Kotlin

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2018-2019)

Kotlin [https://kotlinlang.org/] [https://developer.android.com/kotlin/]

A modern language (like, e.g., Scala and Swift) with advantages over Java:

- **More concise**: Drastically reduce the amount of boilerplate code.
- **Safer**: Avoid entire classes of errors such as null pointer exceptions.

Moreover:

- Interoperable: Leverage existing libraries for the JVM, Android, and the browser.
- Tool-friendly: Choose any Java IDE or build from the command line.

See also [https://kotlinlang.org/docs/reference/comparison-to-java.html]

Outline

Outline

Coding conventions

A coding style guide about:

- Source code organization
- Naming rules
- Formatting
- Documentation comments
- Avoiding redundant constructs
- Idiomatic use of language features
- Coding conventions for libraries

is available at [https://kotlinlang.org/docs/reference/coding-conventions.html]

Defining packages

A source file may start with a package declaration:

```
1 package foo.bar
2
3 import java.util.*
4
5 fun baz() { ... }
6 class Goo { ... }
7
8 // ...
```

- Source files can be placed arbitrarily in the file system.
- All the contents (such as classes and functions) of the source file are contained by the package declared. So, in the example above, the full name of baz() is foo.bar.baz, and the full name of Goo is foo.bar.Goo.
- If the package is not specified, the contents of such a file belong to "default" package that has no name.
- A number of packages are imported into every Kotlin file by default.

Defining functions

Function having two Int parameters with Int return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Function with an expression body and inferred return type:

```
fun sum(a: Int, b: Int) = a + b
```

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Unit return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

Default Arguments

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { ... }
```

Named Arguments

```
fun foo(bar: Int = 0, baz: Int) { ... }

foo(baz = 1) // The default value bar = 0 is used
```

With named arguments we can make the code much more readable:

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

and if we do not need all arguments:

```
reformat(str, wordSeparator = '_')
```

Defining variables

Assign-once (read-only) local variable:

```
val a: Int = 1  // immediate assignment
val b = 2   // ‘Int‘ type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

Mutable variable:

```
var x = 5 // ‘Int‘ type is inferred
x += 1
```

Top-level variables:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

Local Functions (i.e. a function inside another function)

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: Set<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }
    dfs(graph.vertices[0], HashSet())
}
```

Local function can access local variables of outer functions (i.e. the closure), so in the case above, the visited can be a local variable:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }
    dfs(graph.vertices[0])
}
```

Using nullable values and checking for `null`

A reference must be explicitly marked as nullable when null value is possible.
Return `null` if `str` does not hold an integer:

```
1 fun parseInt(str: String): Int? {
2     // ...
3 }
```

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using ‘x * y‘ yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("either ’$arg1’ or ’$arg2’ is not a number")
    }
}
```

or

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

Generic functions and Variable number of arguments

Generic Functions

```
fun <T> singletonList(item: T): List<T> { ... }
```

Variable number of arguments: at most one parameter of a function (normally the last one) may be marked with vararg modifier:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

allowing a variable number of arguments to be passed to the function:

```
val list = asList(1, 2, 3)
```

Inside a function a vararg-parameter of type T is visible as an array of T, i.e. the ts variable in the example above has type Array<out T>.

QUESTION: what is the meaning of the "out" keyword?

Outline

Everything is an object

Everything is an object: we can call member functions and properties on any expression.
QUESTION: what are "member functions" and "properties"?
Some of the types can have a special internal representation (e.g., numbers, characters and booleans can be represented as primitive values at runtime) but to the user they look like ordinary classes.
A presentation about:

- Numbers
- Characters
- Booleans
- Arrays
- Unsigned integers
- Strings

is available at:
`https://kotlinlang.org/docs/reference/basic-types.html`

Representation

On the Java platform, numbers are physically stored as JVM primitive types, unless we need a nullable number reference (e.g. `Int?`) or generics are involved. In the latter cases numbers are boxed.

Note that boxing of numbers does not necessarily preserve identity:

```
1 val a: Int = 10000
2 println(a === a) // Prints 'true'
3 val boxedA: Int? = a
4 val anotherBoxedA: Int? = a
5 println(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

On the other hand, it preserves equality:

```
1 val a: Int = 10000
2 println(a == a) // Prints 'true'
3 val boxedA: Int? = a
4 val anotherBoxedA: Int? = a
5 println(boxedA == anotherBoxedA) // Prints 'true'
```

Explicit Conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)
print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the
     other is Long
```

Smaller types are NOT implicitly converted to bigger types. This means that we cannot assign a value of type`Byte` to an `Int` variable without an explicit conversion

```
val b: Byte = 1 // OK, literals are checked statically
val i: Int = b // ERROR
```

We can use explicit conversions to widen numbers

```
val i: Int = b.toInt() // OK: explicitly widened
print(i)
```

Arrays

Arrays are represented by the Array class, that has `get` and `set` functions (that turn into `[]` by operator overloading conventions), and `size` property, along with a few other useful member functions:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

To create an array, we can use a library function `arrayOf()` and pass the item values to it, so that `arrayOf(1, 2, 3)` creates an array `[1, 2, 3]`. Alternatively, the `arrayOfNulls()` library function can be used to create an array of a given size filled with null elements.

Another option is to use the `Array` constructor that takes the array size and the function that can return the initial value of each array element given its index:

```
1 // Creates an Array<String> with values ["0", "1", "4", "9", "16"]
2 val asc = Array(5, { i -> (i * i).toString() })
3 asc.forEach { println(it) }
```

Unlike Java, **arrays in Kotlin are invariant.** This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure (but you can use `Array<out Any>`.

QUESTION: what is `Any`?

Kotlin also has specialized classes to represent arrays of primitive types without boxing overhead: `ByteArray`, `ShortArray`, `IntArray` and so on. These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```
1 val x: IntArray = intArrayOf(1, 2, 3)
2 x[0] = x[1] + x[2]
```

Strings

Strings are represented by the type String. Strings are immutable. Elements of a string are characters that can be accessed by the indexing operation: `s[i]`. A string can be iterated over with a for-loop:

```
for (c in str) {
    println(c)
}
```

Escaping is done in the conventional way, with a backslash.
A raw string is delimited by a triple quote (`"""`), contains no escaping and can contain newlines and any other characters:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

String Templates

Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign ($) and consists of either a simple name:

```
1 val i = 10
2 println("i = $i") // prints "i = 10"
```

or an arbitrary expression in curly braces:

```
1 val s = "abc"
2 println("$s.length is ${s.length}") // prints "abc.length is 3"
```

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal $ character in a raw string (which doesn't support backslash escaping), you can use the following syntax:

```
1 val price = """
2 ${'$'}9.99
3 """
```

Outline

Classes and Primary constructors

Classes in Kotlin are declared using the keyword `class`:

```
1 class Invoice { ... }
2 class Empty
```

A class can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional type parameters).

```
1 class Person constructor(firstName: String) { ... }
```

If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

```
1 class Person(firstName: String) { ... }
```

QUESTION: what are the available “annotations” and “visibility modifiers”?

Primary constructors and Initializers

The primary constructor cannot contain any code. Initialization code can be placed in **initializer blocks**, which are prefixed with the `init` keyword.
During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {
 val firstProperty = "First property: $name".also(::println)
 init {
  println("First initializer block that prints ${name}")
 }
 val secondProperty = "Second property: ${name.length}".also(::println)
 init {
  println("Second initializer block that prints ${name.length}")
 }
}
```

QUESTION: what is the “also” method?

During the instance initialization the following text is printed:

```
First property: hello
First initializer block that prints hello
Second property: 5
Second initializer block that prints 5
```

Note that parameters of the primary constructor can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

In fact, for declaring properties and initializing them from the primary constructor, there is a concise syntax:

```
cllass Person(val firstName: String, val lastName: String, var age: Int) { ... }
```

Much the same way as regular properties, the properties declared in the primary constructor can be mutable (`var`) or read-only (`val`).

Secondary constructors

The class can also declare secondary constructors, which are prefixed with `constructor`:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the `this` keyword:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Note that code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens as the first statement of a secondary constructor, so the code in all initializer blocks is executed before the secondary constructor body. Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}
```

If a non-abstract class does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments. The visibility of the constructor will be public. If you do not want your class to have a public constructor, you need to declare an empty primary constructor with non-default visibility:

```
1 class DontCreateMe private constructor () { ... }
```

NOTE: On the JVM, if all of the parameters of the primary constructor have default values, the compiler will generate an additional parameterless constructor which will use the default values.

```
1 class Customer(val customerName: String = "")
```

Class members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and Inner Classes
- Object Declarations

To create an instance of a class, we call the constructor as if it were a regular function:

```
1 val invoice = Invoice()
2
3 val customer = Customer("Joe Smith")
```

QUESTION: how to create instances of nested, inner and anonymous inner classes?

Inheritance

All classes in Kotlin have a common superclass `Any` (with members `equals()`, `hashCode()` and `toString()`), that is the default superclass for a class with no supertypes declared:

```
1 class Example // Implicitly inherits from Any
```

To declare an explicit supertype, we place the type after a colon in the class header:

```
1 open class Base(p: Int)
2
3 class Derived(p: Int) : Base(p)
```

If the derived class has a primary constructor, the base class can (and must) be initialized right there, using the parameters of the primary constructor.

QUESTION: what is the meaning of the open modifier?

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the super keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

```
class Example // Implicitly inherits from Any
```

To declare an explicit supertype, we place the type after a colon in the class header:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Overriding Methods

Things are explicit in Kotlin: explicit annotations are required for overridable members (we call them open) and for overrides:

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

A member marked `override` is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use `final`:

```
open class AnotherDerived() : Base() {
    final override fun v() { ... }
}
```

Overriding Properties

Overriding properties works in a similar way to overriding methods; properties declared on a superclass that are then redeclared on a derived class must be prefaced with override, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a getter method.

```
1 open class Foo {
2     open val x: Int get() { ... }
3 }
4
5 class Bar1 : Foo() {
6     override val x: Int = ...
7 }
```

QUESTION: what is the "get" keyword?

