Kotlin

1. Условные конструкции в языке Котлин. Их особенности

- **`if/else`**:
  - Используется для ветвления кода.
  - Особенность: это выражение, т.е. может возвращать значение.

```
val max = if (a > b) a else b
```

- **`if` как блок**: Может содержать несколько инструкций.

```
val result = if (condition) {
    println("True branch")
    42 // возвращаемое значение
} else {
    println("False branch")
    0
}
```

2. Оператор присваивания. Арифметические, логические операторы, операторы сравнения. Приоритет операторов

- **Присваивание (`=`)**: Присваивает значение переменной.

```
val x = 5
```

- **Арифметические операторы**: `+`, `,`, `/`, `%`.

```
val sum = a + b
```

- **Логические операторы**: `&&` (И), `||` (ИЛИ), `!` (НЕ).

```
if (a > 0 && b < 10) println("Valid")
```

- **Операторы сравнения**: `==`, `!=`, `<`, `>`, `<=`, `>=`.

```
val isEqual = a == b
```

- **Приоритет операторов** (сверху вниз):
  1. Унарные (`!`, `+`, `-`)
  2. Арифметические (`*`, `/`, `%`, затем `+`, `-`)
  3. Сравнения (`<`, `>`, `<=`, `>=`)
  4. Логические (`&&`, `||`).

3. Конструкция when. Особенности конструкции when

- Аналог `switch` в других языках, но мощнее.
- Поддерживает проверки значений, диапазонов и условий.
- Может быть выражением (возвращать значение).
- Не требует ключевого слова `break`.

Пример:

```
val result = when (x) {
    in 1..10 -> "Диапазон 1-10"
    20, 30 -> "Числа 20 или 30"
    else -> "Неизвестно"
}
```

4. Массивы в языке Котлин. Их виды

- Массивы представлены классом `Array` и специализированными классами для базовых типов (`IntArray`, `DoubleArray` и т.д.).
- Основные виды:
  - **Общие массивы** (`Array`): Могут хранить любой тип.
  - **Специализированные массивы**: Оптимизированы для примитивов.

5. Синтаксис создания массивов. Массивы базовых типов

- **Создание массивов:**

```
val numbers = arrayOf(1, 2, 3)
val numbers2 = IntArray(5) { it * 2 }
```

- **Базовые типы массивов:** `IntArray`, `DoubleArray`, `CharArray` и т.д.

6. Функции в языке Котлин. Создание функций

- Объявление функции:

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

- Функция без возвращаемого значения:

```
fun printMessage(message: String) {
    println(message)
}
```

7. Аргументы по умолчанию. Однострочные функции

- **Аргументы по умолчанию:**

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

- **Однострочные функции:**

```
fun square(x: Int) = x * x
```

8. Типы возвращаемых значений функций в Котлин

- Тип функции указывается после `:`.
- Функции без возвращаемого значения имеют тип `Unit`.

```
fun sayHello(): Unit = println("Hello!")
```

- Функции могут возвращать любой тип:

```
fun sum(a: Int, b: Int): Int = a + b
```

9. Локальные функции. Пример использования

- Локальные функции объявляются внутри других функций.
- Удобны для организации кода и скрытия деталей реализации.

```
fun outerFunction(x: Int) {
    fun innerFunction(y: Int): Int {
        return y * 2
    }
    println(innerFunction(x))
}
```

10. Перегрузка функций. Для чего это нужно. Особенности перегрузки

- Перегрузка позволяет использовать одно имя функции с разными наборами параметров.
- Упрощает код, обеспечивая гибкость.

```
fun print(value: Int) = println("Int: $value")
fun print(value: String) = println("String: $value")
```

- Особенности:
  - Различие по количеству/типу параметров.
  - Возвращаемый тип не влияет на выбор функции.

11. Переменные-функции в Котлин

- **Переменные-функции** позволяют хранить функции в переменных.
- Пример:

```
val operation: (Int, Int) -> Int = { a, b -> a + b }
println(operation(5, 3)) // 8
```

12. Анонимные функции. Для чего используются. Пример реализации

- **Анонимные функции** — функции без имени, используемые для локальных задач.
- Пример:

```
val multiply = fun(a: Int, b: Int): Int {
    return a * b
}
println(multiply(3, 4)) // 12
```

13. Анонимная функция как аргумент функции

- Анонимные функции часто передаются как аргументы другим функциям.
- Пример:

```
fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

val result = calculate(3, 5, fun(x, y) = x * y)
println(result) // 15
```

14. Лямбда-выражения. Передача параметров

- **Лямбда-выражение** — компактный синтаксис для функций.
- Пример передачи параметров:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
println(sum(4, 6)) // 10
```

15. Возвращение результата. Возвращение лямбда-выражений из функций

- Лямбда возвращает результат последнего выражения:

```
val square = { x: Int -> x * x }
println(square(4)) // 16
```

- Функция может возвращать лямбду:

```
fun getLambda(): (Int) -> Int = { it * 2 }
val double = getLambda()
println(double(5)) // 10
```

16. Понятие класса. Определение, инициализация

- **Класс** — шаблон для создания объектов.
- Определение и создание объекта:

```
class Person(val name: String, var age: Int)

val person = Person("John", 30)
println(person.name) // John
```

17. Структура программы на языке Котлин

- Основные компоненты:
  1. Пакет и импорт:

     ```
     package my.program
     import kotlin.math.*
     ```

  2. Функция `main`:

     ```
     fun main() {
         println("Hello, Kotlin!")
     }
     ```

  3. Классы, функции и другие элементы.

18. Свойства класса. Объявление, инициализация, использование

- Свойства — данные, хранящиеся в классе.
- Пример:

```
class Car(var model: String, var year: Int)

val car = Car("Toyota", 2022)
println(car.model) // Toyota
```

19. Методы класса. Объявление, инициализация, использование

- Методы — функции, принадлежащие классу.
- Пример:

```
class Calculator {
    fun add(a: Int, b: Int): Int = a + b
}

val calc = Calculator()
println(calc.add(3, 5)) // 8
```

20. Парадигма объектно-ориентированного программирования. Основные принципы ООП

- ООП — методология программирования, базирующаяся на **объектах**.
- **Принципы ООП:**
  1. **Инкапсуляция**: Скрытие реализации.
  2. **Наследование**: Классы наследуют свойства/методы.
  3. **Полиморфизм**: Разное поведение одного интерфейса.
  4. **Абстракция**: Фокус на важных деталях.
- Пример инкапсуляции:

```
class BankAccount(private var balance: Double) {
    fun deposit(amount: Double) {
        if (amount > 0) balance += amount
    }
    fun getBalance(): Double = balance
}

val account = BankAccount(100.0)
account.deposit(50.0)
println(account.getBalance()) // 150.0
```

21. Определение сеттера в программе. Особенности работы. Идентификатор `field`. Пример использования

- **Сеттер** — метод, вызываемый при присвоении значения свойству.
- Идентификатор `field` используется для доступа к хранимому значению свойства внутри сеттера.
- Пример:

```
class Person {
    var age: Int = 0
        set(value) {
            field = if (value >= 0) value else 0 // защита от некорректных
данных
        }
}

val person = Person()
person.age = -5
println(person.age) // 0
```

22. Определение геттера в программе. Особенности работы. Пример использования

- **Геттер** — метод, возвращающий значение свойства.
- Используется для вычисляемых свойств или изменения отображения значения.
- Пример:

```
class Person(val firstName: String, val lastName: String) {
    val fullName: String
        get() = "$firstName $lastName"
}

val person = Person("John", "Doe")
println(person.fullName) // John Doe
```

23. Использование геттеров и сеттеров в классах

- Геттеры и сеттеры позволяют реализовать логику чтения и изменения свойств.
- Пример:

```
class Account {
    var balance: Double = 0.0
        get() = field
        set(value) {
            if (value >= 0) field = value
        }
}

val account = Account()
account.balance = 100.0
println(account.balance) // 100.0
```

24. Принцип инкапсуляции в объектно-ориентированном программировании. Применение в Котлин

- **Инкапсуляция**: Сокрытие деталей реализации, доступ к которым предоставляется через методы/свойства.
- В Kotlin это достигается с помощью модификаторов доступа (`private`, `protected`, `public`).
- Пример:

```
class BankAccount(private var balance: Double) {
    fun deposit(amount: Double) {
        if (amount > 0) balance += amount
    }
    fun getBalance(): Double = balance
}
```

25. Генерация псевдослучайных чисел с нужными характеристиками с помощью Котлин

- Используется класс `kotlin.random.Random`.
- Пример:

```kotlin
val randomValue = (1..10).random() // случайное число от 1 до 10
val randomDouble = kotlin.random.Random.nextDouble(0.0, 1.0)
println(randomValue)
println(randomDouble)
```

26. Переопределение методов и свойств в произвольном классе. Принципы переопределения, особенности

- Методы и свойства класса должны быть помечены аннотацией open, чтобы их можно было переопределять.
- Пример:

```kotlin
open class Animal {
    open fun sound() = "Generic sound"
}

class Dog : Animal() {
    override fun sound() = "Bark"
}

val dog = Dog()
println(dog.sound()) // Bark
```

27. Переопределение методов и свойств базового класса. Принцип переопределения, особенности

- Переопределение выполняется с использованием ключевого слова override.
- Базовый класс должен быть помечен как open.
- Пример:

```kotlin
open class Vehicle {
    open val speed: Int = 60
    open fun move() = "Moving at speed $speed"
}

class Car : Vehicle() {
    override val speed: Int = 120
    override fun move() = "Driving at speed $speed"
}
```

28. Аннотация open для переопределения. Пример использования

- **open** делает методы и свойства доступными для переопределения.
- Пример:

```kotlin
open class Shape {
    open fun draw() = "Drawing Shape"
}
```

29. Аннотация override для переопределения. Пример использования

- **override** используется для переопределения методов или свойств.
- Пример:

```kotlin
open class Shape {
    open fun draw() = "Drawing Shape"
}

class Circle : Shape() {
    override fun draw() = "Drawing Circle"
}
```

30. Переопределение геттеров и сеттеров

- Геттеры и сеттеры можно переопределять, добавляя дополнительную логику.
- Пример:

```kotlin
open class Rectangle {
    open var width: Int = 0
        get() = field
        set(value) {
            field = if (value > 0) value else 0
        }
}

class Square : Rectangle() {
    override var width: Int
        get() = super.width
        set(value) {
            super.width = value
            println("Square width set to $value")
        }
}

val square = Square()
square.width = 10 // Square width set to 10
```

31. Переопределение в иерархии наследования классов. Запрет переопределения

- **Переопределение в иерархии**: Методы и свойства можно переопределять на каждом уровне наследования, если они помечены как open.
- **Запрет переопределения**: Используется ключевое слово final.
- Пример:

```kotlin
open class Animal {
    open fun sound() = "Generic sound"
    final fun eat() = "Eating food"
}

class Dog : Animal() {
    override fun sound() = "Bark"
    // fun eat() { } // Ошибка: метод final нельзя переопределить
}
```

32. Обращение к реализации из базового класса. Особенности обращения. Использование ключевого слова

- Для вызова метода или свойства базового класса используется ключевое слово super.

- Пример:

```
open class Animal {
    open fun sound() = "Generic sound"
}

class Dog : Animal() {
    override fun sound(): String {
        return super.sound() + " and Bark"
    }
}

val dog = Dog()
println(dog.sound()) // Generic sound and Bark
```

33. Абстрактные классы. Определение и использование. Ключевое слово для определения такого класса

- Абстрактный класс создаётся с помощью ключевого слова `abstract`.
- Нельзя создать объект абстрактного класса.
- Пример:

```
abstract class Animal {
    abstract fun sound(): String
    fun eat() = "Eating food"
}
```

34. Абстрактные методы. Определение и использование. Ключевое слово для определения такого метода

- Абстрактные методы не имеют реализации в базовом классе и обязаны быть реализованы в наследниках.
- Пример:

```
abstract class Animal {
    abstract fun sound(): String
}

class Dog : Animal() {
    override fun sound() = "Bark"
}
```

35. Реализация абстрактного класса. Абстрактные свойства в первичном конструкторе

- Абстрактные свойства могут быть объявлены в абстрактном классе.
- Пример:

```
abstract class Shape(val name: String) {
    abstract val area: Double
}

class Circle(name: String, val radius: Double) : Shape(name) {
    override val area: Double
        get() = Math.PI * radius * radius
}

val circle = Circle("Circle", 5.0)
println(circle.area) // 78.53981633974483
```

36. Полиморфизм в Котлин. Особенности и использование

- **Полиморфизм** позволяет обращаться к объектам различных типов через общий интерфейс или базовый класс.
- Пример:

```
open class Animal {
    open fun sound() = "Generic sound"
}

class Dog : Animal() {
    override fun sound() = "Bark"
}

val animals: List<Animal> = listOf(Animal(), Dog())
animals.forEach { println(it.sound()) } // Generic sound, Bark
```

37. Типы полиморфизма в Котлин

- **Полиморфизм подтипов**: Использование объектов производных классов как объектов базового типа.
- **Ad hoc-полиморфизм**: Реализация нескольких функций с одним именем (перегрузка).
- **Параметрический полиморфизм**: Использование универсальных параметров (Generics).

38. Полиморфизм по случаю (Ad hoc). Особенности и использование

- Используется при перегрузке функций или операторов.
- Пример:

```
fun add(a: Int, b: Int) = a + b
fun add(a: String, b: String) = a + b

println(add(3, 4)) // 7
println(add("Hello, ", "World")) // Hello, World
```

39. Полиморфизм подтипов. Особенности и использование

- Подразумевает использование объектов наследников как объектов базового класса.
- Пример:

```
open class Animal {
    open fun sound() = "Generic sound"
}

class Dog : Animal() {
    override fun sound() = "Bark"
}
```

```kotlin
val animal: Animal = Dog()
println(animal.sound()) // Bark
```

40. Интерфейсы. Общий синтаксис

- Интерфейс в Kotlin создаётся с помощью ключевого слова `interface`.
- Пример:

```kotlin
interface Animal {
    fun sound(): String
}
```

41. Интерфейсы. Реализация свойств. Применение

- Интерфейсы могут содержать абстрактные и реализованные свойства.
- Пример:

```kotlin
interface Named {
    val name: String
        get() = "Default Name"
}

class Person : Named {
    override val name: String
        get() = "John"
}

val person = Person()
println(person.name) // John
```

42. Интерфейсы. Правила переопределения

- Методы и свойства интерфейса должны быть реализованы в классе, если они не имеют реализации по умолчанию.
- Если класс реализует несколько интерфейсов с одинаковыми методами, необходимо явно указать, какую реализацию использовать:

```kotlin
interface A {
    fun print() = println("A")
}

interface B {
    fun print() = println("B")
}

class C : A, B {
    override fun print() {
        super<A>.print()
    }
}
```

43. Отличия абстрактных классов от интерфейсов

Абстрактные классы	Интерфейсы
Используется ключевое слово `abstract`.	Используется ключевое слово `interface`.
Может содержать состояние (свойства с `field`).	Не может содержать состояния.
Может иметь конструктор.	Конструктор не поддерживается.
Наследуется только один абстрактный класс.	Реализуется несколько интерфейсов.

44. Внутренние (inner) классы в Котлин. Особенности при совпадении имён

- **`inner` классы** имеют доступ к членам внешнего класса.
- При совпадении имён используется ключевое слово `this`:

```kotlin
class Outer(val name: String) {
    inner class Inner(val name: String) {
        fun printNames() {
            println(this.name) // имя Inner
            println(this@Outer.name) // имя Outer
        }
    }
}
```

45. Функции высокого порядка в Котлин. Возвращение функции из функции

- Функция высокого порядка принимает другую функцию в качестве параметра или возвращает функцию.
- Пример:

```kotlin
fun operation(op: String): (Int, Int) -> Int {
    return when (op) {
        "add" -> { a, b -> a + b }
        "mul" -> { a, b -> a * b }
        else -> { _, _ -> 0 }
    }
}

val add = operation("add")
println(add(3, 5)) // 8
```

46. Принцип наследования в объектно-ориентированном программировании. Применение в Котлин

- **Наследование** позволяет создавать классы, которые используют свойства и методы других классов.
- Пример:

```kotlin
open class Animal {
    open fun sound() = "Generic sound"
}

class Dog : Animal() {
    override fun sound() = "Bark"
}
```

47. Вызов конструктора при наследовании в Котлин. Наследование класса с первичным конструктором

- При наследовании класса с первичным конструктором необходимо вызвать его:

```
open class Animal(val name: String)

class Dog(name: String, val breed: String) : Animal(name)
```

48. Расширение базового класса. Переопределение методов и свойств

- Классы могут расширяться, переопределяя свойства и методы:

```
open class Animal {
    open fun sound() = "Generic sound"
}

class Cat : Animal() {
    override fun sound() = "Meow"
}
```

49. Data-классы. Где используются и для чего нужны

- **Data-классы** представляют собой классы для хранения данных.
- Используются для:
  - Упрощения работы с данными.
  - Автоматической генерации методов `toString`, `equals`, `hashCode`, `copy`.
- Пример:

```
data class User(val name: String, val age: Int)
```

50. Автоматически генерируемые методы в data-классе. Какие существуют и для чего нужны

- **Методы:**
  1. `toString`: Читабельный вывод.
  2. `equals`: Сравнение объектов.
  3. `hashCode`: Хеширование.
  4. `copy`: Копирование объекта с возможностью изменения некоторых полей.
- Пример:

```
data class User(val name: String, val age: Int)

val user1 = User("John", 30)
val user2 = user1.copy(age = 35)

println(user1) // User(name=John, age=30)
println(user2) // User(name=John, age=35)
```

51. Декомпозиция data-классов. Для чего нужна и особенности реализации

- **Декомпозиция** позволяет распаковать свойства объекта в переменные с помощью `componentN()` методов.
- Нужна для удобного доступа к данным.
- Пример:

```
data class User(val name: String, val age: Int)

val user = User("John", 25)
val (name, age) = user // Декомпозиция
println("$name is $age years old") // John is 25 years old
```

52. Перечисления Enums. Определение и использование

- **Enum** используется для представления набора фиксированных значений.
- Пример:

```
enum class Direction {
    NORTH, SOUTH, EAST, WEST
}

val direction = Direction.NORTH
println(direction) // NORTH
```

53. Встроенные свойства и вспомогательные методы перечислений

- Встроенные свойства:
  - `name`: Имя элемента в строковом формате.
  - `ordinal`: Порядковый номер элемента.
- Вспомогательные методы:
  - `valueOf()`: Возвращает элемент по имени.
  - `values()`: Возвращает массив всех элементов.

54. Встроенное свойство name. Определение в коде, особенности использования

- Возвращает имя элемента перечисления как строку.
- Пример:

```
val direction = Direction.NORTH
println(direction.name) // NORTH
```

55. Встроенное свойство ordinal. Определение в коде, особенности использования

- Возвращает порядковый номер элемента (начиная с 0).
- Пример:

```
val direction = Direction.NORTH
println(direction.ordinal) // 0
```

56. Другие вспомогательные функции valueOf(), values() при работе с Enum классом

- **`valueOf(name: String)`**: Возвращает элемент по имени.

