Kotlin

Kotlin это

Kotlin это

Остров в Финском заливе:

Kotlin это

Kotlin это

Деревня в Польше:

Coordinates: 51°54′36″N 17°39′0″E

Country	Poland
Voivodeship	Greater Poland
County	Jarocin
Gmina	Kotlin

Kotlin это

Кетчуп:

Kotlin это

Класс эскадренных миноносцев:

Kotlin это

Язык программирования

Kotlin

Kotlinlang.org

Kotlin как язык для разработки фронтенда

Залим Башоров

@bashorov

Временная ссылка на слайды: zal.im/s

Залим Башоров

zalim.bashorov@jetbrans.com

@bashorov

Руковожу направлением Kotlin JS

Kotlin

Kotlin

- Статически типизированный

Kotlin

- Статически типизированный
- Современный

Kotlin

- Статически типизированный
- Современный
- Прагматичный

Kotlin targets

Kotlin targets

- Java byte code

Kotlin targets

- Java byte code
  - Официальный язык разработки для Android

Kotlin targets

- Java byte code
  - Официальный язык разработки для Android
- JavaScript

Kotlin targets

- Java byte code
  - Официальный язык разработки для Android
- JavaScript
- Native

Kotlin JS

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1
  - поддержка более поздних версий ECMAScript запланирована

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1
  - поддержка более поздних версий ECMAScript запланирована
- всегда содержит "use strict"

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1
  - поддержка более поздних версий ECMAScript запланирована
- всегда содержит "use strict"
- быстрый

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1
  - поддержка более поздних версий ECMAScript запланирована
- всегда содержит "use strict"
- быстрый
- читаемый (на сколько это возможно)

Kotlin JS

Генерируемый код

- совместим с ECMAScript 5.1
  - поддержка более поздних версий ECMAScript запланирована
- всегда содержит "use strict"
- быстрый
- читаемый (на сколько это возможно)

- есть поддержка SourceMaps

Введение в Kotlin на примерах

Формат введения

```
// код на Kotlin
```

```
// код на JavaScript
```

Формат введения

```
// код на Kotlin
```

```
// код на JavaScript
```


```
// код с использованием фич ECMAScript 2015
```

Формат введения

```
// код на Kotlin
```

```
// код на JavaScript
```


```
// код с использованием фич ECMAScript 2015
```


```
// код на TypeScript
```

Замечания

Замечания

- Это не батл между языками

Замечания

- Это не батл между языками
- Цель - объяснить семантику

Замечания

- Это не батл между языками
- Цель - объяснить семантику
- Результат трансляции может отличаться

Замечания

- Это не батл между языками
- Цель - объяснить семантику
- Результат трансляции может отличаться
- Это не весь Kotlin

Переменная

Переменная

```
var
```

Переменная

```
var foo
```

Переменная

```
var foo: Int
```

Переменная

```
var foo: Int = 1
```

Переменная

```
var foo: Int = 1
```

```
var foo = 1;
```

JS

Переменная без явного указания типа

```
var foo = 1
```

```
var foo = 1;
```

JS

Переменная без явного указания типа

```
var foo = 1
```

```
var foo = 1;
```

JS

Переменная (ES 2015)

```
var foo = 1
```

```
var foo = 1;
```

JS

```
let foo = 1;
```

ES6

Неизменяемая переменная

```
val foo = 1
```

```
const foo = 1;
```

ES6

Nullable/not-null types

```
val a: String = "ok?" // OK
val b: String = null  // Ошибка компиляции
```

Nullable/not-null types

```
val a: String = "ok?" // OK
val b: String = null  // Ошибка компиляции
```

```
val c: String? = "ok?" // OK
val d: String? = null  // OK
```

Удобная работа с nullable значениями

Safe call

```
var foo: String? = ...
val a = foo?.length
val b = foo?.substring(1)
```

```
var foo = ...
var a;
if (foo != null) a = foo.length else a = null
var b;
if (foo != null) b = foo.substring(1) else b = null
```

