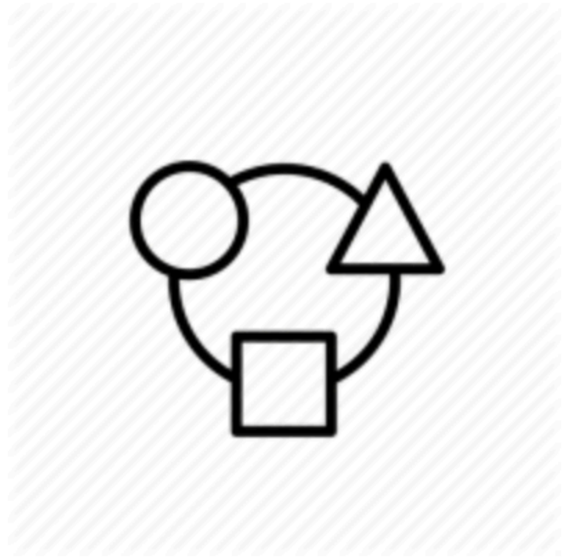
Properties & Fields

Properties & Fields



Kotlin properties and fields offer a richer set of features and variants over Java

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 full syntax for declaring a property is

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

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, as shown below).

Examples:

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

The full syntax of a read-only property declaration differs from a mutable one in two ways: it starts with `val` instead of `var` and does not allow a setter:

```
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter
```

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:

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

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
```

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
```

Backing Properties

If you want to do something that does not fit into this "implicit backing field" scheme, you can always fall back to having a *backing property*:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

In all respects, this is just the same as in Java since access to private properties with default getters and setters is optimized so that no function call overhead is introduced.

Late-Initialized Properties and Variables

Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle this case, you can mark the property with the `lateinit` modifier:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```