What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

* Mobile applications (specially Android apps)
* Desktop applications
* Web applications
* Web servers and application servers
* Games
* Database connection
* And much, much more!

Why Use Java?

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
* It is one of the most popular programming language in the world
* It is easy to learn and simple to use
* It is open-source and free
* It is secure, fast and powerful
* It has a huge community support (tens of millions of developers)
* Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
* As Java is close to [C++](https://www.w3schools.com/cpp/default.asp) and [C#](https://www.w3schools.com/cs/default.asp), it makes it easy for programmers to switch to Java or vice versa

--------------------------------------------------------------------------------------

## Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

## Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by Java (will not be executed).

## Java Multi-line Comments

Multi-line comments start with /\* and ends with \*/.

Any text between /\* and \*/ will be ignored by Java.

--------------------------------------------------------------------------------------

## Java Data Types

As explained in the previous chapter, a [variable](https://www.w3schools.com/java/java_variables.asp) in Java must be a specified data type:

Data types are divided into two groups:

* Primitive data types - includes byte, short, int, long, float, double, boolean and char
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp) (you will learn more about these in a later chapter)

## Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:





Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are [Strings](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), [Classes,](https://www.w3schools.com/java/java_classes.asp)[Interface](https://www.w3schools.com/java/java_interface.asp), etc. You will learn more about these in a later chapter.

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type

## Java Operators

Operators are used to perform operations on variables and values.

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.



## Java Assignment Operators

Assignment operators are used to assign values to variables.

A list of all assignment operators:



## Java Comparison Operators

Comparison operators are used to compare two values:



## Java Logical Operators

Logical operators are used to determine the logic between variables or values:



## Java Strings

Strings are used for storing text.

A String variable contains a collection of characters surrounded by double quotes

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method

There are many string methods available, for example toUpperCase() and toLowerCase()

The indexOf() method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace)

The + operator can be used between strings to combine them. This is called **concatenation**

You can also use the concat() method to concatenate two strings

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

String txt = "We are the so-called "Vikings" from the north.";

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

The sequence \"  inserts a double quote in a string



Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b
* Equal to a == b
* Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

## Short Hand If...Else

There is also a short-hand [if else](https://www.w3schools.com/java/java_conditions.asp), which is known as the **ternary operator** because it consists of three operands.

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

### **Syntax**

variable *= (*condition*) ?* expressionTrue *:*  expressionFalse*;*

## Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

## Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo

## Change an Array Element

To change the value of a specific element, refer to the index number:

cars[0] = "Opel";

**Array Length**

To find out how many elements an array has, use the length property:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4

## Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

## Multidimensional Arrays

A multidimensional array is an array of arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

### **Example**

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



Another example:



So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

## Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

Create a class named "Main" with a variable x

Remember from the [Java Syntax chapter](https://www.w3schools.com/java/java_syntax.asp) that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

## Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new

## Java Class Attributes

In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class

## Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Modifiers

By now, you are quite familiar with the public keyword that appears in almost all of our examples:

**public** class Main

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

* **Access Modifiers** - controls the access level
* **Non-Access Modifiers** - do not control access level, but provides other functionality

## Access Modifiers

For **classes**, you can use either public or default:



For **attributes, methods and constructors**, you can use the one of the following:



## Non-Access Modifiers

For **classes**, you can use either final or abstract:



For **attributes and methods**, you can use the one of the following:



# **Java Encapsulation**

## Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private
* provide public **get** and **set** methods to access and update the value of a private variable

## Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case

Why Encapsulation?

* Better control of class attributes and methods
* Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data

# **Java Inheritance**

## Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass)

Did you notice the protected modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a protected access modifier. If it was set to private, the Car class would not be able to access it.

#### **Why And When To Use "Inheritance"?**

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

# **Java Polymorphism**

## Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Remember from the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) that we use the extends keyword to inherit from a class.

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

#### **Why And When To Use "Inheritance" and "Polymorphism"?**

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

# **Java Abstraction**

## Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

abstract class Animal {

public abstract void animalSound();

public void sleep() {

System.out.println("Zzz");

}

}

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // will generate an error

To access the abstract class, it must be inherited from another class.

#### **Why And When To Use Abstract Classes and Methods?**

To achieve security - hide certain details and only show the important details of an object.

# **Java Interface**

## Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

## Multiple Interfaces

To implement multiple interfaces, separate them with a comma.

## Java equals()

Object class defined equals() method like this:

public boolean equals(Object obj) {

return (this == obj);

}

According to java documentation of equals() method, any implementation should adhere to following principles.

* For any object x, x.equals(x) should return true.
* For any two object x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* For multiple objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* Multiple invocations of x.equals(y) should return same result, unless any of the object properties is modified that is being used in the equals() method implementation.
* Object class equals() method implementation returns true only when both the references are pointing to same object.

## Java hashCode()

Java Object hashCode() is a native method and returns the integer hash code value of the object. The general contract of hashCode() method is:

* Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
* An object hash code value can change in multiple executions of the same application.
* If two objects are equal according to equals() method, then their hash code must be same.
* If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.

## Importance of equals() and hashCode() method

Java hashCode() and equals() method are used in Hash table based implementations in java for storing and retrieving data. I have explained it in detail at [How HashMap works in java?](https://www.journaldev.com/11560/java-hashmap#how-hashmap-works-in-java)

The implementation of equals() and hashCode() should follow these rules.

* If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

# toString()` method in Java

The toString() method returns the string representation of an object. It is widely used for debugging, printing objects’ contents in logs, etc. This post will discuss how to override the toString() method in Java.

The object class already contains the toString() method, which returns a string that “textually represents” the object. The default implementation of the Object.toString() method returns a string consisting of the class name, '@' character, followed by the unsigned hexadecimal representation of the hash code of the object. i.e,

|  |  |
| --- | --- |
| 1  2  3 | public String toString() {      return getClass().getName() + "@" + Integer.toHexString(hashCode());  } |

Since all Java objects inherit from java.lang.Object, you need to override the toString() method to get the desired string representation. Otherwise, the above default implementation of the toString() method will be invoked when you try to print an object.

## Enums

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters

You can access enum constants with the **dot** syntax:

Level myVar = Level.MEDIUM;

**Enum** is short for "enumerations", which means "specifically listed".

## Enum inside a Class

You can also have an enum inside a class

## Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values

## Loop Through an Enum

The enum type has a values() method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum

#### **Difference between Enums and Classes**

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

#### **Why And When To Use Enums?**

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

finalize() Method in Java and How to Override it?

The Java **finalize() method** of [Object class](https://www.geeksforgeeks.org/object-class-in-java/) is a method that the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.

## Finalization:

Just before destroying any object, the garbage collector always calls finalize() method to perform clean-up activities on that object. This process is known as Finalization in Java.

***Note:****The Garbage collector calls the finalize() method only once on any object.*

## Syntax:

protected void finalize throws Throwable{}

Since the Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class, Garbage Collector can call the finalize() method on any java object.

## Why finalize() method is used?

finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

**How to override finalize() method?**

The finalize method, which is present in the Object class, has an **empty implementation**. In our class, clean-up activities are there. Then we have to **override this method** to define our clean-up activities.

JUnit Tutorial | Testing Framework for Java

JUnit tutorial provides basic and advanced concepts of **unit testing in java** with examples. Our junit tutorial is designed for beginners and professionals.

It is an *open-source testing framework* for java programmers. The java programmer can create test cases and test his/her own code.

It is one of the unit testing framework.

To perform unit testing, we need to create test cases. The **unit test case** is a code which ensures that the program logic works as expected.

The **org.junit** package contains many interfaces and classes for junit testing such as Assert, Test, Before, After etc.

## Types of unit testing

There are two ways to perform unit testing: 1) manual testing 2) automated testing.

#### **1) Manual Testing**

If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

#### **2) Automated Testing**

If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

#### **Annotations for Junit testing**

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

**@Test** annotation specifies that method is the test method.

**@Test(timeout=1000)** annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

**@BeforeClass** annotation specifies that method will be invoked only once, before starting all the tests.

**@Before** annotation specifies that method will be invoked before each test.

**@After** annotation specifies that method will be invoked after each test.

**@AfterClass** annotation specifies that method will be invoked only once, after finishing all the tests.

## Assert class

The org.junit.Assert class provides methods to assert the program logic.

#### Methods of Assert class

The common methods of Assert class are as follows:

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

## Varargs

It's possible to pass an arbitrary number of the same type arguments to a method using the special syntax named **varargs (variable-length arguments)**. These arguments are specified by three dots after the type. In the body of the method, you can process this parameter as a regular array of the specified type.

The following method takes an integer **vararg**parameter and outputs the number of arguments in the standard output using the**length** property of arrays.

public static void printNumberOfArguments(int... numbers) {  
    System.out.println(numbers.length);  
}

As you can see, a special syntax **...** is used here to specify a **vararg** parameter.

Now, you can invoke the method passing several integer numbers or an array of ints.

printNumberOfArguments(1);  
printNumberOfArguments(1, 2);  
printNumberOfArguments(1, 2, 3);  
printNumberOfArguments(new int[] { }); // no arguments here  
printNumberOfArguments(new int[] { 1, 2 });

This code outputs:

1  
2  
3  
0  
2

This example also demonstrates the difference between the arguments and parameters of a method. The method has only a single parameter but it can be called with several arguments.

**Varargs and other parameters**

If a method has more than one parameter, the vararg parameter must be the last one in the declaration of the method.

First example is incorrect, second one is correct:

public static void method(double... varargs, int a) { /\* do something \*/ }

public static void method(int a, double... varargs) { /\* do something \*/ }

## JVM

**Writing a program**

As a developer, the first thing you do while creating a program is writing the **source code** in a plain text file and saving it with an extension corresponding to the programming language you've chosen (.java for the Java language, .kt for Kotlin, and so on). A single program consists of one or more such files, which contain instructions specifying what the program does. The source code must follow the syntax rules of the respective language and be easy to read and understand.

**Compilation**

After the code is written, you need to make the computer run the program. As computers don't understand the source code, it needs to be translated into a computer-comprehensible format. That's where a special program called a **compiler** comes in handy. The code obtained after compilation is called native code or low-level code. Each computer platform uses different low-level commands, just like people around the world speak different languages. It creates an additional challenge to use a program on different devices.

In the world of Java, a compiler (the javac tool for Java or the kotlinc tool for Kotlin) translates source code into an intermediate representation known as **Java bytecode**stored in files with a .class extension. Computers can't read bytecode without translation, but a system called the Java Virtual Machine (JVM) can execute it.

**Running a program**

The **Java Virtual Machine** is an application that represents a virtual computer according to the JVM specification document. It executes the compiled Java bytecode and translates it into low-level commands, which the computer understands. Each platform has its own version of the JVM, but since all JVMs match the same specification, your program will behave identically on different devices.

