Dart vs Kotlin vs Swift. Main differences and similarities.

Alex Matuzenko
Experienced Mobile Developer || Tech Enthusiast || Passionate about Backend and complex Systems Design
发布日期: 2019 年 3 月 5 日

Modern native mobile applications are written mainly in Swift (iOS) or Kotlin (Android) programming languages. There are also two major frameworks for cross-platform apps development - React Native (JavaScript) and Flutter with Dart as a programming language. While React Native has its own niche, Flutter looks much more promising and will be used for comparison with Swift and Kotlin.
This article focuses on the main similarities and differences between Dart, Kotlin, and Swift including syntax, functions and classes declarations, and other important details. Luckily there are many parallels in these languages, which makes easier to switch context between Dart or native Kotlin/Swift apps development.

Variables And Constants

Dart

```
// Implicit type
var myVariable = 5;

// Explicit type
String name = 'John';

// Dynamic type
dynamic name = 'John';

// Constants
final name = 'John';
const name = 'John';
```

Kotlin

```
// Implicit type
var myVariable = 5

// Explicit type
var name: String = "John"

// Dynamic type (not for JVM)
var dyn: dynamic = ...

// Constants
val name = "John"
```

Swift

```
// Implicit type
var myVariable = 5

// Explicit type
var name: String = 'John'

// Dynamic type
N/A

// Constants
let name = "John"
```

Basic types

Dart

```
// String -> int
var one = int.parse('1');

// int -> String
String oneAsString = 1.toString();

// Boolean
bool boolValue = false

// Enum
enum Color { red, green, blue }

// Type aliases
N/A

// Check type
if (a is String)...

// Type casting
Movie movie = item as Movie
```

Kotlin

```
// String -> Int
var one = "1".toIntOrNull()

// Int -> String
var oneAsString = 1.toString()

// Boolean
var boolValue: Boolean = false

// Enum
enum class Color { RED, GREEN, BLUE }

// Type aliases
typealias NodeSet = Set<Network.Node>

// Check type
if (a is String)...

// Downcasting
val movie = item as? Movie
```

Swift

```
// String -> Int, 1 or nil
let one = Int("1")

// Int -> String
let oneAsString = String(1)

// Boolean
var boolValue: Bool = false

// Enum
enum Color { case red, green, blue }

// Type aliases
typealias AudioSample = UInt16

// Check type
if a is String...

// Downcasting
let movie = item as? Movie
```

Strings

Dart

```
// Declaration
var s1 = 'String';
var s2 = "String";

// Multiline string
var s1 = """This is a
multi-line string.""";

// String templates
var i = 10;
print('The number is $i.');
print('For objects ${obj.field}');
```

Kotlin

```
// Declaration
val s1 = "String"

// Multiline string
val s1 = """This is a
multi-line string."""

// String templates
val i = 10
print("i = $i") // prints "i = 10"
print("For objects ${obj.field}")
```

Swift

```
// Declaration
let s1 = "String"

// Multiline string
let s1 = """This is a
multi-line string."""

// String templates
let i = 10
let message = "i = \(multiplier)"
print("For objects \(obj.field)")
```

Collections

Dart

```
// Arrays / List
var emptyList = <int>[];
List fixedLengthList = new List(3);
List growableList = new List();
var myList = [1,2,3];

// Sets
var emptySet = Set<String>();
mySet = Set<String>.from(['1', '2']);

// Maps
var emptyMap = Map<String, Int>();
var myMap = {'key1': value1, 'key2',
value2};
```

Kotlin

```
// Arrays / List
val emptyArray = arrayOf<String>()
val emptyList = listOf<String>()
val myArray = arrayOf("1", "2", "3")

// Sets
val emptySet = setOf<String>()
val mySet = setOf("1", "2"")

// Maps
val emptyMap = emptyMap<String, Int>()
val myMap = mutableMapOf("key1" to
"value1", "key2" to "value2")
```

Swift

```
// Arrays
var emptyArray = [Int]()
var myArray = ["1", "2", "3"]

// Sets
var emptySet = Set<String>()
var mySet = Set<String>(["1", "2"])

// Maps
var emptyMap = [String: Int]()
var myMap = ["key1": "value1", "key2":
"value2"]
```

Nullability & Optionals

Dart

```
// Declaration
int id = null;
id.abs(); // Exception
id?.abs(); // Safe call


// Null - aware operators
int id = null;
var userId = id ?? -1; // prints -1


// Optionals
int id;
var userId = id ?? -1; // prints -1
```

Kotlin

```
// Declaration
var id: Int? = null
id.inc() // Compilation error
id!!.inc() // Exception
id?.inc() // Safe call


// Elvis operator
val id: Int? = null
var userId = id ?: -1 // prints -1


// Optionals
val id: Int?
var userId = id ?: -1 // id must be
initialized
```

Swift

```
// Declaration
var id: Int? = nil
id.signum() // Compilation error
id?.signum() // Safe call


// nil-coalescing operator
let id: Int? = nil
var userId = id ?? -1 // prints -1


// Optionals
var id: Int? // optional
var id: Int = 1 // non-optional, must
be initialized
```

