

## Concept and History of Java

The word "Java" may be used to mean the programming language, but also the platform or technology. What are the specifics and differences between these concepts? It is a good idea to start with the definitions offered by Oracle.

The **Java technology** is a combination of a programming language, a platform, and various frameworks (special-purpose software environment).

The **Java programming language** is a strongly typed object-oriented programming language developed by Sun Microsystems Company.

The **Java platform** is a complex of hardware and software aimed to develop and run programs written in the Java language.

Today Java is one of the most popular programming language around the world although it celebrated its 25<sup>th</sup> birthday in 2021. According to the [TIOBE Index](#) and the [The PopularityY of Programming Language Index \(PYPL Index\)](#), Java has held a leading position among all programming languages for years. Java is used to write mobile applications, to work with Big Data, cloud applications, machine learning, navigation systems, and many other things.

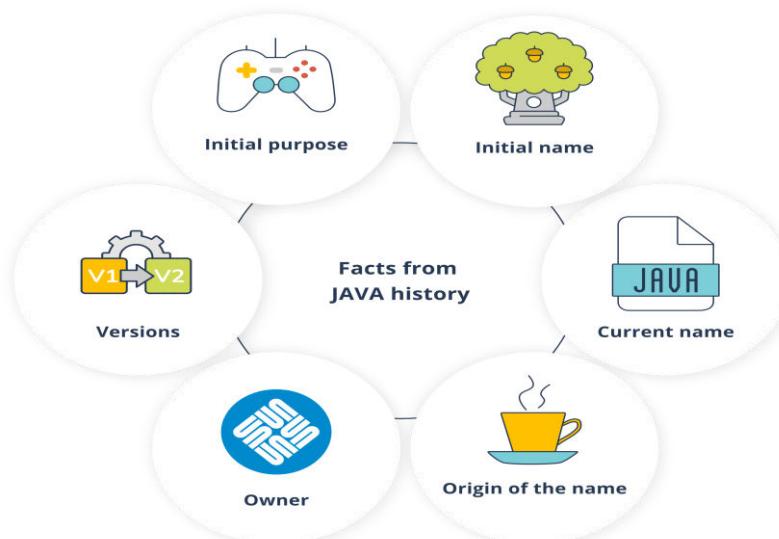
The history of Java started in the early 1990s, when the use of network computing capabilities in everyday life was an innovation. In 1991, a small group of engineers from **Sun Microsystems** under the name of **Green Team (in the photo)** believed that the next stage in the development of computing technologies would be a consolidation of digital consumer devices and computers.



This team, managed by [James Gosling](#), worked almost around the clock. And in the end they created a programming language that caused a revolution in our world. The Green Team demonstrated their new language using the interactive portable remote control for home entertainment Star7 that was initially created for the digital cable TV industry. You can watch the original [video](#) from James Gosling about this device.

<https://www.youtube.com/watch?v=1CsTH9S79qI>

The history of Java features many interesting facts. For example, where does the name come from? What do an oak and coffee have to do with it? Who owns Java? Look at these and some other facts in more detail.



The Java language was developed by the Green Team consisting of James Gosling, Mike Sheridan, and Patrick Norton. The language was initially meant for small embedded systems in electronic devices, such as set-top boxes.

### **Initial name**

James Gosling called the programming language, which had a \*.gt files extension, Greentalk. Soon after the language was called Oak because oaks are a symbol of power and reliability. What's interesting is that the oak is the national tree of many countries.

### **Current name**

In 1995, the name Oak was changed to Java. The reason was that Oak Technologies had already been registered as a trademark. The name Java was one of the best options since it reflected the essence of the new technology: revolutionary, dynamic, living, cool, unique, easily written, and sounding funny.

### **Origin of the name**

Java is an island in Indonesia where the first coffee was produced (the so-called java-coffee, an espresso). James Gosling chose this name because he drank this coffee not far from his office. By the way, Java is simply a name, not an acronym.

### **Owner**

Initially, Java belonged to Sun Microsystems, which is now a subsidiary company of Oracle Corporation.

