
Яндекс

Яндекс

Язык программирования

Степан Чурюканов, Android разработчик

Материалы к презентации:

https://github.com/tewnn/Kotlin-Intro

Online компилятор Kotlin:

https://play.kotlinlang.org

«Обращаться с языком кое-как — значит и мыслить кое-как: неточно, приблизительно, неверно»

Алексей Николаевич Толстой о языке Kotlin

Ожидания от языка программирования

- Удобство и скорость разработки

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов
- Читаемость кода

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов
- Читаемость кода
- Тестируемость

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов
- Читаемость кода
- Тестируемость
- Производительность

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов
- Читаемость кода
- Тестируемость
- Производительность
- Наличие библиотек

Ожидания от языка программирования

- Удобство и скорость разработки
- Уменьшение количества багов
- Читаемость кода
- Тестируемость
- Производительность
- Наличие библиотек
- Очевидность и однозначность

Преимущества языка Kotlin

- Полная совместимость с java

- Полная совместимость с java
- Копиляция в java runtime

- Полная совместимость с java
- Копиляция в java runtime
- Возможность компиляции в JS и нативный код

- Полная совместимость с java
- Копиляция в java runtime
- Возможность компиляции в JS и нативный код
- Статическая типизация

- Полная совместимость с java
- Копиляция в java runtime
- Возможность компиляции в JS и нативный код
- Статическая типизация
- Си-образный синтаксис

- Полная совместимость с java
- Копиляция в java runtime
- Возможность компиляции в JS и нативный код
- Статическая типизация
- Си-образный синтаксис
- Простота и краткость

- Полная совместимость с java
- Копиляция в java runtime
- Возможность компиляции в JS и нативный код
- Статическая типизация
- Си-образный синтаксис
- Простота и краткость
- Поддержка со стороны IDE

- Nullsafety

- Nullsafety
- Куда более богатые возможности по сравнению с java

- Nullsafety
- Куда более богатые возможности по сравнению с java
- Наличие λ выражений

- Nullsafety
- Куда более богатые возможности по сравнению с java
- Наличие λ выражений
- Много синтаксического сахара

- Nullsafety
- Куда более богатые возможности по сравнению с java
- Наличие λ выражений
- Много синтаксического сахара
- Возможность писать скрипты для gradle

- Nullsafety
- Куда более богатые возможности по сравнению с java
- Наличие λ выражений
- Много синтаксического сахара
- Возможность писать скрипты для gradle
- Возможность писать скрипты на kotlin script для инфраструктурных задач

Недостатки языка Kotlin

- Отсутствие устоявшегося code style

- Отсутствие устоявшегося code style
- Инструменты для статического анализа кода

- Отсутствие устоявшегося code style
- Инструменты для статического анализа кода
- Неочевидность ряда конструкций

- Отсутствие устоявшегося code style
- Инструменты для статического анализа кода
- Неочевидность ряда конструкций
- Потенциальные проблемы с производительностью

- Отсутствие устоявшегося code style
- Инструменты для статического анализа кода
- Неочевидность ряда конструкций
- Потенциальные проблемы с производительностью
- Увеличение количество методов приложения

Историческая справка

- Назван в честь острова возле Санкт-Петербурга

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года
- Первый релиз - февраль 2016

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года
- Первый релиз - февраль 2016
- На Google I/O 2017 объявлен официально рекомендуемым языком для Android

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года
- Первый релиз - февраль 2016
- На Google I/O 2017 объявлен официально рекомендуемым языком для Android
- Версия 1.2 с поддержкой JS - ноябрь 2017

Kotlin

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года
- Первый релиз - февраль 2016
- На Google I/O 2017 объявлен официально рекомендуемым языком для Android
- Версия 1.2 с поддержкой JS - ноябрь 2017
- Версия 1.3 с поддержкой корутин - октябрь 2018

- Назван в честь острова возле Санкт-Петербурга
- Разрабатывается с 2011 года
- Первый релиз - февраль 2016
- На Google I/O 2017 объявлен официально рекомендуемым языком для Android
- Версия 1.2 с поддержкой JS - ноябрь 2017
- Версия 1.3 с поддержкой корутин - октябрь 2018
- На Google I/O 2019 объявлен предпочтительным языком для Android

