Idiomatic Kotlin

Andrey Akinshin, JetBrains

Idiomatic

> Idiomatic — using, containing, or denoting expressions that are natural to a native speaker.
>
> — Oxford Dictionary

Agenda

- Expressions
- Loops
- Functions
- StdLib
- References

Expressions

Expression body

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

Expression body

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

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

when as an expression

```
fun parse(number: String): Int? {
    when (number) {
        "one" -> return 1
        "two" -> return 2
        else  -> return null
    }
}
```

when as an expression

```
fun parse(number: String): Int? {
    when (number) {
        "one" -> return 1
        "two" -> return 2
        else  -> return null
    }
}
```

```
fun parse(number: String) =
    when (number) {
        "one" -> 1
        "two" -> 2
        else  -> null
    }
```

when and sealed classes

```
abstract class Result

class Success
    : Result()
class Failure(val message: String)
    : Result()

fun handleResult(result: Result) =
  when (result) {
    is Success ->
      "OK!"
    is Failure ->
      "Failed: ${result.message}"
    else ->
      throw IllegalArgumentException()
  }
```

when and sealed classes

```
abstract class Result

class Success
    : Result()
class Failure(val message: String)
    : Result()

fun handleResult(result: Result) =
  when (result) {
    is Success ->
      "OK!"
    is Failure ->
      "Failed: ${result.message}"
    else ->
      throw IllegalArgumentException()
  }
```

```
sealed class Result

class Success
    : Result()
class Failure(val message: String)
    : Result()

fun handleResult(result: Result) =
  when (result) {
    is Success ->
      "OK!"
    is Failure ->
      "Failed: ${result.message}"
  }
```

try as an expression

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

try as an expression

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

```
fun tryParseInt(number: String) =
  try {
    Integer.parseInt(number)
  } catch (e: NumberFormatException) {
    null
  }
```

Elvis

```
fun foo(name: String?) {
  val s = if (name != null)
    name
  else
    "?"
}
```

Elvis

```
fun foo(name: String?) {
  val s = if (name != null)
    name
  else
    "?"
}
```

```
fun foo(name: String?) {
  val s = name ?: "?"
}
```

Elvis before return

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val age = p.age
  if (age == null) return
  // ...
}
```

Elvis before return

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val age = p.age
  if (age == null) return
  // ...
}
```

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val age : Int = p.age ?: return
  // ...
}
```

Elvis before throw

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val name = p.name
  if (name == null)
    throw IllegalArgumentException(
      "Name required")
  // ...
}
```

Elvis before throw

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val name = p.name
  if (name == null)
    throw IllegalArgumentException(
      "Name required")
  // ...
}
```

```
class Person(
  val name: String?,
  val age: Int?
)

fun processPerson(p: Person) {
  val name = p.name ?:
    throw IllegalArgumentException(
      "Name required")
  // ...
}
```

Loops

Ranges

```
fun isLatinUppercase(c: Char) =
    c >= 'A' && c <= 'Z'
```

Ranges

```
fun isLatinUppercase(c: Char) =
    c >= 'A' && c <= 'Z'
```

```
fun isLatinUppercase(c: Char) =
    c in 'A'..'Z'
```

until in loops

```
fun main(args: Array<String>) {
  for (i in 0..args.size-1) {
    println("$i: ${args[i]}")
  }
}
```

until in loops

```
fun main(args: Array<String>) {
  for (i in 0..args.size-1) {
    println("$i: ${args[i]}")
  }
}
```

```
fun main(args: Array<String>) {
  for (i in 0 until args.size) {
    println("$i: ${args[i]}")
  }
}
```

until in loops

```
fun main(args: Array<String>) {
  for (i in 0..args.size-1) {
    println("$i: ${args[i]}")
  }
}
```

```
fun main(args: Array<String>) {
  for (i in 0 until args.size) {
    println("$i: ${args[i]}")
  }
}
```

```
public infix fun Int.until(to: Int): IntRange {
  if (to <= Int.MIN_VALUE) return IntRange.EMPTY
  return this .. (to - 1).toInt()
}
```

withIndex in loops

```
fun main(args: Array<String>) {
  for (i in 0 until args.size) {
    println("$i: ${args[i]}")
  }
}
```

withIndex in loops

```
fun main(args: Array<String>) {
  for (i in 0 until args.size) {
    println("$i: ${args[i]}")
  }
}
```

```
fun main(args: Array<String>) {
  for ((i, arg) in args.withIndex()) {
    println("$i: $arg")
  }
}
```

deconstructors in loops

```
fun printMap(map: Map<String, String>) {
  for (item in map.entries) {
    println("${item.key}=${item.value}")
  }
}
```

deconstructors in loops

