2,  b = 1.5)
    val e = sum( a = 1.5,  b = 2)
}
```

При этом при перегрузке не учитывает возвращаемый результат функции. Например, пусть у нас будут две следующие версии функции **sum**:

```
fun sum(a: Double, b: Int): Double {
    return a + b
}

fun sum(a: Double, b: Int): String {
    return "$a + $b"
}
```

Они совпадают во всем за исключением возвращаемого типа. Однако в данном случае мы сталкиваемся с ошибкой, так как перегруженные версии должны отличаться именно по типу, порядку или количеству параметров. Отличие в возвращаемом типе не имеют значения.

Пример 1: Разное количество параметров

```
// Базовая версия
fun attack() {  1 Usage
    println("Атака по воздуху!")
}
// Перегруженная версия с параметром
fun attack(enemy: String) {  1 Usage
    println("Атакован $enemy!")
}
```

```kotlin
// Перегруженная версия с двумя параметрами
fun attack(enemy: String, damage: Int) {  1 Usage
    println("Атакован $enemy с уроном $damage HP!")
}
```

```kotlin
fun main() {
    attack() // Атака по воздуху!
    attack( enemy = "Гоблин") // Атакован Гоблин!
    attack( enemy = "Дракон",  damage = 50) // Атакован Дракон с уроном 50 HP!
}
```

Пример 2: Разные типы параметров

```kotlin
// Атака по имени (String)
fun attack(target: String) {  1 Usage
    println("Атакован $target!")
}
// Атака по ID (Int)
fun attack(targetId: Int) {  1 Usage
    println("Атакован враг с ID $targetId!")
}
```

```kotlin
fun main() {
    attack( target = "Орк") // Атакован Орк!
    attack( targetId = 1001) // Атакован враг с ID 1001!
}
```

Пример 3: Разный порядок параметров

```kotlin
// Зелье: название + сила
fun usePotion(name: String, power: Int) {  1 Usage
    println("Использовано $name (+$power HP)")
}
// Зелье: сила + название (другой порядок)
fun usePotion(power: Int, name: String) {  1 Usage
    println("Использовано $name (+$power MP)")
}
```

```kotlin
fun main() {
    usePotion( name = "Лечебное",  power = 50) // Использовано Лечебное (+50 HP)
    usePotion( power = 30,  name = "Магическое") // Использовано Магическое (+30 MP)
}
```

Шаг 10. Анонимные функции

Что это: Функции без имени, которые можно сразу присвоить переменной или передать как аргумент.

Пример:

```
val greet = fun(name: String) {  1 Usage
    println("Привет, $name!")
}
fun main() {
    greet("Игрок")  // Вывод: Привет, Игрок!
}
```

Отличие от обычных функций:

- Нет имени после **fun**.
- Можно использовать как значение (передавать, возвращать).

Анонимные функции выглядят как обычные за тем исключением, что они не имеют имени. Анонимная функция может иметь одно выражение:

```
fun main() {
    val sum = fun(x: Int, y: Int): Int = x + y
    println(sum(10,15))
}
```

Либо может представлять блок кода:

```
fun main() {
    val sum = fun(x: Int, y: Int): Int {
        return x + y
    }
    println(sum(10, 15))
}
```

Шаг 11. Лямбда-выражения

Лямбда-выражения представляют небольшие кусочки кода, которые выполняют некоторые действия. Фактически лямбды представляют сокращенную запись функций.

При этом лямбды, как и обычные и анонимные функции, могут передаваться в качестве значений переменным и параметрам функции.

Лямбда-выражения оборачиваются в фигурные скобки:

```
{println("hello")}
```

В данном случае лямбда-выражение выводит на консоль строку "hello".

Лямбда-выражение можно сохранить в обычную переменную и затем вызывать через имя этой переменной как обычную функцию.

