Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ✓ Functional Programming

DAY 2

- ○ Object-Orientation
- ○ Where to Go From Here

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ✓ Functional Programming

DAY 2

- ○ **Object-Orientation**
- ○ Where to Go From Here

Object-Orientation

1. **Intro**
2. Basics
3. Properties
4. Constructors
5. Inheritance
6. Visibilities
7. Special Classes
8. Objects
9. Generics & Variance

Intro I – Objects

- **Model the world with objects**
  - Data + capabilities
  - Data as attributes
  - Capabilities as methods

- **Objects interact**
  - Well-defined interfaces
  - Information hiding

Intro II – Classes

- **Objects are instances of classes**
  - Generally one class, arbitrary #objects
  - Classes define data + capabilities
- **Inheritance**
  - Classes can inherit data + capabilities
- **Composition**
  - Objects may contain or delegate to other objects

Intro III – Challenges

- **Designing OO systems**
  - Design patterns
  - SOLID principle
- **Structural complexity**
  - Overengineering
- **Reusability**
  - Generic classes and interfaces
- **Modularity**
  - Information hiding, single responsibility, …

Object-Orientation

- ✓ Intro
- **2. Basics**
- 3. Properties
- 4. Constructors
- 5. Inheritance
- 6. Visibilities
- 7. Special Classes
- 8. Objects
- 9. Generics & Variance

Basics I – Classes

- **The basic building blocks**
  - Define the entities of the software
- **Data = properties**
  - Like variables: **`val`** vs **`var`**
- **Capabilities = methods**
  - Like functions: **`fun`**

```
class Student {
  val grade: Double = 3.6
  fun takeExam() { … }
}
```

Basics II – Object construction

- **Objects instantiated from classes**
  - Constructor
  - Parameters => objects differ

- **No new keyword**
  - Still identifiable

```
val student = Student()
```

Object-Orientation

- ✓ Intro
- ✓ Basics
- **3. Properties**
- 4. Constructors
- 5. Inheritance
- 6. Visibilities
- 7. Special Classes
- 8. Objects
- 9. Generics & Variance

Properties I – Late-initialized (1/2)

- **Non-nullable data must be initialized directly**
  - Inconvenient: what if value not yet known?
  - Java: typically initialize with `null` (=> nullable type)

- **Kotlin: `lateinit` modifier**
  - **`lateinit var`** `userData: UserData` (must be **`var`**)
  - Allows to defer initialization

- **Throws `UninitializedPropertyAccessException`**
  - If forgot to initialize

Properties II – Late-initialized (2/2)

- **Use case #1: test cases**
  - Initialization in setup method
  - `@BeforeEach` in JUnit 5

- **Use case #2: dependency injection**
  - Initialization via DI framework
  - e.g. Dagger 2 on Android
  - `@Inject lateinit var student: Student`

Properties III – Delegation

- **Common kinds of properties implemented in stdlib**
- **Lazy properties**
  - Computed only on demand
  - `val maybeNotNeeded by lazy { … }`
- **Observable properties**
  - Can observe value changes
  - `val uiData by observable { … }`
- **Vetoable properties**
  - Can observe and intercept value changes
  - `val validated by vetoable { … }`

Reference: How Delegation Works

```
class C {
  var prop: Type by SomeDelegate()
}
// Compiler-generated code:
class C {
  private val prop$delegate = SomeDelegate()
  var prop: Type
    get() = prop$delegate.getValue(this, this::prop)
    set(value: Type) {
      prop$delegate.setValue(this, this::prop, value)
    }
}
```

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- **4. Constructors**
- 5. Inheritance
- 6. Visibilities
- 7. Special Classes
- 8. Objects
- 9. Generics & Variance

Constructors I – Overview

- **2 types of constructors**
  - One *primary constructor*
  - Any number of *secondary constructors*

- **Intuition**
  - **Primary**: main interface for object creation
  - **Secondaries**: alternatives that internally map to primary