Введение в язык

Базовые понятия

- `package`
- `class`
  - nested class
  - inner class
  - final/open/abstract class
  - enum class
  - sealed class
  - data class
- `interface`
- `object`
  - companion object

- `function`
  - class-level function
  - package-level function
  - high-ordered function
  - inline function
  - extension function
- `property`
  - backed-field property
  - delegated property
  - extension property

- `lambda expression`

Value и variable

```kotlin
val value: Int = 1 // cannot be modified
var variable: Int = value // can be modified
variable++
```

```java
final int value = 1;
int variable = value;
variable++;
```

Value и variable

```kotlin
val value: Int = 1 // cannot be modified
var variable: Int = value // can be modified
variable++
```

```java
final int value = 1;
int variable = value;
variable++;
```

Value и variable

```kotlin
val value: Int = 1 // cannot be modified
var variable: Int = value // can be modified
variable++
```

```java
final int value = 1;
int variable = value;
variable++;
```

Value и variable

```kotlin
val value: Int = 1 // cannot be modified
var variable: Int = value // can be modified
variable++
```

```java
final int value = 1;
int variable = value;
variable++;
```

Все поля (field) - это свойства (properties)

```
var age: Int = 22
```

```
var age: Int = 22
    set(value: Int) {
        field = value
    }
```

```
private var _age: Int = 22
var age: Int
    get() = _age
    set(value: Int) {
        _age = value
    }
```

```
private int age = 22;

public int getAge() {
    return age;
}

public void setAge(int value)
{
    this.age = value;
}
```

Все поля (field) - это свойства (properties)

```
var age: Int = 22


var age: Int = 22
    set(value: Int) {
        field = value
    }


private var _age: Int = 22
var age: Int
    get() = _age
    set(value: Int) {
        _age = value
    }
```

```
private int age = 22;

public int getAge() {
    return age;
}

public void setAge(int value)
{
    this.age = value;
}
```

Все поля (field) - это свойства (properties)

```
var age: Int = 22
```

```
var age: Int = 22
    set(value: Int) {
        field = value
    }
```

```
private var _age: Int = 22
var age: Int
    get() = _age
    set(value: Int) {
        _age = value
    }
```

```
private int age = 22;

public int getAge() {
    return age;
}

public void setAge(int value)
{
    this.age = value;
}
```

Все поля (field) - это свойства (properties)

```
var age: Int = 22
```

```
var age: Int = 22
    set(value: Int) {
        field = value
    }
```

```
private var _age: Int = 22
var age: Int
    get() = _age
    set(value: Int) {
        _age = value
    }
```

```
private int age = 22;

public int getAge() {
    return age;
}

public void setAge(int value)
{
    this.age = value;
}
```

Null Safety

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Nullable types and Non-Null Types

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok


val la = a.length // ok
val lb = b.length // error: variable 'b' can be null
```

Checking for null in conditions

```
var b: String? = "abc"

val lb = if (b != null) b.length else -1
```

Checking for null in conditions

```
var b: String? = "abc"

val lb = if (b != null) b.length else -1
```

Safe Calls

```
var b: String? = "abc"

// length is null when b is null
val l: Int? = b?.length

// length is zero when b is null
val l: Int = b?.length ?: 0

// app crashes when b is null
val l: Int = b!!.length
```

Safe Calls

```
var b: String? = "abc"

// length is null when b is null
val l: Int? = b?.length

// length is zero when b is null
val l: Int = b?.length ?: 0

// app crashes when b is null
val l: Int = b!!.length
```

Safe Calls

```
var b: String? = "abc"

// length is null when b is null
val l: Int? = b?.length

// length is zero when b is null
val l: Int = b?.length ?: 0

// app crashes when b is null
val l: Int = b!!.length
```

Safe Calls

```
var b: String? = "abc"

// length is null when b is null
val l: Int? = b?.length

// length is zero when b is null
val l: Int = b?.length ?: 0

// app crashes when b is null
val l: Int = b!!.length
```

Safe Casts

```
// safe cast: a can have any type
val aInt: Int? = a as? Int

// unsafe cast: a should be null or Int
val aInt: Int? = a as Int?