You can also override a `val` property with a `var` property, but not vice versa. This is allowed because a `val` property essentially declares a getter method, and overriding it as a `var` additionally declares a setter method in the derived class.

Note that you can use the `override` keyword as part of the property declaration in a primary constructor.

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

QUESTION: what about `interface` declarations?

Derived class initialization order

During construction of a new instance of a derived class, the base class initialization is done as the first step (preceded only by evaluation of the arguments for the base class constructor) and thus happens before the initialization logic of the derived class is run.

```
open class Base(val name: String) {
 init { println("Initializing Base") }
 open val size: Int =
   name.length.also { println("Init. size in Base: $it") }
}
class Derived(
    name: String,
    val lastName: String
) : Base(name.capitalize().also { println("Arg. for Base: $it") }) {
    init { println("Initializing Derived") }
    override val size: Int =
        (super.size + lastName.length).also { println("Init. size in Derived: $it")
    }
}
```

- It means that, **by the time of the base class constructor execution, the properties declared or overridden in the derived class are not yet initialized.**
- If any of those properties are used in the base class initialization logic (either directly or indirectly, through another overridden open member implementation), it may lead to incorrect behavior or a runtime failure.
- When designing a base class, you should therefore avoid using `open` members in the constructors, property initializers, and `init` blocks.

SUGGESTION: read the paper:

Gil J.., Shragai T. (2009) Are We Ready for a Safer Construction Environment?. In: Drossopoulou S. (eds) ECOOP 2009 – Object-Oriented Programming. ECOOP 2009. Lecture Notes in Computer Science, vol 5653. Springer, Berlin, Heidelberg [https://doi.org/10.1007/978-3-642-03013-0_23]

and have a look at Swift.

Calling the superclass implementation

Code in a derived class can call its superclass functions and property accessors implementations using the super keyword:

```
1  open class Foo {
2      open fun f() { println("Foo.f()") }
3      open val x: Int get() = 1
4  }
5
6  class Bar : Foo() {
7      override fun f() {
8          super.f()
9          println("Bar.f()")
10     }
11
12     override val x: Int get() = super.x + 1
13 }
```

Inside an `inner` class, accessing the superclass of the outer class is done with the super keyword qualified with the outer class name: `super@OuterClassName`:

```
class Bar : Foo() {
    override fun f() { /* ... */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f() // Calls Foo's implementation of f()
            println(super@Bar.x) // Uses Foo's implementation of x's getter
        }
    }
}
```

Overriding Rules

If a class inherits many implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones).

To denote the supertype from which the inherited implementation is taken, we use super qualified by the supertype name in angle brackets, e.g. `super<Base>`.

```kotlin
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // interface members are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // The compiler requires f() to be overridden:
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}
```

Abstract Classes

A class and some of its members may be declared `abstract`. An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with open – it goes without saying.
We can override a non-abstract open member with an abstract one

```
1 open class Base {
2     open fun f() {}
3 }
4
5 abstract class Derived : Base() {
6     override abstract fun f()
7 }
```

Declaring Properties

Classes in Kotlin can have properties. These can be declared as mutable, using the var keyword or read-only using the val keyword.

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

To use a property, we simply refer to it by name, as if it were a field in Java:

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Getters and Setters

The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer (or from the getter return type).

```
var allByDefault: Int? // error: explicit initializer required, default getter and setter implied
var initialized = 1 // has type Int, default getter & setter
```

We can write custom accessors, very much like ordinary functions, right inside a property declaration. Here's an example of a custom getter:

```
val isEmpty: Boolean
    get() = this.size == 0
```

A custom setter looks like this (by convention, the name of the setter parameter is value, but you can choose a different name if you prefer):

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other
```

Since Kotlin 1.1, you can omit the property type if it can be inferred from the getter:

```
val isEmpty get() = this.size == 0  // has type Boolean
```

If you need to change the visibility of an accessor or to annotate it, but don't need to change the default implementation, you can define the accessor without defining its body:

```
var setterVisibility: String = "abc"
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
    @Inject set // annotate the setter with Inject
```

QUESTION: what is the annotation `@Inject`?

Backing Fields

Fields cannot be declared directly in Kotlin classes. However, when a property needs a backing field, Kotlin provides it automatically. This backing field can be referenced in the accessors using the `field` identifier.

```
var counter = 0 // Note: the initializer assigns the backing field directly
    set(value) {
        if (value >= 0) field = value
    }
```

The `field` identifier can only be used in the accessors of the property.
A backing field will be generated for a property if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the `field` identifier.
For example, in the following case there will be no backing field:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Interfaces

Similar to Java 8: interfaces can contain declarations of abstract methods, or method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties but these need to be abstract or to provide accessor implementations.

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

A class or object can implement one or more interfaces.

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

QUESTION: what is a supertype "object"?

Properties in Interfaces

You can declare properties in interfaces. A property declared in an interface can either be abstract, or it can provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them.

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Interfaces Inheritance

An interface can derive from other interfaces and thus both provide implementations for their members and declare new functions and properties. Quite naturally, classes implementing such an interface are only required to define the missing implementations:

```
interface Named {
    val name: String
}
interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}
data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Resolving overriding conflicts

When we declare many types in our supertype list, it may appear that we inherit more than one implementation of the same method. For example:

```
interface A {
    fun foo() { print("A") }
    fun bar()
}
interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}
class C : A {
    override fun bar() { print("bar") }
}
class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
}
```

Visibility Modifiers

Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers. (Getters always have the same visibility as the property.) There are four visibility modifiers in Kotlin: `private`, `protected`, `internal` and `public`.
The default visibility, used if there is no explicit modifier, is `public`.