JS

Удобная работа с nullable значениями

Elvis operator

```
var foo: String? = ...
val a = foo ?: "foo is null"
val b = foo?.substring(1) ?: "Kool"
```

```
var foo = ...
var a;
if (foo != null) a = foo else a = "foo is null"
var b;
if (foo != null) b = foo.substring(1) else b = "Kool"
```

JS

Функция

Функция

```
fun
```

Функция

```
fun sum
```

Функция

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

Функция

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

Функция

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

Функция

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

```
function sum(a, b) {
    return a + b
}
```

JS

Функция в одну строчку

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

```
function sum(a, b) {
    return a + b
}
```

JS

Функция в одну строчку

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

```javascript
function sum(a, b) {
    return a + b
}
```

JS

Функция в одну строчку

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

```
function sum(a, b) {
    return a + b
}
```

JS

Функция в одну строчку

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

```
function sum(a, b) {
    return a + b
}
```

JS

```
const sum = (a, b) => a + b
```

ES6

Функция как значение

Функция как значение

```
val lambda = { i: Int, s: String -> doSomething() }
```

```
const lambda = (i, s) => { doSomething() };
```

JS

Функция как значение

```
val lambda = { i: Int, s: String -> doSomething() }
```

```
const lambda = (i, s) => { doSomething() };
```

JS

Функция как значение

```
val lambda = { i: Int, s: String -> doSomething() }
```

```
const lambda = (i, s) => { doSomething() };
```

JS

Функция как значение

```
val lambda = { i: Int, s: String -> doSomething() }
```

```
const lambda = (i, s) => { doSomething() };
```

JS

Функция как значение

```kotlin
val lambda = { i: Int, s: String -> doSomething() }
```

```javascript
const lambda = (i, s) => { doSomething() };
```

JS

Вызов:

```kotlin
lambda(1, "string")
```

Функция как параметр другой функции (1)

Функция как параметр другой функции (1)

```
fun foo(f: (s: String) -> Unit ) {...}
```

Функция как параметр другой функции (1)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo({ s -> doSomething(s) })
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Функция как параметр другой функции (1)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo({ s -> doSomething(s) })
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Функция как параметр другой функции (1)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo() { s -> doSomething(s) }
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Функция как параметр другой функции (2)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo { s -> doSomething(s) }
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Type-Safe Html

```
html {
    head {
        title {+"HTML encoding with Kotlin"}
    }
    body {
        h1 {+"HTML encoding with Kotlin"}
        p  {+"this format can be used as an alternative markup to H

        // an element with attributes and text content
        a(href = "http://kotlinlang.org") {+"Kotlin"}

        // content generated by
        p {
            for (arg in args)
                +arg
        }
    }
```

Функция как параметр другой функции (2)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo { s -> doSomething(s) }
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Функция как параметр другой функции (3)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo { doSomething(it) }
}
```

```
function foo(f) {...}

function usage() {
    foo((s) -> doSomething(s))
}
```

JS

Функция как параметр другой функции (4)

```
fun foo(f: (s: String) -> Unit) {...}

fun usage() {
    foo(::doSomething)
}
```

```
function foo(f) {...}

function usage() {
    foo(doSomething)
}
```

JS

Условные операторы

Условные операторы: if

```
if (condition) { ... } else { ... }
```

```
if (condition) { ... } else { ... }
```

JS

Условные операторы: if

```
if (condition)   ...   else   ...
```

```
if (condition)   ...   else   ...
```

JS

Условные операторы: when

```
when {
    condtion1 -> doSomething1()
    condtion2 -> {
        doSomething2()
    }
    else -> doSomethingElse()
}
```

Условные операторы: when

```
when {
    condtion1 -> doSomething1()
    condtion2 -> {
        doSomething2()
    }
    else -> doSomethingElse()
}
```

```
if (condtion1)
    doSomething1()
else if (condtion2) {
    doSomething2()
}
else
    doSomethingElse()
}
```

JS

Условные операторы как выражения: if

```
val message = if (timeOfDay == "morning") "Hi!" else "Bye!"
```

```
const message = timeOfDay === "morning"? "Hi!" : "Bye!"
```

JS

Условные операторы как выражения: if (2)

