THE KOTLIN PROGRAMMING LANGUAGE

Sergey Lukjanov, 2012
slukjanov@mirantis.com

WHAT IS KOTLIN?

- JVM-targeted
- Statically typed
- Object-oriented
- General purpose
- Programming language
- Docs available today
- Open source from Feb 14

OUTLINE

- Motivation
- Design goals
- Feature overview
- Basic syntax
- Classes, types, inheritance
- High-order functions
- Generics
- Tuples
- Pattern matching
- Class objects

MOTIVATION

MOTIVATION

- IDEA codebase ≥ 200MB Java-code, ≥ 50k classes

MOTIVATION

- IDEA codebase ≥ 200MB Java-code, ≥ 50k classes
- Java libraries and community

MOTIVATION

- IDEA codebase ≥ 200MB Java-code, ≥ 50k classes
- Java libraries and community
- There are many languages, why not try?

DESIGN GOALS

DESIGN GOALS

- Full **Java interoperability**

DESIGN GOALS

- Full **Java interoperability**
- **Compiles** as fast as **Java**

DESIGN GOALS

- Full **Java interoperability**
- **Compiles** as fast as **Java**
- **Safer** than **Java**

DESIGN GOALS

- Full **Java interoperability**
- **Compiles** as fast as **Java**
- **Safer** than **Java**
- More **concise** than **Java**

DESIGN GOALS

- Full **Java interoperability**
- **Compiles** as fast as **Java**
- **Safer** than **Java**
- More **concise** than **Java**
- Way **simpler** than **Scala**

FEATURE OVERVIEW 1/2

- Static null-safety guarantees
- Traits
- First-class delegation
- Properties (instead of fields)
- Reified generics
- Declaration-site variance & “Type projections”
- High-order functions (“closures”)
- Extension properties and functions
- Inline-functions (zero-overhead closures)

FEATURE OVERVIEW 2/2

- Tuples
- Modules and build infrastructure
- Pattern matching
- Range expressions
- String templates
- Singletons
- Operator overloading
- Full-featured **IDE** by JetBrains
- Java to Kotlin converting

CODE EXAMPLES

HELLO, WORLD!


```
fun main(args : Array<String>) : Unit {
    println("Hello, World!");
}

fun println(any : Any?) /* : Unit */ {
    System.out?.println(any);
}
```

HELLO, <NAMES>!


```
fun main(args : Array<String>) {
    var names = ""; // names : String

    for(idx in args.indices) {
        names += args[idx]
        if(idx + 1 < args.size) {
            names += ", "
        }
    }

    println("Hello, $names!") // Groovy-style templates
}

val Array<*>.indices : Iterable<Int>
                get() = IntRange(0, size - 1)
```

HELLO, <NAMES>! (FASTER)


```
fun main(args : Array<String>) {
    var names = StringBuilder(); // names : StringBuilder

    for(idx in args.indices) {
        names += args[idx]
        if(idx + 1 < args.size) {
            names += ", "
        }
    }

    println("Hello, $names!") // Groovy-style templates
}

fun StringBuilder.plusAssign(any : Any?) {
    this.append(any)
}
```

HELLO, <NAMES>! (REALISTIC)


```
fun main(args : Array<String>) {
    println("Hello, ${args.join(", ")}!")
}
```

HELLO, <NAMES>! (REALISTIC)


```
fun main(args : Array<String>) {
    println("Hello, ${args.join(", ")}!")
}

fun <T> Iterable<T>.join(separator : String) : String {
    val names = StringBuilder()
    forit (this) {
        names += it.next()
        if (it.hasNext)
            names += separator
    }

    return names.toString() ?: ""
}

fun <T> forit(col : Iterable<T>, f : (Iterator<T>) -> Unit) {
    val it = col.iterator()
    while (it.hasNext)
        f(it)
}
```

NULL-SAFETY 1/2


```
fun parseInt(str : String) : Int? {
    try {
        return Integer.parseInt(str)
    } catch (e : NumberFormatException) {
        return null
    }
}
```