Visibility Modifiers: Packages

Functions, properties and classes, objects and interfaces can be declared on the "top-level", i.e. directly inside a package.

- If you do not specify any visibility modifier, `public` is used by default, which means that your declarations will be visible everywhere;
- If you mark a declaration `private`, it will only be visible inside the file containing the declaration;
- If you mark it `internal`, it is visible everywhere in the same module;
- `protected` is not available for top-level declarations.

QUESTION: what is a "module"?

lasses and Interfaces

For members declared inside a class:

- `private` means visible inside this class only (including all its members);
- `protected` — same as `private` + visible in subclasses too;
- `internal` — any client inside this module who sees the declaring class sees its internal members;
- `public` — any client who sees the declaring class sees its `public` members.

Outer class does not see private members of its inner classes in Kotlin.
If you override a `protected` member and do not specify the visibility explicitly, the overriding member will also have `protected` visibility.

Visibility Modifiers: Constructors and Local declarations

To specify a visibility of the primary constructor of a class, use the following syntax (note that you need to add an explicit `constructor` keyword):

```
1 class C private constructor(a: Int) { ... }
```

Here the constructor is private. By default, all constructors are `public`, which effectively amounts to them being visible everywhere where the class is visible (i.e. a constructor of an `internal` class is only visible within the same module).
Local variables, functions and classes can not have visibility modifiers.

Extensions

Extensions provides the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator. Kotlin supports extension functions and extension properties.

Extension functions

To declare an extension function, we need to prefix its name with the type being extended.

Example: add a swap function to MutableList<Int>:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

The this keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, we can call such a function on any MutableList<Int>:

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

We can make this function generic:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Extensions are resolved **statically**

Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class, but merely make new functions callable with the dot-notation on variables of this type.

Extension functions are dispatched statically, i.e. they are not virtual by receiver type. This means that the extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime. For example:

```
1 open class C
2 class D: C()
3 fun C.foo() = "c"
4 fun D.foo() = "d"
5 fun printFoo(c: C) {
6     println(c.foo())
7 }
8 printFoo(D())
```

This example will print "c", because the extension function being called depends only on the declared type of the parameter c, which is the C class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name is applicable to given arguments, the **member always wins**. For example:

```
class C {
    fun foo() { println("member") }
}
fun C.foo() { println("extension") }
```

If we call `c.foo()` of any `c` of type `C`, it will print "member", not "extension".
However, it's perfectly OK for extension functions to overload member functions which have the same name but a different signature:

```
class C {
    fun foo() { println("member") }
}
fun C.foo(i: Int) { println("extension") }
```

The call to `C().foo(1)` will print "extension".

Nullable Receiver

Extensions can be defined with a nullable receiver type. Such extensions can be called on an object variable even if its value is null, and can check for `this == null` inside the body. This is what allows you to call `toString()` in Kotlin without checking for null: the check happens inside the extension function.

```
1 fun Any?.toString(): String {
2     if (this == null) return "null"
3     // after the null check, 'this' is autocast to a non-null type, so the toString
      () below
4     // resolves to the member function of the Any class
5     return toString()
6 }
```

Extension Properties

Similarly to functions, Kotlin supports extension properties:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

Note that, since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a backing field. This is why **initializers are not allowed for extension properties**. Their behavior can only be defined by explicitly providing getters/setters.

```
val Foo.bar = 1 // error: initializers are not allowed for extension properties
```

Companion Object Extensions

QUESTION: what is a "companion object"?

If a class has a companion object defined, you can also define extension functions and properties for the companion object:

```
class MyClass {
    companion object { }  // will be called "Companion"
}

fun MyClass.Companion.foo() { ... }
```

Just like regular members of the companion object, they can be called using only the class name as the qualifier:

```
MyClass.foo()
```

Scope of Extensions

Most of the time we define extensions on the top level, i.e. directly under packages:

```
1 package foo.bar
2
3 fun Baz.goo() { ... }
```

To use such an extension outside its declaring package, we need to import it at the call site:

```
1 package com.example.usage
2
3 import foo.bar.goo // importing all extensions by name "goo"
4                    // or
5 import foo.bar.*   // importing everything from "foo.bar"
6
7 fun usage(baz: Baz) {
8     baz.goo()
9 }
```

