Table of Contents

Kotlin ABCs

Kotlin fundamentals

Kotlin is fundamentally an object-oriented language. This, along with the constraints imposed by the JVM, means that classes and data will be structured in much the same way as in Java, C++, or any other object-oriented language.

Kotlin's primary advantage to developers is that it manages to be extremely expressive while remaining fairly short; for example, Kotlin code is completely semicolon-free!

Files

Unlike many other languages, the Kotlin compiler has the additional concept of a "file". Whereas all symbols in Java or C# must either be or be contained in *classes*, Kotlin allows properties and functions to be declared at the *top level*:

```
class Test {}

val foo = "This is a top-level property"
fun thisIsATopLevelFunction() {}
```

Top-level symbols are placed in the current package's scope, unless they are `private` — private top-level symbols are only visible within the same file. They can be imported with `[package-name].[symbol-name]`, similarly to classes:

```
// File A
package a

val foo = "This is a top-level property"

// File B
package b
import a.foo

fun main() = println(foo) // "This is a top-level property"
```

Kotlin files typically have the extension `.kt`.

The Kotlin compiler, `kotlinc`, supports dynamic execution of *Kotlin scripts*. Kotlin scripts have the extension `.kts` and do not require a `main` function; all statements found at the top level are executed sequentially.

> If you *do* use a `main` function in a Kotlin script, nothing will happen when it is executed. This is because you are essentially defining a function that is never called, since `main` no longer has any special meaning.

Getting started

You can download `kotlinc`'s binaries here. If you prefer package managers, you can also install it with `pacman` (Arch Linux), Homebrew (macOS/Linux), Snappy (primarily Ubuntu), or Chocolatey (Windows).

`kotlinc` isn’t necessary to compile or run Kotlin code, though. JetBrains' IntelliJ IDEA, unsurprisingly, has first-class Kotlin support built in to the IDE. An easy way to play around with Kotlin is to create a `.kts` *scratch file* (*`Ctrl+Alt+Shift+Insert`*). If you want to create a full Kotlin project, you can easily do so by using JetBrains' Kotlin Maven archetype or by selecting "Kotlin/JVM" in the Gradle project creation dialog. This will generate a full, working Maven project for you with a sample Kotlin entry point and test class.

While there exists an equivalent plugin for Eclipse, it unfortunately tends to be updated quite infrequently, is prone to bugs, and is usually out of date.

`new`

Kotlin does not have a `new` operator. Constructors are called using standard function-call syntax:

```
val person = Person("Test", "Testerson")
```

This is a nice time-saver, and is consistent with C++ syntax, where the keyword originates.

In C++, `new` is used to allocate dynamic memory, call a constructor, and return a pointer, or to otherwise create the object on the stack and return an rvalue (more specifically, a prvalue), which is essentially a value that may be appear on the **r**ight-hand side of an expression. As there is no real functional distinction between heap- and stack-allocated values on the JVM, since one generally does not have to deal with pointers, there is no reason to keep the keyword.

`?`

`?` is an integral part of Kotlin's type system; `?` designates a type as *nullable*:

```
val foo1: String = "bar"  // ok
val foo2: String? = "bar" // ok
val foo1: String = null   // ERROR
val foo2: String? = null  // ok
```

Non-nullable types cannot have the value `null` assigned to them! This is one of Kotlin's advantages — it is extremely difficult to write proper Kotlin code that throws a `NullPointerException`.

Unlike reference/value wrapper types (*cf.* Java's `Optional` and C#'s `Nullable`), Kotlin's nullable values are a real part of the type system and play well with other language features such as generics. This means that using nullable values has no overhead; `T?` and `T` both get compiled to a simple nullable `T` in bytecode.

Under the hood, **non-nullability** is ensured by inserting null-check assertions into methods, constructors, and properties. The compiler will even add the appropriate `@Nullable`/`@NotNull` annotations. This means that no calling code can ever violate these preconditions, which allows a Kotlin library to be safely consumed from other Java/JavaScript/native modules.

Nullable types come with their own useful utilities.

`?.`, the *safe call* operator, can be used to perform operations on nullable values. If the value is `null`, it performs the operation; otherwise, it too returns `null`. This is useful for chaining methods on values that may be null:

```
val input: String? = readLine()
val enteredInt =
    input            // String?
    ?.trim()         // String? -> String?
    ?.toIntOrNull() // String? -> Int?

if (enteredInt == null) println("You did not enter an integer")
```

The Elvis operator (`?:`, try turning it 90° clockwise) is frequently used as the last element in a `?.` chain to return a fallback value. The result of an `?:` expression returns the left operand if it is not `null`; otherwise, it returns the right operand.

