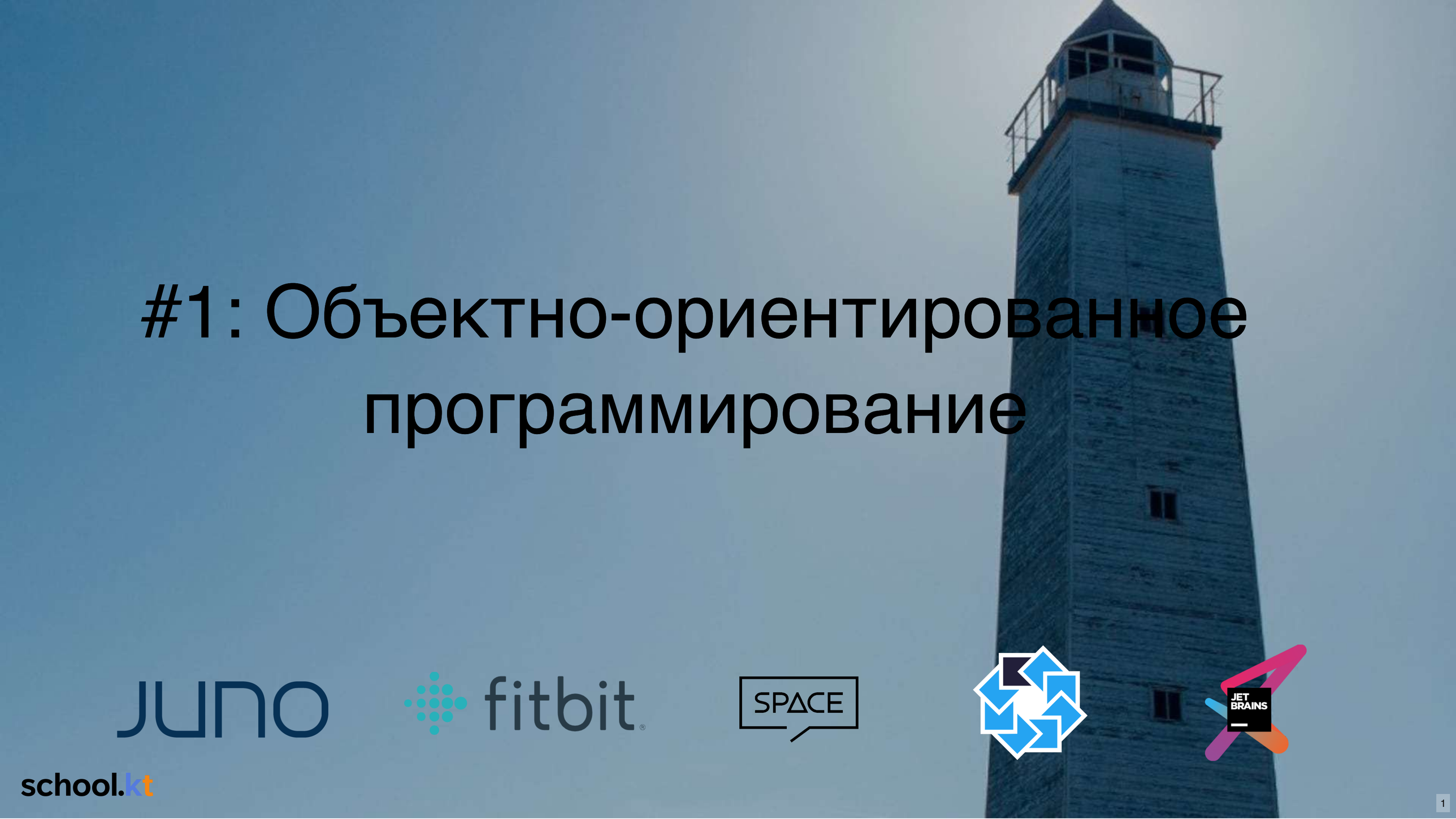
#1: Объектно-ориентированное программирование

