



Overview

- What? Why? Who?
- Basic syntax
- Functions & extension functions
- Null safety
- Classes & properties
- Functional programming

What?

- JVM programming language developed by JetBrains
- Open sourced in 2011. v1.0 on Feb 15th 2016
- 100% interoperable with Java
- First class citizen on Android since Google I/O 2017
- Safe, interoperable, concise

Concise?

```
public class Person {

    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {

        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Person person = (Person) o;

        if (firstName != null ? !firstName.equals(person.firstName) : person.firstName != null) return false;
        return lastName != null ? lastName.equals(person.lastName) : person.lastName == null;
    }

    @Override
    public int hashCode() {
        int result = firstName != null ? firstName.hashCode() : 0;
        result = 31 * result + (lastName != null ? lastName.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
            "firstName='" + firstName + '\'' +
            ", lastName='" + lastName + '\'' +
            '}';
    }
}
```

Concise!

```
data class Person(var firstName: String, var lastName: String)
```

Why?

- Costs nothing to adopt
- No runtime overhead
- Strong IDE support
- Quick to learn
- It comes from industry, not academia
- Null safety

Who?

- Backend Developers
- Android Developers
- iOS Developers
- Windows Developers
- Javascript Developers

Basic syntax

Var

```
var firstName: String = "Jamie"
```

```
var firstName = "Jamie"
```

```
firstName = "Martin"
```

```
firstName = 5 // Compilation error
```

Val

```
val lastName: String = "Magee"
```

```
val lastName = "Magee"
```

```
lastName = "Bloggs" // Compilation error
```

Objects

```
val sb = StringBuilder("Hello")
```

```
sb.append(" World!")
```

```
println(sb) // Hello World!
```

Functions & extension functions

Functions

```
fun maximum(first: Int, second: Int): Int {  
    if (first > second) {  
        return first  
    } else {  
        return second  
    }  
}
```

Functions – ‘if’ is an expression

```
fun maximum(first: Int, second: Int): Int {  
    return if (first > second) {  
        first  
    } else {  
        second  
    }  
}
```

Functions – default values

```
fun maximum(first: Int = 3, second: Int = 4): Int {  
    return if (first > second) {  
        first  
    } else {  
        second  
    }  
}
```

Functions – calling functions

```
var max = maximum(1, 2)
```

```
// Named Parameters
```

```
max = maximum(first = 1, second = 2)
```

```
max = maximum(second = 2, first = 1)
```

```
max = maximum(second = 8)
```


Functions – concise?

```
fun maximum(first: Int, second: Int): Int {  
    return if (first > second) {  
        first  
    } else {  
        second  
    }  
}
```

Functions – concise!

```
fun maximum(first: Int, second: Int): Int =  
    if (first > second) {  
        first  
    } else {  
        second  
    }
```

Functions – concise-er!

```
fun maximum(first: Int, second: Int) =  
    if (first > second) {  
        first  
    } else {  
        second  
    }
```

Functions – concise-est!

```
fun maximum(first: Int, second: Int) = if (first >  
second) first else second
```

Functions – when

```
fun checkResult(value: Any?) = when {  
    value == 3 -> "value is 3"  
    value is Int -> "double the value = ${value * 2}"  
    value == "Hello, world!" -> "Hello, MSE!"  
    else -> "No value"  
}
```

```
val value: Any? = 3  
println(checkResult(value)) // value is 3
```

Functions – when – concise

```
fun checkResult(value: Any?) = when (value) {  
    3 -> "value is 3"  
    is Int -> "double the value = ${value * 2}"  
    "Hello, world!" -> "Hello, MSE!"  
    else -> "No value"  
}
```

```
val value: Any? = 3  
println(checkResult(value)) // value is 3
```

Extension functions

```
fun Int.maximum(other: Int) =  
    if (this > other) this else other
```

```
val max = 3.maximum(4) // 4
```

Extension functions – Infix

```
infix fun Int.maximum(other: Int) =  
    if (this > other) this else other
```

```
val max = 3 maximum 4 // 4
```


