Table of Contents

Classes and objects: Data handling

The basic types in Kotlin are identical to those found in Java and C#.

We have:

- (`abstract`) `class`es
- `interface`s
- `enum class`es
  - Identical to `enum`s in Java; this construct is rare in other languages.
- `annotation class`es
  - Analogous to `@interface`s in Java and `Attribute`s in C#.
- `inner class`es
- `sealed class`es
  - A class with restricted inheritance. Only classes defined in the same file may inherit from it.
  - This is often used for *enum types* (variant types, *not object enums*) similar to Rust's `enum` and C++'s `std::variant`.

Unlike Java, where interfaces have only recently gained support for default implementations, Kotlin permits this for any JVM target level using bytecode magic.

Primary and secondary constructors

Before discussing classes in Kotlin, it is important to understand how constructors work. There are two types of constructors in Kotlin: *primary* and *secondary* constructors.

This is the basic structure of a class in Kotlin:

```
class Foo constructor(bar: String) {
    constructor(baz: Int) { ... }
    ...
}
```

A constructor in Kotlin is declared with the keyword `constructor`, followed by the list of parameters.

The constructor immediately following the type name, *if present*, is called the *primary constructor*. Unlike a secondary constructor, a primary constructor **does not** have a body. Any initialization logic must be wrapped in an `init` block. The parameters of this constructor are visible for all property initializers:

```
class Person constructor(birthYear: Int) {
    val age: Int
    val birthYearMinusOne = birthYear - 1 // birthYear is in scope
here!

    init {
        // Do some extra work here
        age = 2019 - birthYear
    }

    fun foo() {
        println(birthYear) // ERROR: construction has finished,
parameters are no longer available
    }
}
```

Additionally, the `constructor` keyword may be omitted *for the primary constructor only*. It is only required if you wish to apply an access modifier or annotation to the primary constructor:

```
@Component
class MyService @Autowired internal constructor(...)
```

What makes the primary constructor so powerful is that **it can have properties as parameters**. It is no longer necessary to repeat `this.field = field;` for every single constructor parameter!

```
class Person(var name: String, private var ssn: String)
```

This is all the code necessary to create a class with two properties, their respective getters/setters, and the proper constructor!

TIP	Notice that the class or constructor body ({}) can be omitted when it is empty; this is another useful quality-of-life feature.

If a primary constructor is present, all secondary constructors are required to *chain* to it, either directly or through another secondary constructor. This is useful if you wish to have a convenience constructor with different parameters that can be easily converted to the desired type:

```
class Timestamp(val instant: Instant) {
    constructor(ldt: LocalDateTime)
        : this(ldt.toInstant(ZoneOffset.UTC))
    constructor() {
        // ERROR: not chained to primary constructor!
        doSomethingElse()
    }
}
```

`objects`

In addition to these types, Kotlin introduces a new type of class called the `object`. More commonly known as *singletons*, `object`s only can have one global instance for the whole lifetime of a program — this behavior is identical to that of standard singletons and of `static` globals in C/C++. What makes `object`s really stand out, however, is that they require no boilerplate code to implement and are thus free from programmer error. Since they are normal classes, they can implement interfaces, and their instance can be passed around like a normal value.

This means that something like

```java
public interface Bar {
    String baz();
}

// Typical Java singleton
public enum Foo implements Bar {
    INSTANCE {
        @Override String baz() {
            return "Hello, world!";
        }
    }
}

public class Program {
    public static void doSomething(Bar bar) {
        System.out.println(bar.baz());
    }
    public static void main(String[] args) {
        doSomething(Foo.INSTANCE); // Unnecessary qualification :(
    }
}
```

would become

```kotlin
interface Bar {
    fun baz(): String
}

object Foo : Bar {
    override fun baz() = "Hello, world!"
}

fun doSomething(bar: Bar) {
    println(bar.baz())
}

fun main() = doSomething(Foo) // The type name itself refers to the
instance!
```

Kotlin and `static`

This leads us to another important point: *Kotlin does not have the concept of* `static`. The optimal replacement for static utility classes is either a set of top-level functions or an `object`. This is mostly up to your own taste, but typically depends on whether the utility conceptually belongs *at the package level*, or if they should be further grouped according to a certain criterion.

This means that something like

```
public class LzmaUtils {
    private LzmaUtils() {}
    public static void decompressStream(InputStream input) { ... }
}
```

would become

```
package myapp // These utilities may not belong at the top level

object LzmaUtils {
    fun decompressStream(input: InputStream) { ... }
}
```

or, alternatively, simply:

```
package myapp.lzma // This is an appropriate package for these
utilities

fun decompressStream(input: InputStream) { ... }
```

While it is ultimately up to the user to decide, creating top-level symbols is usually considered more idiomatic.

