Extracted from:

Programming Kotlin

Creating Elegant, Expressive, and Performant JVM and Android Applications

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Kotlin 1.3

Create Elegant, Expressive, and Performant JVM and Android Applications

Venkat Subramaniam

edited by Jacquelyn Carter

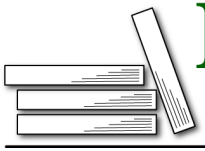
Programming Kotlin

Creating Elegant, Expressive, and Performant JVM and Android Applications

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-635-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2019

Creating Classes

Kotlin differs quite a bit from Java in how classes are defined. Writing classes in Java involves so much boilerplate code. To alleviate the pain of typing all that, programmers rely a lot on IDEs to generate code. Good news, you don't need to type as much. Bad news, you'll have to wade through all that code each day. Kotlin moves the code generation from the IDE to the compiler—kudos for that, especially for a language created by the company that makes the best IDEs in the world.

The syntax for creating a class in Kotlin is closer to the facilities in Scala than in Java. The number of options and flexibilities seem almost endless; let's start small and grow the code for creating a class incrementally.

Smallest Class

Here's the minimum syntax for a class—the `class` keyword followed by the name of the class:

oop/empty.kts
```
class Car
```

We didn't provide any properties, state, or behavior for this class, so it's only fair that it does nothing useful.

Read-Only Properties

Let's define a property in the class:

oop/property.kts
```
class Car(val yearOfMake: Int)
```

We made a few design decisions right there in that highly concise syntax. We wrote a constructor to take an integer parameter and initialize a read-only property named `yearOfMake` of type `Int`—yep, all that in one line. The Kotlin compiler wrote a constructor, defined a field, and added a getter to retrieve the value of that field.

Let's take a closer look at the class definition. That line is a shortcut for this:

```
public class Car public constructor(public val yearOfMake: Int)
```

By default, the access to the class and its members is public and the constructor is public as well. In Kotlin, the line that defines the class is actually defining the primary constructor. The keyword `constructor` isn't needed unless we want to change the access modifier or place an annotation for the primary constructor.

Creating Instances

Let's use the class to create an instance. New in Kotlin, related to creating objects, is there's no new keyword. To create an object use the class name like it's a function:

oop/property.kts

```
val car = Car(2019)
println(car.yearOfMake) //2019
```

The immutable variable car holds a reference to a Car instance created by calling the constructor with the value 2019. The property yearOfMake is accessible directly on the instance car. Efforts to modify it will run into issues, like so:

```
car.yearOfMake = 2019 //ERROR: val cannot be reassigned
```

Much like val local variables, val properties are immutable too.

Read-Write Properties

You can design a property to be mutable if you like:

oop/readwrite.kts

```
class Car(val yearOfMake: Int, var color: String)

val car = Car(2019, "Red")
car.color = "Green"
println(car.color) //GREEN
```

The newly added property color of type String is initialized with the constructor. But, unlike yearOfMake, we may change the value of color anytime on an instance of Car.

Use val to define read-only properties and var for properties that may change.

A Peek Under the Hood—Fields and Properties

In the previous example, yearOfMake and color seem like fields rather than properties to my Java eyes. However, those are properties and not fields. Kotlin doesn't expose fields of classes.

When you call car.yearOfMake, you're actually calling car.getYearOfMake()—the good old JavaBean convention is honored by the compiler. To prove this, let's examine the bytecode generated by the Kotlin compiler.

First, write the class Car in a separate file named Car.kt:

oop/Car.kt

```
class Car(val yearOfMake: Int, var color: String)
```

Next, compile the code and take a look at the bytecode using the javap tool, by running these commands:

```
$ kotlinc-jvm Car.kt
$ javap -p Car.class
```

Here the excerpt of the bytecode generated by the Kotlin Compiler for the Car class is:

```
Compiled from "Car.kt"
public final class Car {
  private final int yearOfMake;
  private java.lang.String color;
  public final int getYearOfMake();
  public final java.lang.String getColor();
  public final void setColor(java.lang.String);
  public Car(int, java.lang.String);
}
```

That concise one line of Kotlin code for the Car class resulted in the creation of two fields—the backing fields for properties, a constructor, two getters, and a setter. Nice.

Let's confirm that when using the object we're not directly accessing the fields. For this, create a file UseCar.kt with the following code:

oop/UseCar.kt

```
fun useCarObject(): Pair<Int, String> {
  val car = Car(2019, "Red")

  val year = car.yearOfMake

  car.color = "Green"

  val color = car.color

  return year to color
}
```

Let's compile using the following commands and take a look at the bytecode generated:

```
$ kotlinc-jvm Car.kt UseCar.kt
$ javap -c UseCarKt.class
```

The formatted excerpt from the output that follows shows that we're not accessing the fields directly; the code follows the JavaBean convention and doesn't breach encapsulation.

```
//...
       7: ldc           #11                 // String Red
       9: invokespecial #15
                             // Method Car."<init>":(ILjava/lang/String;)V
```

```
      12: astore_0
      13: aload_0
      14: invokevirtual #19                 // Method Car.getYearOfMake:()I
      17: istore_1
      18: aload_0
      19: ldc           #21                 // String Green
      21: invokevirtual #25
                            // Method Car.setColor:(Ljava/lang/String;)V
      24: aload_0
      25: invokevirtual #29
                            // Method Car.getColor:()Ljava/lang/String;
//...
```

In Kotlin you access properties by providing the name of the property instead of the getter or setter.