Control flow

Dart

```
// Switch - case
var id = 5;
switch (id) {
  case 1:
    print("id == 1");
    break;
  case 2:
    print("id == 2");
    break;
  default:
    print("id is undefined");
}


// Inclusive for loop
for (var i = 1; i <= 5; i++) {}

// Cascade notation (..)
querySelector('#confirm')
  ..text = 'Confirm'
  ..classes.add('important')
  ..onClick.listen((e) =>
window.alert('Confirmed!'));
```

Kotlin

```
// When
val id = 5
when (id) {
  1 -> print("id == 1")
  2 -> print("id == 2")
else -> {
  print("id is undefined")
  }
}


// Inclusive for loop
for (index in 1..5) {}

// Cascade notation (..)
N/A
You can use the Builder pattern or
scope functions apply or also
```

Swift

```
// Switch - case
let id = 5
switch id {
  case 1:
    print("id == 1")
  case 2:
    print("id == 2")
  default:
    print("id is undefined")
}


// Inclusive for loop
for index in 1...5 {}

// Cascade notation (..)
N/A
Builder, apply{} extension
```

Functions

Dart

```
// Declaration
String greet(String name) {
  return "Hey ${name}!";
}

// Optional named parameters
void greet({name: String}) {}

// Named parameters
void greet({name: String}) {}


// Default parameters
void greet({String name = "John"})
{}

// Var args
N/A

// Tuple return
N/A, you can return array


// Closure / lambda
var square = (int x) => x * x;
print(square(4)); // prints 16


// Extensions
N/A
```

Kotlin

```
// Declaration
fun greet(name: String): String {
  return "Hey ${name}!"
}

// Optional named parameters
fun greet(name: String): String {}

// Named parameters
fun greet({name: String}) {}
greet("John")
greet(name = "John")

// Default parameters
fun greet(name: String = "John") {}

// Var args
fun greet(vararg names: String...)
{}

// Tuple return
N/A, return class instead


// Closure / lambda
val square = { x:Int -> x * x }
print(square(4)) // prints 16


// Extensions
fun Int.square(): Int {
  return this * this
}

println(4.square()) //prints 16
```

Swift

```
// Declaration
func greet(name: String) -> String {
  return "Hey \(name)!"
}

// Optional named parameters
fun greet(_ name: String) {}

// Named parameters
fun greet(name: String) {}


// Default parameters
fun greet(name: String = "John") {}

// Var args
fun greet(_ names: String...) {}

// Tuple return
func getRates() -> (Int, Int, Int) {
    return (15, 20, 30)
}

// Closure / lambda
var square = { (x: Int) -> Int in
  return x * x
}
print(square(4)) // prints 16

// Extensions
extension Int {
  func square() -> Int {
    return self * self
  }
}

print(5.square()) //prints 16
```

Classes

Dart

```
// Declaration
class Square {
  var color = 0
  String greet(name: String) {
      return "Hello $name"
  }
}


// Inheritance
class Employee extends Person {}

// Data class or struct
N/A, use regular class declaration


// Interface & Protocol
abstract class Nameable {
  String name() {}
}

void call<T extends Nameable>
  (Nameable x) {
    print("Name is " + user.name());
  }
```

Kotlin

```
// Declaration
class Square {
  var color = 0
  fun greet(name: String)
      = "Hello $name"
}


// Inheritance
class MyActivity: AppCompatActivity()

// Data class (reference type)
data class User(var name:String)


var user1 = User("John")
var user2 = user1
user1.name = "Bob"
print(user2.name) // prints Bob


// Interface & Protocol
interface Nameable {
    fun name(): String
}

fun call<T: Nameable>(user: T) {
    println("User is " + user.name())
}
```

Swift

```
// Declaration
class Square {
  var color = 0
  func greet(name: String) -> String {
    return "Hello \(name)"
  }
}


// Inheritance
class Controller: UIViewController {}

// Struct (value type)
struct User {
 var name: String
}


var user1 = User(name: "John")
var user2 = user1
user1.name = "Bob"
print(user2.name) // prints John


// Interface & Protocol
protocol Nameable {
    func name() -> String
}

func call<T: Nameable>(user: T) {
    print("User is " + user.name())
}
```

To sum up

All these three languages are well suited to build beautiful modern mobile applications. Luckily now there is much more similar in Kotlin and Swift than it was for Java and Objective-C, and it is a perfect time to start learning new platform or even to move further and to give Flutter a chance.
If you guys have spotted any mistakes or want me to add something more into the article, please let me know. I am more than welcome your feedback!
Looking for an experienced mobile developer to grow your business? Let's keep in touch.

References & Credits

Swift is like Kotlin
Swift vs. Kotlin — the differences that matter
A Comparison of Swift and Kotlin Languages
Online playground - Dart
Online playground - Kotlin
Online playground - Swift