// unsafe cast: a should be Int
val aInt: Int = a as Int
```

Safe Casts

```
// safe cast: a can have any type
val aInt: Int? = a as? Int

// unsafe cast: a should be null or Int
val aInt: Int? = a as Int?

// unsafe cast: a should be Int
val aInt: Int = a as Int
```

Safe Casts

```
// safe cast: a can have any type
val aInt: Int? = a as? Int

// unsafe cast: a should be null or Int
val aInt: Int? = a as Int?

// unsafe cast: a should be Int
val aInt: Int = a as Int
```

Функции

Синтаксис функций

Объявление

```
fun double(x: Int): Int {
    return 2 * x
}

fun double(x: Int): Int = 2 * x

fun double(x: Int) = 2 * x
```

```
public final int double(int x) {
    return 2 * x;
}
```

Вызов

```
val doubled = double(42)
```

```
int doubled = double(42);
```

Синтаксис функций

Объявление

```
fun double(x: Int): Int {
    return 2 * x
}

fun double(x: Int): Int = 2 * x

fun double(x: Int) = 2 * x
```

```
public final int double(int x) {
    return 2 * x;
}
```

Вызов

```
val doubled = double(42)
```

```
int doubled = double(42);
```

Синтаксис функций

Объявление

```
fun double(x: Int): Int {
    return 2 * x
}

fun double(x: Int): Int = 2 * x

fun double(x: Int) = 2 * x
```

```
public final int double(int x) {
    return 2 * x;
}
```

Вызов

```
val doubled = double(42)
```

```
int doubled = double(42);
```

Синтаксис функций

Объявление

```
fun double(x: Int): Int {
    return 2 * x
}

fun double(x: Int): Int = 2 * x

fun double(x: Int) = 2 * x
```

```
public final int double(int x) {
    return 2 * x;
}
```

Вызов

```
val doubled = double(42)
```

```
int doubled = double(42);
```

Синтаксис функций

Объявление

```
fun double(x: Int): Int {
    return 2 * x
}

fun double(x: Int): Int = 2 * x

fun double(x: Int) = 2 * x
```

```
public final int double(int x) {
    return 2 * x;
}
```

Вызов

```
val doubled = double(42)
```

```
int doubled = double(42);
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Значения параметров функций по умолчанию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = true,
             wordSeparator: Char = ' '): String {
    ...
}

reformat(str)
reformat(str, true, true, false, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
reformat(str, wordSeparator = '_')
```

Extension functions

Объявление

```
// extension function
fun Int.double(): Int = 2 * this

fun Int.double() = 2 * this


// extension property
val Int.doubled: Int get() = 2 *
this
```

Вызов

```
// extension function call
val doubled = 42.double()
```

```
// extension property call
val doubled = 42.doubled
```

Extension functions

Объявление

```
// extension function
fun Int.double(): Int = 2 * this

fun Int.double() = 2 * this


// extension property
val Int.doubled: Int get() = 2 * this
```

Вызов

```
// extension function call
val doubled = 42.double()


// extension property call
val doubled = 42.doubled
```

Extension functions

Объявление

```
// extension function
fun Int.double(): Int = 2 * this

fun Int.double() = 2 * this
```

```
// extension property
val Int.doubled: Int get() = 2 * this
```

Вызов

```
// extension function call
val doubled = 42.double()
```

```
// extension property call
val doubled = 42.doubled
```

Extension functions

Объявление

```
// extension function
fun Int.double(): Int = 2 * this

fun Int.double() = 2 * this


// extension property
val Int.doubled: Int get() = 2 * this
```

Вызов

```
// extension function call
val doubled = 42.double()


// extension property call
val doubled = 42.doubled
```

Extension functions

Объявление

```
// extension function
fun Int.double(): Int = 2 * this

fun Int.double() = 2 * this


// extension property
val Int.doubled: Int get() = 2 * this
```

Вызов

```
// extension function call
val doubled = 42.double()


// extension property call
val doubled = 42.doubled
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
package com.example

// package-level fun
fun <T> listOf(vararg elements: T): List<T>
    = if (elements.size > 0) elements.asList() else emptyList()



class Human {
    var age: Int

    // class-level fun (member function)
    fun happyBirthday() = age++
}
```

Function scope