NULL-SAFETY 1/2


```
fun parseInt(str : String) : Int? {
    try {
        return Integer.parseInt(str)
    } catch (e : NumberFormatException) {
        return null
    }
}

fun main(args : Array<String>) {
    val x = parseInt("1027")
    val y = parseInt("Hello, World!") // y == null
    println(x?.times(2)) // can’t write x * 2
    println(x?.times(y)) // times argument can't be nullable
    println(x?.times(y.sure())) // throws NPE if y == null
    if (x != null) {
        println(x * 2)
    }
}
```

NULL-SAFETY 2/2


```
fun String?.isNullOrEmpty() : Boolean {
    return this == null || this.trim().length == 0
}
```

NULL-SAFETY 2/2


```
fun String?.isNullOrEmpty() : Boolean {
    return this == null || this.trim().length == 0
}


fun main(args : Array<String>) {
    println("Hello".isNullOrEmpty())     // false
    println(" World ".isNullOrEmpty())   // false
    println("    ".isNullOrEmpty())      // true
    println(null.isNullOrEmpty())        // true
}
```

AUTOMATIC CASTS


```
fun foo(obj : Any?) {
    if (obj is String) {
        println(obj.get(0));
    }
}
```

WHEN STATEMENT


```
fun foo(obj : Any?) {
    val x : Any? = when (obj) {
        is String -> obj.get(0)     // autocast to String
        is Int -> obj + 1           // autocast to Int
        !is Boolean -> null
        else -> "unknown"
    }

    val i : Int = when (obj) {
        is String -> if(obj.startsWith("a")) 1 else 0
        is Int -> obj
        else -> -1
    }
}
```

