The Kotlin Programming Language

Mooly Sagiv

Slides taken from

Elazar Gershoni (TAU)
Martin Odersky (EPFL)
Donna Malayeri (CMU)
Hila Peleg (Technion)

Ivan Turčinović & Igor Buzatović (Inovatrend)

1. Performance
2. Security
3. Productivity

New PL in Industry

PL	Company	Predecessor	Domain	Concepts
Go	Google	C	Systems	Channels Type Inference ML modules GC
Dart	Google	Javascript	Web	Types
Rust		C++	System Browser	Type Inference Ownership Linear Types
Hack	facebook	PHP	Social Network	Gradual Types

Course Topics

- Induction
- Operational Semantics
- Lambda Calculus
- Type Inference
- Memory Management for PL
  - Closure
  - Scope rules
  - GC
- Ocaml
- Javascript

Schedule

- 20/5 Kotlin
- 27/5 Semantics
- 3/6 Functional Programming (ML+Javascript)
  - Algebraic data types
  - Recursion
  - References
  - Higher order functions
- 10/6 Lambda & Typed Lambda Calculus

Outline

- Motivation
- Simple examples
- Pattern matching
- Higher order programming
- Type Inference
- Type Classes

Modern Functional Programming

- Higher order
- Modules
- Pattern matching
- Statically typed with type inference
- Two viable alternatives
  - Haskel
    - Pure lazy evaluation and higher order programming leads to concise programming
    - Support for domain specific languages
    - I/O Monads
    - Type classes
  - ML/Ocaml/F#
    - Eager call by value evaluation
    - Encapsulated side-effects via references
    - [Object orientation]

ML Dialects

- Standard ML of New Jersey
- Mlton
- CakeML
- F#
- OCaml

Then Why aren't FP adapted?

- Education
- Lack of OO support
  - Subtyping increases the complexity of type inference
- Programmers seeks control on the exact implementation
- Imperative programming is natural in certain situations

The Java Programming Language

- Designed by Sun 1991-95
- Statically typed and type safe
- Clean and powerful libraries
- Clean references and arrays
- Object Oriented with single inheritance
- Interfaces with multiple inheritance
- Portable with JVM
- Effective JIT compilers
- Support for concurrency
- Useful for Internet

Java Critique

- Downcasting reduces the effectiveness of static type checking
  - Many of the interesting errors caught at runtime
    - Still better than C, C++
- Huge code blowouts
  - Hard to define domain specific knowledge
  - A lot of boilerplate code
  - Sometimes OO stands in our way
  - Generics only partially helps
  - Array subtype does not work

Varieties of Polymorphism

- Parametric polymorphism A single piece of code is typed generically
  - Imperative or first-class polymorphism
  - ML-style or let-polymorphism
- Ad-hoc polymorphism The same expression exhibit different behaviors when viewed in different types
  - Overloading
  - Multi-method dispatch
  - Intentional polymorphism
- Subtype polymorphism A single term may have many types using the rule of subsumption allowing to selectively forget information

Parametric Polymorphism

```
let rec map f arg = match arg with
      [] -> []
    | hd :: tl -> f hd :: (map f tl)

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
let inc ls = map (fun x -> x + 1) ls

val inc: int list -> int list
```

Ad-hock Polymorphism

- x * y in ML
- x + y in Javascript
- f(arg) in C++
- f(arg) in Scala

Subtyping

- If S is a subtype of T then operations performed on T can also be performed on S
- Simple example: duck <: bird, ostrich <:bird
  is_equal(b1, b2: bird)
- Another example

```
function max(x: Number, y: Number) =
    if x < y then return y
      else return x
```

  - *max(5, 7)*
  - *max(6.5, 4.3)*

Array Subtyping

```
class Array[A] {
   void set(int index, A val) {
      this[index] = val;
   }
}
```

- Array[String] is not a subtype of Array[Object]
- If it were, we could do this:

```
Array[String] x = new Array[String](1);
Array[Object] y = (Array[Object]) x;
y.set(0, new FooBar());
// just stored a FooBar in a String array!
```

Covariance vs. Contravariance

- Enforcing type safety in the presence of subtyping
- If a function expects a formal argument of type T1 → T2 and the actual argument has a type S1 → S2 then
  - What do have to require?
- If a function assumes a precondition T1 and ensures a postcondition T2
  - If the caller satisfies a precondition S1 and requires that S2 holds after the call
  - What do we have to require?