```
fun main() {
    val hello = {println("Hello Kotlin")}
    hello()
    hello()
}
```

В данном случае лямбда сохранена в переменную **hello** и через эту переменную вызывается два раза. Поскольку лямбда-выражение представляет сокращенную форму функции, то переменная **hello** имеет тип функции () -> **Unit**.

```
val hello: ()->Unit = {println("Hello Kotlin")}
```

Рассмотрим ещё пример:

```
val attack = { enemy: String -> println("Атакован $enemy!") }

fun main() {
    attack("Гоблин")  // Атакован Гоблин!
}
```

Лямбда с возвращаемым значением:

```
val calculateDamage = { base: Int, multiplier: Double ->  1 Usage
    (base * multiplier).toInt()
}

fun main() {
    println("Урон: ${calculateDamage(10, 1.5)}")  // Урон: 15
}
```

Особенности:

- Типы параметров можно не указывать, если они очевидны.
- Если лямбда — последний аргумент функции, её можно вынести за скобки.

Шаг 12. Функции высшего порядка

Что это: Функции, которые принимают другие функции как параметры или возвращают их.

Пример: Применение эффекта к игроку

```
fun applyEffect(effect: (Int) -> Int) {  2 Usages
    val health = 100
    println("Новое здоровье: ${effect(health)}")
}

fun main() {
    applyEffect { it + 20 }  // Новое здоровье: 120 (лечение)
    applyEffect { it - 50 }  // Новое здоровье: 50 (урон)
}
```

Пример: Генератор событий в игре

```
fun onEvent(eventType: String, action: (String) -> Unit) {
    println("Событие: $eventType")
    action(eventType)
}

fun main() {
    onEvent( eventType = "Дождь") {
        println("Игрок получает эффект 'Мокрый'")
    }
}
```

Шаг 13. Замыкания (Closures)

Что это: Лямбды могут "запоминать" переменные из внешней области видимости.

Пример: Счётчик убийств врагов

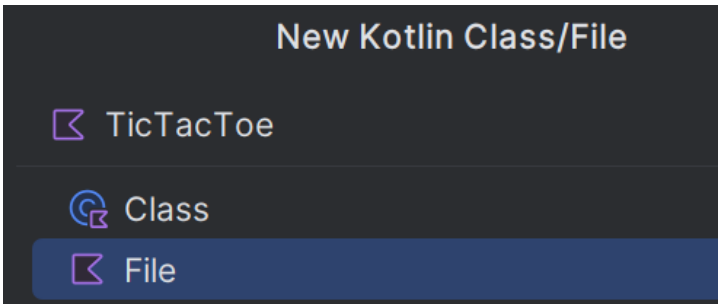
```kotlin
fun createCounter(): () -> Int {  1 Usage
    var count = 0
    return { ++count }  // Захватывает переменную count
}

fun main() {
    val kills = createCounter()
    println(kills())  // 1
    println(kills())  // 2
}
```

Как работает:

- Лямбда **{++count}** сохраняет доступ к **count**, даже после завершения **createCounter**().

Шаг 14. Создание игры крестики-нолики с ботом

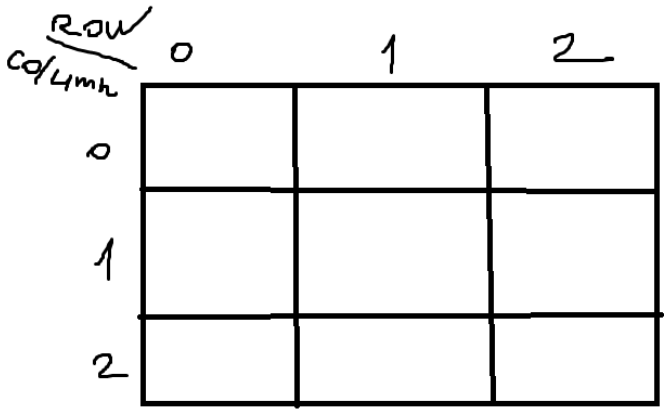
В папке **src** создаём новый файл **TicTacToe**:



Объявляем точку входа в нашу программу методе **main**:

```kotlin
fun main() {


}
```

Начнём с создания пустого игрового поля и функции для его отображения. Поле для игры представляет собой **строки** и **столбцы 3x3** и его нам нужно с вами инициализировать. Вот схематическое представление:



Создадим с вами **функцию printBoard()**. В качестве параметра **board** она будет принимать двумерный массив символов (**Char**). — Эта функция отвечает **только за отображение поля**. Мы отделяем визуализацию от логики — это хорошая практика, даже на начальном этапе:

```
// Функция для отображения поля
fun printBoard(board: Array<Array<Char>>) {

}
```

Перед отображением самого поля, мы выводим **заголовок с номерами столбцов** — 0, 1, 2. — Это помогает игроку понимать, как обращаться к клеткам — например, позиция [1][2] означает **строка 1, столбец 2**.

```
fun printBoard(board: Array<Array<Char>>) {
    println("  0 1 2")  // Нумерация столбцов
```

Далее мы запускаем цикл **for**, который перебирает все **индексы строк** массива **board**. — **board.indices** — это сокращение от 0 until board.size, то есть [0, 1, 2].

📌 Для каждой строки мы:

1. Выводим её номер (например, 0)
2. Выводим содержимое этой строки

```
fun printBoard(board: Array<Array<Char>>) {
    println("  0 1 2")  // Нумерация столбцов
    for (i in board.indices) {

    }
```

Внутри цикла напишем следующий код:

```
for (i in board.indices) {
    print("$i ")  // Нумерация строк
    println(board[i].joinToString( separator = "|") { if (it == ' ') "_" else it.toString() })
}
```

print("$i ") — Слева от строки мы выводим её **номер**, чтобы игрок видел координаты строк — аналогично столбцам. — Это формирует **сетку с номерами**, в которой легко ориентироваться.

println(board[i].joinToString(" | ") { if (it == ' ') "_" else it.toString() })

❖ **board[i]** — Получаем текущую строку массива (список из 3 символов)

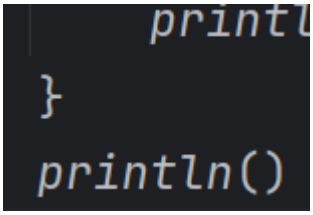
❖ **.joinToString(" | ") { ... }** — Объединяем все 3 символа строки в одну строку с разделителем " | " — это просто вертикальный разделитель, чтобы поле выглядело красиво

❖ **{ if (it == ' ') "_" else it.toString() }** — Здесь для каждой клетки:

- Если в ней пробел ' ' → выводим символ подчёркивания "_" (визуально лучше видно)
- Если там уже есть X или O, мы показываем этот символ

Таким образом, пустые клетки отображаются как _, а заполненные как символы игроков.

Просто ставим **пустую строку**, чтобы чуть отделить поле от следующих сообщений или действий:



Затем в методе **main** мы создаём игровое поле, которое представляет собой **двумерный массив размера 3x3**. Array(3) означает, что у нас будет **3 строки**. Внутри каждой строки — ещё один массив из **3 символов** (Array(3)), и каждый элемент этого массива инициализируется символом ' ' — это **пробел**, обозначающий **пустую клетку**.

И здесь мы вызываем функцию **printBoard** и передаём ей массив **board.** Это нужно, чтобы сразу же после создания поля — **отобразить его в консоли**. Самая первая визуализация помогает понять, как оно выглядит до начала игры.

