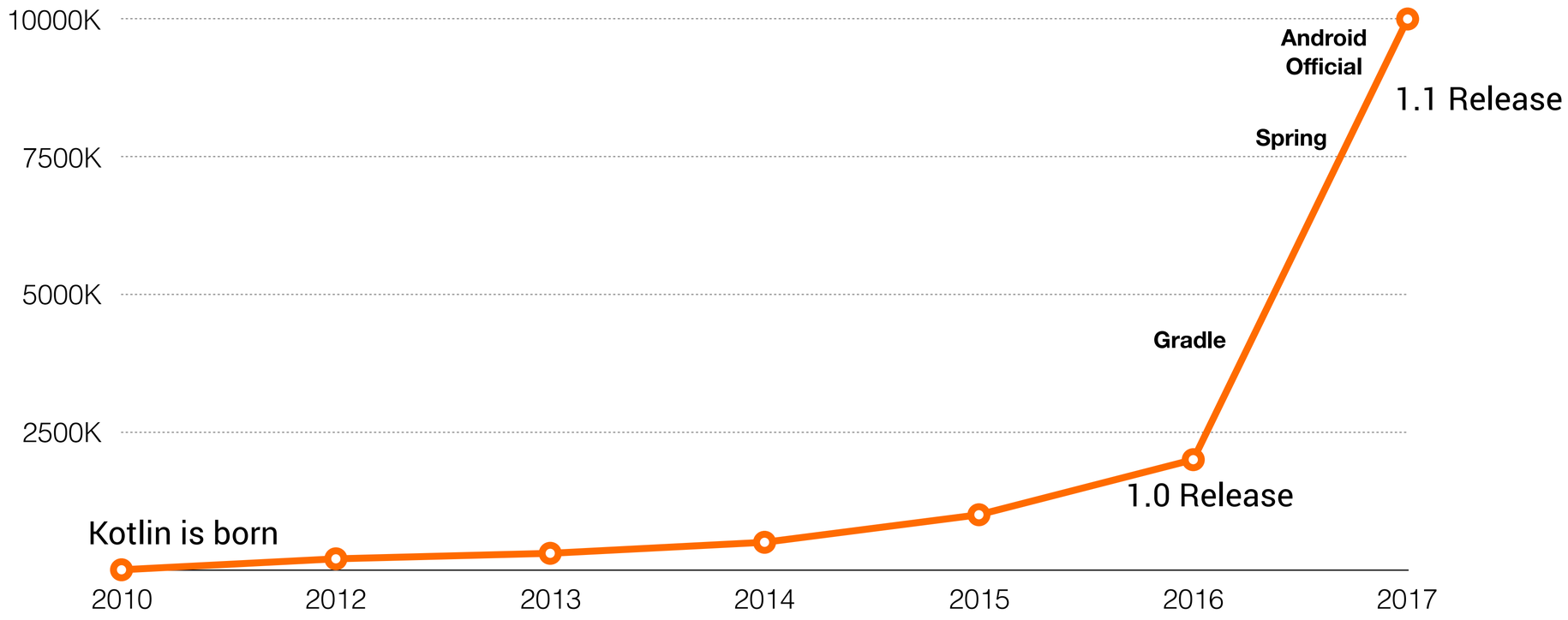
Introduction and Advanced Workshop

Introduction to Kotlin

The Why

- Started in 2010 by JetBrains
- A language that was
  - Concise, Expressive, Toolable, Interoperable
  - Pragmatic

Timeline


10000K
7500K
5000K
2500K
Kotlin is born
1.0 Release
Gradle
Spring
Android
Official
1.1 Release
2010
2012
2013
2014
2015
2016
2017

What is Kotlin

- Statically typed language
- Inspired by Java, Scala, C#, Groovy amongst others
- Targets
  - JVM / Android
  - JavaScript
  - Native

Type of Applications

- Industrial Applications
- Mobile
- Client-Side
  - Web
  - Desktop
- Server-Side