```
val message =
        if (timeOfDay == "morning") {
            doSomething()
            "Hi!"
        } else {
            doSomethingElse()
            "Bye!"
        }
```

Условные операторы как выражения: if (3)

```
val message: String

if (timeOfDay == "morning") {
    doSomething()
    message = "Hi!"
} else {
    doSomethingElse()
    message = "Bye!"
}
```

Условные операторы как выражения: if (3)

```
val message: String

if (timeOfDay == "morning") {
    doSomething()
    message = "Hi!"
} else {
    doSomethingElse()
    message = "Bye!"
}
```

Условные операторы как выражения: when

```
val message = when (timeOfDay) {
    "morning" -> "Hi!"
    "evening" -> "Bye!"
    else -> null
}
```

```
/*  ¯\_(ツ)_/¯ */
```

JS

Условные операторы как выражения: when

```
val message = when (timeOfDay) {
    "morning" -> "Hi!"
    "evening" -> "Bye!"
    else -> null
}
```

```
let message;
switch(timeOfDay) {
    case "morning": message = "Hi!"; break;
    case "evening": message = "Bye!"; break;
    default: message = null;
}
```

ES6

Еще выражениям являются:

- `try-catch-finally`
- `throw`
- `return`

Циклы: while

```
while (condition) {
    // do something
}
```

```
while (condition) {
    // do something
}
```

JS

Циклы: do while

```
do {
    // do something
} while (condition)
```

```
do {
    // do something
} while (condition)
```

JS

Циклы: do while (2)

```
do {
    val foo = ...
    // do something
} while (foo == 0)
```

JS

```
{
    let foo = ...
    do {
        // do something
    } while (foo === 0)
}
```

Циклы: for (1)

```
for (i in 0..10) {
    // do something
}
```

```
for (var i = 0; i <= 10; i++) {
    // do something
}
```

JS

Циклы: for (2)

```
for (element in collection) {
    // do something
}
```

```
for (element of collection) {
    // do something
}
```

ES6

Строки

```
"Hello, HolyJS!"
```

```
"Hello, HolyJS!"
```

JS

Многострочные строки

```
"""Hello,
HolyJS!"""
```

```
`Hello,
HolyJS!`
```

ES6

Строковые шаблоны

(template literals, string interpolation)

```
val name = ...
"Hello, $name"
"Hello, ${name.firstChar()}"
```

```
const name = ...
"Hello, ${name}"
"Hello, ${name.firstChar()}"
```

ES6

Класс

Класс

```
class
```

Класс

```
class MyClass
```

Класс

```
class MyClass {

}
```

Класс

```kotlin
class MyClass {
    val foo = "text"
    fun bar(a: String) = a + foo
}
```

```javascript
class MyClass {
    constructor() {
        this.foo = "text";
    }
    bar(a) {
        return a + this.foo;
    }
}
```

JS

Класс: конструктор

```
class MyClass {
    val foo: String
    constructor(foo: String) {
        this.foo = foo
        doSomething()
    }
}
```

```
class MyClass {
    constructor(foo) {
        this.foo = foo;
        doSomething()
    }
}
```

JS

Класс: много конструкторов

```
class MyClass {
    val foo: String
    constructor(foo: String) { ... }

    constructor(foo: Boolean) {
        this.foo = "Boolean: " + foo
        doSomethingForBool()
    }

    constructor(foo: Int) : this(foo.toString()) {
        doSomethingForInt()
    }
}
```

Класс: primary конструктор

```
class MyClass(foo: String) {




}
```

```
class MyClass {
    constructor(foo) {


    }
}
```

JS

Класс: primary конструктор

```
class MyClass(foo: String) {
    val bar = foo



}
```

```
class MyClass {
    constructor(p) {
        this.bar = p;

    }
}
```

JS

Класс: primary конструктор

```
class MyClass(foo: String) {
    val bar = foo
    val foo: String
    init {
        this.foo = foo
    }
}
```