This is equivalent to the `??` operator in C#.

```
val envvar: String? = System.getProperty("FOO")
val displayValue: String = envvar ?: "No value found" // Not nullable!

if (enteredInt == null) println("The value of FOO is: $displayValue")
```

If you *really* need to force the compiler to dereference a nullable value, the `!!` operator can be used for this purpose:

```
val maybeFoo: Foo? = retrieveMaybeFoo()
val foo = maybeFoo!!
```

Note that this will throw an exception if the value is indeed `null`. Unless you are dealing with complex scenarios (e.g. reflection) where you can be *absolutely sure* that a value will not be null, **never** use this operator. There is always a better way to solve nullability issues, such as using the aforementioned safe call and Elvis operators, combined with `return` or `throw` expressions.

`Unit`

Kotlin, like many functional languages, does not have the concept of "no return type"; every function must return a value. So how do we deal with `void` methods?

Kotlin represents the unit type as Unit. This is equivalent to `()` in Rust or Haskell, and `Unit` in Swift. `Unit` is a singleton value that holds no information, making it a perfect choice for methods that return nothing. It is automatically returned from blocks of code that do not contain a `return` expression:

```
val value = run {}
println(value) // kotlin.Unit
```

It also plays extremely well with generics! Previously, to create a `void` Callable in Java, one would have to specify the type parameter as `void`'s peculiar wrapper type, Void, and then manually return `null` from the implementation of the `call` method:

```
new Callable<Void>() {
    Void call() {
        foo();
        return null;
    }
}
```

This is redundant! Since there exist no valid instances of `Void`, there is no use in returning any sort of value. Furthermore, the client of this API would need to know to discard the returned value.

Fortunately, since `Unit` is implicitly returned, all we need to do in Kotlin is:

```
Callable<Unit> { foo() }
```

This also enables function chains returning `Unit` to compose nicely.

This particular example makes use of SAM conversions.

`Nothing`

While Nothing as a type is fundamentally similar to `Void`, they are extremely different in terms of usage.

A function returning `Nothing` will never return. This is primarily used for functions that will always throw exceptions (i.e. exception helpers), or that will loop forever. All statements following an expression that returns `Nothing` will never execute:

```
fun throwDataException(error: String): Nothing {
    throw DataException("SQL error: $error")
}

try {
    doDatabaseStuff()
} catch(e: SQLException) {
    throwDataException(e.message)
    foo() // Warning: unreachable code
}
```

This is used quite effectively in the standard library by the utility function TODO, often used during development to mark sections of code that are not implemented and should throw an error.

```
if (foo()) {
    handleFoo()
} else {
    // Not done with this yet
    TODO("handleNotFoo()")
    //^ NotImplementedError: "An operation is not implemented:
handleNotFoo()"
}
```

Since `Nothing` cannot hold a value, and `T?` is a union between `T` and `null`, the type `Nothing?` can be used to hold a value that is always null.

`Any`

`Any` is Kotlin's equivalent to Object — it is the implicit base class for all types. It is functionally equivalent to `Object`, except that most of its methods have been removed:

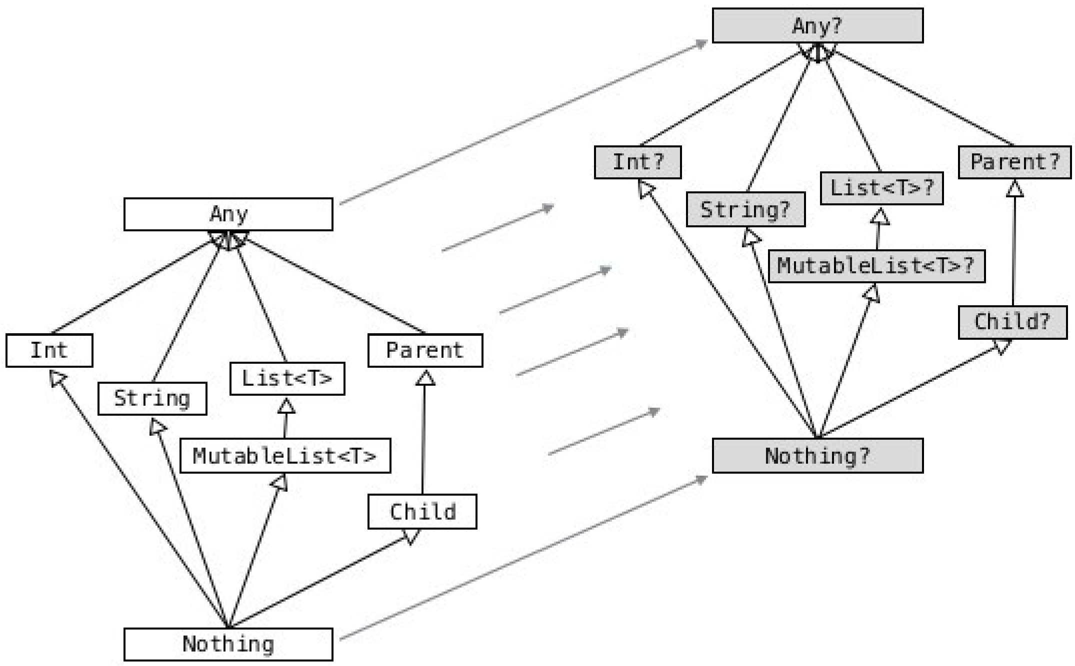
- `clone`
    - Implement `Cloneable` instead, if you *really* need `clone`.

- `finalize`
- `wait`, `notify`, `notifyAll`
  - Use of these methods has been discouraged for years — Kotlin has simplified things by removing them outright.
- `getClass`
  - This method has been replaced by the `::class` operator.

If you need to use any of `Object`'s methods, you can force the compiler to make them visible by casting an object to `Object`:

```
val foo = ...
(foo as java.lang.Object).notify()
```

Kotlin's type hierarchy



The base type for all other types in Kotlin is `Any`. All nullable types are subtypes of their respective non-nullable types. This is important since it allows nullable types to hold a regular, non-null value.

`Nothing`, the type discussed earlier, is at the bottom of the type hierarchy; it is considered a subtype of every other type, meaning that a variable of type `Nothing` cannot be implicitly assigned to.

The only expressions in Kotlin that return `Nothing` are:

- `return`
- `throw`
- `continue`
- `break`

Yes, `return` returns a value! This allows us to extremely easily handle precondition failures, and is a very common Kotlin idiom:

```
fun login(user: User): Boolean {
    val username = user.name ?: return false // User has no name, don't try to log in
    val token = doLogin(user) ?: throw LoginException("Could not log in")
    return true // Success
}
```

In this case, `?:` will either return the preceding value or execute the right-hand expression, forcing the function to return prematurely without too much boilerplate code. This can also be used with `continue` or `return` to prematurely end the loop body.

Of course, this allows us to write meaningless code:

```
return return throw return throw throw return return throw return
```

While the compiler will warn that each of the expressions (except the last) is unreachable, this is valid code.

> It should hopefully be clear that code like this should never be written.

Statements and expressions

Generally, *expressions* are snippets of code that have a *value*. Statements, on the other hand, do not necessarily have any sort of resulting value.

Apart from declarations and assignments, everything in Kotlin is an expression:

```
val password = readLine()
val output = when (password) {
    "hunter2" -> "Authenticated!"
    else -> "Hacker detected!"
}
```

Even an `if` statement returns a value:

```
println(
    if (room.isSmoking) "This is a smoking room"
    else "This is a no-smoking room"
)
```

This is incredibly versatile, since it is possible to place multiple statements within the `if` statement's block — every *block* in Kotlin also returns a value!

> Because `if` is an expression, Kotlin does not have the `? :` ternary operator.

The result of the last statement in a block implicitly becomes the result of the block itself. If the last statement is not an expression, it returns `Unit` instead:

```
val value = run {
    val foo = 40
    foo + 2
}
print(value) // 42
```

> Unlike in most other C-like languages, assignments are not expressions. This means many classic sources of programmer error can be eliminated:

```
_Bool ok = doSomething(...);
if (ok = true) { // = instead of ==, this will always get executed!
    printf("Success\n");
} else {
    // This will never get executed!
    printf("An error occured\n");
    abort();
}
```

Visibility modifiers

Kotlin has the following visibility modifiers:

- `public`
- `internal`
- `protected`
- `private`

`public`, `protected` and `private` members work as they do in Java and C++. `private` *top-level symbols* are visible everywhere in the same file.

> Top-level symbols cannot be `protected`, as this would not make any sense — they do not have anything to do with inheritance.