Declaring Extensions as Members

Inside a class, you can declare extensions for another class. Inside such an extension, there are multiple *implicit receivers* - objects members of which can be accessed without a qualifier. The instance of the class in which the extension is declared is called *dispatch receiver*, and the instance of the receiver type of the extension method is called *extension receiver*.

```
class D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }
    fun D.foo() {
        bar()   // calls D.bar
        baz()   // calls C.baz
    }
    fun caller(d: D) {
        d.foo()   // call the extension function
    }
}
```

In case of a name conflict between the members of the dispatch receiver and the extension receiver, the extension receiver takes precedence. To refer to the member of the dispatch receiver you can use the *qualified this syntax*.

```
class C {
    fun D.foo() {
        toString()         // calls D.toString()
        this@C.toString()  // calls C.toString()
    }
}
```

Extensions declared as members can be declared as open and overridden in subclasses. This means that the dispatch of such functions is virtual with regard to the dispatch receiver type, but static with regard to the extension receiver type.

```kotlin
open class D { }
class D1 : D() { }
open class C {
    open fun D.foo() { println("D.foo in C")
    }
    open fun D1.foo() { println("D1.foo in C")
    }
    fun caller(d: D) {
        d.foo()   // call the extension function
    }
}
class C1 : C() {
    override fun D.foo() { println("D.foo in C1")
    }
    override fun D1.foo() { println("D1.foo in C1")
    }
}
fun main(args: Array<String>) {
    C().caller(D())   // prints "D.foo in C"
    C1().caller(D())  // prints "D.foo in C1" - dispatch receiver is resolved
    virtually
    C().caller(D1())  // prints "D.foo in C" - extension receiver is resolved
    statically
}
```

Extensions utilize the same visibility of other entities as regular functions declared in the same scope would. For example:

- An extension declared on top level of a file has access to the other `private` top-level declarations in the same file;
- If an extension is declared outside its receiver type, such an extension cannot access the receiver's `private` members.

Data Classes

Classes whose main purpose is to hold data, where some standard functionality and utility functions are often mechanically derivable from the data.

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

The compiler automatically derives the following members from all properties declared in the primary constructor:

- `equals()`/`hashCode()` pair;
- `toString()` of the form "User(name=John, age=42)";
- `componentN()` functions corresponding to the properties in their order of declaration;
- `copy()` function (see below).

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

- The primary constructor needs to have at least one parameter;
- All primary constructor parameters need to be marked as `val` or `var`;
- Data classes cannot be abstract, open, sealed or inner;
- `copy()` function (see below).

QUESTION: what is a "sealed class"?

Additionally, the members generation follows these rules with regard to the members inheritance:

- If there are explicit implementations of `equals()`, `hashCode()` or `toString()` in the data class body or final implementations in a superclass, then these functions are not generated, and the existing implementations are used;
- If a supertype has the `componentN()` functions that are open and return compatible types, the corresponding functions are generated for the data class and override those of the supertype. If the functions of the supertype cannot be overridden due to incompatible signatures or being final, an error is reported;
- Deriving a data class from a type that already has a `copy(...)` function with a matching signature is deprecated in Kotlin 1.2 and will be prohibited in Kotlin 1.3.
- Providing explicit implementations for the `componentN()` and `copy()` functions is not allowed.

QUESTION: what is a "matching signature"?

On the JVM, if the generated class needs to have a parameterless constructor, default values for all properties have to be specified.

```
1 data class User(val name: String = "", val age: Int = 0)
```

Properties Declared in the Class Body

Note that the compiler only uses the properties defined inside the primary constructor for the automatically generated functions. To exclude a property from the generated implementations, declare it inside the class body:

```
data class Person(val name: String) {
    var age: Int = 0
}
```

Only the property name will be used inside the `toString()`, `equals()`, hashCode(), and `copy()` implementations, and there will only be one component function `component1()`. While two `Person` objects can have different ages, they will be treated as equal.

```
val person1 = Person("John")
val person2 = Person("John")
person1.age = 10
person2.age = 20
```

Copying

It's often the case that we need to copy an object altering some of its properties, but keeping the rest unchanged. This is what `copy()` function is generated for. For the `User` class

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

its implementation would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

This allows us to write:

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

Data Classes and Destructuring Declarations

Component functions generated for data classes enable their use in destructuring declarations:

```
1 val jane = User("Jane", 35)
2 val (name, age) = jane
3 println("$name, $age years of age") // prints "Jane, 35 years of age"
```

QUESTION: what is a "destructuring declaration"?

Sealed Classes

Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of `enum` classes: the set of values for an `enum` type is also restricted, but each `enum` constant exists only as a single instance, whereas a subclass of a `sealed` class can have multiple instances which can contain state.

```
1 sealed class Expr
2 data class Const(val number: Double) : Expr()
3 data class Sum(val e1: Expr, val e2: Expr) : Expr()
4 object NotANumber : Expr()
```

A sealed class is abstract by itself, it cannot be instantiated directly and can have abstract members.
Sealed classes are not allowed to have non-private constructors (their constructors are private by default).
Note that classes which extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily in the same file.

The key benefit of using sealed classes comes into play when you use them in a `when` expression. If it's possible to verify that the statement covers all cases, you don't need to add an `else` clause to the statement. However, this works only if you use when as an expression (using the result) and not as a statement.