Java 8 strikes back

- Lambdas
- Null safety
- Default methods
- Streams API
- …

Scala

- Scala is an object-oriented and functional language which is completely interoperable with Java (.NET)
- Removes some of the more arcane constructs of these environments and adds instead:

  (1) a uniform object model,

  (2) pattern matching and higher-order functions,

  (3) novel ways to abstract and compose programs

Why Scala?

(Coming from OCaml)

- Runs on the JVM/.NET
  - Can use any Java code in Scala
- Combines functional and imperative programming in a smooth way
- Effective libraries
- Inheritance
- General modularity mechanisms

Kotlin

- A lightweight object-oriented and functional language
- Design principles: Industrial use, tooling, safety
  1. Compiles to JVM byte code and JavaScript
  2. Runtime size 800K
  3. Adapted by Google for Android

Kotlin is now Google’s preferred language for Android app development

https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

More elegant?

Basic syntax and rules

```
package javacro

import addressbook.Person

fun multiply(x: Int, y: Int): Int {
    return x * y
}

fun main(args: Array<String>): Unit {
    var person = Person("John", "Doe")
    person = Person("Mary", "Joe")

    val firstname: String = person.firstname
    println("Hello $firstname")

    val result = multiply(2, 5)
}
```

- Functions - definition in package or in class
- Immutable/mutable variables
- No „new” keyword
- Type inference
- No checked exceptions
- No primitive types
- No static members

More elegant?

Basic syntax and rules cont'd

```
class Person(
        val firstname: String,
        val lastname: String,
        var addresses: List<PostalAddress>,
        var phoneNums: List<PhoneNumber>
) {
    val hasPhoneNums: Boolean
         get() = phoneNums.isEmpty()
}
```

- Primary constructors
- No fields, just properties
- Bean style classes easy to declare
- By default, all classes are final

More elegant?

Null safety

- Null reference – Billion dollar mistake
- Kotlin is designed in a way that aims to eliminate NPE from our code

```
var notNullable : String = "I CANNOT be null"
notNullable = null // compilation error

var nullable : String? = "I CAN be null"
nullable = null // ok

val length1 = notNullable.length()
val length2 = nullable.length() // compilation error
val length3 = nullable!!.length() // for those NPE lovers

bob?.department?.head?.name
```

Default arguments, named arguments

```
//default arguments
fun increase(a: Int, b: Int = 1): Int {
    return a + b;
}

increase(3) // returns 4
increase (3, 2) // returns 5

// pass argument by name
fun format (someTxt : String,
            suffix : String = "END",
            capitalize : Boolean = false) {
    // do something
}

format ("Kotlin rulz", suffix="FINISH")
format ("Kotlin rulz", capitalize = true)
```

- Default argument values can be defined
- Arguments with default values are optional
- No more need for function overloading (almost)
- Kotlin classes can have only one constructor
- Arguments can be called by name
- When passing arguments by name ordering doesn't matter

Ranges

```
// equivalent of 1 <= i && i <= 10
if (i in 1..10) {
    println(i)
}

// equivalent of 1.0 <= x || x >= 3.0
if (x !in 1.0..3.0) println(x)


for (i in 1..4) print(i)
// prints "1234"

for (x in 1.0..2.0) print("$x ")
// prints "1.0 2.0 "

for (i in 4 downTo 1) print(i)
// prints "4321"

for (i in 1..4 step 2) print(i)
// prints "13"
```

