Introduction to Kotlin

What to expect

- focus on unique features
  - not present in Java
  - or differently implemented
- no focus on language syntax
- null-Safety, class types, coroutines, ...
- migration from Java, implementing 2nd day

What is Kotlin?

- open source
- initial release: 2011, stable: 2016
- developed by JetBrains
- statically-typed programming language
- compiled to Java bytecode
  - runs on the JVM
- official language for Android development

Why Kotlin for Java Developers?

- designed to be more concise, expressive, and safe compared to Java.
- smooth learning curve for Java developers due to its similarity
- reduces boilerplate code
- modern features: null safety, higher-order functions, smart casts, etc
- full interoperability with Java
  - can call Kotlin code from Java and vice versa.

Why Kotlin for Java Developers?

- BUT:
  - Java caught up with features - a little different syntax
  - lots of additional features might make it harder to read for beginners

Null Safety

- Kotlin has built-in null safety
  - `?` for nullable types
  - `?.` for safe calls
- the compiler helps you prevent NullPointerExceptions
- you can disable the null safety using `!!`

Null Safety - Example

Java:

```java
String name = null;
if (name != null) {
  name.length();
}
```

Kotlin:

```kotlin
val name: String? = null
name?.length
```

Elvis Operator

- Elvis operator to provide default value if null
- safe call `?.` and the Elvis operator `?:` are concise ways to deal with nullable types.

Elvis Operator - Example

```
val name: String? = null
println(name?.length ?: "Unknown")
```

- Showcase: NullSafety.kt

Classes

- public by default
- open keyword is used to mark a class as inheritable
  - by default, all classes in Kotlin are final
- Kotlin uses primary constructors directly in the class header
- properties should be accessed directly
- `get` and `set` method are generated during compilation
- `new` keyword not needed for instantiation

Classes - Example

Java:

```
public class Person {
  String name;
  public Person(String name) {
    this.name = name;
  }
}
```

Kotlin:

```
open class Person(val name: String)
```

Data Classes

- automatically generate equals(), hashCode(), toString(), and other utility methods
- similar to records in Java, but Java records are immutable
- use `@JvmRecord` to compile to Java record

Data Classes - Example

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

- Showcase: DataClass.kt

Functions

- functions are defined using the fun keyword.
- Kotlin allows type inference, so the return type and parameters can often be omitted
- `void` if return type omitted
- function as method parameter possible

Functions - Example

Java:

```
public String greet(String name) {
  return "Hello, " + name;
}
```

Kotlin:

```
fun greet(name: String): String {
  return "Hello, $name"
}
```

- Showcase: Functions.kt

Coroutines

- write asynchronous, non-blocking code
- more lightweight than traditional threads
- following structured concurrency principles
  - control flow constructs that have clear entry and exit points
  - making code easier to read and debug

Coroutines

- useful for tasks like network requests, file I/O, or heavy computations
- implemented using lightweight continuations and suspendable functions
- higher abstraction than threads, can run e.g. on virtual threads

Coroutines - Example

```
runBlocking {
    val data = async { fetchData() }
    println(data.await())
}
```

- Showcase: Coroutines.kt

Functional Programming

- older than Java's functional API
- supports lambda expressions and allows passing functions as parameters
- `it` implicit
- no `stream` and `collect` unlike in Java

Functional Programming - Example

Java:

```
List<String> names = List.of("Alice", "Bob", "Charlie");
names.stream().map(String::toLowerCase).forEach(System.out::println);
```

Kotlin:

```
val names = listOf("Alice", "Bob", "Charlie")
names.map { it.lowercase() }.forEach { println(it) }
```

Extension Functions

- Java does not support extension functions directly
  - create utility classes
- Kotlin allows to add new functions to existing classes via extension functions

```
fun String.printWithStars() {
  println("*** $this ***")
}

"Hello".printWithStars()
```

Operator Overloading

- custom implementations for predefined set of operators
  - possible operators: +, -, *, /, =, <, >, ...
- readily implemented for some like `BigInteger`

```
"a".toUpperCase() == "A"
BigInteger.ONE + BigInteger.ONE
```

Operator Overloading - Example

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

Resources and more

- Kotlin documentation: https://kotlinlang.org/docs/home.html
- Roman Elizarov: Coroutines and Loom behind the scenes