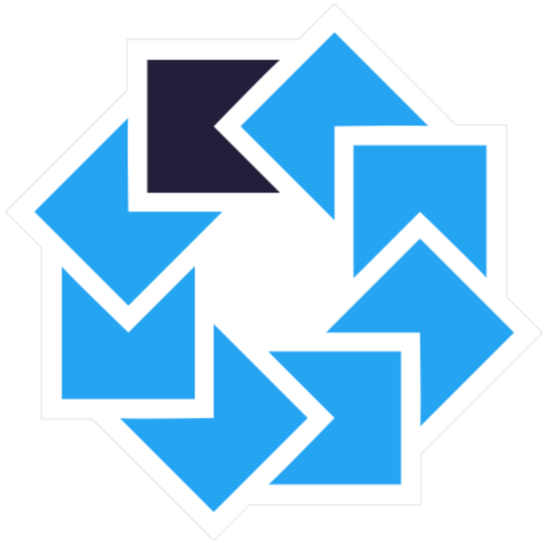


Руслан Ибрагимов

Full Stack Developer @ ObjectStyle

@HeapyHop

ruslan@Ibragimov.by

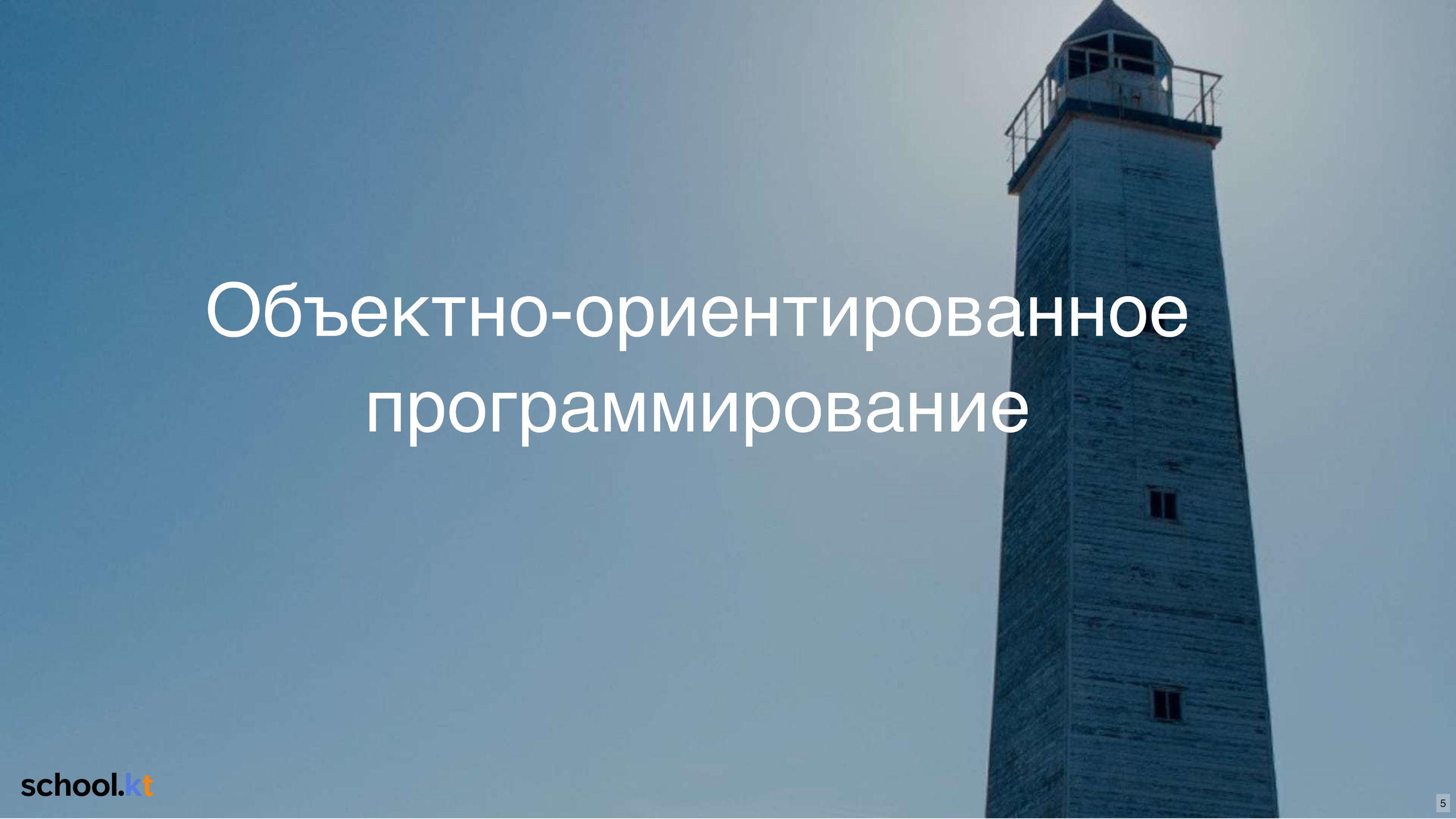


fitbit®

SPACE

juno

Менторы



Объектно-ориентированное программирование

Сегодня

- Классы
- Модификаторы видимости
- Properties (Свойства)
- Наследование
- Интерфейсы
- Объекты
- Делегирование

Классы

Классы (Classes)

```
class Student
```

Классы (Classes)

```
class Student

class Student(val name: String, val age: Int)
```

Классы (Classes)

```
class Student

class Student(val name: String, val age: Int)

class Student(
    val name: String,
    val age: Int
)
```

Классы (Classes)

```
class Student
```

☹
```
class Student(val name: String, val age: Int)
```

☺
```
class Student(
    val name: String,
    val age: Int
)
```

Классы (Classes)

```
class Student(
    val name: String,
    val age: Int
) {

}
```

Классы (Classes)

```
class Student(
    val name: String,
    val age: Int
) {
    fun print() {
        println("Student: $name, $age y.o.")
    }
}
```

Классы (Classes)

```
class Student(
    val name: String,
    val age: Int
) {
    fun print() {
        println("Student: $name, $age y.o.")
    }
}

fun main() {
    Student("Ruslan", 27).print()
}
```

Классы (Classes)

```
class Student(
    val name: String,
    val age: Int
)
```

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

1. Класс public, final

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

1. Класс public, final
2. Поля final

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

1. Класс public, final
2. Поля final
3. Int превратился в int

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

1. Класс public, final
2. Поля final
3. Int превратился в int
4. геттеры public, final

school.kt

Классы (Classes)

```java
public final class Student {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
    public final String getName() { return this.name; }

    public final int getAge() { return this.age; }

    public Student(@NotNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```kotlin
class Student(
    val name: String,
    val age: Int
)
```

1. Класс public, final
2. Поля final
3. Int превратился в int
4. геттеры public, final
5. конструктор

Модификаторы

```
final class FinalUser

open class OpenUser

abstract class AbstractUser
```

Модификаторы

```
final class FinalUser

open class OpenUser

abstract class AbstractUser
```

Effective Java:

Item 17: Design and document for inheritance or else prohibit it

Модификаторы доступа

```
public class PublicUser

private class PrivateUser

internal class InternalUser
```

Модификаторы доступа

```
public class PublicUser

private class PrivateUser

internal class InternalUser
```

Effective Java:

Item 13: Minimize the accessibility of classes and members

Final vs Public

Модификаторы доступа

Modifier	Class member	Top-level declaration
public	visible everywhere	visible everywhere
internal	visible in a module	visible in a module
protected	visible in a subclass	----
private	visible in a class	visible in a file

Модификаторы доступа & JVM

Kotlin	**JVM**
public	public
private	private / package private
protected	protected
internal	?