One of the main concepts of the Java Platform is "write once, run anywhere". It means that a program can run on various devices as long as they have a JVM installed. This concept is also frequently called **platform independence** or portability.

It's important to remember: the code input into the JVM is platform-independent, while the output code is platform-dependent.



*A platform-independent program in the world of Java*

If the JVM is installed on the computer, you can run a compiled JVM program using the java tool. It will open a file with the.class extension to launch the program from this file. The tool is the same for all JVM languages.

The picture below briefly summarizes the work cycle of a JVM program.



**JVM languages**

The Java Platform allows using more than one programming language to create programs. This is achieved by the design of the JVM: it doesn't know anything about any particular programming language. It only understands Java bytecode. If the tools for a programming language can generate bytecode, programs written in this language can be executed on the JVM. Such languages are often called **JVM languages**. They include Java itself, Kotlin, Scala, Groovy, Clojure, and others. So, to create programs in the world of Java, you can choose the most convenient language of your choice.

Nowadays, you can find tools to generate Java bytecode for almost any programming language, which means that there's hardly any language that is not a JVM language.



*Different programming languages can be used to write programs in the world of Java*

Methods provided by the Object class

The Object class provides some common methods to all subclasses. It has nine instance methods (excluding overloaded methods) which can be divided into four groups:

* **threads synchronization:** wait, notify, notifyAll;
* **object identity**: hashCode, equals;
* **object management**: finalize, clone, getClass;
* **human-readable representation**: toString;

This way of grouping methods isn't perfect, but it can help you remember them. Here's a more detailed explanation of the methods:

* The first group of methods (wait, notify, notifyAll) are for working in multithreaded applications.
* hashCode returns a hash code value for the object.
* equals indicates whether some other object is **"equal to"** this particular one.
* finalize is called by the garbage collector (GC) on an object when the GC wants to clean it up. (**Note:**this method has been deprecated as of JDK 9).
* clone creates and returns a copy of the object.
* getClass returns an instance of Class, which has information about the runtime class.
* toString returns a string representation of the object.

Some of the methods listed above are native, which means they are implemented in the **native** code. It is typically written in C or C++. Native methods are usually used to interface with system calls or libraries written in other programming languages.

In the following topics, we will consider these class methods in more detail.

# **Difference between JDK, JRE, and JVM**

### **JVM**

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](https://www.javatpoint.com/os-tutorial)

is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

2

The JVM performs the following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

### **JRE**

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



### **JDK**

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and [applets](https://www.javatpoint.com/java-applet)

. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



# **JVM (Java Virtual Machine) Architecture**

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

### **What is JVM**

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

### **What it does**

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

## JVM Architecture

Let's understand the internal architecture of JVM.



### **1) Classloader**

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside $JAVA\_HOME/jre/lib/ext directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

**2) Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### **3) Heap**

It is the runtime data area in which objects are allocated.

### **4) Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

### **5) Program Counter Register**

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### **6) Native Method Stack**

It contains all the native methods used in the application.

### **7) Execution Engine**

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

### **8) Java Native Interface**

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

## Multiple constructors

##### Theory

Sometimes we need to initialize all fields of an object when creating it, but there are cases in which it might be appropriate to initialize only one or several fields. Fortunately, for this purpose, a class can have several constructors that assign values to the fields in different ways. In this topic, you will learn how to work with multiple constructors and define the way they interact with each other.

##### Constructor overloading

You can define as many constructors as you need. Each constructor should have a name that matches the class name but the parameters should be different. The situation when a class contains multiple constructors is known as **constructor overloading**.

Example in Lesson 10 Robot class:

The class Robot has two constructors:

* Robot() is a no-argument constructor that initializes fields with default values;
* Robot(String name, String model) takes two parameters and assigns them to the corresponding fields.

To create an instance of the class Robot we can use either of the two constructors:

Robot anonymous = new Robot(); // name is "Anonymous", model is "Unknown"  
Robot andrew = new Robot("Andrew", "NDR-114"); // name is "Andrew", model is "NDR-114"

Bear in mind that you cannot define two constructors with the same number, types, and order of the parameters!

##### Invoking constructors from other constructors

We can also invoke a constructor from another one. It allows you to initialize one part of an object by one constructor and another part by another constructor.

Calling a constructor inside another one is done using this. For example:

this(); // calls a no-argument constructor

If you called a constructor that has parameters you can pass some arguments:

this("arg1", "arg2"); // calls a constructor with two string arguments

Remember, the statement for invoking a constructor should be the first statement in the body of a caller constructor.

Now, the class has three constructors:

* Robot() is a no-argument constructor;
* Robot(String name, String model) is a two-argument constructor that invokes another constructor;
* Robot(String name, String model, int lifetime) is a three-argument constructor that fills all fields.

The second constructor invokes the third one and passes name, model, and lifetime = 20 to it. The third constructor, in its turn, initializes all fields of the created object.

Super keyword

## Teory

Sometimes when we define a new subclass we need to access members or constructors of its superclass. Java provides a special keyword super to do this. This keyword can be used in several cases:

* to access instance fields of the parent class;
* to invoke methods of the parent class;
* to invoke constructors of the parent class (no-arg or parameterized).

Let's consider all of these cases with examples.

## Accessing superclass fields and methods

The keyword super can be used to access instance methods or fields of the superclass. In a sense, it is similar to the keyword this, but it refers to the immediate parent class object.

The keyword super is optional if members of a subclass have different names from members of the superclass. Otherwise, using super is the right way to access hidden (with the same name) members of the base class.

**Example.** There are two classes: SuperClass and SubClass. Each class has a field and a method.

In the constructor of SubClass , the superclass field is initialized using the keyword super. We need to use the keyword here because the subclass field hides the base class field with the same name.

In the body of the method printSubValue , the superclass method printBaseValue is invoked. Here, the keyword super is optional. It is required when a subclass method has the same name as a method in the base class. This case will be considered in the topic concerning overriding.

## Invoking superclass constructor

Constructors are not inherited by subclasses, but a superclass constructor can be invoked from a subclass using the keyword super **with parentheses**. We can also pass some arguments to the superclass constructor.

[ALERT-warning]Two important points:

* invoking super(...) must be the first statement in a subclass constructor, otherwise, the code cannot be compiled;
* the default constructor of a subclass automatically calls the no-argument constructor of the superclass.[/ALERT]

**Example.** Here are two classes Person and Employee. The second class extends the first one. Each class has a constructor to initialize fields.

In the provided example, the constructor of the class Employee invokes the parent class constructor for assigning values to the passed fields. In a way, it resembles working with multiple constructors using this().

## Objects

##### Theory

A typical object-oriented program consists of a set of interacting **objects**. Each object has its own state separated from others. Each object is an instance of a particular class (type) that defines common properties and possible behavior for its objects.

All classes from the standard library (**String**, **Date**) and classes defined by programmers are **reference types** which means that variables of these types store addresses where the actual objects are located. In this regard, the comparison and assignment operations work with objects differently than with primitive types.

##### Creating objects

The keyword **new** creates an object of a particular class. Here we create a standard string and assign it to the variable **str**:

String str = new String("hello");

The variable **str** stores a reference to the object **"hello"** located somewhere in the heap memory.

In the same way, we can create an object of any class we know.

Here is a class that describes a patient in a hospital information system:

class Patient {  
    String name;  
    int age;  
}

Here is an instance of this class:

Patient patient = new Patient();

Despite the fact that **String** is a standard class and **Patient** is our own class, both classes are regular reference types. However, there is a big difference between those classes and we will discuss it below.

##### Immutability of objects

There is an important concept in programming called **immutability**. Immutability means that an object always stores the same values. If we need to modify these values, we should create a new object. The common example is the standard **String** class. Strings are immutable objects so all string operations produce a new string. Immutable types allow you to write programs with fewer errors.

The class **Patient** is not immutable because it is possible to change any field of an object.

Patient patient = new Patient();  
  
patient.name = "Mary";  
patient.name = "Alice";

In the following topics, we will look at the existing immutable classes as well as learn how to create new ones and when to use them.

##### Sharing references

More than one variable can refer to the same object.

Patient patient = new Patient();  
  
patient.name = "Mary";  
patient.age = 24;  
  
System.out.println(patient.name + " " + patient.age); // Mary 24  
  
Patient p = patient;  
  
System.out.println(p.name + " " + p.age); // Mary 24

It is important to understand that two variables refer to the same data in memory rather than two independent copies. Since our class is mutable, we can modify the object using both references.

patient.age = 25;  
System.out.println(p.age); // 25

##### Nullability

As for any reference types, a variable of class type can be **null** which means it is not initialized yet.

Patient patient = null;

This is a common feature in Java available for classes since they are reference types.

# **Collections in Java**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList, Vector, LinkedList](https://www.javatpoint.com/java-arraylist)

[, PriorityQueue, HashSet, LinkedHashSet, TreeSet).](https://www.javatpoint.com/java-arraylist)

#### **What is Collection in Java**

A Collection represents a single unit of objects, i.e., a group.

#### **What is a framework in Java**

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

#### **What is Collection framework**

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Do You Know?

* What are the two ways to iterate the elements of a collection?
* What is the difference between ArrayList and LinkedList classes in collection framework?
* What is the difference between ArrayList and Vector classes in collection framework?
* What is the difference between HashSet and HashMap classes in collection framework?
* What is the difference between HashMap and Hashtable class?
* What is the difference between Iterator and Enumeration interface in collection framework?
* How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces?
* What does the hashcode() method?
* What is the difference between Java collection and Java collections?

### **Hierarchy of Collection Framework**

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.





### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### 

### **Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

#### **Methods of Iterator interface**

There are only three methods in the Iterator interface. They are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | | **Method** | | **Description** |
| 1 | public boolean hasNext() | | It returns true if the iterator has more elements otherwise it returns false. | |
| 2 | public Object next() | | It returns the element and moves the cursor pointer to the next element. | |
| 3 | public void remove() | | It removes the last elements returned by the iterator. It is less used. | |

## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

Example in month2.lesson1.

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

Example in month2.lesson1.

## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

Example in month2.lesson1.

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

Example in month2.lesson1.

## Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection5{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. **while**(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

## ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection6{
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Gautam

Karan

Ajay

## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

## HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Vijay

Ravi

Ajay

## LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ajay

## 

## SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

## TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ajay

Ravi

Vijay

# **Java Map Interface**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

|  |  |
| --- | --- |
| **Class** | **Description** |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

### **Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>> entrySet() | It returns the Set view containing all the keys and values. |
| void clear() | It is used to reset the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| int hashCode() | It returns the hash code value for the Map |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

## Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

### **Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

### **Java Map Example:**

***REPO: lessons***

***Package: month2.lesson4***

# **Data Access Object Pattern**

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

* **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
* **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
* **Model Object or Value Object** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

## Implementation

We are going to create a *Student* object acting as a Model or Value Object.*StudentDao* is Data Access Object Interface.*StudentDaoImpl* is concrete class implementing Data Access Object Interface. *DaoPatternDemo*, our demo class, will use *StudentDao* to demonstrate the use of Data Access Object pattern.