```
val direction = Direction.valueOf("NORTH")
println(direction) // NORTH
```

- **`values()`**: Возвращает массив всех элементов.

```
val directions = Direction.values()
println(directions.joinToString()) // NORTH, SOUTH, EAST, WEST
```

57. Хранение состояния. Функция для хранения состояния в Enum классе

- В enum можно добавлять свойства и методы.
- Пример:

```
enum class State(val description: String) {
    STARTED("Process is started"),
    STOPPED("Process is stopped");

    fun printDescription() {
        println(description)
    }
}

val state = State.STARTED
state.printDescription() // Process is started
```

58. Делегирование в Котлин. Делегирование свойств

- **Делегирование** позволяет передать управление свойством другому объекту.
- Пример с lazy:

```
val lazyValue: String by lazy {
    println("Computed!")
    "Hello"
}

println(lazyValue) // Computed! Hello
println(lazyValue) // Hello
```

59. Анонимные классы и объекты. Назначение и особенности

- Используются для создания объектов без явного определения класса.
- Применяются при передаче параметров или реализации интерфейсов на месте.
- Пример:

```
val listener = object : Runnable {
    override fun run() {
        println("Running...")
    }
}
listener.run()
```

60. Ключевое слово для определения анонимного объекта

- Для создания анонимного объекта используется ключевое слово object.
- Пример:

```
val obj = object {
    val name = "Anonymous"
    fun greet() = "Hello, $name"
}

println(obj.greet()) // Hello, Anonymous
```

61. Наследование анонимных объектов. Особенности применения и реализации

- Анонимные объекты могут наследовать классы или реализовывать интерфейсы, но они не имеют имени.
- Ограничение: при передаче анонимного объекта через переменную базового типа, уникальные методы и свойства становятся недоступны.
- Пример:

```
open class Animal {
    open fun sound() = "Generic sound"
}

val dog = object : Animal() {
    override fun sound() = "Bark"
    fun wagTail() = "Wagging tail"
}

println(dog.sound()) // Bark
println(dog.wagTail()) // Wagging tail

val animal: Animal = dog
println(animal.sound()) // Bark
// animal.wagTail() // Ошибка: метод wagTail недоступен через переменную
типа Animal
```

---

62. Анонимный объект как аргумент функции. Особенности применения и реализации

- Анонимный объект может быть передан как аргумент функции, если он соответствует ожидаемому типу.
- Пример:

```
interface ClickListener {
    fun onClick()
}

fun setClickListener(listener: ClickListener) {
    listener.onClick()
}

setClickListener(object : ClickListener {
    override fun onClick() {
        println("Button clicked!")
    }
})
```

---

63. Анонимный объект как результат функции. Особенности применения и реализации

- Функция может возвращать анонимный объект, если тип возвращаемого значения указан как базовый класс или интерфейс.

- Пример:

```
interface Shape {
    fun draw(): String
}

fun createShape(): Shape {
    return object : Shape {
        override fun draw() = "Drawing a circle"
    }
}

val shape = createShape()
println(shape.draw()) // Drawing a circle
```

---

64. Обработка исключений в языке Котлин. Общий синтаксис

- Исключения обрабатываются с помощью блоков `try`, `catch` и `finally`.
- Синтаксис:

```
try {
    // Код, который может вызвать исключение
} catch (e: Exception) {
    // Обработка исключения
} finally {
    // Код, который выполнится в любом случае
}
```

- Пример:

```
try {
    val result = 10 / 0
} catch (e: ArithmeticException) {
    println("Ошибка: ${e.message}")
} finally {
    println("Операция завершена")
}
```

---

65. Класс Exception. Общий тип исключений

- **`Exception`** — базовый класс для всех исключений в Kotlin.
- Поддерживает свойства:
  - `message`: Текст сообщения об исключении.
  - `cause`: Причина исключения.
- Пример создания пользовательского исключения:

```
class CustomException(message: String) : Exception(message)

fun throwException() {
    throw CustomException("Произошла ошибка")
}

try {
    throwException()
} catch (e: CustomException) {
    println("Поймано исключение: ${e.message}")
}
```

61. Наследование анонимных объектов. Особенности применения и реализации

- Анонимные объекты могут наследовать классы или реализовывать интерфейсы, но они не имеют имени.
- Ограничение: при передаче анонимного объекта через переменную базового типа, уникальные методы и свойства становятся недоступны.
- Пример:

```
open class Animal {
    open fun sound() = "Generic sound"
}

val dog = object : Animal() {
    override fun sound() = "Bark"
    fun wagTail() = "Wagging tail"
}

println(dog.sound()) // Bark
println(dog.wagTail()) // Wagging tail

val animal: Animal = dog
println(animal.sound()) // Bark
// animal.wagTail() // Ошибка: метод wagTail недоступен через переменную
типа Animal
```

---

62. Анонимный объект как аргумент функции. Особенности применения и реализации

- Анонимный объект может быть передан как аргумент функции, если он соответствует ожидаемому типу.
- Пример:

```
interface ClickListener {
    fun onClick()
}

fun setClickListener(listener: ClickListener) {
    listener.onClick()
}

setClickListener(object : ClickListener {
    override fun onClick() {
        println("Button clicked!")
```

```
    }
})
```

---

63. Анонимный объект как результат функции. Особенности применения и реализации

- Функция может возвращать анонимный объект, если тип возвращаемого значения указан как базовый класс или интерфейс.
- Пример:

```
interface Shape {
    fun draw(): String
}

fun createShape(): Shape {
    return object : Shape {
        override fun draw() = "Drawing a circle"
    }
}

val shape = createShape()
println(shape.draw()) // Drawing a circle
```

---

64. Обработка исключений в языке Котлин. Общий синтаксис

- Исключения обрабатываются с помощью блоков `try`, `catch` и `finally`.
- Синтаксис:

```
try {
    // Код, который может вызвать исключение
} catch (e: Exception) {
    // Обработка исключения
} finally {
    // Код, который выполнится в любом случае
}
```

- Пример:

```
try {
    val result = 10 / 0
} catch (e: ArithmeticException) {
    println("Ошибка: ${e.message}")
} finally {
    println("Операция завершена")
}
```

---

65. Класс Exception. Общий тип исключений

- **`Exception`** — базовый класс для всех исключений в Kotlin.
- Поддерживает свойства:
  - `message`: Текст сообщения об исключении.
  - `cause`: Причина исключения.
- Пример создания пользовательского исключения:

```
class CustomException(message: String) : Exception(message)

fun throwException() {
    throw CustomException("Произошла ошибка")
}

try {
    throwException()
} catch (e: CustomException) {
    println("Поймано исключение: ${e.message}")
}
```

66. Блоки try/catch/finally. Возвращение значения с помощью try. Обработка исключений с использованием блока catch. Использование finally.

Блоки `try/catch/finally` используются для обработки исключений в Java и Kotlin. В блоке `try` выполняется код, который может вызвать исключение. Если исключение происходит, управление передается в блок `catch`, где можно обработать исключение. Блок `finally` выполняется всегда, независимо от того, было ли выброшено исключение.

Пример:

```
fun example(): Int {
    return try {
        // Код, который может вызвать исключение
        10 / 0
    } catch (e: ArithmeticException) {
        // Обработка исключения
        println("Деление на ноль")
        -1
    } finally {
        // Этот блок выполнится в любом случае
        println("Блок finally")
    }
}
```

67. Базовый класс исключений. Свойства для получения различной информации.

В Kotlin все исключения являются наследниками класса `Throwable`. У этого класса есть несколько полезных свойств:

- `message: String?` — сообщение об ошибке.
- `cause: Throwable?` — причина исключения (если есть).
- `stackTrace: Array<StackTraceElement>` — трассировка стека.

Пример:

```
try {
    // Код, который может вызвать исключение
} catch (e: Exception) {
```

```
    println(e.message) // Сообщение об ошибке
    println(e.cause)   // Причина исключения
    e.printStackTrace() // Трассировка стека
}
```

68. Свойство для получения сообщения об исключении. Особенности применения.

Свойство `message` класса `Throwable` возвращает сообщение об ошибке, которое может быть полезно для логирования или вывода пользователю. Оно может быть `null`, если сообщение не было задано.

Пример:

```
try {
    throw Exception("Произошла ошибка")
} catch (e: Exception) {
    println(e.message) // Выведет: Произошла ошибка
}
```

69. Трассировка стека исключения. Особенности применения.

Трассировка стека показывает путь выполнения программы до момента возникновения исключения. Это полезно для отладки, так как позволяет понять, где именно произошла ошибка.

Пример:

```
try {
    throw Exception("Ошибка")
} catch (e: Exception) {
    e.printStackTrace() // Выведет трассировку стека
}
```

70. Функция printStackTrace(). Применение в коде.

Функция `printStackTrace()` выводит трассировку стека в стандартный поток ошибок (обычно консоль). Это полезно для отладки, но в production-коде лучше использовать логирование.

Пример:

```
try {
    throw Exception("Ошибка")
} catch (e: Exception) {
    e.printStackTrace() // Выведет трассировку стека
}
```

71. Обработка нескольких исключений. Оператор throw.

В Kotlin можно обрабатывать несколько исключений в одном блоке `catch`, используя синтаксис `|`. Оператор `throw` используется для явного выброса исключения.

Пример:

```
try {
    // Код, который может вызвать исключение
} catch (e: ArithmeticException | NullPointerException) {
    println("Обработка нескольких исключений")
} catch (e: Exception) {
    throw e // Повторное выбрасывание исключения
}
```

72. Null и Nullable – типы в Kotlin. Проблемы, связанные с использованием null.

В Kotlin типы по умолчанию не могут принимать значение `null`. Чтобы разрешить `null`, нужно использовать `?` (например, `String?`). Проблемы с `null` включают `NullPointerException`, которые могут возникнуть при неправильной работе с nullable-типами.

Пример:

```
var str: String? = null
println(str?.length) // Безопасный вызов, не вызовет исключение
```

73. Оператор «?:». Особенности применения и использования.

Оператор `?:` (elvis operator) возвращает значение слева, если оно не `null`, иначе возвращает значение справа.

Пример:

```
val length = str?.length ?: -1 // Если str == null, вернет -1
```

74. Оператор «?.». Особенности применения и использования.

Оператор `?.` (safe call operator) выполняет вызов метода или доступ к свойству только если объект не `null`.

Пример:

```
val length = str?.length // Если str == null, вернет null
```

75. Оператор «!!» (not-null assertion operator). Особенности применения и использования.

Оператор `!!` преобразует nullable-тип в not-null тип. Если значение `null`, выбрасывается `NullPointerException`.

Пример:

```
val length = str!!.length // Если str == null, выбросит исключение
```

76. Преобразование типов в Kotlin. Методы для преобразования типов.

В Kotlin есть несколько способов преобразования типов:

- `toInt()`, `toDouble()`, `toString()` и т.д.
- Явное приведение с помощью `as`.

Пример:

```
val str = "123"
val num = str.toInt() // Преобразование строки в число
```

77. Smart cast и оператор is. Использование в программе.

Smart cast — это автоматическое приведение типа после проверки с помощью оператора `is`.

Пример:

```
fun printLength(obj: Any) {
    if (obj is String) {
        println(obj.length) // Smart cast to String
    }
}
```

78. Ограничения умных преобразований. Правила таких ограничений.

Умные преобразования работают только с `val` (неизменяемыми переменными) и только в пределах области видимости, где компилятор может гарантировать, что тип не изменится.

Пример:

```
var obj: Any? = "Hello"
if (obj is String) {
    // obj здесь String
}
// obj снова Any?
```

79. Явные преобразования и оператор as. Возможности оператора.

Оператор `as` используется для явного приведения типов. Если приведение невозможно, выбрасывается исключение `ClassCastException`.

Пример:

```
val obj: Any = "Hello"
val str = obj as String // Явное приведение
```

80. Функции расширения. Общий синтаксис.

Функции расширения позволяют добавлять новые функции к существующим классам без изменения их исходного кода.

Синтаксис:

```
fun ClassName.functionName(parameters): ReturnType {
    // Тело функции
}
```

Пример:

```
fun String.isPalindrome(): Boolean {
    return this == this.reversed()
}

println("madam".isPalindrome()) // true
```

81. Функции области видимости (scope-функции) в Kotlin

Функции области видимости (scope-функции) в Kotlin — это функции, которые позволяют выполнять блок кода в контексте объекта. Они упрощают работу с объектами, делая код более читаемым и выразительным. Основные scope-функции: `let`, `with`, `run`, `apply`, `also`.

---

82. Функция let. Назначение и использование

Функция `let` используется для выполнения блока кода с объектом в качестве аргумента. Она возвращает результат выполнения блока. Часто используется для безопасного выполнения операций с nullable-объектами.

Синтаксис:

```
val result = object.let { it ->
    // Блок кода, где `it` — это объект
    // Возвращает результат выполнения блока
}
```

Пример:

```
val name: String? = "Kotlin"
val length = name?.let {
    println(it) // Выведет: Kotlin
    it.length   // Возвращает длину строки
}
println(length) // Выведет: 6
```

---

83. Функция with. Назначение и использование

Функция `with` позволяет выполнить блок кода с объектом в качестве получателя (this). Она возвращает результат выполнения блока. Удобна для группировки операций над одним объектом.

Синтаксис:

```
val result = with(object) {
    // Блок кода, где `this` — это объект
    // Возвращает результат выполнения блока
}
```

Пример:

```
val numbers = mutableListOf(1, 2, 3)
val sum = with(numbers) {
    add(4)
    add(5)
    sum() // Возвращает сумму элементов
}
println(sum) // Выведет: 15
```

---

84. Функция run. Назначение и использование

Функция `run` похожа на `with`, но вызывается на самом объекте. Она также возвращает результат выполнения блока. Может использоваться как для nullable, так и для non-null объектов.

Синтаксис:

```
val result = object.run {
    // Блок кода, где `this` — это объект
    // Возвращает результат выполнения блока
}
```

Пример:

```
val name: String? = "Kotlin"
val length = name?.run {
    println(this) // Выведет: Kotlin
    length        // Возвращает длину строки
}
println(length) // Выведет: 6
```

---

85. Функция apply. Назначение и использование

Функция `apply` используется для настройки объекта. Она возвращает сам объект после выполнения блока. Удобна для инициализации или конфигурации объектов.

Синтаксис:

```
val result = object.apply {
    // Блок кода, где `this` — это объект
    // Возвращает объект
}
```

Пример:

```
val person = Person().apply {
    name = "Alice"
    age = 25
}
println(person) // Выведет: Person(name=Alice, age=25)
```

---

86. Функция also. Назначение и использование

Функция `also` выполняет блок кода с объектом в качестве аргумента (it) и возвращает сам объект. Удобна для выполнения дополнительных действий, таких как логирование.

Синтаксис:

```
val result = object.also { it ->
    // Блок кода, где `it` — это объект
    // Возвращает объект
}
```

Пример:

```
val numbers = mutableListOf(1, 2, 3).also {
    println("Исходный список: $it") // Выведет: Исходный список: [1, 2, 3]
    it.add(4)
}
println(numbers) // Выведет: [1, 2, 3, 4]
```

---

87. Инфиксная нотация. Ключевое слово. Назначение

Инфиксная нотация позволяет вызывать функции с одним параметром без точки и скобок. Для этого функция должна быть помечена ключевым словом `infix`.

Синтаксис:

```
infix fun ClassName.functionName(parameter: Type) {
    // Тело функции
}
```

Пример:

```
infix fun Int.add(x: Int): Int = this + x

val result = 5 add 10 // Инфиксный вызов
println(result) // Выведет: 15
```

---

88. Коллекции в Kotlin. Типы коллекций. Пакет, в котором располагаются коллекции

Коллекции в Kotlin делятся на два типа:

1. **Неизменяемые (Immutable)** — не поддерживают изменение после создания.
2. **Изменяемые (Mutable)** — поддерживают изменение после создания.

Основные типы коллекций:

- Списки: `List`, `MutableList`
- Множества: `Set`, `MutableSet`
- Ассоциативные массивы: `Map`, `MutableMap`

Коллекции располагаются в пакете `kotlin.collections`.

---

89. Неизменяемые коллекции в Kotlin. Интерфейс Iterable

Неизменяемые коллекции — это коллекции, которые нельзя изменить после создания. Они реализуют интерфейс `Iterable`, который позволяет итерироваться по элементам коллекции.

Пример:

```
val list: List<Int> = listOf(1, 2, 3)
for (item in list) {
```

```
    println(item)
}
```

---

90. Основной интерфейс для работы с неизменяемыми коллекциями

Основной интерфейс для работы с неизменяемыми коллекциями — `Collection`. Он предоставляет базовые операции, такие как:

- `size` — количество элементов.
- `isEmpty()` — проверка на пустоту.
- `contains(element)` — проверка наличия элемента.

Пример:

```
val set: Set<String> = setOf("a", "b", "c")
println(set.size) // Выведет: 3
println(set.contains("a")) // Выведет: true
```

91. Компоненты интерфейса. Интерфейс Map, его особенности в Kotlin

Интерфейс `Map` в Kotlin представляет собой коллекцию пар “ключ-значение”. Он не является наследником интерфейса `Collection`, но входит в стандартную библиотеку Kotlin для работы с ассоциативными массивами.

Особенности `Map` в Kotlin:

- Ключи уникальны, значения могут дублироваться.
- `Map` является неизменяемым (immutable) по умолчанию. Для создания изменяемой версии используется `MutableMap`.
- Доступ к элементам осуществляется по ключу: `map[key]`.
- Основные реализации: `HashMap`, `LinkedHashMap`, `TreeMap`.

Пример:

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
println(map["a"]) // 1
```

---

92. Изменяемые коллекции в Kotlin. Основной интерфейс для работы с изменяемыми коллекциями

В Kotlin коллекции делятся на **неизменяемые** (immutable) и **изменяемые** (mutable). Изменяемые коллекции позволяют добавлять, удалять и изменять элементы.

Основной интерфейс для работы с изменяемыми коллекциями:

- `MutableCollection<T>` — расширяет интерфейс `Collection<T>` и добавляет методы для изменения коллекции (добавление, удаление, очистка).