### **Versions**

The first version of Java was released on January 23, 1996. Since then it has undergone changes that take into consideration modern needs. As of today, the 17th version of Java has been released.

Thus, despite its long history, Java remains one of the leading programming languages.

## **Development and Execution Tools**

Check what is necessary to start working in Java. First, you will need some tools without which it is simply impossible to write, compile, and run programs:

- **JDK** — the Java Development Kit, a set of developer's tools.
- **JRE** — the Java Runtime Environment, the execution system of Java. It is used to organize the processes for running programs.
- **JVM** — the Java Virtual Machine. This is a virtual computing machine that has a set of commands and a memory controller.

The process of using the Java platform tools is shown in the image below. According to it, the original Java program should be first transformed by the compiler into an executable program in the form of bytecode. After that, the bytecode of the Java program can be run in the JVM, i.e., the bytecode is interpreted/transformed into the machine code of the platform where the JVM is used. To write programs in Java, you do not need to know the specifics of bytecode. But understanding bytecode as well as the processes of its generation helps Java programmers in their work.

### **JVM**

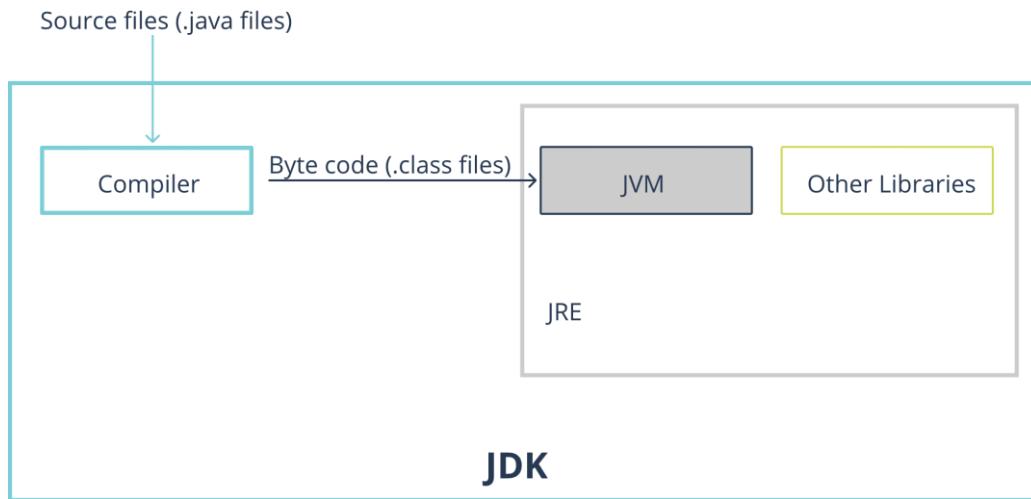
The JVM is the main part of the Java runtime system. The JVM provides a platform-independent way to run code. Programmers may write code without thinking about where it will be run. The JVM performs the following tasks: loads, verifies, and runs code.

### **JRE**

The JRE is the minimum implementation of the virtual machine required to run Java programs. The JRE has no compiler or other development tools. It consists of a virtual machine and libraries of the Java classes. The JRE is part of the JDK; thus, to run a Java program, you first need to install the JDK.

### **JDK**

The JDK is a Java language tool kit for programmers. It includes a compiler, standard libraries of the Java classes, examples, documents, various utilities, and the JRE runtime system. It is free and is freely available from Oracle.



In the next lesson, you will have a chance to install Java tools and start writing programs. But first, you need to get acquainted with the properties of this programming language and its benefits.

## Properties and Benefits of Java

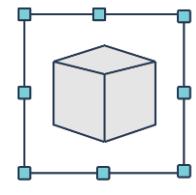
As of today, Java has not lost its relevance due to the following properties.

### Simple



The C-like syntax and object-oriented style are the main factors that make it easy to switch to the Java and quickly develop applications.

### Object-oriented



Java was initially designed as an object-oriented programming language. And this paid off: the object paradigm has become the main trend in the world of software development.

### Safe



*"Java is C++ without the guns, clubs and knives"* — James Gosling.