```
class MyClass {
    constructor(p) {
        this.bar = p;
        this.foo = p;
    }
}
```

JS

Класс: свойство в primary конструкторе

```
class MyClass(val foo: String)
```

```
class MyClass {
    constructor(foo) {
        this.foo = foo;
    }
}
```

JS

Класс: свойство в primary конструкторе

```
class MyClass(val foo: String)
```

```
class MyClass {
    constructor(foo) {
        this.foo = foo;
    }
}
```

JS

```
class MyClass {
    constructor(public foo: string) {
    }
}
```

TS

Интерфейс

```
interface Foo {
    val bar: Int
    fun baz(i: Int): String
}
```

Интерфейс

```kotlin
interface Foo {
    val bar: Int
    fun baz(i: Int): String
}
```

```typescript
interface Foo {
    bar: number
    baz(i: number): string
}
```

Интерфейс

```kotlin
interface Foo {
    val bar: Int
    fun baz(i: Int): String
}

//...

myvar is Foo // OK
```

```typescript
interface Foo {
    bar: number
    baz(i: number): string
}

//...

myvar instanceof Foo // Error
```

Интерфейс с кодом

- Может содержать реализацию
- Но не может иметь состояния

```
interface Foo {
    val bar
        get() = 42

    fun baz(i: Int) = i.toString()

    val boo = 51 // Ошибка компиляции
}
```

Наследование

Наследование

- Классы и их члены `final` по-умолчанию

```
interface A {
    fun foo(): String
}

open class B : A {
    override fun foo() = "B.str"
    open val bar = 1
}
```

Наследование

```
interface A {
    fun foo(): String
}

open class B : A {
    override fun foo() = "B.str"
    open val bar = 1
}
```

Наследование

```
interface A {
    fun foo(): String
}

open class B : A {
    override fun foo() = "B.str"
    open val bar = 1
}

interface C { ... }

class D : B(), C {
    override fun foo() = "D.str"
    override val bar = 2
}
```

Наследование

```kotlin
interface A {
    fun foo(): String
}

open class B : A {
    override fun foo() = "B.str"
    open val bar = 1
}

interface C { ... }

class D : B(), C {
    override fun foo() = "D.str"
    override val bar = 2
}
```

JS

```javascript
class B {
    constructor() {
        this.bar = 1
    }
    foo() {
        return "B.str"
    }
}

class D extends B {
    constructor() {
        this.bar = 2
    }
    foo() {
        return "D.str"
    }
}
```

Наследование на TypeScript

```ts
interface A {
    foo(): string
}

class B {
    bar: number
    constructor() {
        this.bar = 1
    }
    foo() {
        return "B.str"
    }
}
```

```ts
interface C {
    bar: number
}

class D extends B implements C {
    constructor() {
        super()
        this.bar = 2
    }
    foo() {
        return "D.str"
    }
}
```

Взаимодействие со внешним миром

Взаимодействие со внешним миром

- Вызов JavaScript кода из Kotlin

Взаимодействие со внешним миром

- Вызов JavaScript кода из Kotlin
- Вызов Kotlin кода из JavaScript

Вызов JavaScript кода из Kotlin

Вызов JavaScript кода из Kotlin

- Функция `js`

Вызов JavaScript кода из Kotlin

- Функция `js`
- Тип `dynamic`

Вызов JavaScript кода из Kotlin

- Функция `js`
- Тип `dynamic`
- Типизированные декларации

Вызов JavaScript кода из Kotlin

- Функция `js`
- Тип `dynamic`
- Типизированные декларации

(Способы перечислены от наименее безопасного к наиболее)

Функция `js(1)`

Функция `js(1)`

- Функция принимает константную строку

Функция `js(1)`

- Функция принимает константную строку
- Строка на этапе компиляции парсится и встраивается в AST

Функция `js` (1)

- Функция принимает константную строку
- Строка на этапе компиляции парсится и встраивается в AST
- Это не `eval` и не `prepack`

Функция `js` (1)

- Функция принимает константную строку
- Строка на этапе компиляции парсится и встраивается в AST
- Это не `eval` и не `prepack`

