The Kotlin Programming Language

Andrey Breslav

What is Kotlin?

- Statically typed
- object-oriented
- JVM-targeted
- general-purpose
- programming language
- developed by JetBrains
  - ➡ intended for industrial use

- Docs available today
- Public beta is planned for the end of 2011

Goal-wise...

Goal-wise...

- Number of research papers we are planning to publish on Kotlin is

Goal-wise...

- Number of research papers we are planning to publish on Kotlin is
  - ➡ Zero

Goal-wise...

- Number of research papers we are planning to publish on Kotlin is
  - ➡ Zero
  - ➡ ... or really close to that

Outline

- Motivation
- Feature overview
- Basic syntax
- Classes and Types
- Higher-order functions
- Type-safe Groovy-style Builders

Motivation

Motivation

- Why a new language?
  - We are not satisfied with the existing ones
  - And we have had a close look at many of them over 10 years

Motivation

- Why a new language?
  - We are not satisfied with the existing ones
  - And we have had a close look at many of them over 10 years
- Design goals
  - Full **Java interoperability**
  - **Compiles** as fast as **Java**
  - **Safer** than **Java**
  - More **concise** than **Java**
  - Way **simpler** than **Scala**

Feature overview

Feature overview

Static null-safety guarantees

Traits & First-class delegation

Properties (no fields)

Reified generics

Declaration-site variance & "Type projections"

Inline-functions (zero-overhead closures)

Higher-order functions ("closures")

Pattern matching

Extension functions

Modules and Build infrastructure

Feature overview

Static null-safety guarantees

Traits & First-cl… …egation

Properties (no fields)

Reified generics

Full-featured **IDE** by **JetBrains** from the very **beginning**

Declaration-site variance & "Type projections"

Inline-functions (zero-overhead closures)

Higher-order functions ("closures")

Modules and Build infrastructure

Pattern matching

Extension functions

Code examples

- Functions
- Java interoperability
- String templates
- Local variables
- Type inference
- Extension functions and properties
- Null-safety

Hello, world!

```
namespace hello

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

```
fun println(message : String) {
    System.out?.println(message)
}
```

Hello, <names>!

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

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

    println("Hello, $names!")
}

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

Hello, <names>! (Faster version)

```
fun main(args : Array<String>) {
    val names = StringBuilder()

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

    println("Hello, $names!")
}

fun StringBuilder.plusAssign(s : String) {
    this.append(s)
}
```

Hello, <names>! (Realistic version)

```
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()
}
```

join() and forit()

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

Null-safety

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

fun main(args : Array<String>) {
    val x = parseInt("123")
    val y = parseInt("Hello")
    print(x?.times(2))          // Can't say: print(x * 2)

    if (x != null) {
        print(x * 2)
    }
}
```

Types

Syntax	
Class types	`List<Foo>`
Nullable types	`Foo?`
Function types	`fun (Int) : String`
Tuple types	`(Double, Double)`
Self type	`This`

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

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?>?`
`IntArray`		**`int`**`[]`		`IntArray?`
`Nothing`		–		–
`Foo`		`Foo`		`Foo?`

Automatic casts and When

```
fun foo(obj : Any?) {
    if (obj is String) {
      obj.substring(2)
    }
    when (obj) {
        is String => obj[0]
        is Int => obj + 1
        !is Boolean => null
        else => ...
    }
}
```


More on when-expressions

```
fun bar(x : Int) {
    when (x) {
        0 => "Zero"
        1, 2, 3 => "1, 2 or 3"
        x + 1 => "Really strange"
        in 10..100 => "In range"
        !in 100..1000 => "Out of range"
    }
}
```

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
  fun foo() : Int = 1 // open by default
  fun bar() : Int     // abstract by default
}

class Foo(p : Bar) : Class1(p), T1, T2 {
  override fun bar() : Int = foo() + 1
}
```

Disambiguation

```
trait A {
   fun foo() : Int = 1
}

open class B() {
   open fun foo() : Int = 2
}

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

First-class Delegation

```
trait List<T> {
   fun add(t : T)
   fun get(index : Int) : T
}
```

```
class ListDecorator<T>(p : List<T>) : List<T> by p {
  override fun add(t : T) {
    log.message("Added $t")
     super.add(t)
  }

   // override fun get(index : Int) : T = super.get()
}
```

First-class functions

- Functions
  - `fun f(p : Int) : String`
- Function types
  - `fun (p : Int) : String`
  - `fun (Int) : String`
- Function literals
  - `{p => p.toString()}`
  - `{(p : Int) => p.toString()}`
  - `{(p : Int) : String => p.toString()}`

Higher-order functions

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

- `filter(list, {s => s.length < 3})`
  - Sugar: last function literal argument
    - `filter(list) {s => s.length < 3}`
  - Sugar: one-parameter function literal
    - `filter(list) { it.length < 3 }`

Infix function calls & "LINQ"

```
a.contains(b)

// is the same as

a contains b
```

```
users
    .filter { it hasPrivilege WRITE }
    .map { it => it.fullName }
    .orderBy { lastName }
```

Lock example (I)