Unlike Java, Kotlin does not have package-private (default) access. It replaces this with `internal` access, which makes a symbol visible to all other classes *in the same module*. Files outside a project (i.e. in other modules) will not be able to access an `internal` symbol.

> The **default access modifier** for a symbol, when one is not specified, is `public`! This means specifying `public` explicitly is almost always redundant.

Hello, world!

As with any other programming language, to write an executable program we need an entry point. A Kotlin program's entry point is a top-level function called `main`. As many programs do not make use of command-line arguments, the `args` parameter is optional. This means a "hello world" program could look something like:

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

or

```kotlin
fun main() {
    println("Hello, world!")
}
```

Our golfing opportunities don't end here, though. In the interest of enabling terse, functional programming, there exists a shorter syntax for functions that consist of and return a single expression:

```kotlin
fun main() = println("Hello, world!")
```

Basic types

"Primitives"

> Everything in Kotlin is a class! There are no primitive types; the compiler will automatically deal with boxing/unboxing for you, making working with generics a breeze.

Kotlin provides Java's eight primitive types as built-in classes:

- `kotlin.Byte` — 8-bit signed integer
- `kotlin.Short` — 16-bit signed integer
- `kotlin.Int` — 32-bit signed integer
- `kotlin.Long` — 64-bit signed integer
- `kotlin.Float` — 32-bit IEEE 754 floating-point
- `kotlin.Double` — 64-bit IEEE 754 floating-point
- `kotlin.Boolean` — true/false
- `kotlin.Char` — 16-bit Unicode character (**not** a numeric type!)

In addition to these types, Kotlin also provides four *unsigned integer* types:

- `kotlin.UByte` — 8-bit unsigned integer
- `kotlin.UShort` — 16-bit unsigned integer

- `kotlin.UInt` — 32-bit unsigned integer
- `kotlin.ULong` — 64-bit unsigned integer

The format of numeric literals in Kotlin is almost identical to that of Java; octal literals have been removed.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
val creditCardNumber = 1234_5678_9012_3456L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
val bool = true
val char = 'A'
val uint = 20U // UInt
val ulong = 9082348590UL // ULong
val pi = 3.14 // Double
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float
```

[1]

Values of the built-in types cannot be converted between each other implicitly, since in other languages this is a common cause of error or loss of precision. To do so one must use the built-in `toXXX` functions:

```
val char = 'A'
val double: Double = char * 2 // ERROR
val double: Double = char.toDouble() * 2 // ok
```

Often, this is not necessary because the standard arithmetic operators have overloads for most appropriate conversions [2]. For example, adding a `Long` and an `Int` will compile without any conversions, and will result in a `Long` value.

The conversion functions are compiler intrinsics and thus do not introduce any function call overhead.

Arrays

Arrays in Kotlin are represented by the generic `kotlin.Array<T>` class. For performance reasons, primitive arrays have their own specialized types. These map directly to the respective primitive array type (i.e. `IntArray` = `int[]`).

Kotlin does not have array literals (they exist, but are only allowed in annotations); to create an array you can use the built-in `arrayOf` function:

```
val strings: Array<String> = arrayOf("Hello", "world")
val ints: IntArray = intArrayOf(1, 2, 3)
```

Strings

Strings are defined using double quotes.

String templates (also known as string interpolation) are the recommended way to concatenate values with a string:

```
val foo = 3
println("The value of foo is $foo") // "The value of foo is 3"
println("Half of foo is ${foo / 2}") // "Half of foo is 1"
println("Half of foo is ${foo.toDouble() / 2}") // "Half of foo is 1.5"
```

[3]

Multi-line (raw) strings, which have only recently found their way into Java with the publication of JEP 368, can be defined using triple quotes:

```
val text = """
foo
bar
"""
```

Multi-line strings do not support escape characters. If you need to escape a `$` in a raw string, you can replace it with `${'$'}`.

To achieve nicer indentation, you can add a margin and trim it:

```
val text = """
    foo
    bar
""".trimIndent()
```

Don't worry about performance when using `trimIndent` — this function is actually a compiler intrinsic.

Collections

The base class of all collections in Kotlin is `Collection<T>`. Unlike Java, Kotlin's collections provide no mutating operations:

```
val list: List<Int> = ArrayList<Int>()
list.add(3) // ERROR: List has no add method
```

Only `Collection`'s subtype, `MutableCollection`, allows the object to be mutated. The same applies to all of the built-in collection types.

Similarly to arrays, you can create collections using the builder functions `xxxOf`:

```
val unmodifiableList = listOf(1, 2, 3)
val modifiableList = mutableListOf(1, 2, 3)
val arrayList = arrayListOf(1, 2, 3)
val hashMap = hashMapOf("key" to "value", "hello" to "world") // Takes
a list of Pair<,>s
val set = setOf(1, 2, 3, 2) // {1, 2, 3}
```

The syntax `"key" to "value"` makes use of the infix function `to`, which takes two values and constructs a `Pair` from them. You could also write `hashMapOf(Pair("key", "value"), Pair("hello", "world"))`, but it would be less readable.

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!

> 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

```
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

```
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:

```kotlin
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.