Internal

```
class WithInternalMember {
    internal val message = ""
}
```

Internal

```
class WithInternalMember {
    internal val message = ""
}

new WithInternalMember().getMessage$lectures();
```

File Private & JVM

```
// file.kt

private fun callMeMaybe(): Int = 42

class SomePublicClass {
    fun doWork() {
        println(callMeMaybe())
    }
}
```

File Private & JVM

```kotlin
// file.kt

private fun callMeMaybe(): Int = 42

class SomePublicClass {
    fun doWork() {
        println(callMeMaybe())
    }
}
```

```kotlin
FileKt.callMeMaybe();

new SomePublicClass();
```

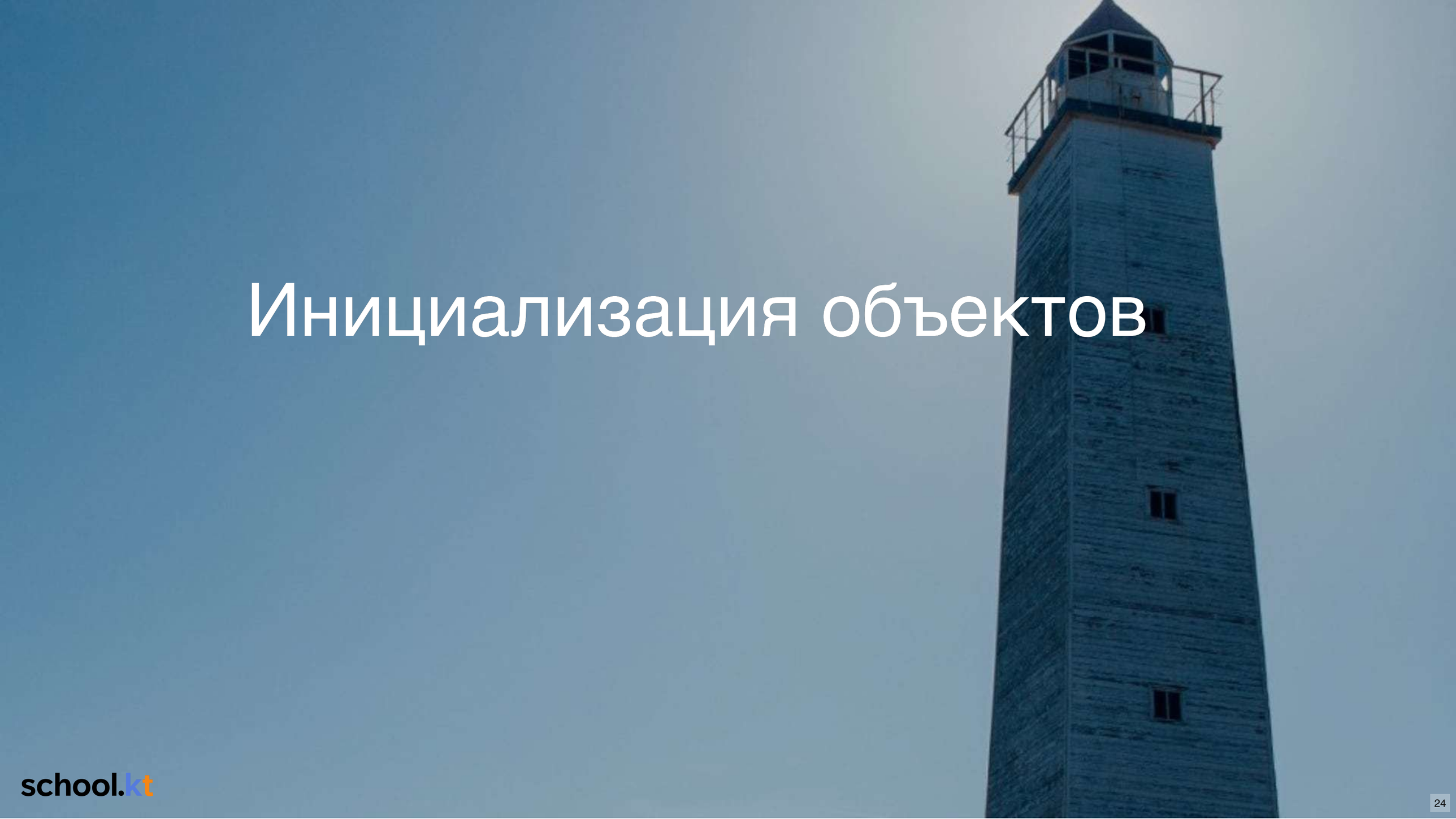
File Private & JVM

```
public final class SomePublicClass {
    public final void doWork() {
        int var1 = FileKt.access$callMeMaybe();
        System.out.println(var1);
    }
}
```

File Private & JVM

```
public final class SomePublicClass {
    public final void doWork() {
        int var1 = FileKt.access$callMeMaybe();
        System.out.println(var1);
    }
}

public final class FileKt {
    private static final int callMeMaybe() { return 42; }

    // $FF: synthetic method
    public static final int access$callMeMaybe() { return callMeMaybe(); }
}
```

Инициализация объектов



Primary Constructor (первичный)

```
class Student(
    val name: String,
    val age: Int
)
```

Primary Constructor (первичный)

```
class Student(
    val name: String,
    val age: Int
)

class Student constructor(
    val name: String,
    val age: Int
)
```

Primary Constructor (первичный)

```
class Student private constructor(
    val name: String,
    val age: Int
)
```

Primary Constructor (первичный)

```
class Student private constructor(
    val name: String,
    val age: Int
)

class Student @Inject constructor(
    val name: String,
    val age: Int
)
```

Primary Constructor (первичный)

```
class Student @Inject @Qualifier constructor(
    val name: String,
    val age: Int
)
```

Primary Constructor (первичный)

```
class Student @Inject @Qualifier constructor(
    val name: String,
    val age: Int
)

class Student @[Inject Qualifier] constructor(
    val name: String,
    val age: Int
)
```

Secondary Constructor (вторичный)

```
class Student(
    val name: String,
    val age: Int
) {
    constructor(name: String) : this(name, age: 42)
}
```

Secondary Constructor (вторичный)

```
class Student(
    val name: String,
    val age: Int
) {
    constructor(name: String) : this(name, age: 42)
}

Student( name: "Luke", age: 42)
Student( name: "Luke")
```

Secondary Constructor (вторичный)

```
class Student(
    val name: String,
    val age: Int
) {
    constructor(name: String) : this(name, age: 42)
}
```

Secondary Constructor (вторичный)

```
class Student(
    val name: String,
    val age: Int
) {
    constructor(name: String) : this(name, age: 42)
}
class Student(
    val name: String,
    val age: Int = 42
)
```

Secondary Constructor (вторичный)

```
class Student @JvmOverloads constructor(
    val name: String,
    val age: Int = 42
)
```