Constructors II – Syntax

- **Primary constructor in class header**
  - `class User(username: String, paid: Boolean) { … }`
  - `init {}` for initialization logic
  - Shorthand
    - `class User(val username: String, val paid: Boolean)`

- **Secondary constructors in class body**
  - `constructor(username: String, plan: Plan) : this(username, true)`
  - Must delegate to primary via `this(…)`

Constructors III – Best Practices

- **Do not use for telescoping**
  - Use default values instead

- **Use secondary constructors for alternative representations**
  - `class Point2D(val x: Double, val y: Double)`
  - `constructor(xy: Pair<Double, Double>) : this(xy.first, xy.second)`

- **Use shorthand primary constructor**
  - Unless custom accessors

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- **5. Inheritance**
- 6. Visibilities
- 7. Special Classes
- 8. Objects
- 9. Generics & Variance

Inheritance I – Idea

- **Classes can inherit data + capabilities**
  - `class Animal(val weight: Int, val extinct: Boolean)`
  - `class Dog(val name: String) : Animal(20, false)`

- **OO = programming of deltas**
  - Inherit similarities
  - Program the differences
  - Theoretically…

- **Challenge: strong coupling**

Inheritance II – Entities

- **Interfaces**
  - Highest level of abstraction
  - Represent abstract capabilities
- **Abstract classes**
  - High level of abstraction
  - Typically implement some general behavior
- **Open classes**
  - Lower level of abstraction
  - All data + capabilities must be concrete
- **Regular classes**: disallow inheritance

Inheritance III – Best Practices

- **Design for inheritance or prohibit it**
  - Enforced through closed-by-default
- **Prefer composition to inheritance**
  - Weaker coupling
  - Zero boilerplate in Kotlin with **by**
- **Define generic capabilities in interfaces**
  - Multiple interfaces implementable
  - Often -`able` suffix

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- ✓ Inheritance
- **6. Visibilities**
- 7. Special Classes
- 8. Objects
- 9. Generics & Variance

Visibilities I – Motivation

- **Information hiding**
  - Well-defined interface
  - Internals inaccessible
  - Enforces invariants, more predictable

- **Visibilities enable information hiding**
  - Define what's accessible where
  - Not very fine-grained, only four possible visibilities

Visibilities II – Overview

- **Public**
  - Default visibility
  - Visible everywhere (if containing class is visible)
- **Internal**
  - Visible inside same module (if containing class is visible)
  - Module = "set of Kotlin files compiled together"
    - IntelliJ module(!), Maven project, Gradle source set, …
- **Protected**
  - Visible inside containing class and children
- **Private**
  - Visible only inside containing class (file)

Visibilities III – Best Practices

- **Use `public` for well-defined API**
  - Or `internal` if only used inside same module
- **Use `private` for class internals**
  - Enforcing information hiding
- **Restrict visibility as much as possible**
  - Can increase on demand (or restructure)
- **Use global variables sparingly**
  - Pollute global namespace
- **Use `protected` to design for inheritance**

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- ✓ Inheritance
- ✓ Visibilities
- **7. Special Classes**
- 8. Objects
- 9. Generics & Variance

Special Classes I – Overview

- **Data classes**
  - May also define methods
  - Convenient, robust
- **Enum classes**
  - For finite number of distinct values
  - e.g. states, directions, modes
- **Sealed classes**
  - Restricted class hierarchies
  - Generalization of enums
  - Allow extending interface afterwards

Special Classes II – Data Classes

- **Generated methods**
  - `equals()`
  - `hashCode()`
  - `toString()`
  - `componentN()` (enables destructuring declaration)
  - `copy()`

```
data class Contact(val name: String, val phone: String)
val friend = Contact("Patrick Pack", "987654321")
val (name, phone) = friend
```

Special Classes III – Enum Classes

- **Used to model several distinct instances**
  - e.g. `LogLevel`, `PostCategory`, or `FileWalkDirection`
- **Enable exhaustive `when`-expressions**
  - Advantage of being fixed at compile-time
- **May contain properties and methods**
  - Though often without

```
enum class LogLevel { DEBUG, WARNING, ERROR }
```