> 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 class`es

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*!

```kotlin
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 reason 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.

> 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.

> 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 Java uses the `Outer.this` syntax, Kotlin uses labels.

```
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.

```kotlin
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.

```kotlin
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

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

instead of

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

Functions and properties

Functions

Kotlin functions are declared using `fun`:

```kotlin
fun functionName(param: Type): Type {...}
```

Parameters, along with all other values in Kotlin, are defined using *Pascal notation*. This means that the type comes after the symbol name.

Parameters can also have *default arguments*, which helps to reduce the need for overloads.

```
fun makePurchase(amount: Int = 1) {
    ...
}
```

As with properties, there are two types of local variables: `val` (immutable) and `var` (mutable). The type of a variable can be omitted, and it will be inferred from its initializer:

```
fun foo() {
    // These are equivalent:
    val a: Int = 3
    val a = 3
}
```

If no return type is specified, it is inferred to be `Unit`. Similarly, if a `return` statement is made with no value, it will implicitly return `Unit`:

```
fun foo() { // Equivalent to fun foo(): Unit
    return // return Unit
}
```

If no return statement is present in the body of a function, and the function is specified to return `Unit`, it is implicitly returned.

If a function's body consists of a single expression, you can make use of a special, shorter syntax:

```
override fun toString() = "..."
```

Single-expression functions can have their return types omitted; they will be inferred from the expression.

Variadic arguments

A parameter can accept an arbitrary number of arguments if it is annotated by `varargs`:

```
fun takeLotsOfStrings(vararg strings: String) {
    for (string in strings) println(string)
}
```

Within the function, `varargs` parameters are treated as arrays and can be used as such.

To pass the contents of an array to a `varargs` parameter, you can use the *spread operator* `*`:

```
val strings = arrayOf("Hello", "world")
// The spread operator can even interleave varargs parameters!
takeLotsOfStrings("Something", *strings, "else")
```

> No more than one parameter per function can be `varargs`. If a parameter is `varargs`, it is not required to be the last parameter. Values for the following parameters can be passed using named arguments, or in the case of a lambda parameter, it may be passed outside the function call parentheses.

Named arguments

When invoking a function, parameters can be referred to by name. This is helpful when invoking a function with **multiple default parameter values**:

```
fun joinToString(
    strings: Iterable<String>,
    separator: CharSequence = ", ",
    prefix: CharSequence = "",
    postfix: CharSequence = "",
    limit: Int = -1,
    truncated: CharSequence = "...",
    transform: ((T) -> CharSequence)? = null
): String { ... }

...

joinToString(listOf("Hello", "world"), separator = "; ", truncated = "
(more)")
```

If named arguments had not been used, redundant values for all intermediate parameters would have had to have been specified.

Extension functions

It is possible to define *extension functions* for external classes. Unlike C#, there is no requirement that they be placed in a "static class"; they are typically placed at the top level.

```
fun Float.inverse(): Float {
    return 1 / this
}

print((3.0f).inverse()) // 0.33333334
```

Within an extension function, `this` refers to the receiver.

> Extension functions can have nullable receivers! This is why you can call `toString` on a `null` value — `toString` is a standard extension function defined as `Any?.toString(): String`.

Properties

Kotlin, like Swift, has no concept of *fields*. All instance attributes are automatically encapsulated in properties.

Properties may be backed by fields internally; this is the default. They may also have custom accessors.

The syntax for defining a property is:

```
[var/val] name: Type = [initialValue]
    get
    set
```

This notation, where the type follows the name, is called *Pascal notation*.