## Step 1

Create Value Object.

*REPO: lessons*

*Package: month2.lesson5*

*Student.java*

## Step 2

Create Data Access Object Interface.

*REPO: lessons*

*Package: month2.lesson5*

*Dao.java*

## Step 3

Create concrete class implementing above interface.

*REPO: lessons*

*Package: month2.lesson5*

*StudentDao.java*

## Step 4

Use the *StudentDao* to demonstrate Data Access Object pattern usage.

*REPO: lessons*

*Package: month2.lesson5*

*Main.java*

## Step 5

Verify the output.

# **Java Exceptions - Try...Catch**

## Java Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

## Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

### **Syntax**

try {

// *Block of code to try*

}

catch(Exception e) {

// *Block of code to handle errors*

}

Consider the following example:

*REPO: lessons*

*Package: month2.lesson6*

*ExceptionExample.java*

This will generate an error, because **myNumbers[10]** does not exist.

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

*REPO: lessons*

*Package: month2.lesson6*

*TryCatch.java*

## Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

*REPO: lessons*

*Package: month2.lesson5*

*TryCatchFinally.java*

## The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

### **Example**

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

If **age** was 20, you would **not** get an exception:

*REPO: lessons*

*Package: month2.lesson5*

*ThrowStatement.java*

# **Java Date and Time**

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time. Following classes are important for dealing with date in Java.

## Java 8 Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

1. java.time.LocalDate class
2. java.time.LocalTime class
3. java.time.LocalDateTime class
4. java.time.MonthDay class
5. java.time.OffsetTime class
6. java.time.OffsetDateTime class
7. java.time.Clock class
8. java.time.ZonedDateTime class
9. java.time.ZoneId class
10. java.time.ZoneOffset class
11. java.time.Year class
12. java.time.YearMonth class
13. java.time.Period class
14. java.time.Duration class
15. java.time.Instant class
16. java.time.DayOfWeek enum
17. java.time.Month enum

## Classical Date/Time API

But classical or old Java Date API is also useful. Let's see the list of classical Date and Time classes.

1. java.util.Date class
2. java.sql.Date class
3. java.util.Calendar class
4. java.util.GregorianCalendar class
5. java.util.TimeZone class
6. java.sql.Time class
7. java.sql.Timestamp class

## Formatting Date and Time

We can format date and time in Java by using the following classes:

1. java.text.DateFormat class
2. java.text.SimpleDateFormat class

## Java Date and Time APIs

Java provide the date and time functionality with the help of two packages java.time and java.util. The package java.time is introduced in Java 8, and the newly introduced classes tries to overcome the shortcomings of the legacy java.util.Date and java.util.Calendar classes.

### **Classical Date Time API Classes**

The primary classes before Java 8 release were:

**Java.lang.System:** The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.

**java.util.Date:** It is used to show specific instant of time, with unit of millisecond.

**java.util.Calendar:** It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.

**java.text.SimpleDateFormat:** It is a class that is used to format and parse the dates in a predefined manner or user defined pattern.

**java.util.TimeZone:** It represents a time zone offset, and also figures out daylight savings.

### **Drawbacks of existing Date/Time API's**

1. **Thread safety:** The existing classes such as Date and Calendar does not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care by developers. The new Date and Time APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.
2. **Bad API designing:** The classic Date and Calendar APIs does not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provides number of different methods for performing operations regarding date, time, duration and periods.
3. **Difficult time zone handling:** To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.

### **New Date Time API in Java 8**

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

**java.time.LocalDate:** It represents a year-month-day in the ISO calendar and is useful for representing a date without a time. It can be used to represent a date only information such as a birth date or wedding date.

**java.time.LocalTime:** It deals in time only. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.

**java.time.LocalDateTime:** It handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.

**java.time.ZonedDateTime:** It combines the LocalDateTime class with the zone information given in ZoneId class. It represent a complete date time stamp along with timezone information.

**java.time.OffsetTime:** It handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

**java.time.OffsetDateTime:** It handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

**java.time.Clock :** It provides access to the current instant, date and time in any given time-zone. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.

**java.time.Instant :** It represents the start of a nanosecond on the timeline (since EPOCH) and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.

**java.time.Duration :** Difference between two instants and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.

**java.time.Period :** It is used to define the difference between dates in date-based values (years, months, days).

**java.time.ZoneId :** It states a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.

**java.time.ZoneOffset :** It describe a time zone offset from Greenwich/UTC time.

**java.time.format.DateTimeFormatter :** It comes up with various predefined formatter, or we can define our own. It has parse() or format() method for parsing and formatting the date time values.

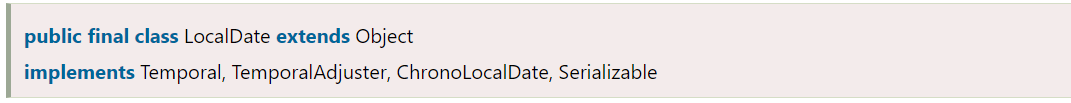
Начало формы

# **Java LocalDate class**

Java LocalDate class is an immutable class that represents Date with a default format of yyyy-mm-dd. It inherits Object class and implements the ChronoLocalDate interface

## Java LocalDate class declaration

Let's see the declaration of java.time.LocalDate class.



### **Methods of Java LocalDate**

|  |  |
| --- | --- |
| **Method** | **Description** |
| LocalDateTime atTime(int hour, int minute) | It is used to combine this date with a time to create a LocalDateTime. |
| int compareTo(ChronoLocalDate other) | It is used to compares this date to another date. |
| boolean equals(Object obj) | It is used to check if this date is equal to another date. |
| String format(DateTimeFormatter formatter) | It is used to format this date using the specified formatter. |
| int get(TemporalField field) | It is used to get the value of the specified field from this date as an int. |
| boolean isLeapYear() | It is used to check if the year is a leap year, according to the ISO proleptic calendar system rules. |
| LocalDate minusDays(long daysToSubtract) | It is used to return a copy of this LocalDate with the specified number of days subtracted. |
| LocalDate minusMonths(long monthsToSubtract) | It is used to return a copy of this LocalDate with the specified number of months subtracted. |
| static LocalDate now() | It is used to obtain the current date from the system clock in the default time-zone. |
| LocalDate plusDays(long daysToAdd) | It is used to return a copy of this LocalDate with the specified number of days added. |
| LocalDate plusMonths(long monthsToAdd) | It is used to return a copy of this LocalDate with the specified number of months added. |
| LocalDate plusMonths(long monthsToAdd) | It is used to return a copy of this LocalDate with the specified number of months added. |
| int getDayOfMonth() | It gets the day-of-month field. |
| DayOfWeek getDayOfWeek() | It gets the day-of-week field, which is an enum DayOfWeek. |
| int getDayOfYear() | It gets the day-of-year field. |
| Month getMonth() | It gets the month-of-year field using the Month enum. |
| int getMonthValue() | It gets the month-of-year field from 1 to 12. |
| int getYear() | It gets the year field. |
| int lengthOfMonth() | It returns the length of the month represented by this date. |
| int lengthOfYear() | It returns the length of the year represented by this date. |
| static LocalDate ofYearDay(int year, int dayOfYear) | It obtains an instance of LocalDate from a year and day-of-year. |
| static LocalDate parse(CharSequence text) | It obtains an instance of LocalDate from a text string such as 2007-12-03 |
| static LocalDate parse(CharSequence text, DateTimeFormatter formatter) | It obtains an instance of LocalDate from a text string using a specific formatter. |

## Java LocalDate Example

Program to demonstrate methods of LocalDate class such as now(), minusDays(), plusDays().

*REPO: lessons*

*Package: month2.lesson8*

*LocalDate.java*

Program to demonstrate isLeapYear() method of LocalDate Class.

*REPO: lessons*

*Package: month2.lesson8*

*LocalDateLeap.java*

Program to demonstrate atTime() method of LocalDate class.

*REPO: lessons*

*Package: month2.lesson8*

*LocalDateAtTime.java*

Program to demonstrate format() method of LocalDate class

*REPO: lessons*

*Package: month2.lesson8*

*LocalDateFormat.java*

Program to demonstrate parse() method of LocalDate class

*REPO: lessons*

*Package: month2.lesson8*

*LocalDateParse.java*

# **Java LocalTime Class**

Java LocalTime class is an immutable class that represents time with a default format of hour-minute-second. It inherits Object class and implements the Comparable interface.

## Java LocalTime class declaration

Let's see the declaration of java.time.LocalTime class.

1. **public** **final** **class** LocalTime **extends** Object
2. **implements** Temporal, TemporalAdjuster, Comparable<LocalTime>, Serializable

### **Methods of Java LocalTime Class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| LocalDateTime atDate(LocalDate date) | It is used to combine this time with a date to create a LocalDateTime. |
| int compareTo(LocalTime other) | It is used to compare this time to another time. |
| String format(DateTimeFormatter formatter) | It is used to format this time using the specified formatter. |
| int get(TemporalField field) | It is used to get the value of the specified field from this time as an int. |
| LocalTime minusHours(long hoursToSubtract) | It is used to return a copy of this LocalTime with the specified number of hours subtracted. |
| LocalTime minusMinutes(long minutesToSubtract) | It is used to return a copy of this LocalTime with the specified number of minutes subtracted. |
| static LocalTime now() | It is used to obtain the current time from the system clock in the default time-zone. |
| static LocalTime of(int hour, int minute, int second) | It is used to obtain an instance of LocalTime from an hour, minute and second. |
| LocalTime plusHours(long hoursToAdd) | It is used to return a copy of this LocalTime with the specified number of hours added. |
| LocalTime plusMinutes(long minutesToAdd) | It is used to return a copy of this LocalTime with the specified number of minutes added. |

## Java LocalTime Example: now()

*REPO: lessons*

*Package: month2.lesson10*

*LocalTimeNow.java*

## Java LocalTime Example: of()

*REPO: lessons*

*Package: month2.lesson10*

*LocalTimeOf.java*

## Java LocalTime Example: minusHours() and minusMinutes(), plusHours() and plusMinutes()

*REPO: lessons*

*Package: month2.lesson10*

*LocalTimeMinus.java*

## Java LocalTime Example

*REPO: lessons*

*Package: month2.lesson10*

*LocalTime.java*



# **Java LocalDateTime class**

Java LocalDateTime class is an immutable date-time object that represents a date-time, with the default format as yyyy-MM-dd-HH-mm-ss.zzz. It inherits object class and implements the ChronoLocalDateTime interface.

## Java LocalDateTime class declaration

Let's see the declaration of java.time.LocalDateTime class.