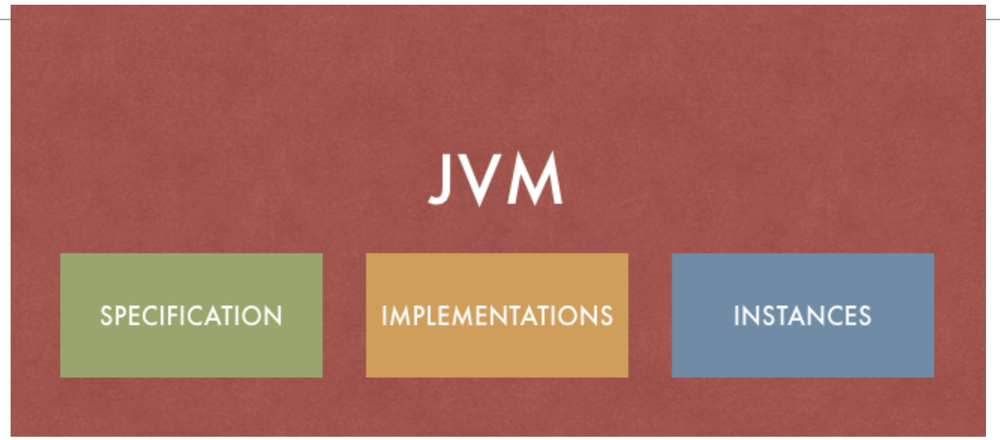
Benefits while adopting

- Similar to other languages
  - Ramp-up time is low
- Interoperable with the platform
  - Java - JVM
  - JavaScript - Packages and Libraries
  - Native - C Interop

Java or JVM

The Java Virtual Machine

- Abstract machine running Java applications
- Single specification, multiple implementations (OpenJDK most common)
- An instance runs an application
- Applications compiled to bytecode
- JRE vs JDK



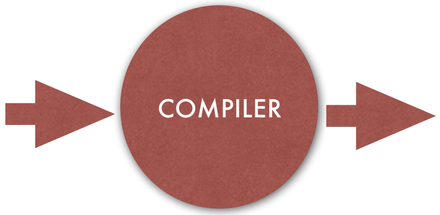
Process

Customer.kt

```
class Customer(val name: String) {

   fun validate(): Boolean {
        . . .
   }

}
```



Customer.class

```
L0
    LINENUMBER 5 L0
    ALOAD 0
    GETFIELD Customer.name . . .
    ARETURN
L1
    LOCALVARIABLE this LCustomer;
    MAXSTACK = 1
```

EXECUTE USING THE JVM

java myApp

Conventions

- Follows Java Coding Conventions
- Lower camelCase for names
- Types in Uppercase
- Methods and Properties in lower camelCase
- Semicolons optional*
- Packages follows reverse notation
  - Multiple classes per file allowed
  - Package does not have to match folder names

Tooling

- JDK
  - Version 6, 7 or 8
- Kotlin Compiler
- Editor or IDE

Build Tools

- Maven
- Gradle
- Kobalt

Introduction

Hands-on

Operator Conventions

Expression	Translates to
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()
a++	a.inc()
a–	a.dec()

Expression	Translates 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)
a in b	b.contains(a)
a !in b	!b.contains(a)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timeAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

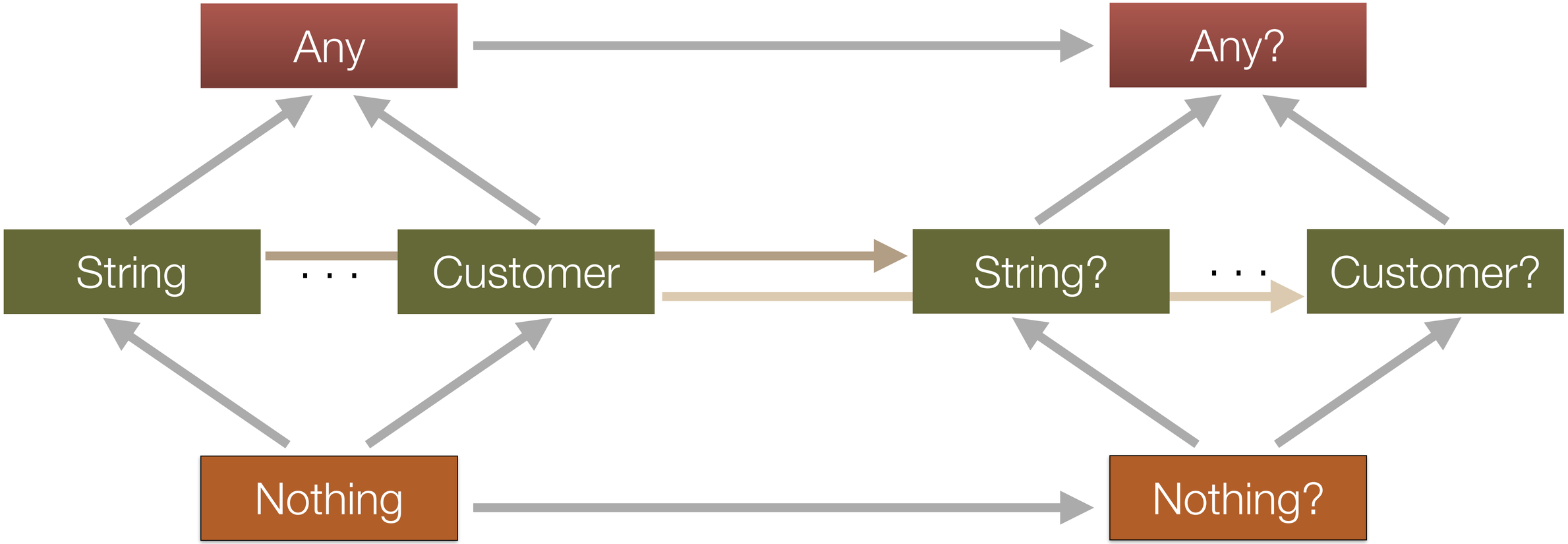
Expression	Translates to
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