```
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

Function scope

```
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

Function scope

```
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

Function scope

```
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

Function scope

```
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

Function scope

```kotlin
/**
 * Depth-first search
 */
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()

    // local fun
    fun dfs(current: Vertex) {
        if (!visited.add(current)) {
            return
        }
        for (v in current.neighbors) {
            dfs(v)
        }
    }

    dfs(graph.vertices[0])
}
```

// see com.example.ktsample.Dfs.kt

Объекты

Объекты как анонимные реализации

```
// anonymous interface implementation
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }
    override fun mouseEntered(e: MouseEvent) { ... }
})
```

“Просто объекты”

```
fun foo() {
    // “just an object” - anonimous entity
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

Object declarations - именованные объекты

```
// declaration
object AllHumansInApp {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    val humans: List<Human> get() = humansArrayList
}


// usage
val human = Human()
AllHumansInApp.register(human)
```

Object declarations - именованные объекты

```
// declaration
object AllHumansInApp {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    val humans: List<Human> get() = humansArrayList
}


// usage
val human = Human()
AllHumansInApp.register(human)
```

Object declarations - именованные объекты

```
// declaration
object AllHumansInApp {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    val humans: List<Human> get() = humansArrayList
}


// usage
val human = Human()
AllHumansInApp.register(human)
```

Object declarations - именованные объекты

```
// declaration
object AllHumansInApp {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    val humans: List<Human> get() = humansArrayList
}


// usage
val human = Human()
AllHumansInApp.register(human)
```

Object declarations - именованные объекты

```
// declaration
object AllHumansInApp {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    val humans: List<Human> get() = humansArrayList
}


// usage
val human = Human()
AllHumansInApp.register(human)
```

Объекты как экземпляры

```
interface Group {
    val humans: List<Human>
}

fun assignToTask(group: Group) {
    ...
}

object AllHumansInApp: Group {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    override val humans: List<Human> get() = humansArrayList
}

assignToTask(AllHumansInApp)
```

Объекты как экземпляры

```
interface Group {
    val humans: List<Human>
}

fun assignToTask(group: Group) {
    ...
}

object AllHumansInApp: Group {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    override val humans: List<Human> get() = humansArrayList
}

assignToTask(AllHumansInApp)
```

Объекты как экземпляры

```
interface Group {
    val humans: List<Human>
}

fun assignToTask(group: Group) {
    ...
}

object AllHumansInApp: Group {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    override val humans: List<Human> get() = humansArrayList
}

assignToTask(AllHumansInApp)
```

Объекты как экземпляры

```
interface Group {
    val humans: List<Human>
}

fun assignToTask(group: Group) {
    ...
}

object AllHumansInApp: Group {
    private val humansArrayList = ArrayList<Human>()

    fun register(human: Human) = humansArrayList.add(human)

    override val humans: List<Human> get() = humansArrayList
}

assignToTask(AllHumansInApp)
```

Companion Objects

```
// declaration
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

// usage
val instance = MyClass.create()
```

Companion Objects

```
// declaration
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

// usage
val instance = MyClass.create()
```

Companion Objects

```
// declaration
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

// usage
val instance = MyClass.create()
```

Companion Objects

```
// declaration
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

// usage
val instance = MyClass.create()
```

Companion Objects

```
// declaration
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

// usage
val instance = MyClass.create()

val instance = MyClass.Factory.create()
```

Unit

```
fun printHello(name: String?) {
    if (name != null) {
        println("Hello ${name}")
    }
    else {
        println("Hi there!")
    }
}
```

Unit

```
fun printHello(name: String?) {
    if (name != null) {
        println("Hello ${name}")
    }
    else {
        println("Hi there!")
    }
}
```

Unit

```
fun printHello(name: String?) {
    if (name != null) {
        println("Hello ${name}")
    }
    else {
        println("Hi there!")
    }
}
```

Unit

```
fun printHello(name: String?): Unit {
    if (name != null) {
        println("Hello ${name}")
    }
    else {
        println("Hi there!")
    }

    // `return Unit` or `return` is optional
    return Unit
}
```

Unit implementation