```
fun main() {
    // Создаём поле 3x3, заполненное пробелами (пустыми клетками)
    val board = Array( size = 3) { Array( size = 3) { ' ' } }
    // Выводим поле в консоль
    printBoard(board)
}
```

Запустите и проверьте результат:

```
   0 1 2
0  |_|_|_
1  |_|_|_
2  |_|_|_
```

Теперь добавим возможность игроку (крестикам) делать ход.

Продолжаем писать код в методе main.

Создаём переменную **currentPlayer**, которая хранит **текущего игрока**. В данном случае 'X' — это человек, а 'O' будет компьютер. Она понадобится, чтобы ставить нужный символ на поле:

```
fun main() {
    val board = Array( size = 3) { Array( size = 3) { ' ' } }
    var currentPlayer = 'X'  // Первым ходит человек (X)
```

Далее выводим короткое приветствие, чтобы игрок знал, что он играет за крестики X, а против него — компьютер (позже мы добавим его ход). И выводим начальное поле: всё ещё пусто, все клетки — символ ' ' (отображаются как _). — Помогает игроку визуально увидеть координаты клеток перед тем, как сделать ход:

```
println("Крестики-нолики: Ты (X) против Компьютера (O)")
printBoard(board)
```

Просим игрока ввести **две координаты** — сначала строку, потом столбец. Подсказка нужна, чтобы не возникло путаницы: какие значения допустимы.

Читаем ввод из консоли и **преобразуем его в целое число**.

```
// Ход игрока
println("Твой ход! Введите строку и столбец (0, 1 или 2):")
val row = readln().toInt()
val col = readln().toInt()
```

Делаем проверку корректности ввода:

```
// Проверяем, что введённые координаты корректны
if (row !in 0 ≤..≤ 2 || col !in 0 ≤..≤ 2) {
    println("Неверные координаты. Должны быть 0, 1 или 2.")
    return
}
```

Проверяем: входят ли координаты в допустимый диапазон (0, 1, 2). Если хотя бы одна выходит за пределы → ошибка. Выводим сообщение и используем return, чтобы **прервать функцию** main — дальнейшие действия не выполняются. Таким образом, мы защищаем игру от выхода за границы массива. Например, board[3][1] вызовет ошибку.

Проверяем занятость клетки:

```
// Проверяем, что клетка свободна
if (board[row][col] != ' ') {
    println("Ячейка уже занята. Попробуй снова.")
    return
}
```

Проверяем: если в выбранной клетке уже **не пробел**, значит туда уже кто-то ходил. Сообщаем об ошибке и выходим из программы. **Таким образом** нельзя ходить туда, где уже стоит X или O. В будущем можно обернуть это в цикл, чтобы игрок ввёл координаты снова.

Выполняем ход:

```
// Записываем ход игрока
board[row][col] = currentPlayer
printBoard(board)
```

В выбранную клетку записываем символ 'X' (человек) — сама клетка на поле теперь будет отображаться как X. Показываем обновлённое поле с ходом игрока. Игрок видит, что его символ оказался в нужной клетке.

Запускаем и проверяем: Введите, например, 1 и 1. Должно получиться:

```
C:\Users\Leontev\.jdks\openjdk-24.0.1\bin\java.ex
Крестики-нолики: Ты (X) против Компьютера (O)
  0 1 2
0 _|_|_
1 _|_|_
2 _|_|_

Твой ход! Введите строку и столбец (0, 1 или 2):
1
1
  0 1 2
0 _|_|_
1 _|X|_
2 _|_|_
```

Теперь добавим функцию **checkWin**(), которая проверяет, выиграл ли игрок:

```
fun checkWin(board: Array<Array<Char>>, player: Char): Boolean {
    // Проверяем строки и столбцы
    for (i in 0..2) {
        // Проверка строки
        if (board[i][0] == player && board[i][1] == player &&
board[i][2] == player) {
            return true
        }
        // Проверка столбца
        if (board[0][i] == player && board[1][i] == player &&
board[2][i] == player) {
            return true
        }
    }

    // Проверка диагоналей
    if (board[0][0] == player && board[1][1] == player &&
board[2][2] == player) {
        return true
    }
    if (board[0][2] == player && board[1][1] == player &&
board[2][0] == player) {
        return true
    }
```

```
    return false
}
```

Разбор функции:

```
fun checkWin(board: Array<Array<Char>>, player: Char): Boolean {
```

- Мы объявляем **функцию** checkWin, которая принимает:
  - board — игровое поле 3×3.
  - player — символ текущего игрока ('X' или 'O').
- Возвращает true, если игрок победил, иначе — false.

Проверка строк и столбцов

```
for (i in 0 ≤ .. ≤ 2) {
```

- Цикл от 0 до 2 — всего 3 строки/столбца.
- Перебираем по индексу каждую строку и соответствующий столбец.

Проверка строки i

```
// Проверка строки
if (board[i][0] == player && board[i][1] == player &&
 board[i][2] == player) {
    return true
}
```

- Проверяем: все 3 ячейки **в одной строке** содержат символ игрока.
- Если да → **победа по строке**.

Проверка столбца i

```
// Проверка столбца
if (board[0][i] == player && board[1][i] == player &&
 board[2][i] == player) {
    return true
}
```

- Проверяем вертикально: ячейки одного **столбца**, но в разных строках.
- Если все 3 — принадлежат игроку → **победа по столбцу**.

Если хотя бы одна из строк или столбцов подходит — возвращаем true.

▼ Проверка диагоналей

После цикла мы отдельно проверяем **две диагонали**:

Левая диагональ: от [0][0] до [2][2]

```
if (board[0][0] == player && board[1][1] == player && board[2][2]
  == player) {
    return true
}
```

Победа, если игрок заполнил всю диагональ слева направо.

Правая диагональ: от [0][2] до [2][0]

```
if (board[0][2] == player && board[1][1] == player && board[2][0]
  == player) {
    return true
}
```

Победа, если игрок заполнил диагональ справа налево.

Если нет победы

```
return false
```

Если ни одна из проверок не прошла — значит **игра продолжается**, победы ещё нет.

Добавляем ход компьютера (O)/ Код:

```
fun getComputerMove(board: Array<Array<Char>>): Pair<Int, Int> {
    val emptyCells = mutableListOf<Pair<Int, Int>>()

    // Собираем все свободные клетки
    for (i in 0..2) {
        for (j in 0..2) {
            if (board[i][j] == ' ') {
                emptyCells.add(Pair(i, j))
            }
        }
    }

    // Выбираем случайную клетку
    return emptyCells[Random.nextInt(emptyCells.size)]
}
```

Разбор кода:

```
fun getComputerMove(board: Array<Array<Char>>): Pair<Int, Int> {
```

- Это функция, которая **выбирает случайную свободную клетку** на игровом поле.
- Она возвращает пару координат: **строка и столбец** (тип Pair<Int, Int>).
- Аргумент board — это само игровое поле, нужно для анализа свободных клеток.

Создание списка пустых клеток

```
val emptyCells = mutableListOf<Pair<Int, Int>>()
```

- Создаём **пустой список**, куда добавим координаты **всех свободных клеток**.
- Каждая клетка хранится как пара индексов: Pair(row, col).

Поиск свободных клеток:

- Проходим по всем строкам (i) и столбцам (j) поля.
- Проверяем каждую клетку — если в ней пробел ' ' (пусто), добавляем её координаты в emptyCells.
- Таким образом, получаем список всех возможных ходов для компьютера.

```
// Собираем все свободные клетки
for (i in 0 ≤ .. ≤ 2) {
    for (j in 0 ≤ .. ≤ 2) {
        if (board[i][j] == ' ') {
            emptyCells.add(Pair( first = i,  second = j))
        }
    }
}
```

Выбор случайной клетки:

```
// Выбираем случайную клетку
return emptyCells[Random.nextInt( until = emptyCells.size)]
```

- Используем Random.nextInt(...), чтобы выбрать **случайный индекс из списка свободных клеток**.
- Возвращаем соответствующую пару координат — **ход компьютера готов**!

Не забываем в начало программы добавить импорт, чтобы использовать функцию **Random.nextInt**(...):