Advanced Kotlin

Advanced

Hands-On

Kotlin Types



Types and Subtypes

- Each Kotlin Class can hold two types, i.e. String and String?

- Each Generic Kotlin Class can hold infinite number of types, i.e. List<Int>

Invariance

Covariance

Producers

Contravariance

Consumers

Use Site and Declaration Site

- Kotlin use Declaration site variance with *out* and *in* modifiers

- Use Site Declaration is supported (type projection)

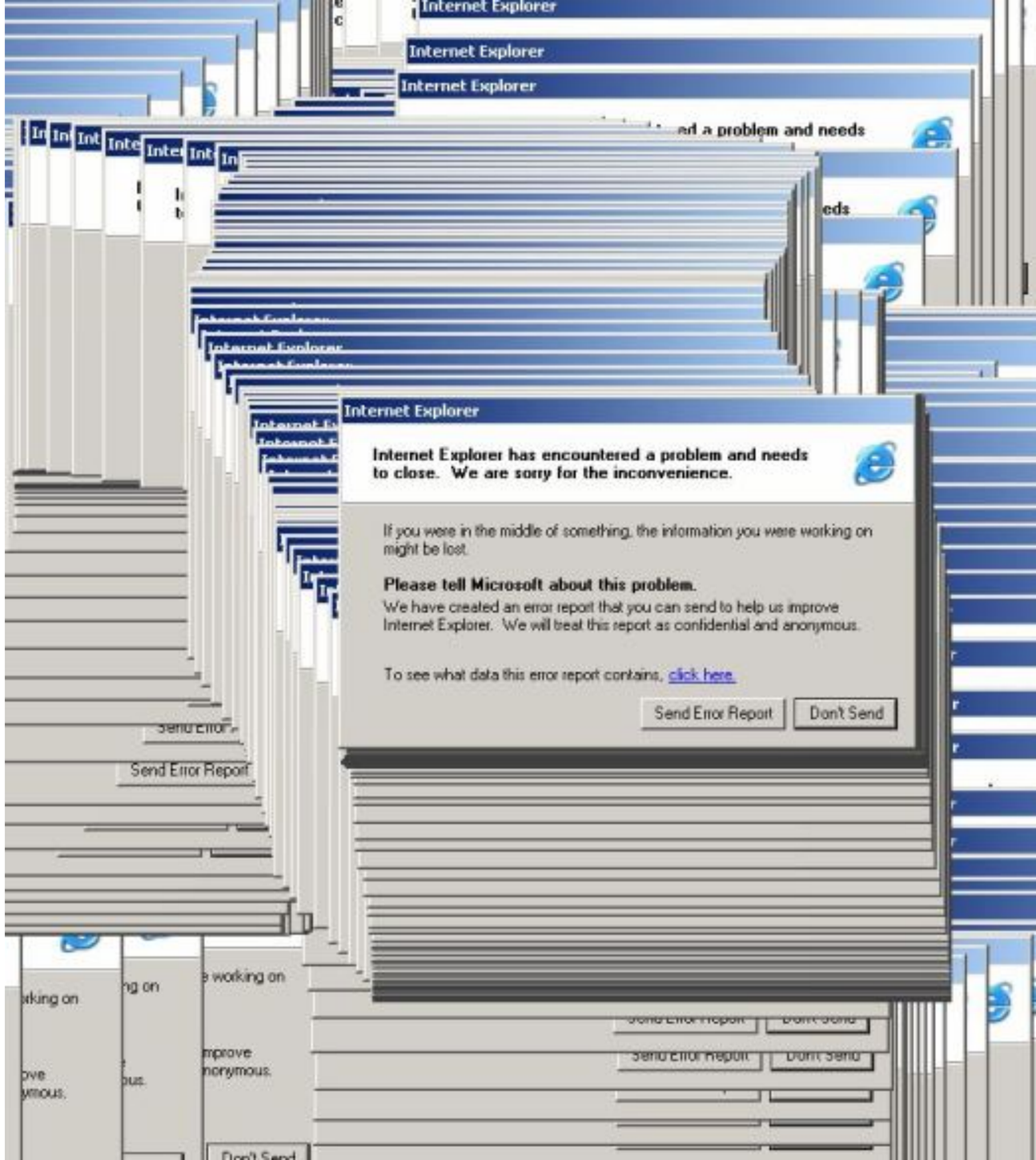
```
fun copy(list: MutableList<out T>)
```