Secondary Constructor (вторичный)

```
class Student(
    val name: String,
    val age: Int
) {
    constructor(name: String) : this(name, age: 42) {
        // additional initialization logic
    }
}
```

Initializer blocks

```
class Student(name: String) {
    val name: String

    init {
        this.name = name
    }
}
```

Initializer blocks

```
class Student(name: String) {
    val name: String

    init {
        this.name = name
    }
}
```

Initializer blocks

```
class Student(name: String) {
    val name: String

    init {
        this.name = name
    }

    init {
        println("Я родился")
    }
}
```

Initializer blocks

```kotlin
class Student(name: String) {
    val name: String

    init {
        this.name = name
    }

    init {
        println("Я родился")
    }
}
```

```java
public final class Student {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() { return this.name; }

    public Student(@NotNull String name) {
        super();
        Intrinsics.checkParameterIsNotNull(name, paramName: "name");
        this.name = name;
        String var2 = "Я родился";
        System.out.println(var2);
    }
}
```

Initializer blocks

```
class Student(name: String) {
    val name: String

    init {
        this.name = name
    }

    init {
        println("Я родился")
    }
}
```

```
public final class Student {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() { return this.name; }

    public Student(@NotNull String name) {
        super();
        Intrinsics.checkParameterIsNotNull(name, paramName: "name");
        this.name = name;
        String var2 = "Я родился";
        System.out.println(var2);
    }
}
```

Quiz

```
open class AbstractFactory {
    init { print("Abstract ") }
}

class ConcreteFactory : AbstractFactory() {
    init { print("Concrete ") }
}

fun main() {
    ConcreteFactory()
}
```

Quiz

```
open class AbstractFactory {
    init { print("Abstract ") }
}

class ConcreteFactory : AbstractFactory() {
    init { print("Concrete ") }
}

fun main() {
    ConcreteFactory()
}
```

a) Concrete
b) Abstract Concrete
c) Concrete Abstract
d) Не скомпилируется

Наследование

```
interface Base
class DefaultBase : Base
```

Наследование

```
interface Base
class DefaultBase : Base

open class Parent
class Child : Parent()
```

Наследование

```
open class Parent(val name: String)
class Child(name: String) : Parent(name)
```

Наследование

```
open class Parent(val name: String)
class Child(name: String) : Parent(name)

open class Parent(val name: String)
class Child : Parent {
    constructor(name: String) : super(name)
}
```

Наследование

```
interface Base {
    fun base(): Int
}
```

Наследование

```
interface Base {
    fun base(): Int
}

class DefaultBase : Base {
    override fun base(): Int = 42
}
```

Наследование

```
interface Base {
    fun base(): Int
}
```

Наследование

```
interface Base {
    fun base(): Int
}

open class DefaultBase : Base {
    final override fun base(): Int = 42
}
```

Наследование

```
interface Base {
    fun base(): Int = 42
}
```

Наследование

```
public interface Base {
    int base();

    public static final class DefaultImpls {
        public static int base(Base $this) { return 42; }
    }
}
```

Наследование

```
class DefaultBase : Base {
    override fun base(): Int {
        return super.base() + 1
    }
}
```

```
public final class DefaultBase implements Base {
    public int base() {
        return Base.DefaultImpls.base( $this: this) + 1;
    }
}
```

Наследование

```
interface Base {
    @JvmDefault
    fun base(): Int = 42
}
```

Наследование

```
interface Base {
    @JvmDefault
    fun base(): Int = 42
}
```

Наследование

```
interface Base {
    @JvmDefault
    fun base(): Int = 42
}
```

```
public interface Base {
    @JvmDefault
    default int base() { return 42; }
}
```

Quiz

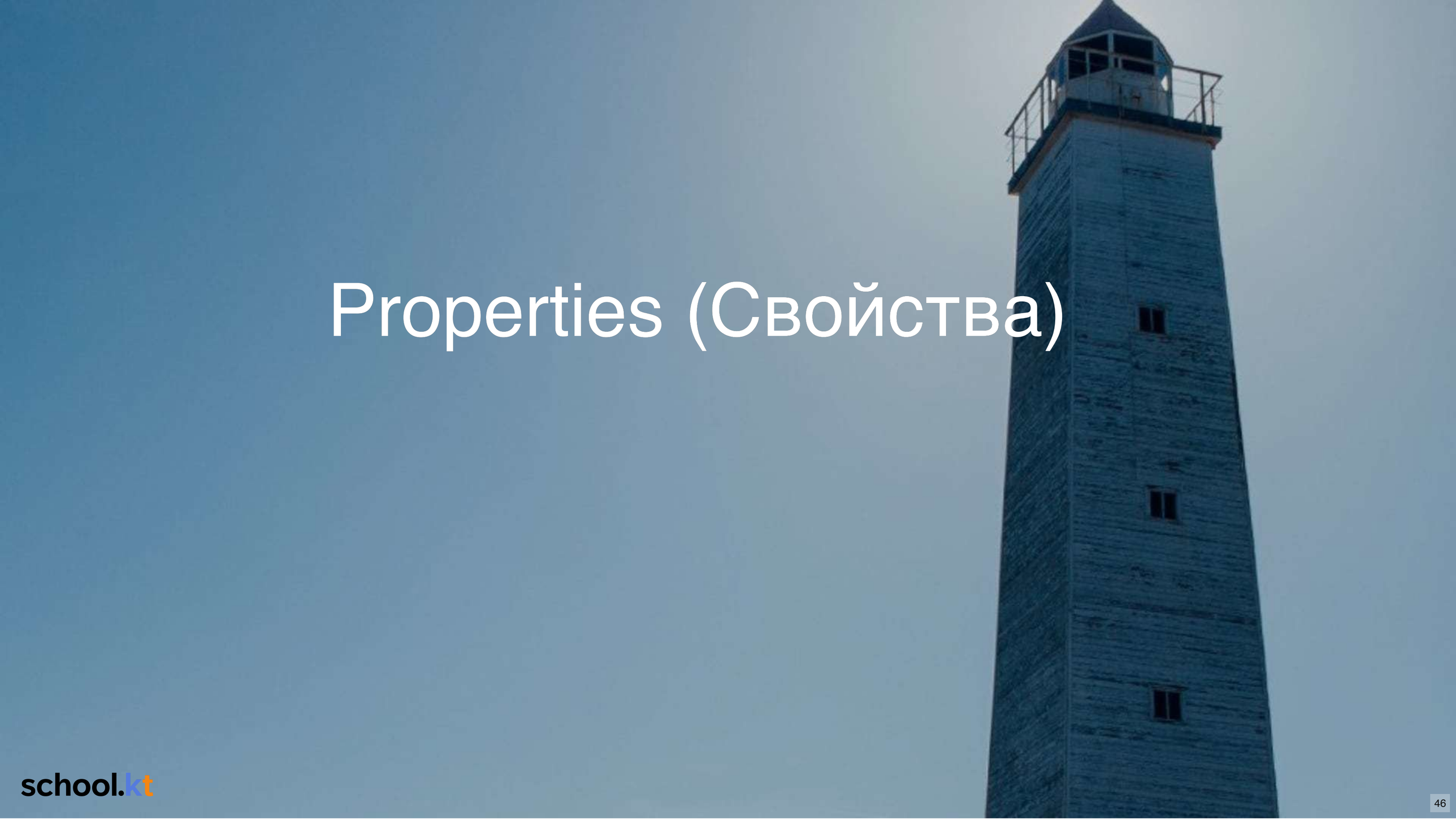
```
abstract class AbstractLogoFactory {
    abstract val name: String
    val logo = name[0].toUpperCase()
}

class KotlinLogoFactory : AbstractLogoFactory() {
    override val name = "Kotlin"
}

fun main() {
    print(KotlinLogoFactory().logo)
}
```