1. **public** **final** **class** LocalDateTime **extends** Object
2. **implements** Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable

### **Methods of Java LocalDateTime**

|  |  |
| --- | --- |
| **Method** | **Description** |
| String format(DateTimeFormatter formatter) | It is used to format this date-time using the specified formatter. |
| int get(TemporalField field) | It is used to get the value of the specified field from this date-time as an int. |
| LocalDateTime minusDays(long days) | It is used to return a copy of this LocalDateTime with the specified number of days subtracted. |
| static LocalDateTime now() | It is used to obtain the current date-time from the system clock in the default time-zone. |
| static LocalDateTime of(LocalDate date, LocalTime time) | It is used to obtain an instance of LocalDateTime from a date and time. |
| LocalDateTime plusDays(long days) | It is used to return a copy of this LocalDateTime with the specified number of days added. |
| boolean equals(Object obj) | It is used to check if this date-time is equal to another date-time. |

## Java LocalDateTime Example

*REPO: lessons*

*Package: month2.lesson12*

*LocalDateTimeEx.java*

## Java LocalDateTime Example: now()

*REPO: lessons*

*Package: month2.lesson12*

*LocalDateTimeNow.java*

## Java LocalDateTime Example: get()

*REPO: lessons*

*Package: month2.lesson12*

*LocalDateTimeGet.java*

## Java LocalDateTime Example: minusPlusDays()

*REPO: lessons*

*Package: month2.lesson12*

*LocalDateTimePlusMinusDays.java*

# **Java Files**

File handling is an important part of any application.

Java has several methods for creating, reading, updating, and deleting files.

## Java File Handling

The File class from the java.io package, allows us to work with files.

To use the File class, create an object of the class, and specify the filename or directory name:

File myObj = new File("filename.txt"); // Specify the filename

The File class has many useful methods for creating and getting information about files. For example:

|  |  |  |
| --- | --- | --- |
| **Method** | **Type** | **Description** |
| canRead() | Boolean | Tests whether the file is readable or not |
| canWrite() | Boolean | Tests whether the file is writable or not |
| createNewFile() | Boolean | Creates an empty file |
| delete() | Boolean | Deletes a file |
| exists() | Boolean | Tests whether the file exists |
| getName() | String | Returns the name of the file |
| getAbsolutePath() | String | Returns the absolute pathname of the file |
| length() | Long | Returns the size of the file in bytes |
| list() | String[] | Returns an array of the files in the directory |
| mkdir() | Boolean | Creates a directory |

# **Java Create and Write To Files**

## Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

*REPO: lessons*

*Package: month3.lesson1*

*CreateFile.java*

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

### **Example**

File myObj = new File("C:\\Users\\MyName\\filename.txt");

## Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

*REPO: lessons*

*Package: month3.lesson1*

*WriteToFile.java*

# **Java Read Files**

## Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter:

*REPO: lessons*

*Package: month3.lesson3*

*ReadFile.java*

## Get File Information

To get more information about a file, use any of the File methods:

*REPO: lessons*

*Package: month3.lesson3*

*GetFileInfo.java*

# **Java Delete Files**

## Delete a File

To delete a file in Java, use the delete() method:

*REPO: lessons*

*Package: month3.lesson3*

*DeleteFile.java*

## Delete a Folder

You can also delete a folder. However, it must be empty:

*REPO: lessons*

*Package: month3.lesson3*

*DeleteFolder.java*

## Differences between @RequestParam and @PathVariable in Spring MVC

Both annotations @RequestParam and @PathVariable in [Spring MVC](https://www.dineshonjava.com/spring-web-mvc-framework-chapter-38/) are used for fetching the values of request parameters. These annotations have similar purpose but some differences in use. The key difference between @RequestParam and @PathVariable is that @RequestParam used for accessing the values of the query parameters where as @PathVariable used for accessing the values from the URI template.

## [@RequestParam](https://www.dineshonjava.com/requestparam-annotation-in-spring-mvc-with-example/)

It is used to get the request parameters. @RequestParam automatically binds the request parameters to the arguments of your handler method. It also provides auto type conversion for some standard type like int, long, float, string, date etc.

**Look at the following request URL:**

***http://localhost:8080/tutorials/bookmark/?site=dineshonjava&id=200***

In the above URL request, the values for site and id can be accessed as below:

@RequestMapping(value = "/tutorials/bookmark")

public String bookmark(

@RequestParam(value="site", required=true) String site,

@RequestParam(value="id", required=false) String id){

...

}

* **defaultValue**– It is String type attribute and the default value to use as a fallback when the request parameter is not provided or has an empty value.
* **name**– It is String type attribute and name of the request parameter to bind to.
* **required**– It is Boolean type attribute whether the parameter is required.
* **value**– It is String type attribute and it is alias for name attribute.

## @PathVariable

It is used to pass parameter along with the url, sometimes we need to pass parameters along with the url to get the data. Spring MVC provides support for customizing the URL in order to get data. To achieving this purpose @PathVariable annotation is used in Spring mvc framework.

**Look at the following request URL:**  
 ***http://localhost:8080/tutorials/bookmark/100?site=dineshonjava&id=200***

In the above URL request, the values for site and id can be accessed as below:

@RequestMapping(value = "/tutorials/bookmark/{siteId}")

public String bookmark(

@PathVariable(value="siteId") String siteId

@RequestParam(value="site", required=true) String site,

@RequestParam(value="id", required=false) String id){

...

}

**value**– It is String type attribute and it is only one attribute of the @PathVariable annotation. It is allowed to use the multiple @PathVariable annotation in the single method. But, ensure that no more than one method has the same pattern.

**Example of @PathVariable and @RequestParam**  
In this example we demonstrate the key difference between @PathVariable and @RequestParam annotations.

## Summary

This tutorial we have explained you to understand the difference between @PathVariable and @RequestParam in the Spring MVC. Basically purpose of both annotation in the same but way of fetching the request parameter value is different. @RequestParam annotation fetch the value of request parameter in the form of passing request parameter with url but @PathVariable annotation fetching value of the parameter in the form request URI template with some placeholder.

# **HTTP Request Methods**

## What is HTTP?

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.

HTTP works as a request-response protocol between a client and server.

Example: A client (browser) sends an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.

## HTTP Methods

* **GET**
* **POST**
* **PUT**
* **HEAD**
* **DELETE**
* **PATCH**
* **OPTIONS**
* **CONNECT**
* **TRACE**

The two most common HTTP methods are: GET and POST.

## The GET Method

GET is used to request data from a specified resource.

Note that the query string (name/value pairs) is sent in the URL of a GET request:

/test/demo\_form.php?name1=value1&name2=value2

**Some notes on GET requests:**

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests are only used to request data (not modify)

## The POST Method

POST is used to send data to a server to create/update a resource.

The data sent to the server with POST is stored in the request body of the HTTP request:

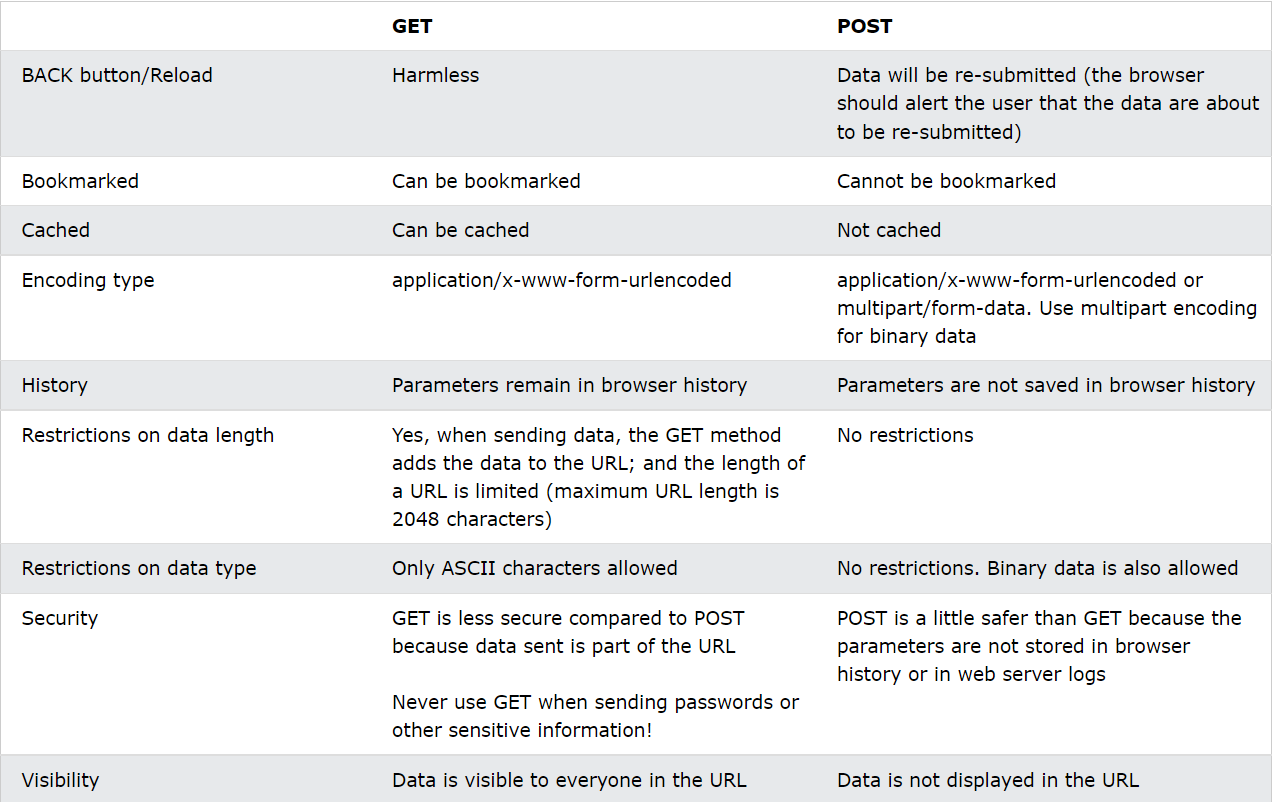
POST /test/demo\_form.php HTTP/1.1  
Host: w3schools.com  
  
name1=value1&name2=value2

**Some notes on POST requests:**

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

## Compare GET vs. POST

The following table compares the two HTTP methods: GET and POST.



## The PUT Method

PUT is used to send data to a server to create/update a resource.

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

## The HEAD Method

HEAD is almost identical to GET, but without the response body.

In other words, if GET /users returns a list of users, then HEAD /users will make the same request but will not return the list of users.

HEAD requests are useful for checking what a GET request will return before actually making a GET request - like before downloading a large file or response body.

## The DELETE Method

The DELETE method deletes the specified resource.

## The PATCH Method

The PATCH method is used to apply partial modifications to a resource.

## The OPTIONS Method

The OPTIONS method describes the communication options for the target resource.