- Star project: safe way to indicate subtype of projection

```
if (element is List<*>) . . .
```

Asynchronous Programming

It’s hard...



Possible solutions

- Threading
- Callbacks
- Promises
- Futures
- Reactive Extensions

Threading

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}
```

Threaded

```
fun preparePost(): Token {
    // makes request & blocks thread
    return token
}
```

Threading

- Threads are expensive - context switches
- Threads are not possible - single threaded targets
- Threads are not desirable - state mutations, UI threads

Callbacks

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}
```

async

callback

```
fun preparePostAsync(cb: (Token) -> Unit) {
    // makes request & return immediately
    // arranges callback to be called later
}
```

Callbacks

- Error handling can be complicated

- Callback hell a.k.a. tilted Christmas Tree

Promises, Futures and Rx

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }
}
```

composable

promise

```
fun preparePostAsync(): Promise<Token> {
    // makes result & return a promise that
    return promise // ... is completed later
}
```

Futures, Promises and Rx

- Error handling can be complicated
- Thinking of different constructs on different platform
- Changing the way of thinking for sync vs async (move to streams)

Kotlin Coroutines

Coroutines

```
fun postItem(item: Item) {
    launch(CommonPool) {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}
```

explicit coroutine context

natural signature

suspending function

```
suspend fun preparePost(): Token {
    // makes request & suspends coroutine
    return suspendCoroutine { /* ... */ }
}
```

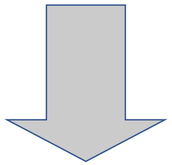
Coroutines

- Same way of thinking for sync and async
- Same way of doing things (exception handling, loops, etc.)
- Same constructs
- Lightweight threads
- Long time existing concepts

How they work

Kotlin

```
suspend fun submitPost(token: Token, item: Item): Post {...}
```



Java/JVM

callback

```
Object submitPost(Token token, Item item, Continuation<Post> cont) {...}
```

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```

How they work

Kotlin

```kotlin
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
```

Java/JVM

```java
switch (cont.label) {
    case 0:
        cont.label = 1;
        preparePost(cont);
        break;
    case 1:
        Token token = (Token) prevResult;
        cont.label = 2;
        submitPost(token, item, cont);
        break;
    case 2:
        Post post = (Post) prevResult;
        processPost(post);
    }
}
```

Constructs

- **Builders (launch, runBlocking, async)**: regular world to coroutine world
- **Suspending functions (suspend):** from coroutine to coroutine
- **suspendCoroutine:** coroutine to callback

Tips and More

Idiomatic Kotlin

Use `when` as expression body

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

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

Use `try` as expression body

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

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

Use `try` as expression

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

```
fun tryParse(number: String): Int? {
  val n = try {
    Integer.parseInt(number)
  } catch (e: NumberFormatException) {
    null
  }
  println(n)
  return n
}
```

Use `elvis` with `return` and `throw`

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

```
fun processPerson(person: Person) {
  val name = person.name
  if (name == null)
    throw IllegalArgumentException(
        "Named required")
  val age = person.age
  if (age == null) return

  println("$name: $age")
}
```

```
fun processPerson(person: Person) {
  val name = person.name ?:
    throw IllegalArgumentException(
      "Name required")
  val age = person.age ?: return

  println("$name: $age")
}
```

Use range checks instead of comparison pairs

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

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

Don’t necessarily create classes just to hold functions