Пример:

```
val mutableList: MutableList<Int> = mutableListOf(1, 2, 3)
mutableList.add(4) // [1, 2, 3, 4]
```

---

93. Другой интерфейс для изменения данных, который расширяет основной интерфейс

Интерфейс `MutableList<T>` расширяет `MutableCollection<T>` и добавляет методы для работы с индексами, такие как:

- `add(index, element)`
- `removeAt(index)`
- `set(index, element)`

Пример:

```
val mutableList = mutableListOf("a", "b", "c")
mutableList.add(1, "d") // [a, d, b, c]
mutableList[0] = "A" // [A, d, b, c]
```

---

94. Методы для удаления и добавления элементов

Основные методы для работы с изменяемыми коллекциями:

- **Добавление:**
  - `add(element)` — добавляет элемент в конец.
  - `add(index, element)` — добавляет элемент по индексу.
- **Удаление:**
  - `remove(element)` — удаляет первое вхождение элемента.
  - `removeAt(index)` — удаляет элемент по индексу.
  - `clear()` — очищает коллекцию.

Пример:

```
val list = mutableListOf(1, 2, 3)
list.add(4) // [1, 2, 3, 4]
list.remove(2) // [1, 3, 4]
list.removeAt(0) // [3, 4]
```

---

95. Операции с коллекциями. Ряд функций, которые предоставляет интерфейс Iterable для выполнения различных операций над коллекциями

Интерфейс `Iterable` предоставляет множество функций для работы с коллекциями:

- **Фильтрация:**
  - `filter(predicate)` — возвращает коллекцию, соответствующую условию.
  - `filterNot(predicate)` — возвращает коллекцию, не соответствующую условию.
- **Преобразование:**
  - `map(transform)` — применяет функцию к каждому элементу.
  - `flatMap(transform)` — преобразует и “разворачивает” коллекцию.
- **Поиск:**
  - `find(predicate)` — возвращает первый элемент, соответствующий условию.

  - `any(predicate)` — проверяет, есть ли хотя бы один элемент, соответствующий условию.
- **Сортировка:**
  - `sorted()` — возвращает отсортированную коллекцию.
  - `sortedBy(selector)` — сортирует по заданному критерию.

Пример:

```
val numbers = listOf(1, 2, 3, 4)
val doubled = numbers.map { it * 2 } // [2, 4, 6, 8]
val even = numbers.filter { it % 2 == 0 } // [2, 4]
```

---

96. Коллекция `List`. Создание и использование

`List` — это упорядоченная коллекция элементов с возможностью доступа по индексу. В Kotlin `List` по умолчанию неизменяем.

Создание:

- `listOf(...)` — создает неизменяемый список.
- `mutableListOf(...)` — создает изменяемый список.

Пример:

```
val list = listOf("a", "b", "c")
println(list[1]) // b
```

---

97. Основные методы коллекции `List`

Основные методы:

- `get(index)` — возвращает элемент по индексу.
- `indexOf(element)` — возвращает индекс первого вхождения элемента.
- `subList(from, to)` — возвращает подсписок.
- `contains(element)` — проверяет наличие элемента.

Пример:

```
val list = listOf("a", "b", "c")
println(list.indexOf("b")) // 1
println(list.subList(1, 3)) // [b, c]
```

---

98. Изменяемые списки `List`. Методы для работы с такими списками

Изменяемые списки (`MutableList`) поддерживают методы для изменения:

- `add(element)` — добавляет элемент.
- `add(index, element)` — добавляет элемент по индексу.
- `remove(element)` — удаляет элемент.
- `removeAt(index)` — удаляет элемент по индексу.
- `set(index, element)` — заменяет элемент по индексу.

Пример:

```
val mutableList = mutableListOf("a", "b", "c")
mutableList.add("d") // [a, b, c, d]
mutableList[1] = "B" // [a, B, c, d]
```

---

99. Коллекция `Set`. Назначение, создание и использование

`Set` — это коллекция уникальных элементов без определенного порядка. В Kotlin `Set` по умолчанию неизменяем.

Назначение:

- Хранение уникальных значений.
- Быстрая проверка наличия элемента.

Создание:

- `setOf(...)` — создает неизменяемый набор.
- `mutableSetOf(...)` — создает изменяемый набор.

Пример:

```
val set = setOf("a", "b", "c", "a") // [a, b, c]
println(set.contains("b")) // true
```

Изменяемый `Set`:

```
val mutableSet = mutableSetOf("a", "b")
mutableSet.add("c") // [a, b, c]
mutableSet.remove("a") // [b, c]
```

100. Методы коллекции `Set`. Изменяемые коллекции `MutableSet`

Методы коллекции `Set`:

- `contains(element)` — проверяет наличие элемента.
- `isEmpty()` — проверяет, пуст ли набор.
- `size` — возвращает количество элементов.
- `intersect(otherSet)` — возвращает пересечение двух наборов.
- `union(otherSet)` — возвращает объединение двух наборов.

Изменяемые коллекции `MutableSet`:

- `add(element)` — добавляет элемент.
- `remove(element)` — удаляет элемент.
- `clear()` — очищает набор.

Пример:

```
val set = mutableSetOf("a", "b", "c")
set.add("d") // [a, b, c, d]
set.remove("a") // [b, c, d]
```

---

101. Коллекция Map. Назначение и особенности

Назначение:

- Хранение пар “ключ-значение”.
- Ключи уникальны, значения могут дублироваться.

Особенности:

- Неизменяемый по умолчанию (`Map`).
- Изменяемая версия — `MutableMap`.
- Основные реализации: `HashMap`, `LinkedHashMap`, `TreeMap`.

Пример:

```
val map = mapOf("a" to 1, "b" to 2)
println(map["a"]) // 1
```

---

102. Обращение к элементам Map. Методы для работы с элементами

Обращение к элементам:

- `map[key]` — возвращает значение по ключу.
- `getValue(key)` — возвращает значение или выбрасывает исключение, если ключ отсутствует.
- `getOrElse(key, defaultValue)` — возвращает значение или результат лямбды, если ключ отсутствует.

Методы:

- `containsKey(key)` — проверяет наличие ключа.
- `containsValue(value)` — проверяет наличие значения.
- `keys` — возвращает набор ключей.
- `values` — возвращает коллекцию значений.