```
/**
 * The type with only one value: the `Unit` object. This type
corresponds to the `void` type in Java.
 */
public object Unit {
    override fun toString() = "kotlin.Unit"
}
```

Классы

Все классы, функции и свойства класса final по умолчанию

```
class Human (
    var age: Int
) {
    fun happyBirthday() {
        age++
    }
}
```

```
public final class Human {
    private int age;
    public Human(int age) {
        this.age = age;
    }
    public final void
happyBirthday() {
        setAge(age + 1)
    }
    public final int getAge() {
        return age;
    }
    public final void setAge(int
value{
        this.age = value;
    }
}
```

Все классы, функции и свойства класса final по умолчанию

```kotlin
class Human (
    var age: Int
) {
    fun happyBirthday() {
        age++
    }
}
```

```java
public final class Human {
    private int age;
    public Human(int age) {
        this.age = age;
    }
    public final void
happyBirthday() {
        setAge(age + 1)
    }
    public final int getAge() {
        return age;
    }
    public final void setAge(int
value{
        this.age = value;
    }
}
```

Все классы, функции и свойства класса final по умолчанию

```
class Human (
    var age: Int
) {
    fun happyBirthday() {
        age++
    }
}
```

```
public final class Human {
    private int age;
    public Human(int age) {
        this.age = age;
    }
    public final void
happyBirthday() {
        setAge(age + 1)
    }
    public final int getAge() {
        return age;
    }
    public final void setAge(int
value{
        this.age = value;
    }
}
```

Все классы, функции и свойства класса final по умолчанию

```
class Human (
    var age: Int
) {
    fun happyBirthday() {
        age++
    }
}
```

```
public final class Human {
    private int age;
    public Human(int age) {
        this.age = age;
    }
    public final void
happyBirthday() {
        setAge(age + 1)
    }
    public final int getAge() {
        return age;
    }
    public final void setAge(int
value{
        this.age = value;
    }
}
```

Открытые для наследования классы и доступные для переопределения функции должны быть явно объявлены

```
open class Base {
    open fun virtual() { ... }
    fun nonVirtual() { ... }
}

class Derived() : Base() {
    override fun virtual() { ... }
}
```

Data Classes

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

Data Classes

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

Будут сгенерированы:

- equals()/hashCode()

Data Classes

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

Будут сгенерированы:

- equals()/hashCode()
- toString() вида "User(name=John, age=42)"

Data Classes

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

Будут сгенерированы:

- equals()/hashCode()
- toString() вида "User(name=John, age=42)"
- componentN() функции, возвращающие значения пропертей класса

Data Classes

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

Будут сгенерированы:

- equals()/hashCode()
- toString() вида "User(name=John, age=42)"
- componentN() функции, возвращающие значения пропертей класса
- copy()

Data Classes

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

Будут сгенерированы:

- equals()/hashCode()
- toString() вида "User(name=John, age=42)"
- componentN() функции, возвращающие значения пропертей класса
- copy()

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

Sealed Classes

- абстрактны

Sealed Classes

- абстрактны
- могут иметь только приватные конструкторы

Sealed Classes

- абстрактны
- могут иметь только приватные конструкторы
- наследники могут быть созданы только в том же классе/файле

Sealed Classes

- абстрактны
- могут иметь только приватные конструкторы
- наследники могут быть созданы только в том же классе/файле

Как результат - наследники образовывают конечное множество сущностей и могут быть перечислены

Sealed Classes

```
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```kotlin
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```kotlin
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```kotlin
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```kotlin
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Sealed Classes

```kotlin
// see com.example.ktsample.Expr.kt
sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Any and Nothing

kotlin.Any

```
/**
 * The root of the Kotlin class hierarchy. Every Kotlin class has [Any] as a
superclass.
 */
public open class Any {

    public open operator fun equals(other: Any?): Boolean

    public open fun hashCode(): Int

    public open fun toString(): String
}
```

kotlin.Nothing

```
/**
 * Nothing has no instances. You can use Nothing to represent "a value that never
exists": for example, if a function has the return type of Nothing, it means that it
never returns (always throws an exception).
 */
public class Nothing private constructor()
```

kotlin.Nothing