## The CONNECT Method

The CONNECT method is used to start a two-way communications (a tunnel) with the requested resource.

## The TRACE Method

The TRACE method is used to perform a message loop-back test that tests the path for the target resource (useful for debugging purposes).

# Postman

## What is Postman?

**Postman** is a scalable API testing tool that quickly integrates into CI/CD pipeline. It started in 2012 as a side project by Abhinav Asthana to simplify API workflow in testing and development. API stands for Application Programming Interface which allows software applications to communicate with each other via API calls.

## Why Use Postman?

With over 4 million users nowadays, Postman Software has become a tool of choice for the following reasons:

1. Accessibility – To use Postman tool, one would just need to log-in to their own accounts making it easy to access files anytime, anywhere as long as a Postman application is installed on the computer.
2. Use of Collections – Postman lets users create collections for their Postman API calls. Each collection can create subfolders and multiple requests. This helps in organizing your test suites.
3. Collaboration – Collections and environments can be imported or exported making it easy to share files. A direct link can also be used to share collections.
4. Creating Environments – Having multiple environments aids in less repetition of tests as one can use the same collection but for a different environment. This is where parameterization will take place which we will discuss in further lessons.
5. Creation of Tests – Test checkpoints such as verifying for successful HTTP response status can be added to each Postman API calls which help ensure [test coverage](https://www.guru99.com/test-coverage-in-software-testing.html).
6. Automation Testing – Through the use of the Collection Runner or Newman, tests can be run in multiple iterations saving time for repetitive tests.
7. Debugging – Postman console helps to check what data has been retrieved making it easy to debug tests.
8. [Continuous Integration](https://www.guru99.com/continuous-integration.html) – With its ability to support continuous integration, development practices are maintained.

# **Spring’s RequestBody and ResponseBody Annotations**

## ****1.****@RequestBody

Simply put, **the @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization** of the inbound HttpRequest body onto a Java object.

First, let's have a look at a Spring controller method:

@PostMapping("/request")

**public** ResponseEntity **postController**(

@RequestBody LoginForm loginForm) {

exampleService.fakeAuthenticate(loginForm);

**return** ResponseEntity.ok(HttpStatus.OK);

}

Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

By default, **the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side controller:**

**public** **class** **LoginForm** {

**private** String username;

**private** String password;

// ...

}

Here, the object we use to represent the HttpRequest body maps to our LoginForm object.

Let's test this using CURL:

curl -i \

-H "Accept: application/json" \

-H "Content-Type:application/json" \

-X POST --data

'{"username": "johnny", "password": "password"}' "https://localhost:8080/.../request"Copy

This is all we need for a Spring REST API and an Angular client using the @RequestBody annotation.

## ****2.****@ResponseBody

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

Suppose we have a custom Response object:

**public** **class** **ResponseTransfer** {

**private** String text;

// standard getters/setters

}

Next, the associated controller can be implemented:

@Controller

@RequestMapping("/post")

**public** **class** **ExamplePostController** {

@Autowired

ExampleService exampleService;

@PostMapping("/response")

@ResponseBody

**public** ResponseTransfer **postResponseController**(

@RequestBody LoginForm loginForm) {

**return** **new** **ResponseTransfer**("Thanks For Posting!!!");

}

}

In the developer console of our browser or using a tool like Postman, we can see the following response:

{"text":"Thanks For Posting!!!"}

**Remember, we don't need to annotate the @RestController-annotated controllers with the @ResponseBody annotation** since Spring does it by default.

# **Introduction to Using Thymeleaf in Spring**

## ****1. Overview****

[Thymeleaf](http://www.thymeleaf.org/) is a Java template engine for processing and creating HTML, XML, JavaScript, CSS and text.

In this tutorial, we will discuss **how to use Thymeleaf with Spring** along with some basic use cases in the view layer of a Spring MVC application.

The library is extremely extensible, and its natural templating capability ensures we can prototype templates without a back end. This makes development very fast when compared with other popular template engines such as JSP.

## 2. ****Thymeleaf in Spring Boot****

Spring Boot provides auto-configuration for Thymeleaf by adding the [spring-boot-starter-thymeleaf](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-boot-starter-thymeleaf%22) dependency:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>

<**version**>2.3.3.RELEASE</**version**>

</**dependency**>

No explicit configuration is necessary. By default, HTML files should be placed in the resources/templates location.

## ****4. Displaying Values From Message Source (Property Files)****

We can use the th:text=”#{key}” tag attribute to display values from property files.

For this to work, we need to configure the property file as a messageSource bean:

Here is the Thymeleaf HTML code to display the value associated with the key welcome.message:

<**span** th:text="#{welcome.message}" />

## ****5. Displaying Model Attributes****

### **5.1. Simple Attributes**

We can use the th:text=”${attributename}” tag attribute to display the value of model attributes.

Let's add a model attribute with the name serverTime in the controller class:

model.addAttribute("serverTime", dateFormat.format(**new** **Date**()));

And here's the HTML code to display the value of serverTime attribute:

Current time is <**span** th:text="${serverTime}" />

### **5.2. Collection Attributes**

If the model attribute is a collection of objects, we can use the th:each tag attribute to iterate over it.

Let's define a Student model class with two fields, id and name:

**public** **class** **Student** **implements** **Serializable** {

**private** Integer id;

**private** String name;

// standard getters and setters

}

Now we will add a list of students as model attribute in the controller class:

List<Student> students = **new** **ArrayList**<Student>();

// logic to build student data

model.addAttribute("students", students);

Finally, we can use Thymeleaf template code to iterate over the list of students and display all field values:

<**tbody**>

<**tr** th:each="student: ${students}">

<**td** th:text="${student.id}" />

<**td** th:text="${student.name}" />

</**tr**>

</**tbody**>

## ****6. Conditional Evaluation****

### **6.1.**if**and**unless

We use the th:if=”${condition}” attribute to display a section of the view if the condition is met. And we use the th:unless=”${condition}” attribute to display a section of the view if the condition is not met.

Let's add a gender field to the Student model:

**public** **class** **Student** **implements** **Serializable** {

**private** Integer id;

**private** String name;

**private** Character gender;

// standard getters and setters

}

Suppose this field has two possible values (M or F) to indicate the student's gender.

If we wish to display the words “Male” or “Female” instead of the single character, we could do this using this Thymeleaf code:

<**td**>

<**span** th:if="${student.gender} == 'M'" th:text="Male" />

<**span** th:unless="${student.gender} == 'M'" th:text="Female" />

</**td**>

### **6.2.**switch **and**case

We use the th:switch and th:case attributes to display content conditionally using the switch statement structure.

Let's rewrite the previous code using the th:switch and th:case attributes:

<**td** th:switch="${student.gender}">

<**span** th:case="'M'" th:text="Male" />

<**span** th:case="'F'" th:text="Female" />

</**td**>

## ****7. Handling User Input****

We can handle form input using the th:action=”@{url}” and th:object=”${object}” attributes. We use th:action to provide the form action URL and th:object to specify an object to which the submitted form data will be bound.

Individual fields are mapped using the th:field=”\*{name}” attribute, where the name is the matching property of the object.

For the Student class, we can create an input form:

<**form** action="#" th:action="@{/saveStudent}" th:object="${student}" method="post">

<**table** border="1">

<**tr**>

<**td**><**label** th:text="#{msg.id}" /></**td**>

<**td**><**input** type="number" th:field="\*{id}" /></**td**>

</**tr**>

<**tr**>

<**td**><**label** th:text="#{msg.name}" /></**td**>

<**td**><**input** type="text" th:field="\*{name}" /></**td**>

</**tr**>

<**tr**>

<**td**><**input** type="submit" value="Submit" /></**td**>

</**tr**>

</**table**>

</**form**>

In the above code, /saveStudent is the form action URL and a student is the object that holds the form data submitted.

The StudentController class handles the form submission:

@Controller

**public** **class** **StudentController** {

@RequestMapping(value = "/saveStudent", method = RequestMethod.POST)

**public** String **saveStudent**(@ModelAttribute Student student, BindingResult errors, Model model) {

// logic to process input data

}

}

The @RequestMapping annotation maps the controller method with the URL provided in the form. The annotated method saveStudent() performs the required processing for the submitted form. Finally, the @ModelAttribute annotation binds the form fields to the student object.

## ****8. Displaying Validation Errors****

We can use the #fields.hasErrors() function to check if a field has any validation errors. And we use the #fields.errors() function to display errors for a particular field. The field name is the input parameter for both these functions.

Let's take a look at the HTML code to iterate and display the errors for each of the fields in the form:

<**ul**>

<**li** th:each="err : ${#fields.errors('id')}" th:text="${err}" />

<**li** th:each="err : ${#fields.errors('name')}" th:text="${err}" />

</**ul**>

Instead of field name, the above functions accept the wild card character \* or the constant all to indicate all fields. We used the th:each attribute to iterate the multiple errors that may be present for each of the fields.

Here's the previous HTML code rewritten using the wildcard \*:

<**ul**>

<**li** th:each="err : ${#fields.errors('\*')}" th:text="${err}" />

</**ul**>

And here we're using the constant all:

<**ul**>

<**li** th:each="err : ${#fields.errors('all')}" th:text="${err}" />

</**ul**>

Similarly, we can display global errors in Spring using the global constant.

Here's the HTML code to display global errors:

<**ul**>

<**li** th:each="err : ${#fields.errors('global')}" th:text="${err}" />

</**ul**>

Also, we can use the th:errors attribute to display error messages.

The previous code to display errors in the form can be rewritten using th:errors attribute:

<**ul**>

<**li** th:errors="\*{id}" />

<**li** th:errors="\*{name}" />

</**ul**>

## ****9. Using Conversions****

We use the double bracket syntax {{}} to format data for display. This makes use of the formatters configured for that type of field in the conversionService bean of the context file.

Let's see how to format the name field in the Student class:

<**tr** th:each="student: ${students}">

<**td** th:text="${{student.name}}" />

</**tr**>

The above code uses the NameFormatter class, configured by overriding the addFormatters() method from the WebMvcConfigurer interface.

For this purpose, our @Configuration class overrides the WebMvcConfigurerAdapter class:

@Configuration

**public** **class** **WebMVCConfig** **extends** **WebMvcConfigurerAdapter** {

// ...

@Override

@Description("Custom Conversion Service")

**public** **void** **addFormatters**(FormatterRegistry registry) {

registry.addFormatter(**new** **NameFormatter**());

}

}

The NameFormatter class implements the Spring Formatter interface.

We can also use the #conversions utility to convert objects for display. The syntax for the utility function is #conversions.convert(Object, Class) where Object is converted to Class type.

Here's how to display student object percentage field with the fractional part removed:

<**tr** th:each="student: ${students}">

<**td** th:text="${#conversions.convert(student.percentage, 'Integer')}" />

</**tr**>

## ****10. Conclusion****

In this article, we've seen how to integrate and use Thymeleaf in a Spring MVC application.