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the ‘else‘ clause is not required because we’ve covered all the cases
}
```

Generics

As in Java, classes in Kotlin may have type parameters:

```
class Box<T>(t: T) {
    var value = t
}
```

In general, to create an instance of such a class, we need to provide the type arguments:

```
val box: Box<Int> = Box<Int>(1)
```

But if the parameters may be inferred, e.g. from the constructor arguments or by some other means, one is allowed to omit the type arguments:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking
    about Box<Int>
```

Variance

Kotlin doesn't have wildcard types. Instead, it has declaration-site variance and type projections.

Why Java needs wildcards? To increase API flexibility. First, generic types in Java are invariant, meaning that `List<String>` is not a subtype of `List<Object>`. Why so? If List was not invariant, it would have been no better than Java's arrays, since the following code would have compiled and caused an exception at runtime:

```
1 // Java
2 List<String> strs = new ArrayList<String>();
3 List<Object> objs = strs; // !!! The cause of the upcoming problem sits here. Java
      prohibits this!
4 objs.add(1); // Here we put an Integer into a list of Strings
5 String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

So, Java prohibits such things in order to guarantee run-time safety. But this has some implications. For example, consider the `addAll()` method from `Collection` interface. What's the signature of this method? Intuitively, we'd put it this way:

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from);
  // !!! Would not compile with the naive declaration of addAll:
  // Collection<String> is not a subtype of Collection<Object>
}
```

That's why the actual signature of `addAll()` is the following:

```
// Java
interface Collection<E> ... {
  void addAll(Collection<? extends E> items);
}
```

The **wildcard type argument** "? `extends E`" indicates that this method accepts a collection of objects of E or some subtype of `E`, not just `E` itself. This means that we can safely **read** `E`'s from items (elements of this collection are instances of a subclass of `E`), but **cannot write** to it since we do not know what objects comply to that unknown subtype of `E`. In return for this limitation, we have the desired behaviour: `Collection<String>` is a subtype of `Collection<? extends Object>`. In "clever words", the wildcard with an **extends**-bound (**upper** bound) makes the type **covariant**.

The key to understanding why this trick works is rather simple: if you can only **take** items from a collection, then using a collection of `Strings` and reading `Objects` from it is fine. Conversely, if you can only **put** items into the collection, it's OK to take a collection of `Objects` and put `Strings` into it: in Java we have `List<? super String>` a **supertype** of `List<Object>`.
The latter is called contravariance, and you can only call methods that take String as an argument on `List<? super String>` (e.g., you can call `add(String)` or `set(int, String)`), while if you call something that returns `T` in `List<T>`, you don't get a `String`, but an `Object`.

Declaration Site Variance

Suppose we have a generic interface `Source<T>` that does not have any methods that take `T` as a parameter, only methods that return `T`:

```
// Java
interface Source<T> {
  T nextT();
}
```

Then, it would be perfectly safe to store a reference to an instance of `Source<String>` in a variable of type `Source<Object>` – there are no consumer-methods to call. But Java does not know this, and still prohibits it:

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! Not allowed in Java
  // ...
}
```

To fix this, we have to declare objects of type `Source<? extends Object>`, which is sort of meaningless, because we can call all the same methods on such a variable as before, so there's no value added by the more complex type. But the compiler does not know that.
In Kotlin, there is a way to explain this sort of thing to the compiler. This is called **declaration-site variance**: we can annotate the **type parameter** `T` of Source to make sure that it is only returned (produced) from members of `Source<T>`, and never consumed. To do this we provide the **out** modifier:

```
1 interface Source<out T> {
2     fun nextT(): T
3 }
4
5 fun demo(strs: Source<String>) {
6     val objects: Source<Any> = strs // This is OK, since T is an out-parameter
7     // ...
8 }
```

The general rule is: **when a type parameter `T` of a class `C` is declared `out`, it may occur only in `out`-position in the members of `C`, but in `return C<Base>` can safely be a supertype of `C<Derived>`.**
In "clever words" they say that the class `C` is `covariant` in the parameter `T`, or that `T` is a `covariant` type parameter. You can think of `C` as being a `producer` of `T`'s, and NOT a `consumer` of `T`'s.
The `out` modifier is called a `variance annotation`, and since it is provided at the type parameter declaration site, we talk about `declaration-site variance`. This is in contrast with Java's `use-site variance` where wildcards in the type usages make the types covariant.

In addition to **out**, Kotlin provides a complementary variance annotation: **in**. It makes a type parameter **contravariant**: it can only be consumed and never produced. A good example of a contravariant type is `Comparable`:

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

Use-site variance: Type projections

It is very convenient to declare a type parameter `T` as `out` and avoid trouble with subtyping on the use site, but some classes **can't** actually be restricted to only return `T`'s! A good example of this is `Array`:

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { ... }
    fun set(index: Int, value: T) { ... }
}
```

This class cannot be either co- or contravariant in `T`. And this imposes certain inflexibilities. Consider the following function:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

This function is supposed to copy items from one array to another.