```
fun printMap(map: Map<String, String>) {
  for (item in map.entries) {
    println("${item.key}=${item.value}")
  }
}
```

```
fun printMap(map: Map<String, String>) {
  for ((key, value) in map) {
    println("$key=$value")
  }
}
```

Functions

Top level functions

```
class StringUtils {
  companion object {
     fun containsZeros(s: String) =
        s.contains('0')
  }
}
```

Top level functions

```
class StringUtils {
  companion object {
     fun containsZeros(s: String) =
        s.contains('0')
  }
}
```

```
fun containsZeros(s: String) =
  s.contains('0')
```

Extension functions

```
fun containsZeros(s: String) =
  s.contains('0')

// Usage
val x = containsZeros("0123")
```

Extension functions

```
fun containsZeros(s: String) =
  s.contains('0')

// Usage
val x = containsZeros("0123")
```

```
fun String.containsZeros() =
  contains('0')

// Usage
val x = "0123".containsZeros()
```

Lambda expressions

```
fun evenCount(list: List<Int>) =
  list.count({ x -> x % 2 == 0 })
```

Lambda expressions

```
fun evenCount(list: List<Int>) =
  list.count({ x -> x % 2 == 0 })
```

```
fun evenCount(list: List<Int>) =
  list.count { it % 2 == 0 }
```

Safe let call

```
class Person(
  val name: String,
  val address: String?
)

fun printAddress(person: Person) {
  if (person.address != null) {
    println(person.address)
  }
}
```

Safe let call

```
class Person(
  val name: String,
  val address: String?
)

fun printAddress(person: Person) {
  if (person.address != null) {
    println(person.address)
  }
}
```

```
class Person(
  val name: String,
  val address: String?
)

fun printAddress(person: Person) {
  person.address?.let {
    println(it)
  }
}
```

Safe let call

```
class Person(
  val name: String,
  val address: String?
)

fun printAddress(person: Person) {
  if (person.address != null) {
    println(person.address)
  }
}
```

```
class Person(
  val name: String,
  val address: String?
)

fun printAddress(person: Person) {
  person.address?.let {
    println(it)
  }
}
```

```
public inline fun <T, R> T.let(block: (T) -> R): R {
  return block(this)
}
```

Initialization via apply

```
class Label {
  val text: String = ""
  val tooltip: String = ""
}

fun createLabel(): Label {
  val label = Label()
  label.text = "Click here"
  label.tooltip = "Help"
  return label
}
```

Initialization via apply

```
class Label {
  val text: String = ""
  val tooltip: String = ""
}

fun createLabel(): Label {
  val label = Label()
  label.text = "Click here"
  label.tooltip = "Help"
  return label
}
```

```
class Label {
  val text: String = ""
  val tooltip: String = ""
}

fun createLabel() =
  Label().apply {
    text = "Click here"
    tooltip = "Help"
  }
```

Initialization via apply

```
class Label {
  val text: String = ""
  val tooltip: String = ""
}

fun createLabel(): Label {
  val label = Label()
  label.text = "Click here"
  label.tooltip = "Help"
  return label
}
```

```
class Label {
  val text: String = ""
  val tooltip: String = ""
}

fun createLabel() =
  Label().apply {
    text = "Click here"
    tooltip = "Help"
  }
```

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
  block()
  return this
}
```

Overloads

```
class Phonebook {
  fun print() {
    print(",")
  }

  fun print(separator: String) {
  }
}

fun main(args: Array<String>) {
  Phonebook().print("|")
}
```

Overloads

```
class Phonebook {
  fun print() {
    print(",")
  }

  fun print(separator: String) {
  }
}

fun main(args: Array<String>) {
  Phonebook().print("|")
}
```

```
class Phonebook {
  fun print(separator: String = ",") {
  }
}

fun main(args: Array<String>) {
  Phonebook().print(separator = "|")
}
```

Return multiple values

```
fun namedNum(): Pair<Int, String> =
  1 to "one"

fun main(args: Array<String>) {
  val pair = namedNum()
  val number = pair.first
  val name = pair.second
}
```

Return multiple values

```
fun namedNum(): Pair<Int, String> =
  1 to "one"

fun main(args: Array<String>) {
  val pair = namedNum()
  val number = pair.first
  val name = pair.second
}
```

```
data class NamedNumber(
  val number: Int,
  val name: String
)

fun namedNum() = NamedNumber(1, "one")

fun main(args: Array<String>) {
  val (number, name) = namedNum()
}
```

Multideclarations and lists

```
data class FileInfo(
  val name: String,
  val ext: String?
)

fun getInfo(fileName: String): FileInfo
  if ('.' in fileName) {
    val parts : List<String> =
      fileName.split('.', limit = 2)
    return FileInfo(parts[0], parts[1])
  }
  return FileInfo(fileName, null)
}
```