Quiz

```
abstract class AbstractLogoFactory {
    abstract val name: String
    val logo = name[0].toUpperCase()
}

class KotlinLogoFactory : AbstractLogoFactory() {
    override val name = "Kotlin"
}

fun main() {
    print(KotlinLogoFactory().logo)
}
```

a) K
b) NullPointerException
c) IllegalStateException
d) Не скомпилируется

Properties (Свойства)



Properties (Свойства)

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

Properties (Свойства)

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

Getters/Setters

```
val s = Student( name: "Ruslan", age: 27)
val name :String = s.name
s.age = 28
```

Getters/Setters

```kotlin
val s = Student( name: "Ruslan", age: 27)
val name :String = s.name
s.age = 28
```

```java
final var s = new Student( name: "Ruslan", age: 27);
final var name = s.getName();
s.setAge(28);
```

Getters/Setters

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

```
class Student(
    val firstName: String,
    val lastName: String,
    age: Int
) {
    val name: String
        get() = "$firstName $lastName"

    var age: Int = age
        set(value) {
            check(value > 0)
            field = value
        }
}
```

Getters/Setters

```
class Student(
    val name: String,
    age: Int
) {
    var age: Int = age
        private set(value) {
            check(value > 0)
            field = value
        }

    fun incrementAge() {
        age++
    }
}
```

```
fun main() {
    val s = Student( name: "Ruslan", age: 42)
    s.incrementAge()
    s.age = 42
}
```

Getters/Setters

```
class Student(
    val name: String,
    age: Int
) {
    var age: Int = age
        private set(value) {
            check(value > 0)
            field = value
        }

    fun incrementAge() {
        age++
    }
}
```

```
fun main() {
    val s = Student( name: "Ruslan", age: 42)
    s.incrementAge()
    s.age = 42
}
```

Backing Field

```
class Student(
    val name: String,
    age: Int
) {
    var age: Int = age
        private set(value) {
            check(value > 0)
            field = value
        }

    fun incrementAge() {
        age++
    }
}
```

Backing Field

```
class Student(
    val name: String,
    age: Int
) {
    var age: Int = age
        private set(value) {
            check(value > 0)
            field = value
        }

    fun incrementAge() {
        age++
    }
}
```

Оптимизации

```
class Counter {
    var count: Int = 0

    fun increment() {
        count++
    }
}
```

Оптимизации

```
class Counter {
    var count: Int = 0

    fun increment() {
        count++
    }
}
```

```
public final class Counter {
    private int count;

    public final int getCount() { return this.count; }

    public final void setCount(int var1) { this.count = var1; }

    public final void increment() { int var10001 = this.count++; }
}
```

Свойства в интерфейсах

```
interface Contact {
    val name: String

    val age: Int
        get() = 42
}
```

Свойства в интерфейсах

```
interface Contact {
    val name: String

    val age: Int
        get() = 42
}
```

```
class Student(
    override val name: String,
    override val age: Int
) : Contact
```

Свойства в интерфейсах

```
interface Contact {
    val name: String
    var age: Int
}
```

Свойства в интерфейсах

```kotlin
interface Contact {
    val name: String
    var age: Int
}
```

```java
public interface Contact {
    @NotNull
    String getName();

    int getAge();

    void setAge(int var1);
}
```

Свойства в интерфейсах

```
interface Contact {
    val name: String
    val age: Int get() = 42
}

class Student(
    override val name: String
) : Contact

fun main() {
    println(Student("Ruslan").age)
}
```

Lateinit

```
class UserController {
    var service: UserService? = null
}
```

Lateinit

```
class UserController {
    var service: UserService? = null
}

class UserController {
    lateinit var service: UserService
}
```

Lateinit

```
class UserController {
    var service: UserService? = null
}

class UserController {
    lateinit var service: UserService
}
class UserController {
    lateinit var service: UserService

    fun check() {
        this::service.isInitialized
    }
}
```