- Simpler “is it in range” check
- Can be used for any type that implements Comparable
- “both ends“ included
- “..” operator is translated to “rangeTo” function
- “rangeTo” function is implemented as extension function on Comparable
- Numerical ranges can be iterated over
- In both directions and in arbitrary steps

Generics

Java

```
public void doSomething(List<String> strs) {
    List<Object> items = strs;
}
```

```
public void doSomething(List<String> strs) {
    List<? extends Object> items = strs;
}
```

Kotlin

```
fun doSomething(strs:List<String>) {
    var items : List<Any> = strs;
}
```

```
public trait List<out E> : Collection<E> {

fun copy(from : Array<out Any>,
         to : Array<Any>) {
```

Java survey: 39% of wildcards usage can be replaced by declaration-site variance

When Object Oriented Stands in Your Way

```
abstract class Exp {
    int eval();
}

class Const extends Exp {
    int val;
    int eval() { return val; }
}

class PlusExp extends Exp {
    Exp left, right;
    int eval() { return left.eval() + right.eval(); }
}
```

- Adding a new type of expression is easy
- Adding a new feature is hard

Smart Type Casts

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) {
    print("Not a String")
}

when (x) {
    is Int -> print(x)
    is List<*> -> print(x.size())
    !is Number -> print("Not even a number")
    else -> print("can't do anything")
}
```

- Java's instaceof not very practical
- No !instanceof
- Meet Kotlin's "is"
- "is" negation "!is"
- Automatic type cast when "is" evaluates true inside if / when blocks

Expression Example

```
sealed class Exp {
    abstract fun eval() : Int
}

class ConstExp(val v: Int) : Exp() {
    override fun eval() : Int = v
}

class PlusExp(val left: Exp, val right: Exp) : Exp() {
    override fun eval() : Int = left.eval() + right.eval()
}

fun myToString(e: Exp) : String = when (e) {
    is ConstExp -> e.toString()
    is PlusExp -> myToString(e.left) + myToString(e.right)
}
```

Data classes

```
class User ( val firstName : String,
             val lastName : String )


fun main(args:Array<String>) {

    var user1 = User("John", "Doe");
    var user2 = User("John", "Doe");

    println (user1);
    println( user1.hashCode() );
    println( user2.hashCode() );
    println( user1.equals(user2) );
}
```

```
javacro.User@41a7d9e7
1101519335
1926426205
false
```

Data classes

```
data class User ( val firstName : String,
                  val lastName : String )
```

```
fun main(args:Array<String>) {

    var user1 = User("John", "Doe");
    var user2 = User("John", "Doe");

    println (user1);
    println( user1.hashCode() );
    println( user2.hashCode() );
    println( user1.equals(user2) );
}
```

User(firstName=John,lastName=Doe)

71819599

71819599

true

Operator overloading

Expression	Translated to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)

- Operators are translated to corresponding functions
- Operator set is fixed
- Operator overloading is achieved by overriding corresponding functions of specific class
- By adding extension functions we can enable certain operator for classes that don't support it originally
- On our own classes we can enable certain operator by implementing corresponding functions

Extension Functions

```
val a = BigDecimal(5)
val b = BigDecimal(3)
val c = a + b
```

```
fun BigDecimal.plus(that: BigDecimal) =
    BigDecimal(this.doubleValue() + that.doubleValue())
```

More cool stuff

Type-safe Groovy-style builders

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    body {
        h1 {+"XML encoding with Kotlin"}
        p {+"this format can be used instead of HTML"}

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

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

- For generating tree-like structures (UI layouts, 3D scene graphs)
  - Tree-like syntax – better insight in what is being generated
  - Functions create, initialize and return builder specific objects
- Better then xml: no scheme required, type safe, interoperable with other code

Type-safe Groovy-style builders