`var` properties are *mutable*; they have getters and setters, whereas `val` properties only have getters. By default, the getter and setter can be omitted, unless you wish to apply an access modifier or annotation to it.

```
var property: Any? = null
    @Inject internal set
```

Getters and setters may also have bodies, as in C#:

```
var first = "Name"
var last = "Nameington"

var fullName: String
    get() {
        return "$first $last"
    }
    set(value) {
        val split = value.split(" ")
        first = split[0]
        last = split.skip(1).joinToString(" ")
    }
```

If the body of a getter or setter contains a single expression, the braces may be omitted, similarly to functions:

```
val fullName: String
    get() = "$first $last"
```

The type of the property may usually be omitted; it will be automatically inferred from its getter or initial value.

```
// These are equivalent:
val foo: Int = 3
val foo = 3
```

Backing fields

Within the body of a field-backed property's getter or setter, a **magic variable** called `field` is accessible. This is a mutable reference to the property's backing field, and can be used to easily add additional logic to a property setter.

```
var positiveInt: Int = 1
    get
    set(value) {
        if (value > 0) field = value
    }
```

If a property is not given an initial value, or if `field` is never used, the property will not have a backing field; it is purely compiled to a getter (and a setter, if the property is a `var`). This is useful for creating e.g. "compound properties" (like the aforementioned `fullName`) that should not be stored in memory and are the result of performing cheap operations.

Properties with no backing field can be declared as `inline`; no getters or setters will be generated, as they will be inlined into the calling code.

`lateinit` properties

The `lateinit` modifier allows a property or local variable of a non-nullable type to initially have no value. This is especially useful for classes that do not have their fields initialized at construction time; Android activities, test fixtures, or Spring services are common examples.

```
@Test
class FooServiceTest {
    lateinit var fooService: FooService

    @BeforeClass
    fun init() {
        this.fooService = ...
    }
}
```

Without `lateinit`, the property would have to be declared as nullable because its initial value is `null`; every operation on it would either have to make use of the `!!` operator or use unnecessary `?.` chaining.

Since Kotlin 1.2, local variables can also be `lateinit`.

Extension properties

It is also possible to define *extension properties* that act identically to extension functions:

```kotlin
val Float.integerPart: Float get() = this.toInt().toFloat()

print((2.6543f).integerPart) // 2.0
```

Compile-time constants

Kotlin supports compile-time constants similar to `public static final`s in Java or `static const`s in C/C++, which can also be used in annotations:

```kotlin
const val PROGRAM_NAME = "MyApp"

@ProgramName(PROGRAM_NAME)
class Foo
```

`const val`s must be initialized to a literal value. They are usually inlined by the compiler into code that uses them, for performance reasons.

Nesting

Local functions and classes

Classes and functions can be declared *locally*, that is, within other functions:

```kotlin
fun foo() {
    fun bar() {
        ...
    }
    class Quux

    bar()
}

fun baz() {
    bar() // ERROR
    Quux::class // ERROR
}
```

This is an invaluable tool — the scope of symbols should be restricted as much as possible, and if a certain subroutine or data class is only needed within a function, it is a great idea to make them local to that function.

Classes, functions, properties, and inheritance

Kotlin classes, functions and properties are `final` by default. They can be made virtual by adding the `open` modifier:

```
open class Foo // This class can be extended!
```

Abstract classes and functions do not need to be declared as `open`, as this would defeat their purpose.

> Many frameworks require certain types to be extensible. Spring, for example, requires classes to be `open` for its AOP tools to work properly. You can easily get around this by adding JetBrains' all-open compiler plugin to your project. The plugin can be configured to make all classes open by default, or only the classes that have a certain annotation (i.e. `@Component`).

Inheritance

To extend a class, add it after the type name using the C++-style extension syntax:

```
class Derived : Base
```

The base class must be initialized in the class header:

```
abstract class Base

class Derived : Base() // Primary constructor is called
```

Alternatively, if the base class has no primary constructor, its secondary constructors can chain to `super`:

```
class Derived : Base {
    constructor() : super()
}
```

This ensures that the superclass constructor has finished by the time the subclass's initializers run.

> Interfaces cannot be initialized in the class header because they do not have constructors.

`abstract` and `override`

Functions and properties can also be `abstract` members of abstract classes and interfaces. Interface functions are implicitly `abstract`. Default implementations for interface functions are easy to specify — just give the function a body:

```
interface Comparer<T1, T2> {
    fun compare(a: T1, b: T2): Boolean {
        return true // Default implementation always returns true
    }
}
```

To override a function or property, declare it in the subtype using the `override` modifier.

> Unlike in Java, where `@Override` is an annotation, `override` is a keyword in Kotlin.