We have also seen examples of how to display fields, accept input, display validation errors, and convert data for display.

Logging in Java

Java provides the ability to capture the log files.

**The need for Log capture**

There are multiple reasons why we may need to capture the application activity.

* Recording unusual circumstances or errors that may be happening in the program
* Getting the info about whats going in the application

The details which can be obtained from the logs can vary. Sometimes, we may want a lot of details regarding the issue, or sometimes some light information only.

Like when the application is under development and is undergoing testing phase, we may need to capture a lot of details.

**Log Levels**

The log levels control the logging details. They determine the extent to which depth the log files are generated. Each level is associated with a numeric value and there are 7 basic log levels and 2 special ones.

We need to specify the desired level of logging every time, we seek to interact with the log system. The basic logging levels are:

* 2016-06-16 17:02:13 TRACE Trace log message
* 2016-06-16 17:02:13 DEBUG Debug log message
* 2016-06-16 17:02:13 INFO Info log message
* 2016-06-16 17:02:13 ERROR Error log message

# **Java 8 - Optional Class**

Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.

## Class Declaration

Following is the declaration for **java.util.Optional<T>** class −

public final class Optional<T> extends Object

## Class Method

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **static <T> Optional<T> empty()**  Returns an empty Optional instance. |
| 2 | **boolean equals(Object obj)**  Indicates whether some other object is "equal to" this Optional. |
| 3 | **Optional<T> filter(Predicate<? super <T> predicate)**  If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional. |
| 4 | **<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**  If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional. |
| 5 | **T get()**  If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| 6 | **int hashCode()**  Returns the hash code value of the present value, if any, or 0 (zero) if no value is present. |
| 7 | **void ifPresent(Consumer<? super T> consumer)**  If a value is present, it invokes the specified consumer with the value, otherwise does nothing. |
| 8 | **boolean isPresent()**  Returns true if there is a value present, otherwise false. |
| 9 | **<U>Optional<U> map(Function<? super T,? extends U> mapper)**  If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result. |
| 10 | **static <T> Optional<T> of(T value)**  Returns an Optional with the specified present non-null value. |
| 11 | **static <T> Optional<T> ofNullable(T value)**  Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| 12 | **T orElse(T other)**  Returns the value if present, otherwise returns other. |
| 13 | **T orElseGet(Supplier<? extends T> other)**  Returns the value if present, otherwise invokes other and returns the result of that invocation. |
| 14 | **<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)**  Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier. |
| 15 | **String toString()**  Returns a non-empty string representation of this Optional suitable for debugging. |

This class inherits methods from the following class −

* java.lang.Object

# Spring Boot Upload/Download File to/from Database example

In this tutorial, I will show you how to upload and download files to/from database with a Spring Boot Rest APIs. We also use Spring Web MultipartFile interface to handle HTTP [multi-part](https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html) requests.

## Spring Boot Rest APIs for uploading Files to Database

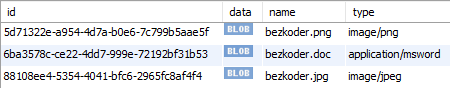
Our Spring Boot Application will provide APIs for:

* uploading File to PostgreSQL/MySQL database
* downloading File database with the link
* getting list of Files’ information (file name, url, type, size)

These are APIs to be exported:

| **Methods** | **Urls** | **Actions** |
| --- | --- | --- |
| POST | /upload | upload a File |
| GET | /files | get List of Files (name, url, type, size) |
| GET | /files/[fileId] | download a File |

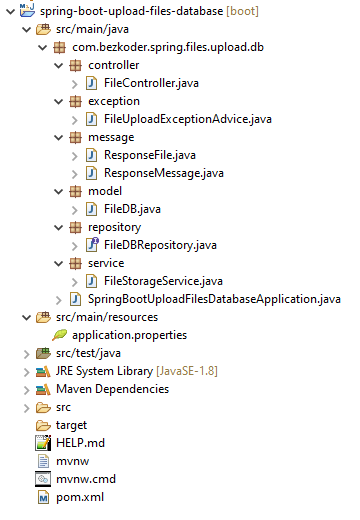
The uploaded files will be stored in PostgreSQL/MySQL Database **files** table with these fields: id, name, type and data as **BLOB** type (Binary Large Object is for storing binary data like file, image, audio, or video).



## Technology

* Java 8
* Spring Boot 2 (with Spring Web MVC)
* PostgreSQL/MySQL Database
* Maven 3.6.1

## Project Structure



– FileDB is the data model corresponding to **files** table in database.  
– FileDBRepository extends Spring Data JpaRepository which has methods to store and retrieve files.  
– FilesStorageService uses FileDBRepository to provide methods for saving new file, get file by id, get list of Files.  
– FilesController uses FilesStorageService to export Rest APIs: POST a file, GET all files’ information, download a File.  
– FileUploadExceptionAdvice handles exception when the controller processes file upload.  
– ResponseFile contains information of the file (name, url, type, size) for HTTP response payload.  
– application.properties contains configuration for Servlet Multipart and PostgreSQL/MySQL database connection.  
– pom.xml for Spring Boot, Spring Data JPA and PostgreSQL/MySQL connector dependency.

# **Using Spring ResponseEntity to Manipulate the HTTP Response**

## 1. Introduction

Using Spring, we usually have many ways to achieve the same goal, including fine-tuning HTTP responses.

In this short tutorial, we'll see how to set the body, status, and headers of an HTTP response using ResponseEntity.

## 2. ResponseEntity

ResponseEntity **represents the whole HTTP response: status code, headers, and body**. As a result, we can use it to fully configure the HTTP response.

If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

ResponseEntity is a generic type. Consequently, we can use any type as the response body:

@GetMapping("/hello")

ResponseEntity<String> **hello**() {

**return** **new** **ResponseEntity**<>("Hello World!", HttpStatus.OK);

}

Since we specify the response status programmatically, we can return with different status codes for different scenarios:

@GetMapping("/age")

ResponseEntity<String> **age**(

@RequestParam("yearOfBirth") **int** yearOfBirth) {

**if** (isInFuture(yearOfBirth)) {

**return** **new** **ResponseEntity**<>(

"Year of birth cannot be in the future",

HttpStatus.BAD\_REQUEST);

}

**return** **new** **ResponseEntity**<>(

"Your age is " + calculateAge(yearOfBirth),

HttpStatus.OK);

}

Additionally, we can set HTTP headers:

@GetMapping("/customHeader")

ResponseEntity<String> **customHeader**() {

**HttpHeaders** headers = **new** **HttpHeaders**();

headers.add("Custom-Header", "foo");

**return** **new** **ResponseEntity**<>(

"Custom header set", headers, HttpStatus.OK);

}

Furthermore, ResponseEntity **provides two nested builder interfaces**: HeadersBuilder and its subinterface, BodyBuilder. Therefore, we can access their capabilities through the static methods of ResponseEntity.

The simplest case is a response with a body and HTTP 200 response code:

@GetMapping("/hello")

ResponseEntity<String> **hello**() {

**return** ResponseEntity.ok("Hello World!");

}

For the most popular [HTTP status codes](https://www.baeldung.com/cs/http-status-codes) we get static methods:

BodyBuilder **accepted**();

BodyBuilder **badRequest**();

BodyBuilder **created**(java.net.URI location);

HeadersBuilder<?> noContent();

HeadersBuilder<?> notFound();

BodyBuilder **ok**();

In addition, we can use the BodyBuilder status(HttpStatus status) and the BodyBuilder status(int status) methods to set any HTTP status.

Finally, with ResponseEntity<T> BodyBuilder.body(T body) we can set the HTTP response body:

@GetMapping("/age")

ResponseEntity<String> **age**(@RequestParam("yearOfBirth") **int** yearOfBirth) {

**if** (isInFuture(yearOfBirth)) {

**return** ResponseEntity.badRequest()

.body("Year of birth cannot be in the future");

}

**return** ResponseEntity.status(HttpStatus.OK)

.body("Your age is " + calculateAge(yearOfBirth));

}

We can also set custom headers:

@GetMapping("/customHeader")

ResponseEntity<String> **customHeader**() {

**return** ResponseEntity.ok()

.header("Custom-Header", "foo")

.body("Custom header set");

}

Since BodyBuilder.body() returns a ResponseEntity instead of BodyBuilder, it should be the last call.

Note that with HeaderBuilder we can't set any properties of the response body.

While returning ResponseEntity<T> object from the controller, we might get an exception or error while processing the request and would like to **return error-related information to the user represented as some other type, let's say E**.

Spring 3.2 brings support for a global **@ExceptionHandler with the new @ControllerAdvice annotation,**which handles these kinds of scenarios. For in-depth details, refer to our existing article [here](https://www.baeldung.com/exception-handling-for-rest-with-spring).

**While ResponseEntity is very powerful, we shouldn't overuse it.** In simple cases, there are other options that satisfy our needs and they result in much cleaner code.

## 3. Alternatives

### 3.1. @ResponseBody

In classic Spring MVC applications, endpoints usually return rendered HTML pages. Sometimes we only need to return the actual data; for example, when we use the endpoint with AJAX.

In such cases, we can mark the request handler method with @ResponseBody, and **Spring treats the result value of the method as the HTTP response body** itself.

### 3.2. @ResponseStatus

When an endpoint returns successfully, Spring provides an HTTP 200 (OK) response. If the endpoint throws an exception, Spring looks for an exception handler that tells which HTTP status to use.

We can mark these methods with @ResponseStatus, and therefore, Spring **returns with a custom HTTP status**.