```
js("console.log('hello')")
```

Функция `js` (1)

- Функция принимает константную строку
- Строка на этапе компиляции парсится и встраивается в AST
- Это не `eval` и не `prepack`

```
js("console.log('hello')")
```

```
console.log('hello')
```

Функция `js` (2)

- Строка проверяется на синтаксическу корректность

Функция `js` (2)

- Строка проверяется на синтаксическу корректность

```
js("console log")
```

Функция `js` (2)

- Строка проверяется на синтаксическу корректность

```
js("console log")
```

Компилятор покажет ошибку:

```
error: JavaScript: missing ; before statement
    js("console log")
                  ^
```

Tun dynamic

Тип dynamic

- Доступен только в Kotlin JS

Тип `dynamic`

- Доступен только в Kotlin JS
- Значение типа `dynamic` можно присваивать куда угодно

```
var d: dynamic = ...; var i: Int = 1; var s: String = ""

i = d // OK
s = d // OK

i = s // Error
s = i // Error
```

Тип dynamic

- Доступен только в Kotlin JS
- Значение типа `dynamic` можно присваивать куда угодно
- В переменную типа `dynamic` можно писать что угодно

```
var d: dynamic = ...; var i: Int = 1; var s: String = ""

d = i    // OK
d = s    // OK
d = null // OK
```

Тип dynamic

- Доступен только в Kotlin JS
- Значение типа `dynamic` можно присваивать куда угодно
- В переменную типа `dynamic` можно писать что угодно
- Можно вызывать любую функцию и обращаться к любому полю

```
var d: dynamic = ...;
d.foo()
d.bar(1)
d.baz = "z"
```

Тип `dynamic`

- Доступен только в Kotlin JS
- Значение типа `dynamic` можно присваивать куда угодно
- В переменную типа `dynamic` можно писать что угодно
- Можно вызывать любую функцию и обращаться к любому полю
- Операторные вызовы компилируются как есть

```
d[0] = "a"
d + a
d > 1
```

```
d[0] = "a";
d + a;
d > 1;
```

JS

Тип `dynamic`

- Доступен только в Kotlin JS
- Значение типа `dynamic` можно присваивать куда угодно
- В переменную типа `dynamic` можно писать что угодно
- Можно вызывать любую функцию и обращаться к любому полю
- Операторные вызовы компилируются как есть

Типизированные декларации

- Ключевое слово: `external`
- Опционально, аннотации: `JsName`, `JsModule`, `JsQualifier`
- Из external декларации не порождается никакой код

external свойства / переменные

```JS
var property = 1
```

external свойства / переменные

```
var property = 1
```

JS

```
external var property: Int
```

external функции

```JS
/*
    s       {string}
    n       {number}
    returns {string}
 */
function foo(s, n) {
//...
}
```

external функции

JS

```
/*
    s       {string}
    n       {number}
    returns {string}
 */
function foo(s, n) {
//...
}
```

```
external fun foo(
                    s: String,
                    n: Double
                    ): String
```

external интерфейсы и классы

```
external interface A {
    val foo: Int
    fun bar(i: Int): Unit
}

external class B {
    val baz: Int
    fun boo(): Int
}
```

external интерфейсы и классы

```
external interface A {
    val foo: Int
    fun bar(i: Int): Unit
}

external class B {
    val baz: Int
    fun boo(): Int
}
```

- `external interface` существует только на этапе компиляции

DefinitelyTyped*

* definitelytyped.org

DefinitelyTyped*

- Большой репозиторий деклараций для TypeScript (>3000 деклараций)
- Написанные вручную

* definitelytyped.org

DefinitelyTyped*

- Большой репозиторий деклараций для TypeScript (>3000 деклараций)
- Написанные вручную

А причем тут Kotlin?

* definitelytyped.org

DefinitelyTyped*

- Большой репозиторий деклараций для TypeScript (>3000 деклараций)
- Написанные вручную

А причем тут Kotlin?

- Есть конвертор TypeScript деклараций в Kotlin (`ts2kt`)

* definitelytyped.org

DefinitelyTyped*

- Большой репозиторий деклараций для TypeScript (>3000 деклараций)
- Написанные вручную

А причем тут Kotlin?

- Есть конвертор TypeScript деклараций в Kotlin (`ts2kt`)
- Планируем создать подобный репозиторий с декларациями для Kotlin

* definitelytyped.org

Вызов Kotlin кода из JavaScript

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)