```
import kotlin.random.Random
```

Теперь объединим всё в бесконечный цикл, который прервётся при победе или ничье и добавим ход игрока в метод main:

```
fun main() {
    val board = Array(3) { Array(3) { ' ' } }
    var currentPlayer = 'X' // Человек — X
    var moves = 0

    println("Крестики-нолики: Ты (X) против Компьютера (O)")
    printBoard(board)
```

```
    while (true) {
        if (currentPlayer == 'X') {
            println("Твой ход! Введите строку и столбец (0, 1 или 2):")
            val row = readln().toInt()
            val col = readln().toInt()

            if (row !in 0..2 || col !in 0..2) {
                println("Неверные координаты. Повторите попытку.")
                continue
            }

            if (board[row][col] != ' ') {
                println("Ячейка уже занята. Попробуй снова.")
                continue
            }

            board[row][col] = currentPlayer
        } else {
            println("Ход компьютера:")
            val (row, col) = getComputerMove(board)
            println("Компьютер выбрал: $row $col")
            board[row][col] = currentPlayer
        }

        moves++
        printBoard(board)

        if (checkWin(board, currentPlayer)) {
            println(if (currentPlayer == 'X') "Ты победил!" else
"Компьютер победил!")
            break
        } else if (moves == 9) {
            println("Ничья!")
            break
        }

        currentPlayer = if (currentPlayer == 'X') 'O' else 'X'
    }

    println("Игра окончена.")

}
```

Шаг 15. Создание игры камень-ножницы-бумага

Теперь на основе всего разобранного создадим с вами игру камень-ножницы-бумага.

Цель: выбрать вариант, который побеждает вариант, выбранный компьютером.

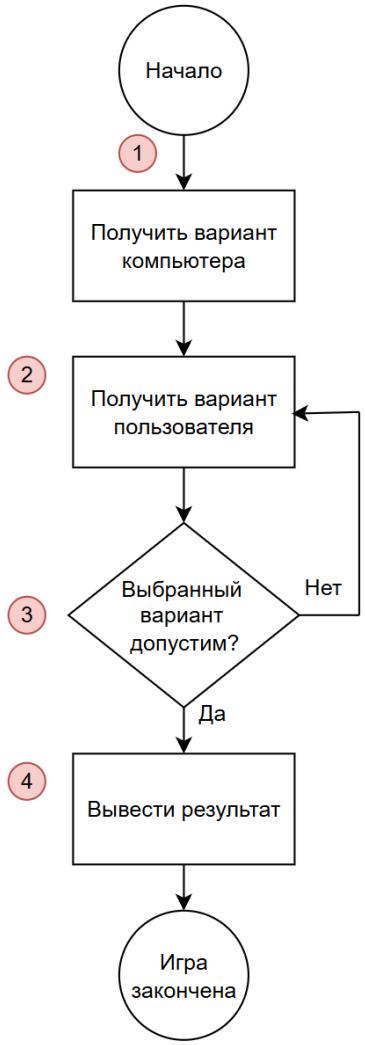
Подготовка: при запуске приложения игра случайным образом выбирает один из трех вариантов: «**Камень**», «**Ножницы**» или «**Бумага**». Затем она предлагает вам выбрать один из этих вариантов.

Правила: игра сравнивает два варианта. Если они совпадают, игра заканчивается вничью. Если же варианты различны, победитель определяется по следующим правилам:

Варианты	Результат
Ножницы, бумага	«Ножницы» побеждают («ножницы режут бумагу»).
Камень, ножницы	«Камень» побеждает («камень разбивает ножницы»).
Бумага, камень	«Бумага» побеждает («бумага накрывает камень»).

Прежде чем переходить к написанию кода, необходимо составить план работы приложения.

Сначала необходимо определить общий ход игры. Основная последовательность действий выглядит так:



1. Вы запускаете игру.

Приложение случайным образом выбирает один из вариантов: «камень», «ножницы» или «бумага».

2. Приложение запрашивает ваш вариант.

Вы вводите свое решение в окне вывода IDE.

3. Приложение проверяет ваш выбор.

Если выбран недопустимый вариант, приложение возвращается к шагу 2 и предлагает ввести другой вариант. Это происходит до тех пор, пока не будет введен допустимый вариант.

4. Игра выводит результат.

Она сообщает, какие варианты были выбраны вами и приложением, а также результат: выиграли вы, проиграли или же партия завершилась вничью.

При построении игры необходимо реализовать несколько высокоуровневых задач:

1. Заставить игру выбрать вариант.

Мы создадим новую функцию с именем **getGameChoice**, которая будет случайным образом выбирать один из вариантов: «Камень», «Ножницы» или «Бумага».

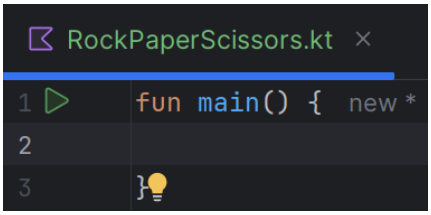
2. Запросить у пользователя выбранный им вариант.

Для этого мы напишем еще одну функцию с именем **getUserChoice**, которая будет запрашивать у пользователя выбранный им вариант. Функция проверяет введенный вариант, и, если пользователь ввел недопустимое значение — запрашивает данные заново, пока не будет введен правильный вариант.

3. Вывести результат.

Мы напишем функцию с именем **printResult**, которая будет определять результат: выигрыш/проигрыш пользователя или ничья. Затем функция выводит результат.

Создайте новый файл в проекте с названием **RockPaperScissors** и добавьте точку входа:



Первое, что нужно сделать, — заставить компьютерного игрока выбрать один из вариантов («Камень», «Ножницы» или «Бумага») случайным образом.

Для создания массива будет использоваться функция arrayOf. Этот код будет добавлен в функцию main приложения, чтобы массив создавался при запуске приложения:

```
val options = arrayOf("Камень", "Бумага", "Ножницы")
```

После создания массива необходимо определить новую функцию **getGameChoice**():

```
fun getGameChoice(optionsParam: Array<String>) =  1 Usage   Paltos
    optionsParam[(Math.random() * optionsParam.size).toInt()]
```

- Math.random() генерирует число от 0.0 до 1.0.
- Умножение на optionsParam.size (3) дает диапазон 0.0..3.0.
- Приведение к Int отбрасывает дробную часть → получаем индекс 0, 1 или 2.

- **Пример:**

Если Math.random() вернет 0.7 → 0.7 * 3 = 2.1 → toInt() = 2 → "Ножницы".

Далее создадим функцию **getUserChoice**():

```
fun getUserChoice(optionsParam: Array<String>): String
 {  1 Usage    Paltos
    var isValidChoice = false
    var userChoice = ""
    while (!isValidChoice) {
        print("Пожалуйста введите одно из следующих значений: ")
        for (item in optionsParam) print(" $item")
        println()
        val userInput = readLine()
        if (userInput != null && userInput in optionsParam) {
            isValidChoice = true
            userChoice = userInput
        }
        if (!isValidChoice) println("Неверный ввод.")
    }
    return userChoice
}
```

1. Бесконечный цикл while, пока пользователь не введет корректное значение.
2. Вывод подсказки с вариантами: "Камень Бумага Ножницы".
3. Чтение ввода (readLine()).
4. Проверка:
    - userInput != null — ввод не пустой.
    - userInput in optionsParam — ввод есть в массиве options.
5. Если проверка пройдена — выход из цикла, иначе повтор запроса.

Следующей определим функцию **printResult**():

