

# Properties And Fields

## Field accessors in Java

In **Java** world, we are so accustomed to writing *getters* and *setters* for our *fields*, that the advice by [Effective Java Item 14](#): **In public classes, use accessor methods, not public fields** sort of goes without saying. All major IDEs help us here: they generate getters and setters, so that it is not that hard to produce a class like this:

```
// Java
public class Address {

    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String getZip() {
        return zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }
}
```

Most of the lines in this class are pure boilerplate code.

## Getting rid of field/get/set triples

In *Kotlin*, there's no way to declare a *field*. All you have is *properties*. Read/write properties are declared with the `var` keyword, and read-only ones – with `val` keyword. Thus, the class above can be rewritten as follows:

```
public class Address() { // parentheses denote a _primary constructor_
    public var name : String = ...
    public var street String = ...
    public var city : String = ...
    public var state : String? = ...
    public var zip : String = ...
}
```

Here we have five *mutable* properties, each of which has a *backing field* that stores the value, and two accessors: getter and setter. Thus, the byte-code generated from this class will be almost equivalent to the one for the Java class above. The only difference will be property initializers, see [Null-safety](#).

(For even better option look [here](#)).

To use a property, one simply refers 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
}
```

## Declaring properties and accessors

The full syntax for a *mutable* property declaration is as follows:

```
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 base class member being overridden .

Examples:

```
var
allByDefault
    : Int? // error: explicit initializer required, default getter and setter implied
var initialized = 1 // has type Int, default getter and setter
var setterVisibility : String = "abc" // Initializer required, not a nullable type
private set // the setter is private and has the default implementation
```

Note that types are not inferred for properties exposed as parts of the public API, i.e. **public** and **protected**, because changing the initializer may cause an unintentional change in the public API then. For example

```
public val
example
    = 1 // A public property must have a type specified explicitly
```

The full syntax of an *immutable* 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
```

Since this property is purely derived from others, the compiler will not generate a *backing field* for it.

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
    }
```

## Backing fields

As we mentioned above, some properties have *backing fields*, i.e. from the client's point of view, a property is a pair of accessors (or just one getter), but physically the accessors may read and write data from/to a real *field*. One can not *declare* a field explicitly in *Kotlin*, the compiler figures it out for us.

In the simple cases, when we do not provide custom accessor implementations, it is obvious that a property must have a backing field, otherwise what should the default accessors do in the following case?

```
var counter : Int = 0
```

But when there is a custom accessor, it *may* or *may not* rely on a backing field.

To access a backing field of a property *x*, one says *\$x* (the dollar sign cannot be used as a part of an identifier in *Kotlin*):

```
var counter = 0 // the initializer value is written directly to the backing field
    set(value) {
        if (value >= 0)
            $counter = value
    }
```

The *\$counter* field can be accessed only from inside the class where the *counter* property is defined.

The compiler looks at the accessors' bodies, and if they use the backing field (or the accessor implementation is left by default), a backing field is generated, otherwise it is not.

For example, in the following case there will be *no backing field*:

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

The backing field is not needed because the only accessor does not refer to it.

## What if I want to do ... ?

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.

## Overriding properties

See [Overriding properties](#).

## Best practices related to this feature

J. Bloch. [Effective Java Second Edition](#)

**Item 14:** In public classes, use accessor methods, not public fields

See also: [JavaBeans](#)

## Similar features in other languages

Java IDEs generate accessors [automatically](#).

For **Java**, there's [Project Lombok](#) : the syntax for properties is based on Java annotations.

In **C#**, [Groovy Beans](#) and [Gosu](#) one still writes getters and setters alongside their backing fields that are declared explicitly, although the access looks like property access. [Scala](#) does not distinguish between a field and a property, but to customize a setter one needs to write a separate function named by convention.

## What's next

- [Basic types](#)