```
/**
 * Nothing has no instances. You can use Nothing to represent "a value that never
exists": for example, if a function has the return type of Nothing, it means that it
never returns (always throws an exception).
 */
public class Nothing private constructor(): Any
```

kotlin.Nothing

```
/**
 * Nothing has no instances. You can use Nothing to represent "a value that never
exists": for example, if a function has the return type of Nothing, it means that it
never returns (always throws an exception).
 */
public class Nothing private constructor(): Any

fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

Higher-Order Functions

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}


listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action.invoke(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Функция, которая принимает в качестве аргумента или возвращает другую функцию.

```
// see com.example.ktsample.ForEach.kt

fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}

listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
listOf(1, 2, 3, 4).myForEach(::print)
listOf(1, 2, 3, 4).myForEach(printer::print)
```

Higher-Order Function

Более сложный пример функции высшего порядка

```
// see com.example.ktsample.Fold.kt
```

Inline function

```
fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}
```

```
listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
```

```
for (element in listOf(1, 2, 3, 4)) action(element) {
   println(it.toString())
}
```

Inline function

```
inline fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}
```

```
listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
```

```
for (element in listOf(1, 2, 3, 4)) action(element) {
   println(it.toString())
}
```

Inline function

```
inline fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}
```

```
listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
```

```
for (element in listOf(1, 2, 3, 4)) action(element) {
   println(it.toString())
}
```

Inline function

```
inline fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}
```

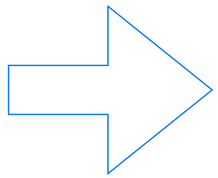
```
listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
```

```
for (element in listOf(1, 2, 3, 4)) action(element) {
   println(it.toString())
}
```

Inline function

```
inline fun <T> Iterable<T>.myForEach(action: (T) -> Unit) {
    for (element in this) action(element)
}
```

```
listOf(1, 2, 3, 4).myForEach{ println(it.toString()) }
```



```
for (element in listOf(1, 2, 3, 4)) action(element) {
   println(it.toString())
}
```

Delegated properties

Lazy property

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

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

Lazy property

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

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

Lazy property

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

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

Lazy property

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

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

Lazy property

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

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

```
>computed!
>Hello
>Hello
```

Реализация DelegatedProperty

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}

// see com.example.ktsample.DelegatedProperty.kt
```

Delegated implementation

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Delegated implementation

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Collections

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

List and MutableList

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)          // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)     // prints "[1, 2, 3, 4]"
readOnlyView.clear()      // -> does not compile
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for collections creation

```
listOf(1, 2, 3)
mutableListOf(1, 2, 3)

setOf(1, 2, 3)
mutableSetOf(1, 2, 3)

mapOf(1 to “1”, 2 to “2”, 3 to “3”)
mutableMapOf(1 to “1”, 2 to “2”, 3 to “3”)
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Useful functions for operations with collections

```
// see com.example.ktsample.Collections.kt

fun fibonachiJazz() {
    val chars = listOf(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711)
        .filter { it % 2 == 0 }
        .map<Int, String> { it.toString() }
        .flatMap<String, Char> { numStr -> numStr.toList() }
        .distinct()
        .sorted()

    println(chars)
}

// Result: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Немного о сахаре

- apply
- with
- let
- also
- run

Реализация with

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}
```

Использование with

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
val person: Person = getPerson()

with(person) {
    print(name)
    print(age)
}
```

```
val person: Person = getPerson()

print(person.name)
print(person.age)
```

Использование with

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
val person: Person = getPerson()

with(person) {
    print(name)
    print(age)
}
```

```
val person: Person = getPerson()

print(person.name)
print(person.age)
```

Использование with

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
val person: Person = getPerson()

with(person) {
    print(name)
    print(age)
}
```

```
val person: Person = getPerson()

print(person.name)
print(person.age)
```

Использование with

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
val person: Person = getPerson()

with(person) {
    print(name)
    print(age)
}
```

```
val person: Person = getPerson()

print(person.name)
print(person.age)
```

Реализация apply

```
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Использование apply

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
fun getPerson() = Person().apply {
    // only access properties in apply block!
    name = "Peter"
    age = 18
}
```

```
fun getPerson() {
    val peter = Person()
    clark.name = "Peter"
    clark.age = 18
    return peter
}
```