To override a member and prevent further overriding, declare it as `final`, like in C++:

```
interface A {
    // `abstract` is implied, since this is an interface
    fun foo()
    val bar: String
}

open class B : A {
    final override fun foo()
    override val bar get() = "Baz"
}

class C : B {
    override fun foo() // ERROR
    override val bar get() = "Quux" // ok
}
```

`override` can even be used on primary constructor property parameters! This is especially useful for use with data classes.

Explicit `super`

If a class inherits the same member from multiple supertypes, it must provide its own implementation to avoid the diamond problem: [4]

```
open class Rectangle {
    open fun draw() { ... }
}

interface Polygon {
    fun draw() { ... } // Default implementation
}

class Square : Rectangle(), Polygon {
    // The compiler requires draw to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw
        super<Polygon>.draw() // call to Polygon.draw
    }
}
```

Function objects

Kotlin treats functions as first-class language citizens; they can be stored in variables and passed around. To accomplish this, Kotlin uses *function types*:

```
val intConsumer: (Int) -> Unit = fun(int: Int) {
    println(int)
}
```

There is a shorter, more concise syntax for creating an anonymous function — the *lambda expression*.

```
val intConsumer: (Int) -> Unit = { int ->
    println(int)
}
```

Kotlin lambdas are entirely contained within `{}`; like in Swift, the parameters are declared within the body itself:

```
{
    arg1, ... ->
    [lambda body]
    [last expression, or implicit Unit]
}
```

[5]

The value of the last expression of a lambda is implicitly returned, unless the return type of the lambda is `Unit`, in which case `Unit` is always returned.

> If a lambda has only one parameter, the parameter list can be left out; the parameter receives the special name `it`.

> Avoid using implicit parameter names with nested lambdas — it quickly becomes unclear just *which* `it` is being referred to.

Lambdas as function parameters

Lambdas can be passed to functions as parameters. There are many functions in the standard library which accept lambda expressions; most of their charm comes from the fact that *the last argument, if it is a lambda expression, can be placed outside of the function call.*

This might not sound like much at first, but this:

```
val list = listOf(1, 2, 3)
list.forEach({
    println(it)
})
```

becomes

```
val list = listOf(1, 2, 3)
list.forEach {
    println(it)
}
```

This is what helps many of Kotlin's standard library functions fit into the language so well — they *look* like they could be built-in language features, but they are really just normal, convenient functions.

A great example of this is repeat:

```
// For loop
for (i in 0 until 7) {
    foo()
}

// `repeat`
repeat(7) {
    foo()
}
```

`repeat` is much more readable, looks like a built-in keyword, and the intent of the code becomes much clearer at no additional cost.

Inlining lambdas

Functions can be declared as `inline` — this means that they will not be compiled to an actual function; their code will simply be pasted into the call site wherever it is used. The usefulness of this feature becomes apparent when it is combined with lambdas. Literal lambdas passed to inline methods will also become inlined:

```
inline fun run(block: () -> Unit) {
    block()
}

run { println("Hello, world!") }
```

is compiled to simply

```
println("Hello, world!")
```

This removes the entire object overhead of lambda expressions, and allows useful functional utilities to be built.

> Most standard-library functions that affect control flow (i.e. `repeat`, `forEach`, `map`, etc.) are inlined and thus incur no performance penalties!

If you need a lambda to remain in "object form" and not be inlined into an inline function (e.g. if you need to store it in a list), you can annotate the parameter as `noinline`.

If an inlined lambda needs to be cross-inlined into *another* inline function (e.g. this inline function calls another one), you must annotate the parameter as `crossinline`.

Because inline lambdas are inserted directly into the call site, `return`ing from them may have unexpected behavior!

```
fun loopList() {
    list.forEach { item ->
        if (item == 3) return // This will return from the ENTIRE
function,
                              // not just from the lambda!
    }
}
```

This is called a *non-local return*, which is often seen in standard loops. It is important to keep in mind that `return` has no meaning within a lambda and always affects the enclosing function scope. Generally, it is therefore advisable to avoid using `return` within a lambda unless necessary.

`reified` generics

Inline functions can be used to provide compile-time reified generics.

Since the functions are inlined, they have access to generic type data:

```
inline fun <reified T> checkType(any: Any): Boolean {
    return any is T // This would not work in Java,
                    // or in a non-inline function!
}
```

This allows `is`, `as`, and `::class` to be safely used on generic type parameters.