Java assures an extended check during compilation followed by a second check when running programs. Moreover, the memory control model is extremely simple: there are no indices defined by the programmer and there are no arithmetic indices; Java assembles objects automatically.

### Platform-Independent



The Java compiler generates bytecode that is interpreted as machine code by the platform where the JVM is launched. One and the same Java bytecode will run on any platform!

### Multi-threading



The Java platform supports multithreading on the language level and provides efficient synchronization mechanisms, as well as a conflict-free access for parallel execution threads.

### Dynamic



The Java compiler strictly follows a static check during compilation. The language and runtime system are dynamic. The classes are linked only when needed. New code modules can be linked on demand from various sources, including the network.

### Distributed



The Java programming language was initially aimed to create distributed network applications for various users in various environments, from network devices to the global network and desktops.

Java remains a top language not just because of its properties. It has several benefits as compared to other object-oriented programming languages, which you can read about below.

#### Backward compatibility

#### Code portability

#### Flexible security system

You can run a code written many years ago on new Java versions. This feature is not typical for all programming languages. This is possible due to **backward compatibility** — one of the key Java benefits. For example, JDK 8 can run code compiled in JDK 7 or JDK 6. There is a Compatibility Manual for each Java release that separately lists features not compatible with previous versions. You can study the indicators of backward compatibility of different versions [here](#).

One of the Java benefits is the high indicator of **code portability**. It is achieved on the back of complete independence of the executed Java code on the operating system and hardware. This is what allows Java to run programs on any device for which a JVM exists.

A **flexible security system** is an accomplishment of Java technology. It is achieved through the total control of the virtual machine over the program execution. Any operations exceeding authorizations set for the program (for example, unauthorized access to data or network connection) lead to an interruption of the program.

## Conclusion

In this lesson, you explored the history of Java and reviewed interesting facts about it:

- The Java platform is a set of software products and specifications that together provide a system for application software development.
- Java is used on a variety of devices: from mobile phones and embedded systems to large corporate servers and supercomputers.

- The Java platform is not specialized for just one processor or operating system. The virtual machine and interpreter with a set of libraries are implemented for different platforms. This is done so that Java programs can run everywhere consistently.

## Check Your Knowledge!

1. What category of programming languages can Java be assigned to?

Procedure-oriented

Scripting

**Object-oriented**

Functional

Answer

Correct:

Java was created when object-oriented programming was at the peak of its popularity, and it includes all of its main paradigms.

2. What is the acronym form of the name of a virtual computing machine that has a set of commands and a memory controller?

JDK

**JVM**

JRE

Answer

Correct:

The JVM or Java Virtual Machine is the acronym of the virtual computing machine's name in Java.

3. How are the JVM, JDK, and JRE related?

The JVM allows the development of programs, and the JDK assures their execution in cooperation with JRE.

The JVM uses the JRE to develop programs, and the JDK runs them.

**The JVM runs programs, the JRE provides a runtime environment, and the JDK allows the development of programs.**

Answer

Correct:

The JVM is the main part of the Java runtime system. The JRE is the minimum implementation of the virtual machine required to run programs. While the JDK is the Java language tool kit for programmers.

4. Which benefit of Java allows the JDK, which had a later release date, to run code compiled in an earlier version?

Code portability

Multithreading

Platform independence

**Backward compatibility**

Answer

Correct:

Due to backward compatibility, JDK 8, for example, can run code compiled in JDK 6. Additionally, there is a Compatibility Manual that lists separately those elements not compatible with the previous versions.

# Installation of the JDK

## Introduction

In this lesson, you will get the necessary instructions to install the JDK on your computer.

## Installing the JDK in Windows

The **JDK** (Java Development Kit) is a developer's toolkit for developing applications in Java. It includes:

- A compiler
- Standard libraries of the Java classes
- Examples
- Documents
- Various utilities
- The JRE – Java Runtime Environment

To install the JDK to the Windows operating system, go to [Oracle's official website](#) to download the necessary files.

It is recommended to download and install the latest JDK version.

Below you will find step-by-step instructions for the JDK installation.

