Basic Java Keywords

Introduction

Hi, welcome to my free Java 8 course! My goal is to teach you Java in the clearest possible way, and to help you become a pragmatic perfectionist and a clean code craftsman.

In this article I'm going to introduce you to fifteen Java keywords / concepts that are central to Java. To make the concepts more concrete, I'll illustrate them using the analogy of cars in a garage.

Please note: I really try not to use any term before I have officially introduced it to you. Getting a keyword explained using concepts I didn't know was something that always ticked me off in classes, so I aim never to do this to you. For this reason, I might intentionally not explain things one hundred percent correctly at first, but later, if necessary, I'll add the academically correct explanation.

package

Ok, let's start off with our first keyword: **`package`**. A `package` is like the folder structure on your computer. It organizes the different files of your program. A `package` name must be unique since there are millions of other programs out there, and you don't want to have the same program file name as someone else. Since Java code is heavily shared, the packages are used to prevent name clashes. If two different code files in one program have the same name, including their package name, only one of them will be loaded and the other code file will be completely ignored, which will usually result in errors.

To guarantee that a package is unique it is commonplace to include one's domain name as part of the package. Usually, the top or root folder of a package structure is the organization's domain name in reverse order. Using a package name is never required, but it is highly recommended, even in the most basic programs. When declaring a `package`, you must declare it in the first line of your program. Lowercase and singular nouns are commonly used for each part of your package name. In my example, I'm starting the package name with my top level domain, `com`, followed by my domain name, `marcusbiel`, then `java8course` since that is the project we are working on. As I stated earlier, the sample code in this article is related to a garage, so we will end the package name with "garage". Our full package name will therefore be:

```
package com.marcusbiel.java8course.garage;
```
Example 1

Where possible, use terms that the client can understand and that relate to the business (garage in my case) rather than technical terms like “frontend” or “backend”.

import

To simplify programs and reduce redundancy, many program files reuse existing code. To do so, you could use the code file’s full name including its full package name. However, to make your code more readable, you would usually add the file to the list of imports, which will allow you to address the code by its simple file name, without having to constantly reference it by its full package location.

The import statement, or statements, is written directly after the package declaration. When you import code, you include the package name of the file. You can import specific files from the package, or the entire package, by using a star symbol.

Let’s begin our code by importing the first car into our garage, a `Bmw`.

```
package com.marcusbiel.java8course.garage;

import com.marcusbiel.java8course.car.Bmw;
```
Example 2

Class

Programs can easily consist of thousands, if not tens of thousands, of lines of code. When you have so much text in one place, it’s easy to get lost. To make code clearer, Java programmers classify their code into different *units*, called **classes**.

As a programmer, you are free to name your packages and your classes whatever you like, but your goal should always be clear. When you’re working with a business client, the client defines *what* he wants; it is your job to know *how* to express this. If you can create a common language to bridge the gap between the two, it will be easier for your client to understand the code without much coding knowledge. For this reason, programmers tend to name their classes using a noun that describes the class, starting with a capital letter.

For example we could create a `class Car`, which will later contain all the code related to a car (Example 2). At the beginning of a class, there is an opening curly brace, and at the end of our code there is a closing curly brace that shows the end of the class.

Classes should only be focused on one topic, and you should try to make them as small as possible for the sake of readability and maintainability.

```
class Car {

}
```

Example 3

Method

A class will consist of one or more methods. A method defines how a certain task will be completed using code. For example, a method `timesTwo` could double the amount of a given input. Methods are also sometimes called functions. While this term isn't totally academically correct, you may use it. By convention, a method is usually named after a verb that describes what actions it performs. Methods can operate on anything: numbers, colors, sounds - you name it.

```
timesTwo(2) => 4
add("ab", "c") => "abc"
print("Hello") => will print "Hello"
```

Example 4

Imagine your code as a book. A book has chapters (the packages), paragraphs (classes), and sentences (methods). Methods can also call other methods, creating a chain of methods. As a programmer, you want to structure your code by thinking of the current level of abstraction, just like writing a book. Your code should end up being readable like a book!

An example of a method calling other methods would be a "`prepareDinner`()" method internally calling a "`prepareAppetizer`()" method, followed by calling a method called "`prepareMainCourse`()", followed by a method called "`prepareDessert`()".

If our `prepareAppetizer`() method then has to call three more methods, "`washLettuce`()", "`addTomatoes`()" and "`tossSalad`()", we've created a readable and understandable hierarchy in our code. We could, of course, have `prepareDinner`() directly call all three of these methods, instead of `prepareAppetizer`(), but that would clutter our code and make it difficult to read.

Coming up with a clean structure and making your code "speak" is important. The harder it is to understand what your program does, the easier it will be to introduce an error. As a rule of thumb, try to keep your methods shorter than twenty lines. Personally, I aim for a method length of just 1-3 lines.