Special Classes IV – Sealed Classes

- **Allows fixed hierarchies**
  - Can only be extended inside same file
  - Control over hierarchy
- **May be combined with data classes**

```
sealed class Expression
data class Const (val value: Int) : Expression()
data class Plus  (val left: Expression,
                  val right: Expression) : Expression()
data class Minus (val left: Expression,
                  val right: Expression) : Expression()
```

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- ✓ Inheritance
- ✓ Visibilities
- ✓ Special Classes
- **8. Objects**
- 9. Generics & Variance

Objects I – Overview

- **Object declarations**
  - Singletons
- **Object expressions**
  - Ad-hoc objects
  - Instead of anonymous inner classes
- **Companion objects**
  - Alternative to static members

Objects II – Object Declarations

- **Declare named objects**
  - Only ever one instance => *Singleton*

- **Use if only one object necessary**
  - e.g. `DataCache`, `Registry`, `UserRepository`, `Presenter`, …

```
object HomeController {
  fun updateUi() { … }
}

HomeController.updateUi()
```

Objects III – Object Expressions

- **Ad-hoc objects**
  - ```
    val center = object {
      val x = centerX
      val y = centerY
    }
    ```
- **Implementing interfaces**
  - ```
    view.setOnClickListener(object : ClickListener {
      override fun click(event: ClickEvent) { … }
    })
    ```
  - For SAM interfaces, prefer lambdas
    - ```
      view.setOnClickListener { … }
      ```
      ```

Objects IV – Companion Objects

- **No `static` keyword in Kotlin**
  - Alternative 1: file-level declarations
  - Alternative 2: companion objects
- **Companion objects**
  - Object declaration inside class with **`companion`** modifier
  - Accessible directly on class, like static members

```
class Car {
  companion object CarFactory { fun defaultCar() = … }
}
val car = Car.defaultCar()
```

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- ✓ Inheritance
- ✓ Visibilities
- ✓ Special Classes
- ✓ Objects
- **9. Generics & Variance**

Generics I – Idea

- **Increase API reusability**
  - `Box<T>` instead of `IntBox`, `StringBox`, `PersonBox`, …
  - `List<T>` instead of `IntList`, `DoubleList`, …
  - Data types: same generic logic
  - DRY principle

- **Generic classes and functions**
  - `class Stack<T> { fun add(t: T) { … } }`
  - `fun <T> stackOf()`
  - *Generic type parameter* name is arbitrary

Generics II – Covariance (1/2)

- **Covariance = subtype usable in place of supertype**
  - e.g. `val n: Number = 23`

- **Can be unsound/unsafe**
  - Java arrays are covariant and unsafe:
    ```
    Number[] numbers = new Integer[];
    numbers.add(1.234);  // error at runtime
    ```
  - Safe if type is a producer only, not consumer

Generics III – Covariance (2/2)