*Click on the arrow to see the JDK installation steps for Windows.*

### Step 1



- Choose the appropriate version of the installation file, and download it to your computer. By default, the JDK is installed in the following catalog "**C:\ Program Files \ Java \ jdk-{version number}.{update number}**".
- Accept the offered default values, and follow the instructions on the screen.
- Upon completion of the JDK installation, such tools as the compiler "**javac.exe**" and the runtime environment "**java.exe**" will be located in the sub-catalog "**bin**" of the installed JDK catalog.
- To run the JDK programs, you need to include the "bin" catalog in the PATH environment variable of the Windows operating system. You can find detailed instructions on how to set the environment variable PATH by following this [link](#).

### Step 2

Check the installed JDK:

- Run the "command line" application.
- Enter and execute the "path" command to see the value of the PATH environment variable, and make sure that the "bin" of your JDK has been included in the PATH variable.
- path
- PATH=...;c:\Program Files\Java\jdk-15\bin;...

### Step 3

Check that the chosen versions of the JDK and JRE have been installed correctly. To do this, enter and run the following commands:

```
// Display the JDK version
```

```
javac -version
```

**Output:**

```
javac 15
```

```
// Display the JRE version
```

```
java -version
```

**Output:**

```
java version "15" 2020-009-15
Java(TM) SE Runtime Environment (build 15+36-1562)
Java HotSpot(TM) 64-Bit Server VM (build 15+36-1562, mixed mode, sharing)
```

## Installing the JDK in Ubuntu

To install Java based on the example of the Ubuntu installation package, enter the following command in the console:  
`sudo apt-get install default-jdk`

Note that when installing the JDK using standard **apt-get** methods, there is a chance that you will not install the latest version of Java. You can check the versions of the compiler and runtime environment by entering the following commands in the console:

```
javac -version
java -version
```

## Installing the JDK on Mac

Below you will find the easiest method of the JDK installation for the MacOS. Note the necessary system requirements for installation of Java 7 and later versions:

- Processor Intel version 10.7.3 and higher
- Administrator rights
- 64-bit browser (for example, Safari)

## Conclusion

In this lesson, you have reviewed how to install the JDK in various operating systems: Windows, Ubuntu, MacOS.

## Check Your Knowledge!

1. What is the JDK?

A developer's toolkit for developing applications in Java. correct

A Java runtime environment

A virtual computing machine that has a set of commands and a memory controller

Answer

Correct:

The Java Development Kit is a developer's toolkit for developing applications in Java.

2. Which components are part of the JDK?

A compiler and a library of classes

A compiler, utilities, a runtime system, a library of classes, documents correct

A compiler, a library of classes, a virtual machine

A compiler, a virtual machine, a runtime system, a library of classes

Answer

Correct:

The JDK includes a compiler, standard libraries of the Java classes, examples, documents, various utilities, and the JRE runtime system.

# Keywords

## Introduction

In this lesson, you will study keywords. You will explore groups of keywords and what they are used for.

## Concept of Keywords in Java

In their classic meaning, **keywords** are tokens used by the programming language:

- To denote primary types, algorithmic branching and looping constructs, access modifiers
- To declare, import, create, return, and open methods
- In case of multithreaded programming

Keywords exist in all programming languages and are constantly updated.

The list of all Java keywords is given below.



 Keywords may not be used as identifiers: names of variables, classes, or methods.

 Pay attention to two keywords: **goto** and **const**. These words are reserved but are not used. If you try using them, you will get a compilation error.

Another interesting keyword is **var**; it was added in the Java 10 version. It is used to denote a variable that will receive its data type only after the assignment operator (=) is executed. The variable type depends on the value to the right of the assignment operator.

## Classification

The classification of keywords by the area they are applied to allows the establishment of eight main groups.