### 3.3. Manipulate the Response Directly

Spring also lets us access the javax.servlet.http.HttpServletResponse object directly; we only have to declare it as a method argument:

@GetMapping("/manual")

**void** **manual**(HttpServletResponse response) **throws** IOException {

response.setHeader("Custom-Header", "foo");

response.setStatus(200);

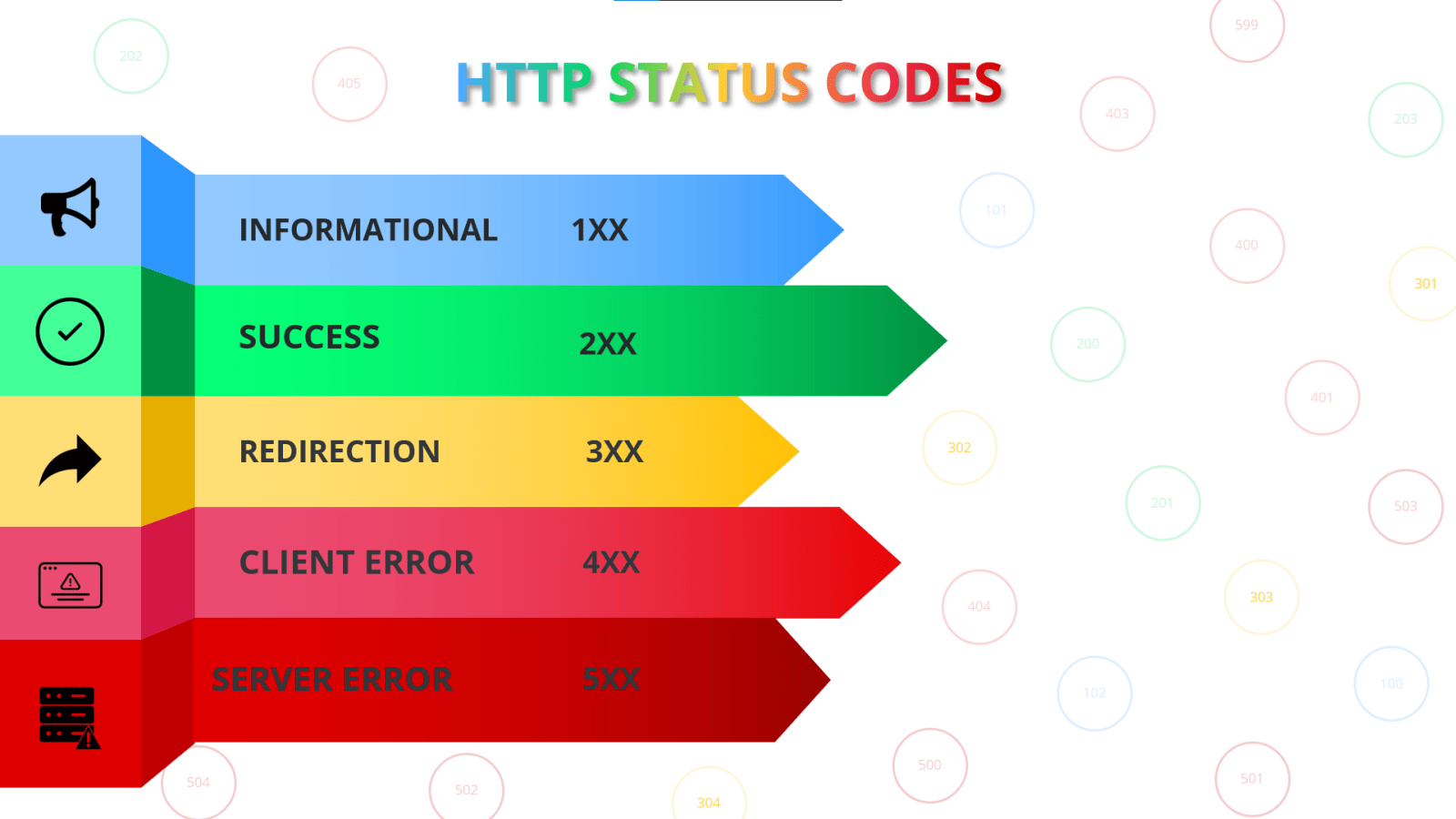
response.getWriter().println("Hello World!");

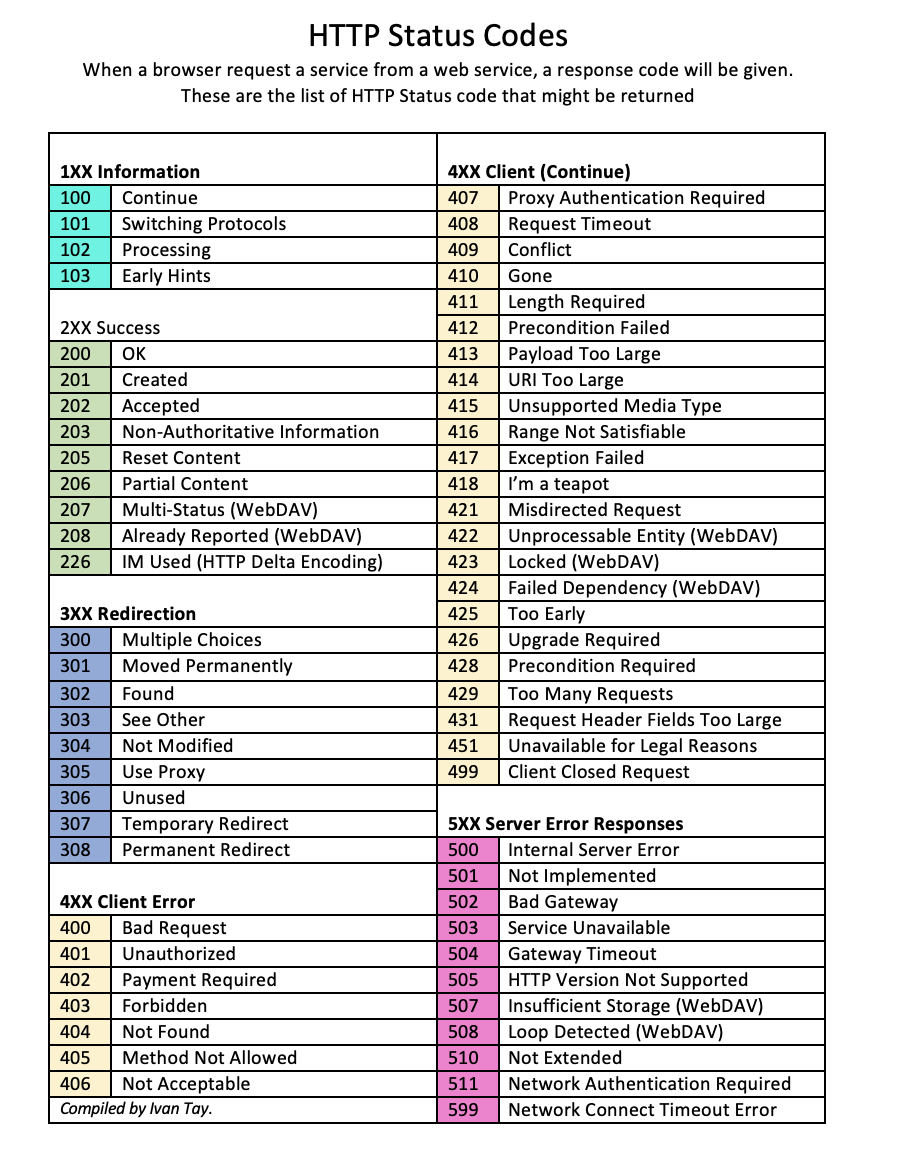
}

Since Spring provides abstractions and additional capabilities above the underlying implementation, **we shouldn't manipulate the response this way**.

## 4. Conclusion

In this article, we discussed multiple ways to manipulate the HTTP response in Spring, and examined their benefits and drawbacks.





# **A Guide To Spring Redirects**

## ****1. Overview****

This tutorial will focus on **implementing a Redirect in Spring** and will discuss the reasoning behind each strategy.

## ****2. Why Do a Redirect?****

Let’s first consider **the reasons why we may need to do a redirect** in a Spring application.

There are many possible examples and reasons of course. For example, we might need to POST form data, work around the double submission problem or just delegate the execution flow to another controller method.

A quick side note here: The typical Post/Redirect/Get pattern doesn't adequately address double submission issues, and problems such as refreshing the page before the initial submission has completed may still result in a double submission.

## ****3. Redirect With the****RedirectView

Let's start with this simple approach and **go straight to an example**:

@Controller

@RequestMapping("/")

**public** **class** **RedirectController** {

@GetMapping("/redirectWithRedirectView")

**public** RedirectView **redirectWithUsingRedirectView**(

RedirectAttributes attributes) {

attributes.addFlashAttribute("flashAttribute", "redirectWithRedirectView");

attributes.addAttribute("attribute", "redirectWithRedirectView");

**return** **new** **RedirectView**("redirectedUrl");

}

}

Behind the scenes, RedirectView will trigger an HttpServletResponse.sendRedirect(), which will perform the actual redirect.

Notice here how **we're injecting the redirect attributes into the method.** The framework will do the heavy lifting and allow us to interact with these attributes.

We're adding the model attribute attribute, which will be exposed as HTTP query parameter. The model must contain only objects — generally Strings or objects that can be converted to Strings.

## ****4. Redirect With the Prefix****redirect:

The previous approach — using RedirectView — is suboptimal for a few reasons.

First, we're now coupled to the Spring API because we're using the RedirectView directly in our code.

Second, we now need to know from the start, when implementing that controller operation, that the result will always be a redirect, which may not always be the case.

A better option is using the prefix redirect:. The redirect view name is injected into the controller like any other logical view name. **The controller is not even aware that redirection is happening.**

Here's what that looks like:

@Controller

@RequestMapping("/")

**public** **class** **RedirectController** {

@GetMapping("/redirectWithRedirectPrefix")

**public** ModelAndView **redirectWithUsingRedirectPrefix**(ModelMap model) {

model.addAttribute("attribute", "redirectWithRedirectPrefix");

**return** **new** **ModelAndView**("redirect:/redirectedUrl", model);

}

}

When a view name is returned with the prefix redirect:, the UrlBasedViewResolver (and all its subclasses) will recognize this as a special indication that a redirect needs to happen. The rest of the view name will be used as the redirect URL.

A quick but important note is that when we use this logical view name here — redirect:/redirectedUrl — we're doing a redirect **relative to the current Servlet context.**

We can use a name such as a redirect: http://localhost:8080/spring-redirect-and-forward/redirectedUrl if we need to redirect to an absolute URL.

## ****5. Forward With the Prefix**** forward:

Let’s now see how to do something slightly different: a forward.

Before the code, let's go over **a quick, high-level overview of the semantics of forward vs redirect**:

* redirect will respond with a 302 and the new URL in the Location header; the browser/client will then make another request to the new URL.
* forward happens entirely on a server side. The Servlet container forwards the same request to the target URL; the URL won't change in the browser.

Now let's look at the code:

@Controller

@RequestMapping("/")

**public** **class** **RedirectController** {

@GetMapping("/forwardWithForwardPrefix")

**public** ModelAndView **redirectWithUsingForwardPrefix**(ModelMap model) {

model.addAttribute("attribute", "forwardWithForwardPrefix");

**return** **new** **ModelAndView**("forward:/redirectedUrl", model);

}

}

Same as redirect:, the forward: prefix will be resolved by UrlBasedViewResolver and its subclasses. Internally, this will create an InternalResourceView, which does a RequestDispatcher.forward() to the new view.

To wrap up, compared to the two requests that we had in the case of the redirect solution, in this case, we only have a single request going out from the browser/client to the server side. The attribute that was previously added by the redirect is, of course, missing as well.