```
html {
    this.head { /* ... */ }
    this.body { /* ... */ }
}

html {
    head { /* ... */ }
    body { /* ... */ }
}
```

```
fun head(init : Head.() -> Unit) : Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}
```

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

- Objects initialized using function literal argument
- Extension function literal allows referencing to object enclosing element
- Kara web framework using it for HTML layouts, CSS
- Android apps, XML layouts

Higher Order Functions

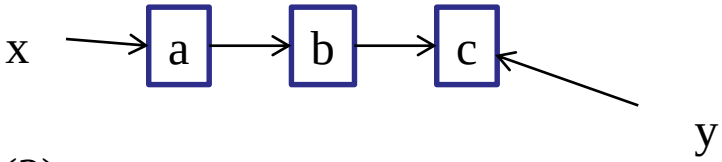
```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

Application

```
val items = listOf(1, 2, 3, 4, 5)


items.fold(0, { acc: Int, i: Int -> acc + I })


items.fold(0, { acc: Int, i: Int ->
    print("acc = $acc, i = $i, ")
    val result = acc + i
    println("result = $result")
    result
})
```

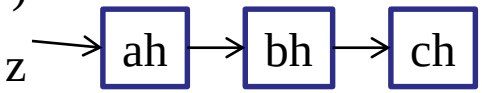
Mutable vs. Immutable Data Structures

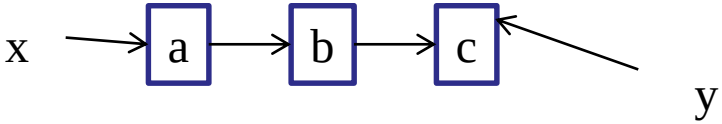
- Basic data structures in Kotlin are immutable
- Operations will copy (if they must)



y = x.drop(2)

z = x.map(_ + "h")



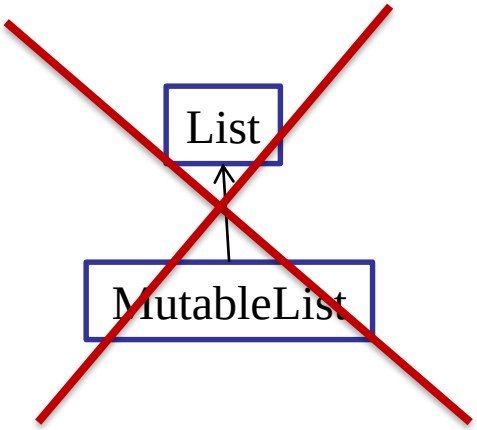


- Many positive consequences

Mutable vs. Immutable

- Mutable and immutable collections are not the same type hierarchy!
- Have to copy the collection to change back and forth, can't cast

