The Kotlin Programming Language

Andrey Breslav
Dmitry Jemerov

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 end of 2011

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

- 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

- Language features
  - Static null-safety guarantees
  - Higher-order functions ("closures")
  - Mixins & First-class delegation
  - Properties (no fields)
  - Reified generics
  - Declaration-site variance & "Type projections"
  - Extension functions
  - Modules and Build infrastructure
  - Inline-functions (zero-overhead closures)
  - Pattern matching
  - ...
- Full-featured IDE by JetBrains from the very beginning

Basic syntax

- IDE demo
  - ➡ functions
  - ➡ variables
  - ➡ operator overriding
  - ➡ extension functions
  - ➡ null-safety
  - ➡ automatic casts
  - ➡ when-expressions

Hello, world!

```
namespace demo1

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

String templates

```
namespace demo2

fun main(args : Array<String>) {
   print("Hello, args' size is ${args.size}!")
}

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

Assign-once locals

```
fun main(args : Array<String>) {
    val text = "Hello, world!"
    print(text)
}

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

... and globals

```
val text = "Hello, world!"

fun main(args : Array<String>) {
    print(text)
}

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

Local functions

```
fun main(args : Array<String>) {
    fun text() = "Hello, world!"
    print(text())
}

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

Mutable variables

```
fun main(args : Array<String>) {
    var v = "Hello"
    v += ", " + "world!"
    print(v)
}

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

Custom operators

```
object Console {
    fun plusAssign(s : String) {
        System.out?.println(s)
    }
}

fun main(args : Array<String>) {
    var v = "Hello"
    v += ", " + "world!"
    Console += v
}
```

Extension functions

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

fun String.print() {
  System.out?.println(this)
}
```

X

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")
    x?.times(2)

    if (x != null) {
        x.times(2)
    }
}
```

X

Automatic casts and When

```
fun foo(obj : Any?) {
    if (obj is String) {
      obj.get(0)
    }
    when (obj) {
        is String => obj.get(0)
        is Int => obj.plus(1)
        !is Boolean => null
    }
}

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"
    }
}
```

X

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?>?
Array<Int>		**int**[]		Array<Int>?
Nothing		–		–
Foo		Foo		Foo?

Classes

```
class Foo(bar : Bar) : Buzz(bar) {
  ...
}
```

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

Multiple inheritance?

- Requirements
  - ➡ Subtyping
  - ➡ Implementation reuse
- Problems
  - ➡ Ambiguities
  - ➡ Obscure initialization logic

- Unrestricted (C++)
- Interface-only (Java, C#)
- Traits (Scala)
- Mixins (Ada, CZ, ...)

Traits/Mixins (Envisioned)

```
trait class Trait1 : Class1
  with OtherTrait {
  // No state
}

class Foo(p : Bar) : Class2(p)
  with Trait1, Trait2 { ... }

class Decorator(p : Class3) : Class3 by p
  with Trait1, Trait2 { ... }
```

Disambiguation

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

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

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

Automatic disambiguation

- If all the inherited implementations come from the same source (trait), there's no need to override?
- Issues
  - Binary compatibility
  - Internal vs API

Generics (I)

```
class Producer<out T> {
  fun produce() : T          Producer<Int> <: Producer<Any>
}

class Consumer<in T> {
  fun consume(t : T)         Consumer<Any> <: Consumer<Int>
}

class Ouroboros<T> {
  fun consume(t : T)         Ouroboros<Int> >:< Ouroboros<Any>
  fun produce() : T
}
```

Generics (II)

`Ouroboros<out Int> <: Ouroboros<out Any>`

- consume not available

`Ouroboros<in Any> <: Ouroboros<in Int>`

- produce on `Ouroboros<in Int>` returns `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
    }
}
```

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

- `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 }`

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

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) {
  myLock.lock()
  try {
    body()
  }
  finally {
    myLock.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() : TagWithText("html") {
  fun head(init : fun Head.() : Unit) { … }
  fun body(init : fun Body.() : Unit) { … }
}
```

Resources

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