```
package foo.bar.baz
...
```

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)
- Сгенерированное имя декларации

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)
- Сгенерированное имя декларации

```
fun foo() {...}
```

```
function foo() {...}
```

JS

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)
- Сгенерированное имя декларации

```
fun foo() {...}
```

```
function foo() {...}
```

JS

```
fun boo(s: String) {...}
```

```
function boo_61zpoe$(s) {...}
```

JS

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)
- Сгенерированное имя декларации

```
<module>.<path.to.package>.<mangled_name>()
```

Вызов Kotlin кода из JavaScript

Полное имя любой декларации состоит из следующих частей:

- Ссылка на модуль
- Имена вложенных пакетов (package, namespace)
- Сгенерированное имя декларации

```
<module>.<path.to.package>.<mangled_name>()
```

```
someModule.foo.bar.baz.boo_61zpoe$("Hi!")
```
JS

Улучшим API при помощи `JsName`

```
someModule.foo.bar.baz.boo_61zpoe$("Hi!")
```
JS

Улучшим API при помощи `JsName`

```
someModule.foo.bar.baz.boo_61zpoe$("Hi!")
```

JS

```
@JsName("boo")
fun boo(s: String) {...}
```

Улучшим API при помощи `JsName`

```
someModule.foo.bar.baz.boo("Hi!")
```

JS

```
@JsName("boo")
fun boo(s: String) {...}
```

```
function boo(s) {...}
```

JS

Улучшим API при помощи `JsName`

```
someModule.foo.bar.baz.abc("Hi!")
```
JS

```
@JsName("abc")
fun boo(s: String) {...}
```

```
function abc(s) {...}
```
JS

Инструменты

Инструменты

- Intellij IDEA

Инструменты

- Intellij IDEA
- Инкрементальная компиляция (WIP)

Инструменты

- Intellij IDEA
- Инкрементальная компиляция (WIP)
- Sourcemaps

Инструменты

- Intellij IDEA
- Инкрементальная компиляция (WIP)
- Sourcemaps
- Оптимизатор (WIP)

Инструменты

- Intellij IDEA
- Инкрементальная компиляция (WIP)
- Sourcemaps
- Оптимизатор (WIP)
- Конвертор TypeScript деклараций в Kotlin (ts2kt)

Инструменты сборки

- gradle
  - kotlin-frontend-plugin
- maven
- ant
- webpack (через kotlin-frontend-plugin)

Планы Kotlin JS

- Инкрементальная компилция

Планы Kotlin JS

- Инкрементальная компилция
- Unit testing

Планы Kotlin JS

- Инкрементальная компилция
- Unit testing
- Оптимизация размера и скорости

Планы Kotlin JS

- Инкрементальная компилция
- Unit testing
- Оптимизация размера и скорости
- Поддержка новых версий ECMAScript

Планы Kotlin JS

- Инкрементальная компилция
- Unit testing
- Оптимизация размера и скорости
- Поддержка новых версий ECMAScript
- WebAssembly (?)

Полезные ссылки

- Сайт Kotlin: kotlinlang.org
  - документация: kotlinlang.org/docs/reference/
  - Try Kotlin: try.kotl.in
  - Kotlin Koans: try.kotl.in/koans
- Slack: kotlinlang.slack.com
  - регистрация slack.kotl.in
  - канал про Kotlin JS: #javascript

Спасибо!

Залим Башоров

Kotlin (kotl.in)

zalim.bashorov@jetbrans.com

@bashorov

Ссылка на слайды: zal.im/slides/holyjs17

Временная ссылка на слайды: zal.im/s