Controlling Change to Properties

In our Car class, yearOfMake is immutable but color is mutable. Immutability is safe, but uncontrolled change to a mutable property is unsettling. Kotlin will prevent someone changing or setting color to null since it's a String and not String? type—that is, it's not a nullable reference type. But what if someone sets it to an empty string? Let's fix the class to prevent that. Along the way we'll add another property to the class, but this one won't be given a value through the constructor.

oop/setter.kts

```
class Car(val yearOfMake: Int, theColor: String) {
  var fuelLevel = 100

  var color = theColor
    set(value) {
      if (value.isBlank()) {
        throw RuntimeException("no empty, please")
      }

      field = value
    }
}
```

To reiterate, in Kotlin you never define fields—backing fields are synthesized automatically, but only when necessary. If you define a field with both a custom getter and custom setter and don't use the backing field using the field keyword, then no backing field is created. If you write only a getter or a setter for a property, then a backing field is synthesized.

In our constructor we defined one property, yearOfMake, and one parameter, not field, named theColor—we didn't place val or var for this parameter. If you really wanted to call this color, you could have done that as well.

Within the class, we created a property named fuelLevel and initialized it to a value of 100. Its type is inferred to be Int by the compiler. This property isn't being set using any parameter of the constructor.

Then we created a property named color and assigned it to the value in the constructor parameter theColor. If we had called the constructor parameter color instead of theColor, then this line would have read like this:

```
var color = color
```

But named thecolor, it reads as follows:

```
var color = theColor
```

Using the same name for a property and a parameter is like the practice in Java where the this.color = color syntax is used for assignment, but if it's confusing, use different names for properties and parameters.

Kotlin will synthesize a getter and a setter for the fuelLevel property. For the color property it will only synthesize a getter, but use the setter provided in code. The setter throws an exception if the value given for the property is empty. You may call the parameter something other than value if you like. Also, you may specify the type for the parameter of set if you desire, but in this case the compiler knows the type, based on the type of the property initialization. If the given value is acceptable, then assign it to the field which is referenced by a special keyword field. Since Kotlin synthesizes fields internally, it doesn't give access to that name in code. You may refer to it using the keyword field only within getters and setters for that field.

Let's use the modified class to access the fields.

oop/setter.kts

```
val car = Car(2019, "Red")
car.color = "Green"
car.fuelLevel--

println(car.fuelLevel) //99

try {
  car.color = ""
} catch(ex: Exception) {
  println(ex.message) //no empty, please
}

println(car.color) //Green
```

No issue arose to change fuelLevel or to set color to "Green". But the attempt to change color to an empty string failed with a runtime exception. Following Kotlin convention, we got the details of the exception from the message property

instead of using the getMessage() method of the Exception class. In the last line we verify that the color property is the same as what was set before the try expression.

Access Modifiers

The properties and methods of a class are public by default in Kotlin. The possible access modifiers are: public, private, protected, and internal. The first two have the same meaning as in Java. The protected modifier gives permission to the methods of the derived class to access that property. The internal modifier gives permission for any code in the same module to access the property or method, where a module is defined as all source code files that are compiled together. The internal modifier doesn't have a direct bytecode representation. It's handled by the Kotlin compiler using some naming conventions without posing any runtime overhead.

The access permission for the getter is the same as that for the property. You may provide a different access permission for the setter if you like. Let's change the access modifier for fuelLevel so that it can be accessed from only within the class.

oop/privatesetter.kts

```
var fuelLevel = 100
  private set
```

To make the setter private, simply place the keyword private before the keyword set. If you don't have an implementation to provide for the setter, then leave out the parameter value and the body. If you don't write a setter, or if you don't specify the access modifier for the setter, then its permission is the same as that of the property.

Initialization Code

The primary constructor declaration is part of the first line. Parameters and properties are defined in the parameter list for the constructor. Properties not passed through the constructor parameters may also be defined within the class. If the code to initialize the object is more complex than merely setting values, then we may need to write the body for the constructor. Kotlin provides a special space for that.

A class may have zero or more init blocks. These blocks are executed as part of the primary constructor execution. The order of execution of the init blocks is top-down. Within an init block you may access only properties that are already defined above the block. Since the properties and parameters declared in the primary constructor are visible throughout the class, any init block

within the class can use them. But to use a property defined within the class, you'll have to write the init block after the said property's definition.

Just because we can define multiple init blocks doesn't mean we should. Within your class, first declare your properties at the top, then write one init block, but only if needed, and then implement the secondary constructors (again only if needed), and finally create any methods you may need.

If you're curious to see an init block, here's one to set the value of the fuelLevel based on the yearOfMake property.

oop/initialization.kts

```
class Car(val yearOfMake: Int, theColor: String) {
  var fuelLevel = 100
    private set

  var color = theColor
    set(value) {
      if (value.isBlank()) {
        throw RuntimeException("no empty, please")
      }

      field = value
    }

  init {
    if (yearOfMake < 2020) { fuelLevel = 90 }
  }
}
```

Within the init block we change the value of fuelLevel based on the value of yearOfMake. Since this requires access to fuelLevel, it can't be earlier than the declaration of fuelLevel.

You may wonder if, instead of the init block, could we have accomplished the task at the location of fuelLevel definition? Sure thing—you may remove the above init block entirely and write the following:

```
var fuelLevel = if (yearOfMake < 2020) 90 else 100
  private set
```

Don't write more than one init block, and avoid it if you can. The less work we do in constructors, the better from the program safety and also performance point of view.