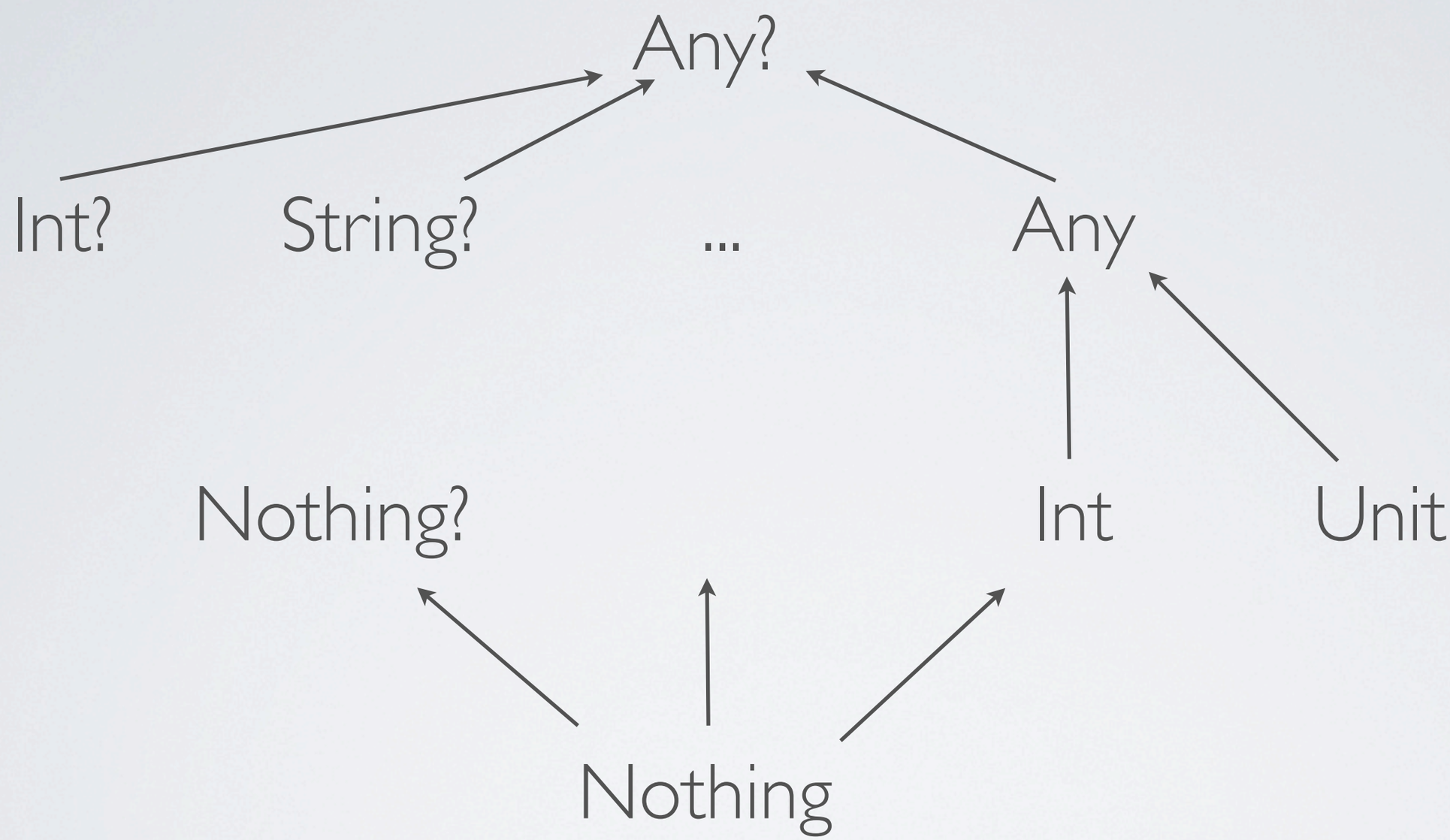
TYPES 1/2

Syntax	
Class types	`List<Foo>`
Nullable types	`Foo?`
Function types	`(Int) -> String`
Tuple types	`(Int, Int)`
Self types	`This`

TYPES 2/2

Special types	
Top	`Any?`
Bottom	`Nothing`
No meaningful return value	`Unit`

TYPES HIERARCHY



Complete lattice

MAPPING TO JAVA TYPES

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
`Any`		`Object`		`Any?`

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
`Any`		`Object`		`Any?`
`Unit`		`void`		`Unit`

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
`Any`		`Object`		`Any?`
`Unit`		`void`		`Unit`
`Int`		`int`		`Int`

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
`Any`		`Object`		`Any?`
`Unit`		`void`		`Unit`
`Int`		`int`		`Int`
`Int?`		`Integer`		`Int?`

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
`Any`		`Object`		`Any?`
`Unit`		`void`		`Unit`
`Int`		`int`		`Int`
`Int?`		`Integer`		`Int?`
`String`		`String`		`String?`

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
Any		Object		Any?
Unit		void		Unit
Int		int		Int
Int?		Integer		Int?
String		String		String?
Array<Foo>		Foo[]		Array<Foo?>?

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
Any		Object		Any?
Unit		void		Unit
Int		int		Int
Int?		Integer		Int?
String		String		String?
Array<Foo>		Foo[]		Array<Foo?>?
Array<Int>		int[]		Array<Int>?

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
Any		Object		Any?
Unit		void		Unit
Int		int		Int
Int?		Integer		Int?
String		String		String?
Array<Foo>		Foo[]		Array<Foo?>?
Array<Int>		int[]		Array<Int>?
List<Int>		List<Integer>		List<Int?>?

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
Any		Object		Any?
Unit		void		Unit
Int		int		Int
Int?		Integer		Int?
String		String		String?
Array<Foo>		Foo[]		Array<Foo?>?
Array<Int>		int[]		Array<Int>?
List<Int>		List<Integer>		List<Int?>?
Nothing		-		-

MAPPING TO JAVA TYPES

Kotlin	GEN	Java	LOAD	Kotlin
Any		Object		Any?
Unit		void		Unit
Int		int		Int
Int?		Integer		Int?
String		String		String?
Array<Foo>		Foo[]		Array<Foo?>?
Array<Int>		int[]		Array<Int>?
List<Int>		List<Integer>		List<Int?>?
Nothing		-		-
Foo		Foo		Foo?