*Click on the tabs to learn more about the keywords in Java.*

▼ Group 1 — Primitive data types

▼ Group 2 — Branching and looping

▼ Group 3 — Access modifiers

▼ Group 4 — Declaration and import

▼ Group 5 — Creation, returns, and calls

▼ Group 6 — Handling exceptions

▼ Group 7 — Multithreaded programming

▼ Group 8 — Other

Keywords denoting **primitive data types**:

- **byte**: an integer data type occupying only one byte
- **short**: an integer data type, a short integer
- **int**: an integer data type, a regular integer
- **long**: an integer data type, a long integer
- **char**: character data type
- **float**: a data type for floating-point numbers (with a floating point and single precision)
- **double**: a data type for floating-point numbers (with a floating point and double precision)
- **boolean**: a data type for logical values

Keywords used in such algorithmic constructs as **branching and looping**:

- **if**: checks the condition for executing a block of instructions
- **else**: executes an alternative block of instructions
- **switch**: is used to determine a multiple selection parameter
- **case**: determines the label and the block of instructions that will be executed if the value of the multiple selection parameter matches the label
- **default**: determines the block of instructions that will be executed if the value of the multiple selection parameter does not match any of the specified labels
- **while**: determines the loop that repeats a block of instructions as well as the condition for continuing the loop
- **do**: determines the loop that repeats a block of instructions
- **break**: is used to exit some construct
- **continue**: is used to exit the current looping iteration and enter the next one
- **for**: determines the loop that repeats a block of instructions a certain number of times

Keywords used to specify the **visibility scope (access modifiers)**:

- **private**: indicates that elements of other classes cannot access a method or variable of the given class
- **protected**: indicates that other elements of the given class, its subclasses, or classes of the same package can access a method or variable of that class
- **public**: indicates that other elements of any class can access a method or variable of the given class

## Keywords used for **declaration** as well as for **import**:

- **import**: specifies the classes and packages that will be imported (as an option, imported statically)
- **package**: declares a package
- **class**: declares a class
- **interface**: declares an interface
- **enum**: declares an enumerated type
- **extends**: is used to declare a class and indicates a superclass
- **implements**: is used to declare that a class implements interfaces
- **static**: indicates that attributes/methods of a given class relate to the whole class not just an instance and they may be used without creating an object
- **final**: is used for creating constants (non-changeable data), non-changeable methods, non-heritable classes
- **void**: specifies that a method returns no value
- **abstract**: is used to declare classes and methods; such a class cannot have an instance, and the method does not have a body
- **native**: specifies that a method is implemented in a platform-dependent code, often in the C language; this modifier has, for example, the method *hashcode* in **Object**

## Keywords used for **creation, returns, and calls**:

- **var**: declares a variable
- **new**: creates objects
- **return**: returns a value from a method
- **this**: refers to the current object in a method
- **super**: refers to superclass (parent) objects

## Keywords used to **handle exceptions**:

- **try**: specifies a block of code that controls the occurrence of instruction exceptions
- **catch**: specifies a block of code that is executed if an exception occurs
- **finally**: specifies a block of code that will be executed no matter whether there is an exception or not
- **throw**: is used to generate (throw) an exception
- **throws**: indicates that method could generate (throw) an exceptions

## Keywords used for **multithreaded programming**:

- **synchronized**: specifies that a method or a block of code can only be accessed by one thread at a time
- **volatile**: specifies that a change in a variable by one thread will be accessible by all the other threads