Пример:

```
val map = mapOf("a" to 1, "b" to 2)
println(map.containsKey("a")) // true
println(map.getOrElse("c") { 0 }) // 0
```

---

103. Интерфейс MutableMap. Назначение и особенности

Назначение:

- Расширяет `Map` и добавляет методы для изменения данных.

Особенности:

- Позволяет добавлять, удалять и изменять элементы.
- Основные методы: `put`, `remove`, `clear`.

Пример:

```
val mutableMap = mutableMapOf("a" to 1, "b" to 2)
mutableMap["c"] = 3 // {a=1, b=2, c=3}
mutableMap.remove("a") // {b=2, c=3}
```

---

104. Методы для работы с интерфейсом MutableMap. Коллекции, которыми реализуется этот интерфейс

Методы:

- `put(key, value)` — добавляет или обновляет пару.
- `remove(key)` — удаляет пару по ключу.
- `putAll(map)` — добавляет все пары из другой карты.
- `clear()` — очищает карту.

Реализации:

- `HashMap` — не гарантирует порядок.
- `LinkedHashMap` — сохраняет порядок добавления.
- `TreeMap` — сортирует элементы по ключам.

Пример:

```
val map = hashMapOf("a" to 1, "b" to 2)
map.put("c", 3) // {a=1, b=2, c=3}
```

---

105. Функции для трансформации коллекции/последовательности

- `map(transform)` — применяет функцию к каждому элементу.
- `flatMap(transform)` — преобразует и “разворачивает” коллекцию.
- `filter(predicate)` — фильтрует элементы по условию.
- `groupBy(keySelector)` — группирует элементы по ключу.

Пример:

```
val numbers = listOf(1, 2, 3)
val squared = numbers.map { it * it } // [1, 4, 9]
```

---

106. Группировка элементов коллекции/последовательности. Назначение и применяемые функции

Назначение:

- Группировка элементов по определенному критерию.

Функции:

- `groupBy(keySelector)` — возвращает карту, где ключ — результат лямбды, а значение — список элементов.

Пример:

```
val words = listOf("a", "ab", "abc")
val grouped = words.groupBy { it.length } // {1=[a], 2=[ab], 3=[abc]}
```

---

107. Сортировка коллекции/последовательности. Интерфейс для сортировки

Интерфейс:

- `Comparable<T>` — используется для естественной сортировки.
- `Comparator<T>` — используется для кастомной сортировки.

Функции:

- `sorted()` — сортирует по естественному порядку.
- `sortedWith(comparator)` — сортирует с использованием компаратора.

Пример:

```
val numbers = listOf(3, 1, 2)
val sorted = numbers.sorted() // [1, 2, 3]
```

---

108. Логика сортировки для базовых встроенных типов

- Числа сортируются по возрастанию.
- Строки сортируются лексикографически (по алфавиту).
- Даты сортируются от более ранних к более поздним.

Пример:

```
val strings = listOf("b", "a", "c")
val sorted = strings.sorted() // [a, b, c]
```

---

109. Реализация интерфейса Comparable. Функция compareTo()

Интерфейс `Comparable`:

- Определяет естественный порядок сортировки.
- Метод `compareTo(other)` возвращает:
  - Отрицательное число, если текущий объект меньше.
  - Ноль, если объекты равны.
  - Положительное число, если текущий объект больше.

Пример:

```
class Person(val age: Int) : Comparable<Person> {
    override fun compareTo(other: Person) = this.age - other.age
}
```

---

110. sortedWith и интерфейс Comparator. Назначение и реализация

Интерфейс `Comparator`:

- Используется для кастомной сортировки.
- Метод `compare(a, b)` возвращает:
  - Отрицательное число, если `a < b`.
  - Ноль, если `a == b`.
  - Положительное число, если `a > b`.

Пример:

```
val people = listOf(Person(20), Person(10))
val sorted = people.sortedWith(compareBy { it.age }) // [Person(10), Person(20)]
```

---

111. Сортировка на основе критерия. Функции sortedBy() и sortedByDescending()

- `sortedBy(selector)` — сортирует по возрастанию на основе критерия.
- `sortedByDescending(selector)` — сортирует по убыванию на основе критерия.

Пример:

```
val people = listOf(Person(20), Person(10))
val sorted = people.sortedBy { it.age } // [Person(10), Person(20)]
```

---

112. Сортировка. Функции reverse() и shuffle()

- `reverse()` — возвращает коллекцию в обратном порядке.
- `shuffle()` — перемешивает элементы случайным образом.

Пример:

```
val list = listOf(1, 2, 3)
val reversed = list.reversed() // [3, 2, 1]
val shuffled = list.shuffled() // [2, 1, 3] (случайно)
```

---

113. Агрегатные операции в коллекциях/последовательностях

- `sum()` — возвращает сумму элементов.
- `average()` — возвращает среднее значение.
- `minOrNull()` — возвращает минимальный элемент.
- `maxOrNull()` — возвращает максимальный элемент.
- `count()` — возвращает количество элементов.

Пример:

```
val numbers = listOf(1, 2, 3)
println(numbers.sum()) // 6
```

---

114. Обобщения. Обобщенные классы и функции. Назначение

Назначение:

- Позволяют создавать классы и функции, которые работают с любыми типами данных.

Пример:

```
class Box<T>(val value: T)
val box = Box(10) // Box<Int>
```

---

115. Обобщенные типы. Применение нескольких параметров

Можно использовать несколько обобщенных параметров:

```
class Pair<K, V>(val key: K, val value: V)
val pair = Pair("a", 1) // Pair<String, Int>
```

---

116. Обобщенные функции. Назначение и реализация

Назначение:

- Функции, которые работают с любыми типами данных.

Пример:

```
fun <T> printValue(value: T) {
    println(value)
}
printValue(10) // 10
```

---

117. Ограничения обобщений. Назначение и реализация

Назначение:

- Ограничивают типы, которые можно использовать в обобщениях.

Пример:

```
fun <T : Comparable<T>> max(a: T, b: T): T {
    return if (a > b) a else b
}
```

---

118. Установка нескольких ограничений. Ограничения в классах

Можно установить несколько ограничений с помощью `where`:

```
class Box<T>(val value: T) where T : Comparable<T>, T : Serializable
```

Пример:

```
fun <T> printIfComparable(value: T) where T : Comparable<T>, T : Serializable {
    println(value)
}
```