CLASSES


```
open class Parent(p : Bar) {
    open fun foo() {
    }

    fun bar() {
    }
}

class Child(p : Bar) : Parent(p) {
    override fun foo() {
    }
}
```

- Any is the default supertype
- Constructors must initialize supertypes
- Final by default, explicit override annotations

TRAITS


```
trait T1 : Class1, OtherTrait {
    // no state
}

class Foo(p : Bar) : Class1(p), T1, T2 {
    // ...
}

class Decorator(p : T2) : Class2(), T2 by p {
    // ...
}
```

DISAMBIGUATION


```
trait A {
    fun foo() : Int = 1 // open by default
}

open class B() {
    open fun foo() : Int = 2 // not open by default
}

class C() : B(), A {
    override fun foo() = super<A>.foo() // returns 1
}
```

FIRST-CLASS FUNCTIONS


```
fun foo(arg : String) : Boolean // function


(p : Int) -> Int // function type
(Int) -> Int // function type


(a : Int) -> a + 1 // function literal
(b) : Int -> b * 2 // function literal
c -> c.times(2) // function literal
```

HIGH-ORDER FUNS


```
fun <T> filter( c : Iterable<T>, f: (T)->Boolean):Iterable<T>

filter(list, { s -> s.length < 3 })
```

HIGH-ORDER FUNS


```
fun <T> filter( c : Iterable<T>, f: (T)->Boolean):Iterable<T>

filter(list, { s -> s.length < 3 })

filter(list) { s -> s.length < 3 }
```

HIGH-ORDER FUNS


```
fun <T> filter( c : Iterable<T>, f: (T)->Boolean):Iterable<T>

filter(list, { s -> s.length < 3 })

filter(list) { s -> s.length < 3 }
// if only one arg:
```

HIGH-ORDER FUNS


```
fun <T> filter( c : Iterable<T>, f: (T)->Boolean):Iterable<T>

filter(list, { s -> s.length < 3 })

filter(list) { s -> s.length < 3 }
// if only one arg:
filter(list) { it.length < 3 }
```

LOCAL FUNCTIONS


```
fun reachable(from : Vertex, to : Vertex) : Boolean {
    val visited = HashSet<Vertex>()

    fun dfs(current : Vertex) {
        // here we return from the outer function:
        if (current == to) return@reachable true

        // And here - from local function:
        if (!visited.add(current)) return

        for (v in current.neighbors)
            dfs(v)
    }

    dfs(from)
    return false // if dfs() did not return true already
}
```

INFIX FUNCTION CALLS


```
// regular call:
a.contains("123")

// infix call:
a contains "123"
```

INFIX FUNCTION CALLS


```
// regular call:
a.contains("123")

// infix call:
a contains "123"


// “LINQ”
users
    .filter { it hasPrivilege WRITE }
    .map { it -> it.fullName }
    .orderBy { it.lastName }
```

LOCK EXAMPLE


```
myLock.lock()
try {
    // do something
} finally {
    myLock.unlock()
}
```

LOCK EXAMPLE


```
myLock.lock()
try {
    // do something
} finally {
    myLock.unlock()
}
```

```
lock(myLock) {
    // do something
}
```

LOCK EXAMPLE


```
myLock.lock()
try {
    // do something
} finally {
    myLock.unlock()
}
```

```
lock(myLock) {
    // do something
}
```

```
inline fun <T> lock(l : Lock, body : () -> T) : T {
    l.lock()
    try {
        return body()
    } finally {
        l.unlock()
    }
}
```

GENERICS: INVARIANCE


```
class List<T> {
    fun add(t : T)
    fun get(idx : Int) : T
}

val intList = List<Int>()
// We should not be able to do it:
val anyList : List<Any> = intList
anyList.add("1") // Cause of the problem
val i : Int = intList.get(0) // !!!
```

DECLARATION-SITE VARIANCE


```
class List<T> {
    fun add(t : T)
    fun get(idx : Int) : T
}
```

```
val intList = List<Int>()
val anyList : List<Any> = intList
```

DECLARATION-SITE VARIANCE


```
class List<T> {
    fun add(t : T)
    fun get(idx : Int) : T
}

class Producer<out T> {
    fun get() : T
}
```

```
val intList = List<Int>()
val anyList : List<Any> = intList

val intProd = Producer<Int>()
val anyProd : Producer<Any> = intProd
```