## ****6. Attributes With the****RedirectAttributes

Next, let's look closer at **passing attributes in a redirect**, making full use of the framework with RedirectAttributes:

@GetMapping("/redirectWithRedirectAttributes")

**public** RedirectView **redirectWithRedirectAttributes**(RedirectAttributes attributes) {

attributes.addFlashAttribute("flashAttribute", "redirectWithRedirectAttributes");

attributes.addAttribute("attribute", "redirectWithRedirectAttributes");

**return** **new** **RedirectView**("redirectedUrl");

}

As we saw before, we can inject the attributes object in the method directly, which makes this mechanism very easy to use.

Notice also that **we are adding a flash attribute as well.** This is an attribute that won't make it into the URL.

With this kind of attribute, we can later access the flash attribute using @ModelAttribute(“flashAttribute”) **only in the method that is the final target of the redirect**:

@GetMapping("/redirectedUrl")

**public** ModelAndView **redirection**(

ModelMap model,

@ModelAttribute("flashAttribute") Object flashAttribute) {

model.addAttribute("redirectionAttribute", flashAttribute);

**return** **new** **ModelAndView**("redirection", model);

}

That way, using RedirectAttributes instead of a ModelMap gives us the ability only to share**some attributes between the two methods** that are involved in the redirect operation.

## ****7. An Alternative Configuration Without the Prefix****

Let's now explore an alternative configuration: a redirect without using the prefix.

To achieve this, we need to use an org.springframework.web.servlet.view.XmlViewResolver:

<**bean** class="org.springframework.web.servlet.view.XmlViewResolver">

<**property** name="location">

<**value**>/WEB-INF/spring-views.xml</**value**>

</**property**>

<**property** name="order" value="0" />

</**bean**>

This is instead of org.springframework.web.servlet.view.InternalResourceViewResolver we used in the previous configuration:

<**bean**

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

</**bean**>

We also need to define a RedirectView bean in the configuration:

<**bean** id="RedirectedUrl" class="org.springframework.web.servlet.view.RedirectView">

<**property** name="url" value="redirectedUrl" />

</**bean**>

Now we can **trigger the redirect by referencing this new bean by id**:

@Controller

@RequestMapping("/")

**public** **class** **RedirectController** {

@GetMapping("/redirectWithXMLConfig")

**public** ModelAndView **redirectWithUsingXMLConfig**(ModelMap model) {

model.addAttribute("attribute", "redirectWithXMLConfig");

**return** **new** **ModelAndView**("RedirectedUrl", model);

}

}

## ****8. Redirecting an HTTP POST Request****

For use cases like bank payments, we might need to redirect an HTTP POST request. Depending on the HTTP status code returned, POST request can be redirected to an HTTP GET or POST.

As per HTTP 1.1 protocol [reference](https://tools.ietf.org/html/rfc7238), status codes 301 (Moved Permanently) and 302 (Found) allow the request method to be changed from POST to GET. The specification also defines the corresponding 307 (Temporary Redirect) and 308 (Permanent Redirect) status codes that don't allow the request method to be changed from POST to GET.

Let's look at the code for redirecting a post request to another post request:

@PostMapping("/redirectPostToPost")

**public** ModelAndView **redirectPostToPost**(HttpServletRequest request) {

request.setAttribute(

View.RESPONSE\_STATUS\_ATTRIBUTE, HttpStatus.TEMPORARY\_REDIRECT);

**return** **new** **ModelAndView**("redirect:/redirectedPostToPost");

}Copy

@PostMapping("/redirectedPostToPost")

**public** ModelAndView **redirectedPostToPost**() {

**return** **new** **ModelAndView**("redirection");

}

## 9. Forward With Parameters

Now let's consider a scenario where we would want to send some parameters across to another RequestMapping with a forward prefix.

In that case, **we can use an HttpServletRequest to pass in parameters between calls.**

Here's a method forwardWithParams that needs to send param1 and param2 to another mapping forwardedWithParams:

@RequestMapping(value="/forwardWithParams", method = RequestMethod.GET)

**public** ModelAndView **forwardWithParams**(HttpServletRequest request) {

request.setAttribute("param1", "one");

request.setAttribute("param2", "two");

**return** **new** **ModelAndView**("forward:/forwardedWithParams");

}

In fact, the mapping forwardedWithParams can exist in an entirely new controller and need not be in the same one:

@RequestMapping(value="/forwardWithParams", method = RequestMethod.GET)

@Controller

@RequestMapping("/")

**public** **class** **RedirectParamController** {

@RequestMapping(value = "/forwardedWithParams", method = RequestMethod.GET)

**public** RedirectView **forwardedWithParams**(

**final** RedirectAttributes redirectAttributes, HttpServletRequest request) {

redirectAttributes.addAttribute("param1", request.getAttribute("param1"));

redirectAttributes.addAttribute("param2", request.getAttribute("param2"));

redirectAttributes.addAttribute("attribute", "forwardedWithParams");

**return** **new** **RedirectView**("redirectedUrl");

}

}

As we can see, param1 and param2 traveled from the first controller to the second one. Finally, they showed up in the redirect named redirectedUrl that forwardedWithParams points to.

## ****10. Conclusion****

This article illustrated **three different approaches to implementing a redirect in Spring**, how to handle/pass attributes when doing these redirects and how to handle redirects of HTTP POST requests.

# ModelAndView

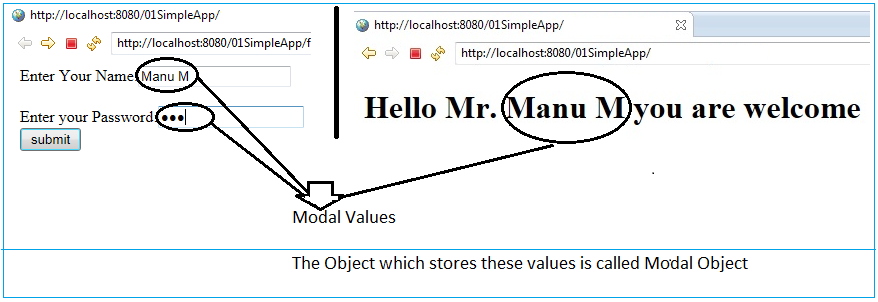
ModelAndView is a class present in org.springframework.web.servlet package. ModelAndView is a value object designed to hold model and view making it possible for a handler to return both model and view in a single return value. ModelAndView is resolved by DispatcherServlet using special framework objects ViewResolver and View.

## View

It is a String which is the name of a JSP file or which is the value given for the name attribute of a <bean> tag where this <bean> tag corresponds to the class which implements View interface.

## Model

It is an object which stores the values that need to be sent to views/JSP. It can be a String object which has stored a string value. When we have to send many model values, we can store them in Map and object of Map can be used.



## Constructors of ModelAndView class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ModelAndView() | Default constructor for bean-style usage: populating bean properties instead of passing in constructor arguments. |
| ModelAndView(String viewName) | Convenient constructor when there is no model data to expose. |
| ModelAndView(String viewName, HttpStatus status) | Create a new ModelAndView given a view name and HTTP status. |
| ModelAndView(String viewName, Map< String,?> model) | Create a new ModelAndView given a view name and a model. |
| ModelAndView(String viewName, Map<String,?> model, HttpStatus status) | Create a new ModelAndView given a view name, model, and HTTP status. |
| [ModelAndView](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html#ModelAndView-java.lang.String-java.lang.String-java.lang.Object-)(String viewName, String modelName, Object modelObject) | Convenient constructor to take a single model object. |
| ModelAndView(View view, String modelName, Object modelObject) | Convenient constructor to take a single model object. |
| [ModelAndView](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html#ModelAndView-org.springframework.web.servlet.View-)([View](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/View.html) view) | Convenient constructor when there is no model data to expose. |
| [ModelAndView](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html#ModelAndView-org.springframework.web.servlet.View-java.util.Map-)([View](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/View.html) view, Map< String,?> model) | Create a new ModelAndView given a View object and a model. |

## Important methods of ModelAndView class

|  |  |
| --- | --- |
| **Method** | **Description** |
| **public** **boolean** hasView() | Indicate whether or not this ModelAndView has a view, either as a view name or as a direct View instance. |
| **public** **boolean** isEmpty() | Return whether this ModelAndView object is empty, i.e. whether it does not hold any view and does not contain a model. |
| **public** **boolean** isReference() | Return whether we use a view reference, i.e. true if the view has been specified via a name to be resolved by the DispatcherServlet via a ViewResolver. |
| **public** Map getModel() | Return the model map. Never returns null. To be called by application code for modifying the model. |
| **public** ModelAndView addAllObjects( Map< String,?> modelMap) | Add all attributes contained in the provided Map to the model. |
| **public** ModelAndView addObject(String attributeName, Object attributeValue) | Add an attribute to the model. |
| **public** ModelMap getModelMap() | Return the underlying ModelMap instance (never null). |
| **public** String getViewName() | Return the view name to be resolved by the DispatcherServlet via a ViewResolver, or null if we are using a View object. |
| **public** View getView() | Return the View object, or null if we are using a view name to be resolved by the DispatcherServlet via a ViewResolver. |
| **public** **void** setView(View view) | Set a View object for this ModelAndView. Will override any pre-existing view name or View. |
| **public** **void** setViewName String viewName) | Set a view name for this ModelAndView, to be resolved by the DispatcherServlet via a ViewResolver. Will override any pre-existing view name or View. |

# **Using Spring @ResponseStatus to Set HTTP Status Code**

## 1. Introduction

In Spring MVC, we have many ways to **set the status code of an HTTP response**.

In this short tutorial, we will see the most straightforward way: using the @ResponseStatus annotation.

## 2. On Controller Methods

When an endpoint returns successfully, Spring provides an HTTP 200 (OK) response.

If we want to specify the **response status of a controller method**, we can mark that method with @ResponseStatus. It has two interchangeable arguments for the desired response status: code, and value.

Note, that when we set reason, Spring calls HttpServletResponse.sendError(). Therefore, it will send an **HTML error page to the client, which makes it a bad fit for REST endpoints**.

Also note, that Spring only uses @ResponseStatus, when **the marked method completes successfully** (without throwing an Exception).

## 3. With Error Handlers

We have three ways to use @ResponseStatus to convert an Exception to an HTTP response status:

* using @ExceptionHandler
* using @ControllerAdvice
* marking the Exception class

In order to use the first two solutions, we have to define an error handler method. You can read more about this topic in [this article](https://www.baeldung.com/exception-handling-for-rest-with-spring).

We can use @ResponseStatus with these error handler methods **the same way we did with regular MVC methods** in the previous section.

When Spring catches this Exception, it uses the settings we provided in @ResponseStatus.

Note, that when we mark an Exception class with @ResponseStatus, Spring always calls HttpServletResponse.sendError(), whether we set reason or not.

Also note, that Spring uses the same configuration for subclasses, unless we mark them with @ResponseStatus, too.

## 4. Conclusion

In this article, we saw how we can use @ResponseStatus to set HTTP response code in different scenarios, including error handling.