Multideclarations and lists

```
data class FileInfo(
  val name: String,
  val ext: String?
)

fun getInfo(fileName: String): FileInfo
  if ('.' in fileName) {
    val parts : List<String> =
      fileName.split('.', limit = 2)
    return FileInfo(parts[0], parts[1])
  }
  return FileInfo(fileName, null)
}
```

```
data class FileInfo(
  val name: String,
  val ext: String?
)

fun getInfo(fileName: String): FileInfo
  if ('.' in fileName) {
    val (name, ext) =
      fileName.split('.', limit = 2)
    return FileInfo(name, ext)
  }
  return FileInfo(fileName, null)
}
```

lateinit

```
class MyTest {
  class State(val data: String)

  var state: State? = null

  @Before
  fun setup() {
    state = State("abc")
  }

  @Test
  fun foo() {
    Assert.assertEquals(
        "abc",
        state!!.data
    )
  }
}
```

lateinit

```
class MyTest {
  class State(val data: String)

  var state: State? = null

  @Before
  fun setup() {
    state = State("abc")
  }

  @Test
  fun foo() {
    Assert.assertEquals(
        "abc",
        state!!.data
    )
  }
}
```

```
class MyTest {
  class State(val data: String)

  lateinit var state: State

  @Before
  fun setup() {
      state = State("abc")
  }

  @Test
  fun foo() {
      Assert.assertEquals(
          "abc",
          state.data
      )
  }
}
```

StdLib

require

```
fun calculate(n: Int) {
  if (n <= 0)
    throw IllegalArgumentException(
      "n should be positive")
  // ...
}
```

require

```java
fun calculate(n: Int) {
  if (n <= 0)
    throw IllegalArgumentException(
      "n should be positive")
  // ...
}
```

```java
fun calculate(n: Int) {
  require(n > 0) {
    "n should be positive"
  }
  // ...
}
```

filterIsInstance

```
fun getStrings(objs: List<Any>) =
  objs.filter { it is String }

val x: List<Any> = getStrings(objs)
```

filterIsInstance

```
fun getStrings(objs: List<Any>) =
  objs.filter { it is String }

val x: List<Any> = getStrings(objs)
```

```
fun getStrings(objs: List<Any>) =
  objs.filterIsInstance<String>()

val x: List<Strings> = getStrings(objs)
```

mapNotNull

```
data class Result(
  val value: Any?,
  val error: String?
)

fun listErrors(results: List<Result>)
    : List<String> =
  results
    .map { it.error }
    .filterNotNull()
```

mapNotNull

```
data class Result(
  val value: Any?,
  val error: String?
)

fun listErrors(results: List<Result>)
    : List<String> =
  results
    .map { it.error }
    .filterNotNull()
```

```
data class Result(
  val value: Any?,
  val error: String?
)

fun listErrors(results: List<Result>)
    : List<String> =
  results.mapNotNull { it.errorMessage }
```

compareBy

```
class Person(
  val name: String,
  val age: Int
)

fun sortPersons(persons: List<Person>) =
  persons.sortedWith(
    Comparator<Person> { p1, p2 ->
      val rc = p1.name
                 .compareTo(p2.name)
      if (rc != 0)
        rc
      else
        p1.age - p2.age
    })
```

compareBy

```
class Person(
  val name: String,
  val age: Int
)

fun sortPersons(persons: List<Person>) =
  persons.sortedWith(
    Comparator<Person> { p1, p2 ->
      val rc = p1.name
                 .compareTo(p2.name)
      if (rc != 0)
        rc
      else
        p1.age - p2.age
    })
```

```
class Person(
  val name: String,
  val age: Int
)

fun sortPersons(persons: List<Person>) =
    persons.sortedWith(
      compareBy(Person::name,
                Person::age))
```

groupBy

```
class Request(
  val url: String,
  val remoteIP: String,
  val timestamp: Long
)

fun analyzeLog(log: List<Request>) {
  val map = mutableMapOf<
    String, MutableList<Request>>()
  for (request in log) {
      map.getOrPut(request.url) {
        mutableListOf()
      }.add(request)
  }
}
```

groupBy

```
class Request(
  val url: String,
  val remoteIP: String,
  val timestamp: Long
)

fun analyzeLog(log: List<Request>) {
  val map = mutableMapOf<
    String, MutableList<Request>>()
  for (request in log) {
      map.getOrPut(request.url) {
        mutableListOf()
      }.add(request)
  }
}
```

```
class Request(
  val url: String,
  val remoteIP: String,
  val timestamp: Long
)

fun analyzeLog(log: List<Request>) {
  val map = log.groupBy(Request::url)
}
```