Let's try to apply it in practice:

```
1 val ints: Array<Int> = arrayOf(1, 2, 3)
2 val any = Array<Any>(3) { "" }
3 copy(ints, any)
4 //   ^ type is Array<Int> but Array<Any> was expected
```

Here we run into the same familiar problem: `Array<T>` is **invariant** in `T`, thus neither of `Array<Int>` and `Array<Any>` is a subtype of the other. Why? Again, because copy **might** be doing bad things, i.e. it might attempt to **write**, say, a `String` to `from`, and if we actually passed an array of `Int` there, a `ClassCastException` would have been thrown sometime later.

Then, the only thing we want to ensure is that `copy()` does not do any bad things. We want to prohibit it from **writing** to `from`, and we can:

```
1 fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

What has happened here is called type projection: we said that from is not simply an array, but a restricted (projected) one: we can only call those methods that return the type parameter `T`, in this case it means that we can only call `get()`. This is our approach to use-site variance, and corresponds to Java's `Array<? extends Object>`, but in a slightly simpler way.

You can project a type with **in** as well:

```
1 fun fill(dest: Array<in String>, value: String) { ... }
```

`Array<in String>` corresponds to Java's `Array<? super String>`, i.e. you can pass an array of `CharSequence` or an array of `Object` to the `fill()` function.

Star-projections

Sometimes you want to say that you know nothing about the type argument, but still want to use it in a safe way. The safe way here is to define such a projection of the generic type, that every concrete instantiation of that generic type would be a subtype of that projection
Kotlin provides so called star-projection syntax for this:

- For `Foo<out T : TUpper>`, where `T` is a covariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>`. It means that when the `T` is unknown you can safely read values of `TUpper` from `Foo<*>`.
- For `Foo<in T>`, where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>`. It means there is nothing you can write to `Foo<*>` in a safe way when `T` is unknown.
- For `Foo<T : TUpper>`, where `T` is an invariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

If a generic type has several type parameters each of them can be projected independently. For example, if the type is declared as interface `Function<in T, out U>` we can imagine the following star-projections:

- `Function<*, String>` means `Function<in Nothing, String>`;
- `Function<Int, *>` means `Function<Int, out Any?>`;
- `Function<*, *>` means `Function<in Nothing, out Any?>`.

Note: star-projections are very much like Java's raw types, but safe.

Generic functions

Not only classes can have type parameters. Functions can, too. Type parameters are placed before the name of the function:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString() : String {  // extension function
    // ...
}
```

To call a generic function, specify the type arguments at the call site after the name of the function:

```
val l = singletonList<Int>(1)
```

Type arguments can be omitted if they can be inferred from the context, so the following example works as well:

```
val l = singletonList(1)
```

Generic constraints

The set of all possible types that can be substituted for a given type parameter may be restricted by generic constraints. The most common type of constraint is an upper bound that corresponds to Java's `extends` keyword:

```
fun <T : Comparable<T>> sort(list: List<T>) {  ... }
```

Only a subtype of `Comparable<T>` may be substituted for `T`. For example:

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype
     of Comparable<HashMap<Int, String>>
```

The default upper bound (if none specified) is `Any?`. Only one upper bound can be specified inside the angle brackets. If the same type parameter needs more than one upper bound, we need a separate where-clause:

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

Type erasure

The type safety checks that Kotlin performs for generic declaration usages are only done at compile time. At runtime, the instances of generic types do not hold any information about their actual type arguments. The type information is said to be erased. For example, the instances of `Foo<Bar>` and `Foo<Baz?>` are erased to just `Foo<*>`.

Therefore, there is no general way to check whether an instance of a generic type was created with certain type arguments at runtime, and the compiler prohibits such is-checks.

Type casts to generic types with concrete type arguments, e.g. `foo as List<String>`, cannot be checked at runtime. These unchecked casts can be used when type safety is implied by the high-level program logic but cannot be inferred directly by the compiler. The compiler issues a warning on unchecked casts, and at runtime, only the non-generic part is checked (equivalent to `foo as List<*>`).

Object Expressions and Declarations

Sometimes we need to create an object of a slight modification of some class, without explicitly declaring a new subclass for it. Java handles this case with anonymous inner classes. Kotlin slightly generalizes this concept with object expressions and object declarations.

Object Expressions

To create an object of an anonymous class that inherits from some type (or types), we write:

```
fwindow.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
})
```

If a supertype has a constructor, appropriate constructor parameters must be passed to it. Many supertypes may be specified as a comma-separated list after the colon:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { ... }

val ab: A = object : A(1), B {
    override val y = 15
}
```

If, by any chance, we need "just an object", with no nontrivial supertypes, we can simply say:

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

Note that anonymous objects can be used as types only in local and private declarations. If you use an anonymous object as a return type of a public function or the type of a public property, the actual type of that function or property will be the declared supertype of the anonymous object, or `Any` if you didn't declare any supertype. Members added in the anonymous object will not be accessible.

```
class C {
    // Private function, so the return type is the anonymous object type
    private fun foo() = object {
        val x: String = "x"
    }

    // Public function, so the return type is Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // Works
        val x2 = publicFoo().x  // ERROR: Unresolved reference 'x'
    }
}
```

Just like Java's anonymous inner classes, code in object expressions can access variables from the enclosing scope. (Unlike Java, this is not restricted to final variables.)