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

Lock example (II)

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

```
fun lock(l : Lock, body : fun () : Unit)
```

Lock example (III)

```
inline fun lock(l : Lock, body : fun () : Unit) {
  l.lock()
  try {
    body()
  }
  finally {
    l.unlock()
  }
}
```

Extension functions

- Functions
  - ```
    fun Foo.f(p : Int) : String
    ```
- Function types
  - ```
    fun Foo.(p : Int) : String
    ```
  - ```
    fun Foo.(Int) : String
    ```
- Function literals
  - ```
    {Foo.(p : Int) => this.toString()}
    ```
  - ```
    {Foo.(p : Int) : String => this.toString()}
    ```


    ```

Builders in Groovy

```
html {
    head {
      title "XML encoding with Groovy"
    }
    body {
      h1 "XML encoding with Groovy"
      p "this format can be used as an alternative markup to XML"

      /* an element with attributes and text content */
      ahref:'http://groovy.codehaus.org' ["Groovy"]
    }
}
```

Builders in Kotlin

```
html {
    head {
      title { +"XML encoding with Kotlin" }
    }
    body {
      h1 { +"XML encoding with Kotlin" }
      p { +"this format is now type-safe" }

      /* an element with attributes and text content */
      a(href="http://jetbrains.com/kotlin") { +"Kotlin" }
    }
}
```

Builders: Implementation (I)

- Function definition

```
fun html(init : fun HTML.() : Unit) : HTML {
  val html = HTML()
  html.init()
  return html
}
```

- Usage

```
html {
  this.head { ... }
}
```

Builders: Implementation (II)

- Function definition

```
fun html(init : fun HTML.() : Unit) : HTML {
  val html = HTML()
  html.init()
  return html
}
```

- Usage

```
html {
  head { ... }
}
```

Builders: Implementation (III)

```
abstract class Tag(val name : String) : Element {
    val children = ArrayList<Element>()
    val attributes = HashMap<String, String>()
}

abstract class TagWithText(name : String) : Tag(name) {
    fun String.plus() {
      children.add(TextElement(this))
    }
}

class HTML() : Tag("html") {
    fun head(init : fun Head.() : Unit) { }
    fun body(init : fun Body.() : Unit) { }
}
```

Builders in Kotlin

```
html {
    head {
      title { +"XML encoding with Kotlin" }
    }
    body {
      h1 { +"XML encoding with Kotlin" }
      p { +"this format is now type-safe" }

      /* an element with attributes and text content */
      a(href="http://jetbrains.com/kotlin") { +"Kotlin" }
    }
}
```

Generics: Invariance

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

val ints = List<Int>()
val anys : List<Any> = ints
anys.add("1") // Cause of the problem
val i : Int = ints.get(0) // !!!
```

Generics: Declaration-site variance

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

```
List<Int> >:< List<Any>
val ints = List<Int>()
val anys : List<Any> = ints
```

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

```
val ints = Producer<Int>()
val anys : Producer<Any> = ints
```

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

```
val anys = Consumer<Any>()
val ints : Consumer<Int> = anys
```

Generics: Use-site variance

```
val ints = List<Int>()

val anysOut : List<out Any> = ints

anysOut.add("1") // Not available
```

```
val i : Int = ints.get() // No problem
```

Generics: Use-site variance

```
val ints = List<Int>()

val anysOut : List<out Any> = ints

anysOut.add("1") // Not available

val i : Int = ints.get() // No problem
```

```
val anys = List<Any>()

val intsIn : List<in Int> = anys

intsIn.add(0)

val obj = intsIn.get() // : Any?
```

Reified generics

- Type information in retained at runtime
  - `foo is List<T>`
  - `Array<T>(3)`
  - `T.create()`
- Java types are still erased
  - `foo is java.util.List<*>`

Class objects (I)

- Classes have no static members
- Each class may have a **class object** associated to it:

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

val e = Example.create()
```

Class objects (II)

- Class objects can have supertypes:

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

val factory : Factory<Example> = Example
val e : Example = factory.create()
```

Class objects (III)

- Generic constraints for class objects:

```
class Lazy<T>()
  where class object T : Factory<T>
{
  private var store : T? = null
  public val value : T
    get() {
      if (store == null) {
        store = T.create()
      }
      return store
    }
}
```

We are hiring

- Full-time
  - ➡ Back-end development
  - ➡ Standard library development
  - ➡ Static analyses and Refactorings
  - ➡ Incremental compilation
- Internships

Resources

- Documentation:
  - ➡ http://jetbrains.com/kotlin
- Blog:
  - ➡ http://blog.jetbrains.com/kotlin
- Twitter:
  - ➡ @project_kotlin
  - ➡ @abreslav

Practical Type Systems (seminar)

- Topics
  - Type systems of industrial languages (e.g. C#, Java, Scala, Kotlin)
  - Cutting-edge work on OOP and Generic programming
  - Formalizing Kotlin
- Location
  - JetBrains, Kantemirovskaya, 2A (m. "Lesnaya")
- Day/Time
  - TBD
- To participate
  - **andrey.breslav@jetbrains.com**