coerceIn

```
fun updateProgress(value: Int) {
  val actualValue = when {
    value < 0   -> 0
    value > 100 -> 100
    else        -> value
  }
}
```

coerceIn

```
fun updateProgress(value: Int) {
  val actualValue = when {
    value < 0   -> 0
    value > 100 -> 100
    else        -> value
  }
}
```

```
fun updateProgress(value: Int) {
  val actualValue =
    value.coerceIn(0, 100)
}
```

zip

```
fun calcDurations(start: LongArray,
                  finish: LongArray) =
    start.mapIndexed { index, s ->
        finish[index] - s
    }
```

zip

```
fun calcDurations(start: LongArray,
                  finish: LongArray) =
    start.mapIndexed { index, s ->
        finish[index] - s
    }
```

```
fun calcDurations(start: LongArray,
                  finish: LongArray) =
    (start zip finish).map {
      (s, f) -> f - s
    }
```

References

Kotling Reference

https://kotlinlang.org/docs/reference/

Kotlin LEARN COMMUNITY TRY ONLINE

Reference | Tutorials | Books | More resources

- Overview
- Getting Started
- Basics
- Classes and Objects
- Functions and Lambdas
- Multiplatform Programming
- Other
- Core Libraries
- Reference
- Java Interop

Learn Kotlin

Edit Page

All Materials | Getting Started | Migrating from Java

Documentation

Kotlin documentation is a good place to start, check out these links to get your feet wet:

— Basics
— Idioms
— Interop with Java

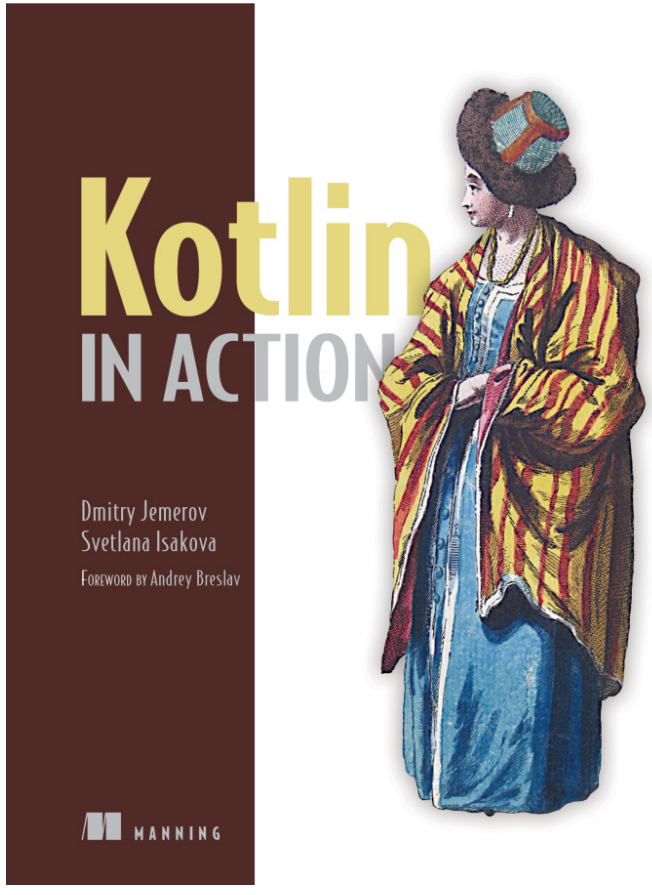
IDE

Many modern IDEs support Kotlin and help in writing idiomatic Kotlin code:

— Kotlin Educational Plugin
— Java2Kotlin converter

Kotlin in action

https://www.manning.com/books/kotlin-in-action



Ideomatic Kotlin Repo

https://github.com/yole/idiomatic-kotlin/

yole / idiomatic-kotlin

Watch 10 | Star 67 | Fork 13

Code | Issues 2 | Pull requests 0 | Projects 0 | Insights

Examples for "Idiomatic Kotlin" talk

14 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Find File | Clone or download

yole Consistent name — Latest commit 1e70c8b on 4 Apr 2018

gradle/wrapper	Group examples by category; add some new ones	a year ago
src/main/kotlin	Consistent name	a year ago
.gitignore	Add .idea to gitignore	a year ago
build.gradle	Add DSL examples	a year ago
gradlew	Initial commit	2 years ago
gradlew.bat	Initial commit	2 years ago
settings.gradle	Initial commit	2 years ago

Questions?

Andrey Akinshin

https://aakinshin.net

https://github.com/AndreyAkinshin

https://twitter.com/andrey_akinshin

andrey.akinshin@gmail.com