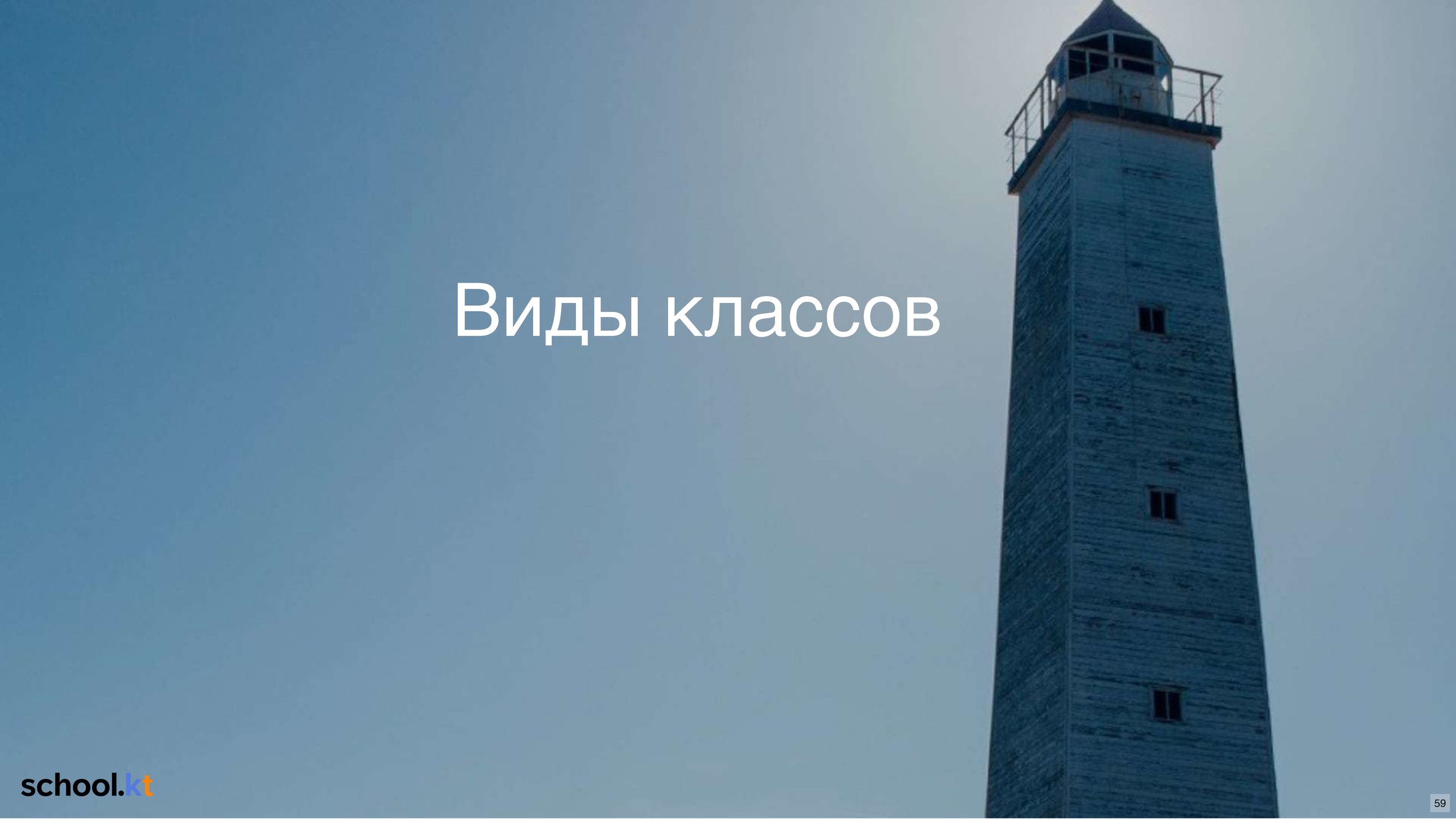
Extension Properties

```
val <T> List<T>.lastElement: T
    get() = this[size - 1]

fun main() {
    println(listOf(1, 2, 3).lastElement)
}
```

Extension Properties

```
var <T> MutableList<T>.lastElement: T
    get() = this[size - 1]
    set(value) {
        this[size - 1] = value
    }

fun main() {
    val list = mutableListOf(1, 2, 3)
    list.lastElement = 4
    println(list)
}
```