Null safety

The nullability problem

```
private String myString = "A value";
```

```
// myString is reassigned  
myString = null;
```

```
// Uh oh  
myString.equals("Another value");
```

Nullable type

```
val a: String = "Cannot be null"
```

```
val b: String? = null
```

Unwrapping nullables

`b?.equals("Other string") // Safe`

`b!!.equals("Other String") // Unsafe`

Chaining nullables

```
b?.toDoubleOrNull()?.compareTo(1)
```

```
if (b != null) {  
    if (Double.valueOf(b) != null) {  
        return Double.valueOf(b).compareTo(1.0);  
    }  
}  
return null;
```

Elvis operator

```
val result = b?.toDoubleOrNull()?.equals(1) ?: false
```



Classes & properties

Classes

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


Classes

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

Classes

```
class Person constructor(fName: String, lName:  
String) {  
    val firstName: String = fName  
    var lastName = lName  
}
```

Classes

```
public class Person {  
    @NotNull  
    private final String firstName;  
    @NotNull  
    private String lastName;  
  
    public Person(@NotNull String firstName, @NotNull String lastName) {  
        Intrinsics.checkNotNull(firstName, "firstName");  
        Intrinsics.checkNotNull(lastName, "lastName");  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    @NotNull  
    public String getFirstName() {  
        return firstName;  
    }  
  
    @NotNull  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        Intrinsics.checkNotNull(lastName, "lastName");  
        this.lastName = lastName;  
    }  
}
```

Properties

```
class Person constructor(private val firstName: String,  
private var lastName: String) {  
  
    fun getFullName() = "$firstName $lastName"  
  
}
```

Properties

```
public final class Person {  
    @NotNull  
    private final String firstName;  
    @NotNull  
    private String lastName;  
    @NotNull  
    public final String getFullName() {  
        return "" + this.firstName + ' ' + this.lastName;  
    }  
    public Person(@NotNull String firstName, @NotNull String  
lastName) {  
        ...  
    }  
}
```

Properties

```
class Person(private var firstName: String,  
             private var lastName: String) {  
  
    var fullName  
        get() = "$firstName $lastName"  
        set(value) {  
            val split = value.split(" ")  
            firstName = split[0]  
            lastName = split[1]  
        }  
}
```

Properties

```
public final class Person {  
    @NotNull  
    private String firstName;  
    @NotNull  
    private String lastName;  
    @NotNull public final String getFullName() {  
        return "" + this.firstName + ' ' + this.lastName;  
    }  
    public final void setFullName(@NotNull String value) {  
        List split = StringsKt.split$default(...);  
        this.firstName = (String)split.get(0);  
        this.lastName = (String)split.get(1);  
    }  
    public Person(@NotNull String firstName, @NotNull String  
lastName) {  
        ...  
    }  
}
```

Functional programming

Lambdas

```
val sum: (x: Int, y: Int) -> Int = { x, y -> x + y }
```

```
val sum2: (Int, Int) -> Int = { x, y -> x + y }
```

```
val sum3: { x: Int, y: Int -> x + y }
```

Lambdas

```
val sum3: { x: Int, y: Int -> x + y }
```

```
/** A function that takes 2 arguments. */
```

```
public interface Function2<in P1, in P2, out R> :
```

```
Function<R> {
```

```
    /** Invokes the function with the specified  
arguments. */
```

```
    public operator fun invoke(p1: P1, p2: P2): R  
}
```

Collections

```
val list = arrayOf(1, 2, 3, 4, 5)
```

```
val result = list.map { it + 2 }  
    .filter { value -> value % 2 == 0 }  
    .firstOrNull { it > 5 }
```

Sequences

```
val list = arrayOf(1, 2, 3, 4, 5)
```

```
val result = list.asSequence()  
    .map { it + 2 }  
    .filter { value -> value % 2 == 0 }  
    .firstOrNull { it > 5 }
```

Questions?

Resources

- <https://try.kotlinlang.org>
- <https://kotlinlang.org/docs/reference/>
- <https://kotlin.link>
- <http://slack.kotlinlang.org>
- <https://www.pluralsight.com/courses/kotlin-fundamentals>