```
class StringUtils {
  companion object {
    fun isPhoneNumber(s: String) =
        s.length == 7 &&
        s.all { it.isDigit() }
  }
}
```

```
object StringUtils {
  fun isPhoneNumber(s: String) =
    s.length == 7 &&
    s.all { it.isDigit() }
}
```

```
fun isPhoneNumber(s: String) =
    s.length == 7 &&
    s.all { it.isDigit() }
```

Use extension functions copiously

```
fun isPhoneNumber(s: String) =
    s.length == 7 &&
    s.all { it.isDigit() }
```

```
fun String.isPhoneNumber() =
    length == 7 &&
    all { it.isDigit() }
```

Avoid using member extension functions (except for DSLs)

```
class PhoneBook {
  fun String.isPhoneNumber() =
      length == 7 &&
      all { it.isDigit() }
}
```

```
class PhoneBook {
}

private fun
String.isPhoneNumber() =
    length == 7 &&
    all { it.isDigit() }
```

Don’t use member extensions with containing class as receiver

```
class PhoneBook {
  fun PhoneBook.find(name: String)=
      "1234567"
}
```

```
class PhoneBook {
  fun find(name: String) =
      "1234567"
}
```

Consider extracting non-essential API into extensions

```
class Person(val firstName: String,
             val lastName: String) {

  val fullName: String
    get() = "$firstName $lastName"
}
```

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

val Person.fullName: String
    get() = "$firstName $lastName"
```

Use default parameters instead of 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 = "|")
}
```

Use `lateinit` when you can’t init in constructor

```
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)
  }
}
```

Use `typealias` for functional types and collections

```
class EventDispatcher {
  fun addClickHandler(
    handler: (Event) -> Unit) {
  }

  fun removeClickHandler(
    handler: (Event) -> Unit) {
  }
}
```

```
typealias ClickHandler =
    (Event) -> Unit

class EventDispatcher {
  fun addClickHandler(
      handler: ClickHandler) {
  }

  fun removeClickHandler(
      handler: ClickHandler) {
  }
}
```

Beyond Pair and Triple, use data classes

```
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()
}
```

Use destructuring in loops

```
fun printMap(m: Map<String, String>) {
  for (e in m.entries) {
    println("${e.key} -> ${e.value}")
  }
}
```

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

Use destructuring with lists

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

```
fun splitNameExt(fn: String): NameExt {
  if ('.' in fn) {
    val parts = fn.split('.', limit = 2)
    return NameExt(parts[0], parts[1])
  }
  return NameExt(fn, null)
}
```

```
fun splitNameExt(fn: String): NameExt {
   if ('.' in fn) {
      val (name, ext) =
          fn.split('.', limit = 2)
      return NameExt(name, ext)
   }
   return NameExt(fn, null)
}
```

Use `copy` method for data classes

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

fun happyBirthday(person: Person) {
    person.age++
}
```

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

fun happyBirthday(person: Person) =
    person.copy(
        age = person.age + 1)
```

Use `coerceIn` to ensure number is in range

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

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

Use `apply` for object initialisation

```
fun createLabel(): JLabel {
  val label = JLabel("Foo")
  label.foreground = Color.RED
  label.background = Color.BLUE
  return label
}
```

```
fun createLabel() =
    JLabel("Foo").apply {
      foreground = Color.RED
      background = Color.BLUE
    }
```

Use `filterIsInstance` to filter a list by object type

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

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

Use ‘mapNotNull’ to apply a function and select items for which it returns a non-null value

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

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

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

Use `compareBy` for multi-step comparison

```
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
        })
```

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

Use `groupBy` to group items in a collection

```
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)
  }
}
```

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

Use String methods for string parsing

```
data class PathParts(val dir: String,
                     val name: String)
```

```
val pattern = Regex("(.+)/([^/]*)")

fun splitPath(path: String): PathParts {
  val match = pattern.matchEntire(path)
        ?: return PathParts("", path)

  return PathParts(match.groupValues[1],
      match.groupValues[2])
}
```

```
fun splitPath(path: String) =
  PathParts(
    path.substringBeforeLast('/', ""),
    path.substringAfterLast('/'))
```

https://github.com/jetbrains/kotlin-workshop