Methods are defined similarly to classes. First you define the name of the method, usually a verb beginning with a lowercase letter. Methods are normally verbs because they do something. Classes are usually nouns because they are the objects that the methods are acting on/in. The name of the method is followed by a pair of parentheses. Inside these parentheses you can define anywhere from zero to an unlimited number of input fields. When you call a method, you need to send specific values, called arguments, to the specific fields, called method parameters. Arguments are the actual values you are sending to the method. In the context of the method, the general values it will receive are called method parameters. I recommend **an absolute maximum** of three to five method parameters, because more than that harms readability. The fewer parameters, the better. After your parameters, you begin your method definition with an opening curly brace and end it with a closing curly brace. In between, you put the code for your method. Let's take another look at our `Car` class:

```
package com.marcusbiel.java8course.garage;

import com.marcusbiel.java8course.car.Bmw;

class Car {

     drive(speed) {

     }
}
```

Example 5

Object

As I said before, a class is used to structure code in Java in the form of units of code. However, this is only part of the story. A class is like a blueprint for what you want to do when the program is running. The class `Car` defines how a car will behave. However, at runtime (when the program is running), there will be a number of cars, each with its own set of values.

So a class only acts as a template for the objects that are created in your program. Each object has the same set of behaviors, as defined by the class, but it also exists as its own set of values in the program that could change as the program runs. For example, you could have two Objects of class `Car` that both drive, but they could have different values for their respective speeds.

Variable

A variable, as the name implies, varies in value. It is a **placeholder** for a value that you can name and set. Variables can be used as the input and output for methods. Variables can be a variety of different data types. There are two categories of data types: primitive data types and object types. Primitive data types such as `int`, `char`, and `boolean` store things like numbers and single characters. Object types are used to store objects and are referenced by object reference variables. An example of a possible reference variable name for an object of type Car would be `myCar`, `myPorsche`, or `momsCar`.

Dot

In Java, a "`.`" is not used to indicate the end of a sentence. Instead, a "`.`" is used by a variable to *execute* or *call* a method. For example, we could have a variable `car` call the method `drive()`.

Semicolon

Since the "`.`" is used to indicate a method being called, the "`;`" is used to indicate the end of a command, statement, or declaration.

```
car.drive();
```

Example 6

Variable Declaration

Before we can use a variable, we need to define it. You declare a variable by writing the type of the variable, followed by its name, followed by a "`;`".

```
Car myPorsche;
```

Example 7

Object Allocation

Once you've declared a variable, you can allocate it to a specific object. You can also do both of these things in one line. First, you can create a variable of type Car, called `myPorsche` and then, using an equals sign, assign it to a new Car object, with a first value of 1 and a second value of 320.

```
Car myPorsche = new Car(1, 320);
```
Example 8

After we declare the variable and assign it to the object, whenever we use `myPorsche` we are referencing this object created in the memory. You might also notice that we put two values into the constructor of Car, but without actually looking at the constructor of Car we wouldn't know what their meanings were. This is one of the reasons to have as few fields as possible in methods and constructors.

public

`public` is an access modifier that defines that the class or method that is tagged as `public` can be used by any other class from any other package. Besides `public`, there are also other access modifiers such as `private`, default, and `protected`, which I will cover in more detail later.

void

For every method, it is required that you return a value, even when you don't want to. However, the tag `void` defines that the method will not return a value. If a method isn't `void`, it can return any kind of data type. A clean way to design your methods is to make each method fulfill one of two roles. A **query method** returns a value but doesn't alter the state of a class. A **command method** alters the state of a class, but doesn't return a value. `drive(speed)` is a command method. It performs an action ("driving"), but it doesn't return a value (and thus we define it as void).

```
package com.marcusbiel.java8course.garage;

import com.marcusbiel.java8course.car.Bmw;

public class Car {

     public void drive(speed) {

     }
}
```
Example 9

@Test

The `@` symbol indicates an annotation. Annotations are a way for the programmer to express to the compiler certain background information about a method or a class that isn't directly part of the code. For now, I want to highlight one important annotation. Just know that any method annotated with @Test indicates that a method is a test.

camelCase

When a class, variable, or method consists of more than one word, you use an uppercase letter to indicate the new word. This is called camelCase, because the "humps" from the upper case letters make the words look like camel humps. In some other programming languages, multiple word names are separated with underscores, but in Java "camelCase" is used instead.

Object Oriented Language

Each programming language has a design in mind for its structure. Java is an object oriented language. That is, everything in Java is focused around objects. A clever Java programmer works towards fully understanding the problem and translating his theoretical model into objects and classes that help him achieve his goals.

The main purpose in writing a program is to solve a problem in the most concise way possible. The better job a programmer does, the easier the code will be to maintain, the faster the program will run, and the fewer errors the program will have.

Java's power comes from the idea that a model takes names and concepts from the real world, allowing programmers to communicate easily with less code-savvy clients. While you might be thinking about the code behind your Car class, your client might be thinking of a real car, but both of you will understand each other.

That concludes the Basic Keywords article of my free Java 8 course.

Thanks for reading!