DECLARATION-SITE VARIANCE


```
class List<T> {
    fun add(t : T)
    fun get(idx : Int) : T
}
```

```
val intList = List<Int>()
val anyList : List<Any> = intList
```

```
class Producer<out T> {
    fun get() : T
}
```

```
val intProd = Producer<Int>()
val anyProd : Producer<Any> = intProd
```

```
class Consumer<in T> {
    fun add(t : T)
}
```

```
val anyCons = Consumer<Any>()
val intCons : Consumer<Int> = anyCons
```

USE-SITE VARIANCE


```
val intList = List<Int>()
val anyListOut : List<out Any> = intList
anyListOut.add("1") // Not available
val i : Int = intList.get(0) // No problem
```

```
val anyList = List<Any>()
val intListIn : List<in Int> = anyList
intListIn.add(123)
val obj = intListIn.get(0) // : Any?
```

REIFIED GENERICS


```
// Type information is retained in runtime
foo is List<T>
Array<T>(10)
T.create()
T.javaClass
```

REIFIED GENERICS


```
// Type information is retained in runtime
foo is List<T>
Array<T>(10)
T.create()
T.javaClass

// Java types is still erased...
foo is java.util.List<*>
```

TUPLES


```
class Tuple2<out T1, out T2>(
  val _1 : T1,
  val _2 : T2
)

val pair : #(Int, String) = #(1, "")
// same as 'Tuple2<Int, String>(1, "")'

when (x) {
  is #(null, *) => throw NullPointerException()
  is #(val a, val b) => print(a, b)
}

print("left = ${pair._1}, right = ${pair._2}")

val point : #(x : Int, y : Int) // labeled tuple
print("x = ${point.x}, y = ${point.y}")
val point : #(x : Int, y : Int) = #(y = 10, x = 5)
```

PATTERN MATCHING 1/2


```
when (a) {
    is Tree#(*, null) -> print("no right child")
    is Tree#(val l is Tree, val r is Tree) -> print("$l and $r")
    is Tree -> print("just a tree")
    is #(*, val b in 1..100) -> print(b)
    else -> print("unknown")
}
```

PATTERN MATCHING 1/2


```
when (a) {
    is Tree#(*, null) -> print("no right child")
    is Tree#(val l is Tree, val r is Tree) -> print("$l and $r")
    is Tree -> print("just a tree")
    is #(*, val b in 1..100) -> print(b)
    else -> print("unknown")
}

class Tree(val left : Tree?, val right : Tree?)
```

PATTERN MATCHING 1/2


```
when (a) {
    is Tree#(*, null) -> print("no right child")
    is Tree#(val l is Tree, val r is Tree) -> print("$l and $r")
    is Tree -> print("just a tree")
    is #(*, val b in 1..100) -> print(b)
    else -> print("unknown")
}
```

```
class Tree(val left : Tree?, val right : Tree?)

decomposer fun Any?.Tree() : #(Tree?, Tree?)? {
    return if (this is Tree) #(this.left, this.right) else null
}
```

PATTERN MATCHING 2/2


```
when (d) {
    is mmddyy#(02, 16, val a) -> print("Feb 16th of $a"))
}


class Date(val timestamp : Long) {
  fun mmddyy() : #(Int, Int, Int)? = #(month, day, year)
}

fun Date.mmddyy() : #(Int, Int, Int)? = #(month, day, year)
```

CLASS OBJECTS 1/2


```
class Example() {
    class object {
        fun create() = Example()
    }
}

val e = Example.create()
```

CLASS OBJECTS 2/2


```
trait Factory<T> {
    fun create() : T
}

class Example2() {
    class object : Factory<Example2> {
        override fun create() = Example2()
    }
}

val factory : Factory<Example2> = Example2
val e2 : Example2 = factory.create()
```

RESOURCES

Documentation: http://jetbrains.com/kotlin
Blog: http://blog.jetbrains.com/kotlin

THANKS

This presentation based on slides and speeches of Andrey Breslav, author of Kotlin language.

Q&A

Sergey Lukjanov, 2012
slukjanov@mirantis.com