# Table of Contents


Functions and properties . . . 1
- Functions . . . 1
  - Variadic arguments . . . 1
  - Named arguments . . . 2
- Properties . . . 2
  - Backing fields . . . 4
  - `lateinit` properties . . . 4
- Nesting . . . 5
  - Local functions and classes . . . 5

Classes, functions, properties, and inheritance . . . 5
- Inheritance . . . 6
- `abstract` and `override` . . . 6
- Explicit `super` . . . 7
- Function objects . . . 8
- Anonymous objects . . . 8

# Functions and properties

## Functions

Kotlin functions are declared using `fun`:

```
fun functionName(params): Type {...}
```

Parameters, along with all other values in Kotlin, are defined using *Pascal notation*. Parameters can also have *default arguments*, which helps to reduce the need for overloads.

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

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")
```

| | |
|---|---|
| **CAUTION** | No more than one parameter per function can be `varargs`. |

### 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.

## 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 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.

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.

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

**NOTE** 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.

**TIP** 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 = ...
    }
}
```

## Nesting

### Local functions and classes

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

```
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.

## 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.

**TIP** 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.

> **TIP** 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
}
```

> **TIP** `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: [1]

```kotlin
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

TODO lambda

## Anonymous objects

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

```kotlin
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:

```kotlin
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.

| | |
|---|---|
| **TIP** | 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/classes.html#overriding-rules