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

Object Declarations

Singleton may be useful in several cases, and Kotlin (after Scala) makes it easy to declare singletons:

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

This is called an object declaration, and it always has a name following the `object` keyword.
Just like a variable declaration, an object declaration is not an expression, and cannot be used on the right hand side of an assignment statement.
Object declaration's initialization is thread-safe.
To refer to the object, we use its name directly:

```
DataProviderManager.registerDataProvider(...)
```

Such objects can have supertypes:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

NOTE: object declarations can't be local (i.e. be nested directly inside a function), but they can be nested into other object declarations or non-inner classes.

Semantic difference between object expressions and declarations

There is one important semantic difference between object expressions and object declarations:

- object expressions are executed (and initialized) immediately, where they are used;
- object declarations are initialized lazily, when accessed for the first time.

Outline

Higher-Order Functions

A higher-order function is a function that takes functions as parameters, or returns a function. A good example is the functional programming idiom fold for collections, which takes an initial accumulator value and a combining function and builds its return value by consecutively combining current accumulator value with each collection element, replacing the accumulator:

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

In the code above, the parameter combine has a function type `(R, T) -> R`, so it accepts a function that takes two arguments of types `R` and `T` and returns a value of type `R`. It is invoked inside the for-loop, and the return value is then assigned to `accumulator`.

To call fold, we need to pass it an instance of the function type as an argument, and lambda expressions are widely used for this purpose at higher-order function call sites:

```
val items = listOf(1, 2, 3, 4, 5)

// Lambdas are code blocks enclosed in curly braces.
items.fold(0, {
    // When a lambda has parameters, they go first, followed by '->'
    acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
    val result = acc + i
    println("result = $result")
    // The last expression in a lambda is considered the return value:
    result
})

// Parameter types in a lambda are optional if they can be inferred:
val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

// Function references can also be used for higher-order function calls:
val product = items.fold(1, Int::times)
```

Function types

Function types have a special notation that corresponds to the signatures of the functions, i.e. their parameters and return values:

- All function types have a parenthesized parameter types list and a return type: `(A, B) -> C` denotes a type that represents functions taking two arguments of types `A` and `B` and returning a value of type `C`. The parameter types list may be empty, as in `() -> A`. The `Unit` return type cannot be omitted.
- Function types can optionally have an additional receiver type, which is specified before a dot in the notation: the type `A.(B) -> C` represents functions that can be called on a receiver object of `A` with a parameter of `B` and return a value of `C`. Function literals with receiver are often used along with these types.

The function type notation can optionally include names for the function parameters: `(x: Int, y: Int) -> Point`. These names can be used for documenting the meaning of the parameters.

Instantiating a function type

There are several ways to obtain an instance of a function type:

- Using a code block within a function literal, in one of the forms:
  - a lambda expression: `{ a, b -> a + b }`,
  - an anonymous function: `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  "Function literals with receiver" can be used as values of function types with receiver.
- Using a callable reference to an existing declaration:
  - a top-level, local, member, or extension function: `::isOdd`, `String::toInt`,
  - a top-level, member, or extension property: `List<Int>::size`,
  - a constructor: `::Regex`

  These include bound callable references that point to a member of a particular instance: `foo::toString`.

Instantiating a function type (cont.)

- Using instances of a custom class that implements a function type as an interface:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

The compiler can infer the function types for variables if there is enough information:

```
val a = { i: Int -> i + 1 } // The inferred type is (Int) -> Int
```

Instantiating a function type (cont.)

Non-literal values of function types with and without receiver are interchangeable, so that the receiver can stand in for the first parameter, and vice versa. For instance, a value of type `(A, B) -> C` can be passed or assigned where a `A.(B) -> C` is expected and the other way around:

```
1 val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
2 val twoParameters: (String, Int) -> String = repeatFun // OK
3
4 fun runTransformation(f: (String, Int) -> String): String {
5     return f("hello", 3)
6 }
7 val result = runTransformation(repeatFun) // OK
```

Note that a function type with no receiver is inferred by default, even if a variable is initialized with a reference to an extension function. To alter that, specify the variable type explicitly.

Invoking a function type instance

Non-literal values of function types with and without receiver are interchangeable, so that the receiver can stand in for the first parameter, and vice versa. A value of a function type can be invoked by using its invoke(...) operator: `f.invoke(x)` or just `f(x)`.
If the value has a receiver type, the receiver object should be passed as the first argument. Another way to invoke a value of a function type with receiver is to prepend it with the receiver object, as if the value were an extension function: `1.foo(2)`.

```
1 val stringPlus: (String, String) -> String = String::plus
2 val intPlus: Int.(Int) -> Int = Int::plus
3
4 println(stringPlus.invoke("<-", "->"))
5 println(stringPlus("Hello, ", "world!"))
6
7 println(intPlus.invoke(1, 1))
8 println(intPlus(1, 2))
9 println(2.intPlus(3)) // extension-like call
```