Использование apply

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
fun getPerson() = Person().apply {
    // only access properties in apply block!
    name = "Peter"
    age = 18
}
```

```
fun getPerson() {
    val peter = Person()
    clark.name = "Peter"
    clark.age = 18
    return peter
}
```

Использование apply

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
fun getPerson() = Person().apply {
    // only access properties in apply block!
    name = "Peter"
    age = 18
}
```

```
fun getPerson() {
    val peter = Person()
    clark.name = "Peter"
    clark.age = 18
    return peter
}
```

Использование apply

```
class Person {
    var name: String? = null,
    var age: Int? = null
}
```

```
fun getPerson() = Person().apply {
    // only access properties in apply block!
    name = "Peter"
    age = 18
}
```

```
fun getPerson() {
    val peter = Person()
    clark.name = "Peter"
    clark.age = 18
    return peter
}
```

Реализация also

```
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
```

Использование also

```
class Book(author: Person) {
    val author = author.also {
      //does not mutate its receiver parameter
      requireNotNull(it.age)
      print(it.name)
    }
}
```

```
class Book(val author: Person) {
    init {
      requireNotNull(author.age)
      print(author.name)
    }
}
```

Использование also

```
class Book(author: Person) {
    val author = author.also {
      //does not mutate its receiver parameter
      requireNotNull(it.age)
      print(it.name)
    }
}
```

```
class Book(val author: Person) {
    init {
      requireNotNull(author.age)
      print(author.name)
    }
}
```

Использование also

```
class Book(author: Person) {
    val author = author.also {
      //does not mutate its receiver parameter
      requireNotNull(it.age)
      print(it.name)
    }
}
```

```
class Book(val author: Person) {
    init {
      requireNotNull(author.age)
      print(author.name)
    }
}
```

Использование also

```
class Book(author: Person) {
    val author = author.also {
      //does not mutate its receiver parameter
      requireNotNull(it.age)
      print(it.name)
    }
}
```

```
class Book(val author: Person) {
    init {
      requireNotNull(author.age)
      print(author.name)
    }
}
```

Использование also

```
class Book(author: Person) {
    val author = author.also {
      //does not mutate its receiver parameter
      requireNotNull(it.age)
      print(it.name)
    }
}
```

```
class Book(val author: Person) {
    init {
      requireNotNull(author.age)
      print(author.name)
    }
}
```

Реализация let

```
inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}
```

Использование let

```
getNullablePerson()?.let {
    // only executed when not-null
    promote(it)
}

val driversLicence: Licence? = getNullablePerson()?.let {
    // convert nullable person to nullable driversLicence
    licenceService.getDriversLicence(it)
}

val person: Person = getPerson()
getPersonDao().let { dao ->
    // scope of dao variable is limited to this block
    dao.insert(person)
}
```

Использование let

```
getNullablePerson()?.let {
    // only executed when not-null
    promote(it)
}

val driversLicence: Licence? = getNullablePerson()?.let {
    // convert nullable person to nullable driversLicence
    licenceService.getDriversLicence(it)
}

val person: Person = getPerson()
getPersonDao().let { dao ->
    // scope of dao variable is limited to this block
    dao.insert(person)
}
```

Использование let

```
getNullablePerson()?.let {
    // only executed when not-null
    promote(it)
}

val driversLicence: Licence? = getNullablePerson()?.let {
    // convert nullable person to nullable driversLicence
    licenceService.getDriversLicence(it)
}

val person: Person = getPerson()
getPersonDao().let { dao ->
    // scope of dao variable is limited to this block
    dao.insert(person)
}
```

Сравнение with, also, apply, let, run

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}

inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}

inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}

inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}

inline fun <T, R> T.run(block: T.() -> R): R {
    return block()
}
```

Generics

Синтаксис шаблонов

```
class Box<T>(t: T) {
    var value = t
}


val box: Box<Int> = Box<Int>(1)

// 1 has type Int, so the compiler figures out that we are talking about Box<Int>
val box = Box(1)
```

Синтаксис шаблонов

```
class Box<T>(t: T) {
    var value = t
}


val box: Box<Int> = Box<Int>(1)