`companion object`s

What if one wants to mix static functions and instance methods within a single class? This is often not an indicator of good design choices — if you can, think about making these into (private) top-level functions instead.

This is possible, however, using `companion object`s:

```
class Person {
    var name = "Gagagegg" // Instance property

    companion object {
        fun createPerson(): Person = Person()
    }
}

...

Person.createPerson() // Person(name="Gagagegg")
```

A companion object is essentially an embedded `object` with the same name as a class: it is accessed using the enclosing class's name, can implement interfaces or abstract classes, and is treated as a value. This effectively removes the "non-object-orientedness" of static methods from the language, making it truly object-oriented.

Companion objects can be used to remove the need for boilerplate code. The most common application of this is in logging frameworks:

```
abstract class LoggerCompanion {
    val LOGGER = ...
}

class MyApplication {
    private companion object : LoggerCompanion()

    fun foo() {
        LOGGER.log("Hello, world!")
    }
}
```

All properties and functions of the companion object are pulled into the enclosing class's scope.

NOTE

Companion objects are compiled to a static singleton field, and are accessible as such from Java code. If you want companion object functions to be compiled to real static methods, annotate them with `@JvmStatic`.

`data classes`

Data classes are one of Kotlin's most loved features. If you need to store complex objects in memory and have all of the boilerplate abstracted away, they are the feature for you.

Data classes:

- must have a primary constructor with one or more parameters
- must have a primary constructor with no non-property parameters
- **should** generally be immutable
- cannot be inherited from

In most ways they behave identically to regular classes, except that `equals`, `toString` *and* `hashCode` *are automatically generated*!

```
data class Student(val name: String, val id: String, val graduation:
Year)
```

This single line of code will generate a `Student` class with a proper implementation of all of the following:

- constructors
- `getName`, `getId`, `getGraduation`
- `equals`, `hashCode`, `toString`
- `copy`

`copy` is automatically generated for all data classes and allows the user to construct an exact copy of the specified object, with the specified changes:

```
val john = Student("John", "1234", Year.of(2020))
val jane = john.copy(name = "Jane", id = "5678")
```

Thanks to `copy`, there is often no to make data classes mutable — new, modified instances can be created easily. Immutability comes with a large amount of benefits, including the elimination of defensive copying; data classes should generally be

made immutable.

NOTE	It may be worth mentioning that Java is adopting this syntax in JDK 14, for its proposed records feature! While this is a preview feature and may not necessarily make it into the full release, this is impressive progress.

Nesting

Classes can be nested in Kotlin.

CAUTION	Unlike in Java, to comply with scope rules, outer classes cannot access private members of inner classes!

Nested classes

This is equivalent to `static` classes in Java; the class is placed in the scope of the enclosing class and can access its private members, but is otherwise unrelated to it.

Inner classes

This is equivalent to normal nested classes in Java. Inner classes hold a reference to an instance of the outer class. In Kotlin, an inner class is denoted by the `inner` keyword:

```
class Outer {
    inner class Inner
}
```

To access the outer class instance, use `this@Outer`. Whereas in Java you would use `Outer.this`, Kotlin uses labels to accomplish this.

```
class Outer {
    fun foo() {
        println("Outer")
    }

    inner class Inner {
        fun foo() {
            println("Inner")
        }

        fun bar() {
            this@Outer.foo() // "Outer"
            this.foo() // "Inner"
        }
    }

    class Nested {
        fun bar() {
            this@Outer.foo() // ERROR: this is not an inner class,
                             // so it has no Outer instance!
        }
    }
}
```

Type aliases

Type aliases are often used to shorten long generic type names, or to alias function types.

```
typealias EventHandler = (EventData, String, Any) -> Boolean
typealias HandlerList = List<EventHandler>

val handlers: HandlerList // Instead of List<(EventData, String, Any)
-> Boolean>
```

Like C's `typedef` and C++'s `using`, typealiases are equivalent to and interchangeable with their aliased types.

```
typealias Id = String
fun retrieveId(): Id

val id: String = retrieveId() // ok
```

This is a good alternative to defining functional interfaces. Since Kotlin already has built-in function types, one can simply write

```
typealias Predicate<T> = (T) -> Boolean
```

instead of

```
@FunctionalInterface
interface Predicate<T> {
    boolean test(T t);
}
```