Виды классов

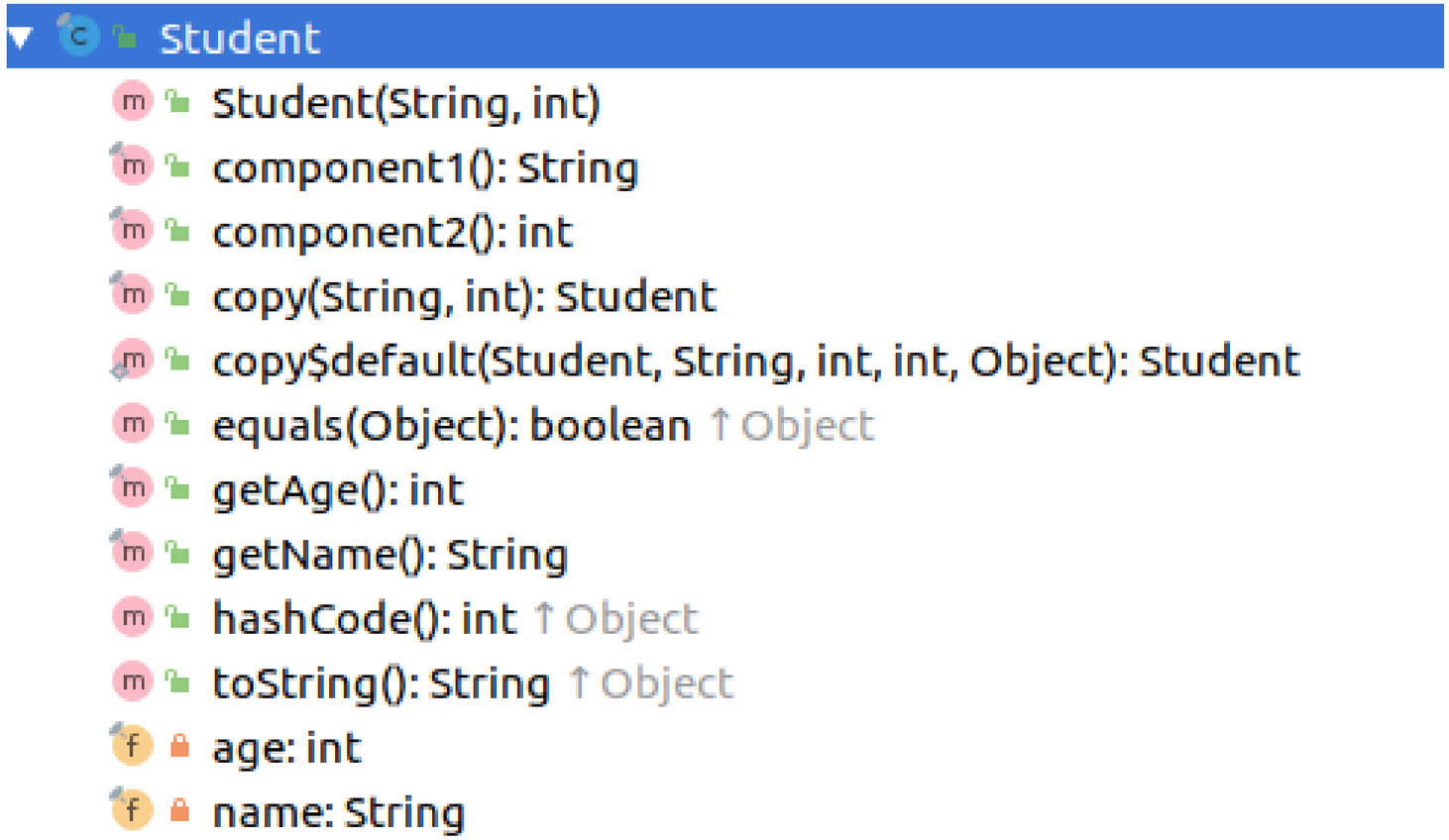


Data

```
class Student(
    val name: String,
    val age: Int
)

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

Data

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



Data

```
val (name: String, age: Int) = Student( name: "Ruslan", age: 27)
```

Data

```
val (name: String, age: Int) = Student( name: "Ruslan", age: 27)

val s1 = Student( name: "Ruslan", age: 27)
val s2: Student = s1.copy(age = 28)
```

Data

```
val (name: String, age: Int) = Student( name: "Ruslan", age: 27)

val s1 = Student( name: "Ruslan", age: 27)
val s2: Student = s1.copy(age = 28)

val s1 = Student( name: "Ruslan", age: 27)
val s2 = Student( name: "Ruslan", age: 27)
s1 == s2 // true
```

Data

```
val (name: String, age: Int) = Student( name: "Ruslan", age: 27)

val s1 = Student( name: "Ruslan", age: 27)
val s2: Student = s1.copy(age = 28)

val s1 = Student( name: "Ruslan", age: 27)
val s2 = Student( name: "Ruslan", age: 27)
s1 == s2 // true

// Student(name=Ruslan, age=27)
println(Student( name: "Ruslan", age: 27))
```

Sealed (Изолированные)

```
sealed class Expr

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

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

object NotANumber : Expr()
```

Sealed (Изолированные)

```
sealed class Expr

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

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

object NotANumber : Expr()
```

Sealed (Изолированные)

```
fun eval(expr: Expr): Double = when (expr) {
    is Const → expr.number
    is Sum → eval(expr.e1) + eval(expr.e2)
    NotANumber → Double.NaN
}
```

Sealed (Изолированные)

```
fun eval(expr: Expr): Double = when (expr) {
    is Const → expr.number
    is Sum → eval(expr.e1) + eval(expr.e2)
    NotANumber → Double.NaN
}
```

Sealed (Изолированные)

```
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
}

fun main() {
    println(eval(Sum(Const(1.0), Const(2.0))))
}
```

Inner & Nested

Java	Kotlin	Класс определенный внутри другого класса
static class A	**class A**	nested class
class A	inner class A	inner class

Inner & Nested

Java	Kotlin	Класс определенный внутри другого класса
static class A	**class A**	nested class
class A	inner class A	inner class

```
class Contact {
    class Address(
        val street: String,
        val building: String
    )
}
```

Inner & Nested

Java	Kotlin	Класс определенный внутри другого класса
static class A	**class A**	nested class
class A	inner class A	inner class

```
class Contact {
    class Address(
        val street: String,
        val building: String
    )
}
```

Inner & Nested

Java	Kotlin	Класс определенный внутри другого класса
static class A	**class A**	nested class
class A	inner class A	inner class

```
class Contact {
    class Address(
        val street: String,
        val building: String
    )
}
```

```
class Contact {
    inner class Address(
        val street: String,
        val building: String
    )
}
```

Inline

```
inline class UInt(val x: Int)
```

Inline

```
inline class UInt(val x: Int)

fun main() {
    println(UInt( x: 1).x)
}
```

Inline

```
inline class UInt(val x: Int)

fun main() {
    println(UInt( x: 1).x)
}

int var0 = UInt.constructor-impl(1);
System.out.println(var0);
```

Enum

Enum

```
enum class Color(
    val r: Int,
    val g: Int,
    val b: Int
) {
    BLUE( r: 0, g: 0, b: 255),
    GREEN( r: 0, g: 255, b: 0),
    RED( r: 255, g: 0, b: 0);

    fun rgb() :Int = (r * 256 + g) * 256 + b
}
```

Enum

```
enum class Color(
    val r: Int,
    val g: Int,
    val b: Int
) {
    BLUE( r: 0, g: 0, b: 255) {
        override fun self() :String = "blue"
    },
    GREEN( r: 0, g: 255, b: 0) {
        override fun self() :String = "green"
    },
    RED( r: 255, g: 0, b: 0) {
        override fun self() :String = "red"
    };

    abstract fun self(): String
}
```

Object

Object

```
object Single {
    fun callMe() {
        println("Hello!")
    }
}


fun main() {
    Single.callMe()
}
```

Object

```
public final class Single {
    public static final Single INSTANCE;

    public final void callMe() {
        String var1 = "Hello!";
        System.out.println(var1);
    }

    private Single() {
    }

    static {
        Single var0 = new Single();
        INSTANCE = var0;
    }
}
```

Object

```
interface CallMeable {
    fun callMe()
}

object Single : CallMeable {
    override fun callMe() {
        println("Hello!")
    }
}
```

Quiz

```
open class A(val x: Any?) {
    override fun toString() = javaClass.simpleName
}

object B : A(C)
object C : A(B)

fun main() {
    println(B.x)
    println(C.x)
}
```

Quiz

```
open class A(val x: Any?) {
    override fun toString() = javaClass.simpleName
}

object B : A(C)
object C : A(B)

fun main() {
    println(B.x)
    println(C.x)
}
```

a) null; null
b) C; null
c) ExceptionInInitializerError
d) Не скомпилируется

Object Expression

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

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

Object Expression

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}

println(adHoc.x + adHoc.y)
```