Anonymous objects

Objects of an anonymous type can be created using `object` literals.

```
val runnable = object : Runnable {
    override fun run() {
        foo()
    }
}
```

However, when using SAM (single-abstract-method) interfaces that are defined in Java code, this is unnecessary, because Kotlin will automatically create helper constructors for these interfaces to allow for a nicer syntax. This is called *SAM conversion*:

```
val runnable = Runnable {
    foo()
}
```

Additionally, anonymous objects that do not extend any class can be created. This is similar to C#'s `new {}`.

```
val list = listOf(3)
val mapped = list.map { int ->
    object {
        val value = int
    }
}

for (item in mapped) {
    println(item.value)
}
```

While the type cannot be referred to by name, it is available to the compiler and can thus be used within the same scope. Since anonymous objects have no proper type, they cannot be returned from methods.

It is often better to use local data classes instead of untyped anonymous objects, as they are named and they more clearly express the intent of the code.

```
fun foo() {
    data class TempData(...) // Local data class!
}
```

[1] https://kotlinlang.org/docs/reference/basic-types.html

[2] https://kotlinlang.org/docs/reference/basic-types.html#explicit-conversions

[3] This is somehow the context for my second-most popular StackOverflow answer: https://stackoverflow.com/a/48800990

[4] https://kotlinlang.org/docs/reference/classes.html#overriding-rules

[5] This is largely paraphrased from my StackOverflow answer here.

Control flow

`for` loops

Kotlin only provides for loops that iterate through iterable objects; that is, any object that has an `operator iterator` function.

```
for (item in collection) { ... }
```

The standard C-style loop is not supported. To iterate through a range of indices, you can take the classic approach:

```
var index = 0
while (index < count) {
    ...
    ++index
}
```

or make use of Kotlin's ranges.

Ranges

The syntax of a range is `lower..upper`; ranges are inclusive, unlike in languages like Python or Swift, they include both of their bounds.

```
for (index in 1..count) { ... }
for (index in 0..(count - 1)) { ... }
```

Since ranges are end-inclusive, the standard library provides the infix function `until` to simplify the creation of exclusive ranges.

```
for (index in 0 until count) { ... }
```

With the standard range-manipulating infix functions, you can create all sorts of ranges — backwards ranges, ranges that skip values, you name it.

```
println((1..10).toList()) // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
println((1 until 10).toList()) // [1, 2, 3, 4, 5, 6, 7, 8, 9]
println((10 downTo 1).toList()) // [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
println((10 downTo 1 step 3).toList()) // [10, 7, 4, 1]
```

In the special case of iterating over an array's indices and elements, you can use `forEachIndexed`:

```
val list = listOf(1, 2, 3, 4)
list.forEachIndexed { index, i -> println("$index: $i") }
```

> If you need to do any complex manipulation of the current loop index, you will have to fall back to a `while` loop and a manual index. This can often be avoided; for common use cases, there is often a standard library function that does what you want. windowed can be used for the common "rolling window" approach, for example.

`while` / `do…while`

These loop types function as they do in C.

`when`

`when` is Kotlin's replacement for the switch statement. It is much more powerful and does not suffer from accidental fall-through; in fact, **fall-through is impossible**!

You can match values of any type, check the type of a value, or check if it is contained in a collection (the `in` operator):

```
val x = readLine()?.toIntOrNull()!!
when (x) {
    1 -> println("x = 1")
    in 12..38 -> println("x is between 12 and 38")
    !in 3..5 -> println("x is not between 3 and 5")
    else -> println("None of the conditions were satisfied")
}

val anObject: Any = ...
when (anObject) {
    is String -> println("The object is a string")
    is Number -> println("The object is a number")
    else -> println("I don't know what type the object is")
}
```

Kotlin's `when` isn't quite as powerful as Scala's `match` expression (there is no support for pattern matching), but it's a nice step up from Java.

Nested scopes (labels)

Similarly to Java, Kotlin can make use of *labels* to return from a nested scope:

```
outer:
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        if (i == 1 && j == 2) break outer;
    }
}
```

would become:

```
outer@for (i in 0 until 3) {
    for (j in 0 until 3) {
        if (i == 1 && j == 2) break@outer
    }
}
```

Labels can also be used to return from lambdas. We can use this to fix our earlier problem with non-local returns:

```kotlin
fun loopList() {
    list.forEach { item ->
        if (item == 3)
            return@forEach // This will only return from the lambda,
                           // skipping the current item,
                           // so this is equivalent to a `continue`
    }
}
```