```
fun printResult(userChoice: String, gameChoice: String)
 {  1 Usage    Paltos *
    val result: String
    if (userChoice == gameChoice) {
        result = "Ничья!"
    } else if ((userChoice == "Камень" && gameChoice == "Ножницы") ||
        (userChoice == "Бумага" && gameChoice == "Камень") ||
        (userChoice == "Ножницы" && gameChoice == "Бумага")
    ) {
        result = "Вы победили!"
    } else {
        result = "Вы проиграли!"
    }
    println("Ваш выбор $userChoice. Я выбираю $gameChoice. $result")
}
```

- **Логика определения победителя:**
  - **Ничья:** Одинаковые значения.
  - **Победа:**
    - Камень > Ножницы
    - Бумага > Камень
    - Ножницы > Бумага
  - **Проигрыш:** Все остальные случаи.
- **Примеры:**
  - Пользователь: "Камень", Компьютер: "Ножницы" → "Вы победили!".
  - Пользователь: "Бумага", Компьютер: "Ножницы" → "Вы проиграли!".

И в методе main создадим выбор компьютера gameChoice и ввод пользователя userChoice и после выведем наш результат:

```
fun main() {  Paltos
    val options = arrayOf("Камень", "Бумага", "Ножницы")
    val gameChoice = getGameChoice( optionsParam = options)
    val userChoice = getUserChoice( optionsParam = options)
    printResult(userChoice, gameChoice)
}
```

Самостоятельные задания

Задание 1

Задача: Создайте функцию **showGameLogo**(), которая выводит название игры (например, "=== EPIC QUEST ==="). Вызовите её дважды в **main**().

Задание 2

Задача: Создайте функцию showEnemy(name: String, level: Int), которая выводит: **"Враг: [name] (Уровень: [level])".**

Вызовите её для двух разных противников.

Задание 3

Задача: Создайте функцию spawnEnemy(name: String, health: Int = 100, isBoss: Boolean = false), которая выводит:

"Враг: $name (Здоровье: $health HP${if (isBoss) " | БОСС!" else ""}". Вызовите её:

1. С обычным врагом (только имя).
2. С врагом-боссом (имя и isBoss = true).

Задание 4

Задача: Используйте функцию spawnEnemy из задания 3, чтобы создать:

1. Врага с кастомным здоровьем (300 HP), но не босса.
2. Босса с именем "Король Тьмы" и здоровьем 500 HP.

Задание 5

Создайте функцию createQuest(title: String, reward: Int = 50, isMain: Boolean = false), которая выводит:

"Квест: $title (Награда: $reward золота${if (isMain) " | ОСНОВНОЙ" else ""})". Вызовите её:

1. С обычным квестом ("Найти кота").
2. С основным квестом и наградой 100 ("Спасти принцессу").

Задание 6

Задача: Напишите функцию calculateXP(level: Int): Int, которая возвращает количество опыта для достижения уровня по формуле:

XP = level × 1000.

Вызовите её для 5 уровня и выведите результат.

Задание 7

Задача: Создайте функцию checkInventory(item: String): Boolean, которая возвращает:

- true — если предмет есть в инвентаре (например, "Меч").
- false — если предмета нет (например, "Зелье").

Задание 8

Задача: Создайте перегруженные функции castSpell:

1. Без параметров — выводит "Каст случайного заклинания!".
2. С параметром spell: String — выводит "Каст заклинания $spell!".
3. С параметрами spell: String и power: Int — выводит "Каст $spell с силой $power!".

Задание 9

Задача: Создайте перегруженные функции createWeapon:

1. Принимает name: String — возвращает "Оружие: $name".
2. Принимает name: String и damage: Int — возвращает "Оружие: $name (Урон: $damage)".
3. Принимает damage: Int и isMagic: Boolean — возвращает "${if (isMagic) "Магическое" else "Обычное"} оружие (Урон: $damage)".

Задание 10

Создайте перегруженные функции heal():

1. Без параметров — лечит 10 HP.
2. С параметром amount: Int — лечит указанное количество HP.
3. С параметрами amount: Int и isPotion: Boolean — если isPotion=true, выводит "Выпито зелье (+$amount HP)", иначе — "Заклинание лечения (+$amount HP)".