Companion Object

```
data class Student(
    val name: String,
    val age: Int
) {
    companion object {
        fun of(name: String) :Student  = Student(name,  age: 42)
    }
}

fun main() {
    Student.of( name: "Ruslan")
}
```

Companion Object

```
data class Student(
    val name: String,
    val age: Int
) {
    companion object {
        fun of(name: String) :Student  = Student(name,  age: 42)
    }
}

fun main() {
    Student.of( name: "Ruslan")
}
```

Companion Object

```
data class Student(
    val name: String,
    val age: Int,
    private val weight: Int
) {
    companion object {
        fun printWeight(student: Student) {
            println(student.weight)
        }
    }
}
```

Companion Object

```
data class Student(
    val name: String,
    val age: Int,
    private val weight: Int
) {
    companion object {
        fun printWeight(student: Student) {
            println(student.weight)
        }
    }
}
```

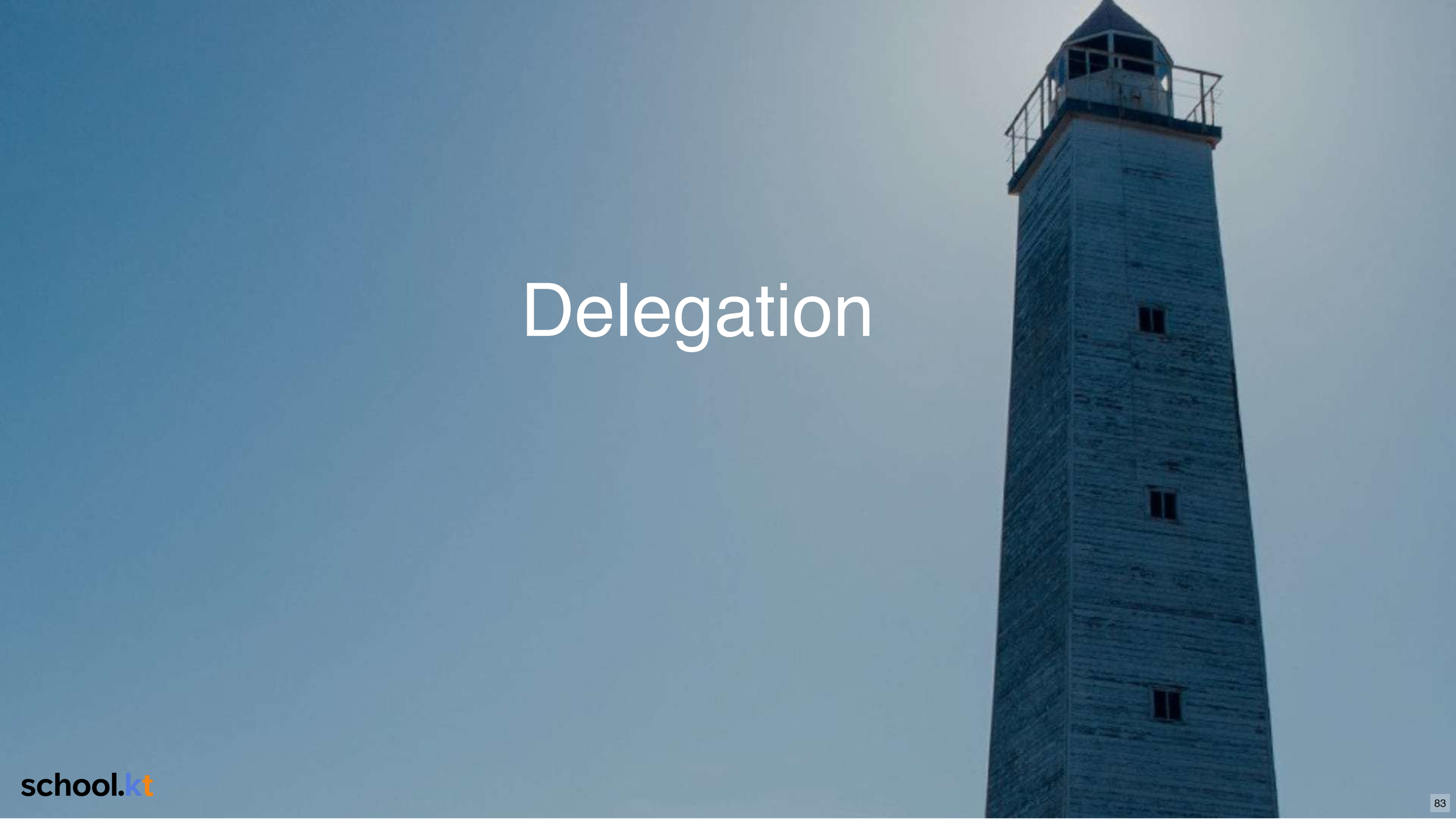
Companion Object

```
class Student(
    val name: String,
    val age: Int
) {
    companion object
}

fun Student.Companion.default(): Student {
    return Student( name: "Ruslan", age: 27)
}
```

Companion Object

```
class Student(
    val name: String,
    val age: Int
) {
    companion object
}

fun Student.Companion.default(): Student {
    return Student( name: "Ruslan", age: 27)
}

fun main() {
    Student.default()
}
```

Delegation



Class Delegation

```
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( x: 10)
    Derived(b).print()
}
```

Class Delegation

```
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( x: 10)
    Derived(b).print()
}
```

Class Delegation

```
data class Container(
    val name: String,
    private val items: List<Int>
) : List<Int> by items

fun main() {
    val (name, items) = Container("Kotlin", listOf(1, 2, 3))
    println ("Hello $name, $items")
}
```

Class Delegation

```
data class Container(
    val name: String,
    private val items: List<Int>
) : List<Int> by items

fun main() {
    val (name, items) = Container("Kotlin", listOf(1, 2, 3))
    println ("Hello $name, $items")
}
```

a) Hello Kotlin, [1, 2, 3]
b) Hello Kotlin, 1
c) Hello 1, 2
d) Hello Kotlin, 2


Спасибо!!!
bit.ly/2Nu9SjO