x.toList



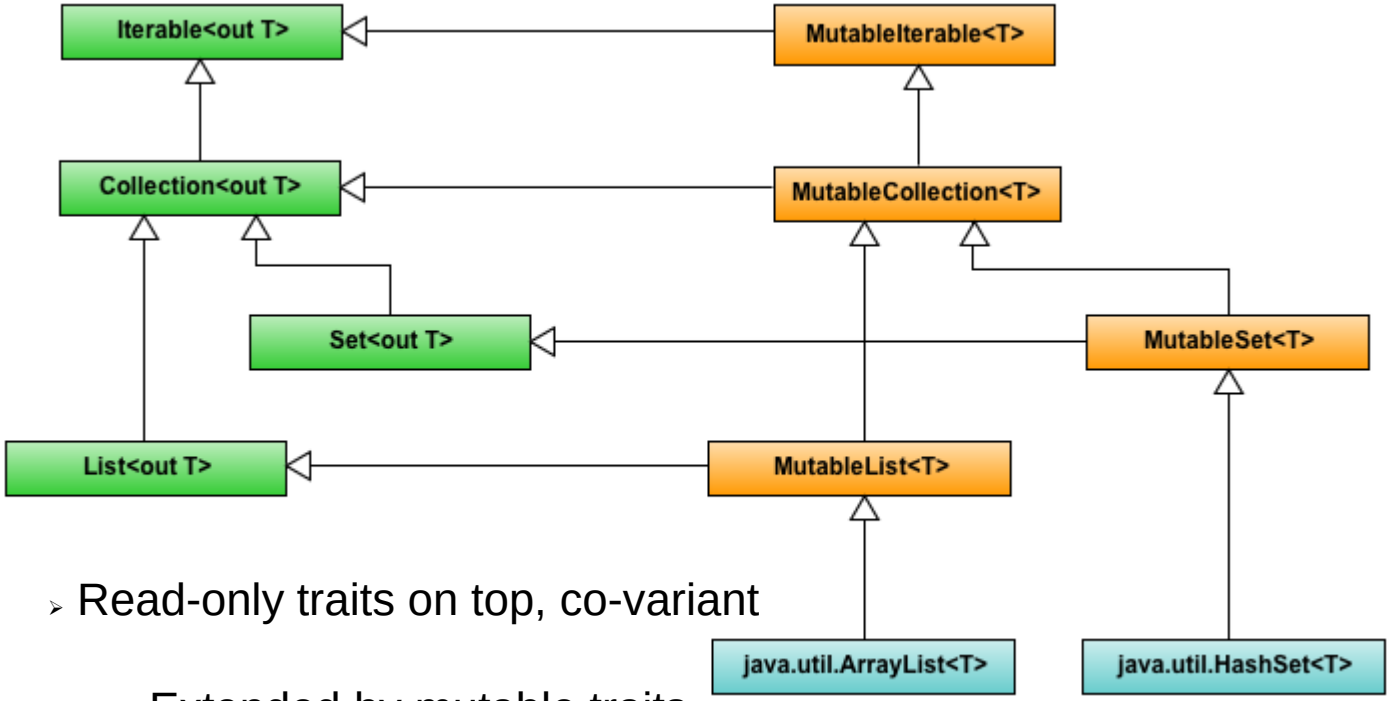
Dependent Multiple Inheritance (C++)

```
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Kotlin object system

- Class-based
- Single inheritance
- Interfaces follow Java 8
  - Can contain declarations of abstract methods and as well as method implementations
  - Cannot store state

Kotlin collections



- Read-only traits on top, co-variant
  - Extended by mutable traits
- Implemented by JDK classes (compile time modifications)
  - Read-only traits do not guarantee immutability
- Top level package functions for instantiating collections:
  - Read-only: setOf, listOf, mapOf

Interfaces

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

Inheritance

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String
    override val name: String
      get() = "$firstName $lastName"
}
data class Employee(
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Multiple Inheritance

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() {  print("bar") }
}
  class D : A, B {
      override fun foo() {
          super<A>.foo() ;  super<B>.foo()    }
      override fun bar() {
          super<B>.bar()    }
  }
```

Delegated Properties

- Reuse code for certain tasks
- Lazy properties
  - the value gets computed only upon first access
- observable properties
  - listeners get notified about changes to this property

Delegated Properties

```
class Example {
    var p: String by Delegate()
}

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }
    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

Interface 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(10)
    Derived(b).print()
}
```

Overriding Delegation

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}
```

```
class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}
```

```
class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}
    fun main() {
        val b = BaseImpl(10)
        Derived(b).printMessage()
        Derived(b).printMessageLine()
    }
```

Summary

- An integration of OO and FP
  - Also available in Javascript/Ruby but with dynamic typing
- Static typing
- Concise
- Efficient
- Support for concurrency
- Already adapted
- But requires extensive knowledge

Languages

- Ocaml
- Javascript

Concepts &Techniques

- Syntax
  - Context free grammar
  - Ambiguous grammars
  - Syntax vs. semantics
- Static semantics
  - Scope rules
- Semantics
  - Small vs. big step
- Runtime management

- Functional programming
  - Lambda calculus
  - Recursion
  - Higher order programming
  - Lazy vs. Eager evaluation
  - Pattern matching
  - Closure
- Types
  - Type safety
  - Static vs. dynamic
  - Type checking vs. type inference
  - Most general type
  - Polymorphism
  - Type inference algorithm