// 1 has type Int, so the compiler figures out that we are talking about Box<Int>
val box = Box(1)
```

Синтаксис шаблонов

```
class Box<T>(t: T) {
    var value = t
}


val box: Box<Int> = Box<Int>(1)

// 1 has type Int, so the compiler figures out that we are talking about Box<Int>
val box = Box(1)
```

out T

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
}
```

out T

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
}
```

out T

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
}
```

out T

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
}
```

out T

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
}
```

In T

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

In T

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

In T

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

In T

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

In T

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

Type erasure

```
Foo<Bar>     ====>     Foo<*>
Foo<Baz?>    ====>     Foo<*>
```

Reified

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}

treeNode.findParentOfType(MyTreeNode::class.java)

treeNode.findParentOfType<MyTreeNode>()
```

Reified

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

```
treeNode.findParentOfType<MyTreeNode>()
```

Reified

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}

treeNode.findParentOfType(MyTreeNode::class.java)

treeNode.findParentOfType<MyTreeNode>()
```

Reified

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}

treeNode.findParentOfType(MyTreeNode::class.java)

treeNode.findParentOfType<MyTreeNode>()
```

Reified

```
treeNode.findParentOfType<MyTreeNode>()

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

Reified

```
treeNode.findParentOfType<MyTreeNode>()

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

Reified

```
treeNode.findParentOfType<MyTreeNode>()

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

Reified

```
treeNode.findParentOfType<MyTreeNode>()

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

Reified

```
treeNode.findParentOfType<MyTreeNode>()

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

Заключение

Рассмотренные темы

- преимущества и недостатки языка Kotlin

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety
- функции высшего порядка и лямбда-выражения

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety
- функции высшего порядка и лямбда-выражения
- inline

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety
- функции высшего порядка и лямбда-выражения
- inline
- объекты

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety
- функции высшего порядка и лямбда-выражения
- inline
- объекты
- работа с коллекциями

Рассмотренные темы

- преимущества и недостатки языка Kotlin
- синтаксис
- nullsafety
- функции высшего порядка и лямбда-выражения
- inline
- объекты
- работа с коллекциями
- generics

Рассмотренные темы

- Delegated properties

Рассмотренные темы

- Delegated properties
- Delegated implementation

Не были рассмотрены

- Reflection

Не были рассмотрены

- Reflection
- Coroutines

Не были рассмотрены

- Reflection
- Coroutines
- Sequences

Дополнительные материалы

https://kotlinlang.org - официальная документация языка

https://kotlinlang.ru - фанатский перевод документации

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro
2. Разобраться со следующими примерами:
   - Collections.kt
   - Dfs.kt
   - Expr.kt
   - Fold.kt
   - ForEach.kt

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro
2. Разобраться со следующими примерами:
   - Collections.kt
   - Dfs.kt
   - Expr.kt
   - Fold.kt
   - ForEach.kt
3. Поиграться с play.kotlinlang.org

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro
2. Разобраться со следующими примерами:
   - Collections.kt
   - Dfs.kt
   - Expr.kt
   - Fold.kt
   - ForEach.kt
3. Поиграться с play.kotlinlang.org
4. Посмотреть java bytecode для delegated property и понять в чём проблема с ней.

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro
2. Разобраться со следующими примерами:
   - Collections.kt
   - Dfs.kt
   - Expr.kt
   - Fold.kt
   - ForEach.kt
3. Поиграться с play.kotlinlang.org
4. Посмотреть java bytecode для delegated property и понять в чём проблема с ней.
5. Посмотреть Sequences.

Домашнее задание

1. Склонировать проект github.com/tewnn/Kotlin-Intro
2. Разобраться со следующими примерами:
   - Collections.kt
   - Dfs.kt
   - Expr.kt
   - Fold.kt
   - ForEach.kt
3. Поиграться с play.kotlinlang.org
4. Посмотреть java bytecode для delegated property и понять в чём проблема с ней.
5. Посмотреть Sequences.
6. Пройти Kotlin Koans online или в Android Studio/Idea.

Яндекс

Спасибо за внимание!

Степан Чурюканов

tewnn@yandex.ru

https://t.me/tewnn