- **Return types are covariant (in Kotlin & Java)**
  - `interface Measureable { fun measure(): Number }`
  - ```
    class Ingredient() : Measureable {
      override fun measure(): Int
    }
    ```
- **Kotlin's read-only collections are covariant**
  - Safe because producers only
  - `val invited: List<Person> = listOf<Friend>(…)`
- **Mutable collections are not covariant – why?**
    ```

Generics IV – Contravariance

- **Contravariance = supertype usable in place of subtype**
  - Counterpart to covariance
  - e.g. `val salaryComp: Compare<Int> = NumberCompare()`

- **Safe if consumer only, not producer**
  - e.g. `Compare<T>`, `Repair<T>`, `Handle<T>`, …
  - How to handle supertype implies how to handle subtype

Generics V – Invariance

- **Invariance = cannot use subtype as supertype or vice versa**
  - Only safe choice for many classes
  - If both consumes *and* produces `T`

- **Consume**
  - `T` in *in-position*
  - `fun consume(t: T) { … }`

- **Produce**
  - `T` in *out-position*
  - `fun produce(): T { … }`

Generics VI – Declaration-Site Variance

- **Declare class as variant (w.r.t. certain type parameter)**
  - `class Stack<out E>`
  - `class MutableStack<E>`
  - `class Compare<in T>`
  - `class Function<in T, out R>`

- **Allows variance at all use-sites**
  - `val deck: Stack<Drawable> = stackOf<Card>(…)`
  - `val expert: Compare<Car> = VehicleExpert()`

- **Use if safe to increase reusability**

Generics VII – Use-Site Variance

- **Only way in Java**
  - Also possible in Kotlin

- **To make invariant types variant at a particular use-site**
  - `val producer: MutableList<out Number> = mutableListOf<Int>()`
  - `producer.add(?)  // not possible (parameter type Nothing)`
  - In Java: `List<? extends Number> producer = new ArrayList<>();`

- **PECS: producer-extends, consumer-super (Java)**
  - Easy in Kotlin: producer-out, consumer-in ☺

Generics VIII – Star Projections

- **Use if no information about generic type**
  - Uses most restrictive type that's still safe
- **For covariant `Producer<out T>`**
  - Becomes `Producer<out Any?>`
  - May produce anything
- **For contravariant `Consumer<in T>`**
  - Becomes `Consumer<in Nothing>`
  - May consume nothing
- **For invariant `Invariant<T>`**
  - Becomes `Invariant<out Any?>` when reading
  - Becomes `Invariant<in Nothing>` when writing

Generics IX – Best Practices

- **Prefer generic classes and functions**
  - If logic is independent of generic type
  - Useful for data types
  - Increase reusability

- **Prefer declaration-site variance to use-site variance**
  - If type represents consumer or producer

- **Use star projections if no type info but want type safety**

Object-Orientation

- ✓ Intro
- ✓ Basics
- ✓ Properties
- ✓ Constructors
- ✓ Inheritance
- ✓ Visibilities
- ✓ Special Classes
- ✓ Objects
- ✓ Generics & Variance

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ✓ Functional Programming

DAY 2

- ✓ Object-Orientation
- ○ Where to Go From Here

Where To Go From Here

1. **Resources**
2. Community

Resources I – The Docs(!)

- **Kotlin's official docs are excellent**
  - https://kotlinlang.org/docs/reference/
- **In-depth and precise info**
- **First place to go for detailed information**

Resources II – The Koans

- **Solve tasks in Kotlin while learning the language**
  - You can even do them online
  - https://try.kotlinlang.org/#/Kotlin%20Koans/Introduction/Hello,%20world!/Task.kt

- **Great place to get more practice after this training!**

Where To Go From Here

✓ Resources

2. Community

Community – Slack Channel

- **Extremely active and helpful Slack channel**
  - Developers like you, many doing Android
  - Developers from Jetbrains
  - #getting-started channel

- **How to join**
  - Get an invite: http://slack.kotlinlang.org/
  - Join the channel: https://kotlinlang.slack.com/

- **All community resources: http://kotlinlang.org/community/**

Where To Go From Here

- ✓ Resources
- ✓ Community

Stay in Touch!

- @petersommerhoff
- LinkedIn
- http://petersommerhoff.com

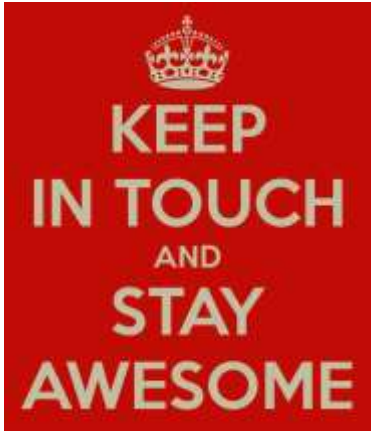

Image source: https://www.askideas.com/26-very-best-keep-in-touch-pictures/