## Keywords that were not included in any of the above groups:

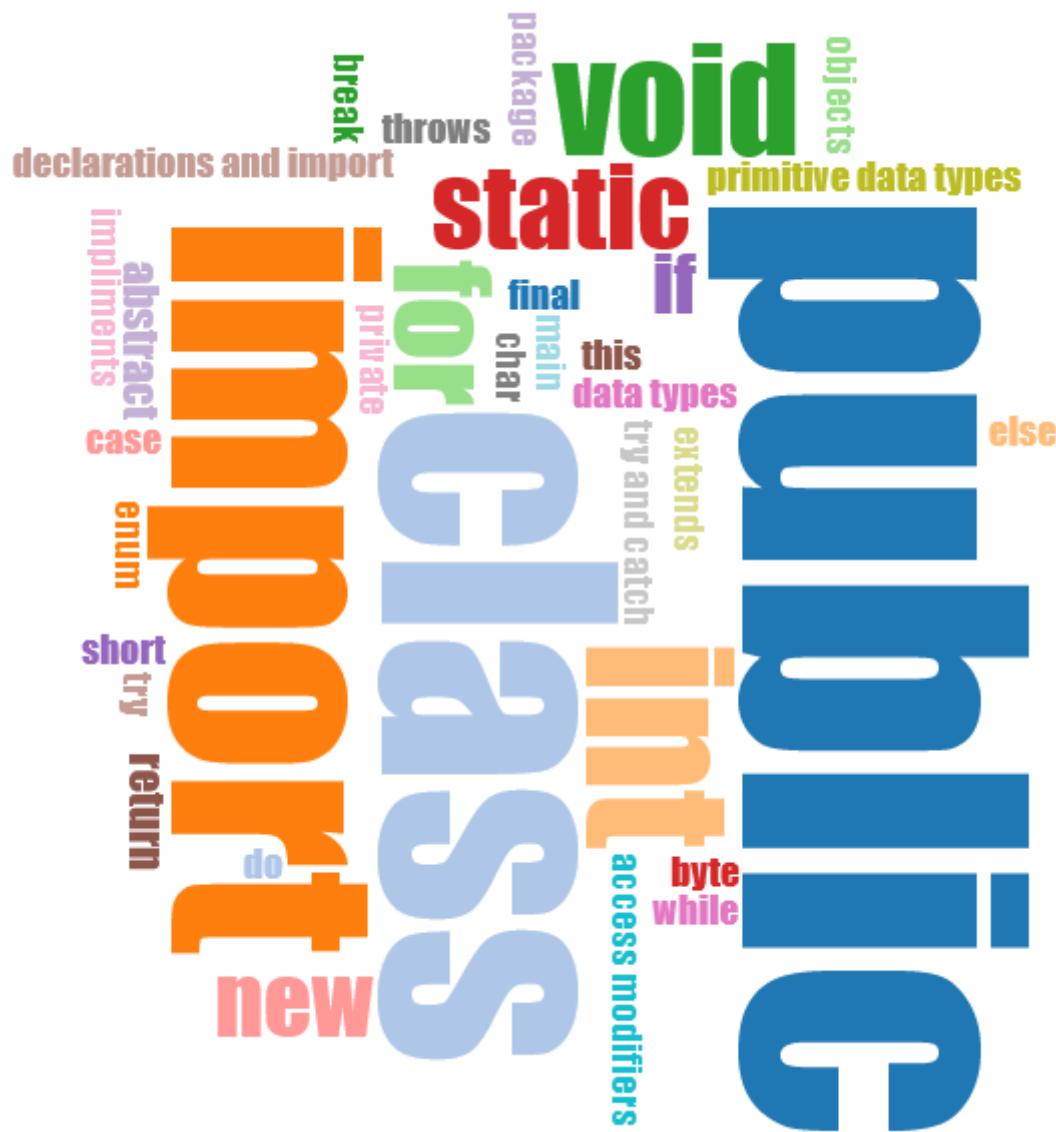
- **instanceof**: an operator used to check whether an object is an instance of a specific class or an interface
- **assert**: allows to verify data and break execution
- **transient**: is applied to class attributes and means that an attribute is not part of an object's persistent state
- **strictfp**: guarantees that calculations with a floating point will be done consistently on all platforms

*Remember that **null**, **true** and **false** are not keywords although in all development environments they are highlighted in the same way as keywords. These are literals.*

## Keyword Cloud

In your opinion, what are the three Java keywords used by programmers most often?

*Enter your answer option in each block and then click Save. View the responses of other program participants. Note that there are no right or wrong answers here.*



**144** words submitted in total.

Your words were:

- **import** 12%
  - **class** 13%
  - **public** 16%

## Check Your Knowledge!

## What are keywords used for?

**To identify primary types, algorithmic branching and looping constructs, access modifiers**

## To declare, import, create, return, and call methods

To indicate the names of variables

### In case of multithreaded programming

## Answer

Correct:

Keywords are used for different purposes, except for indicating the names of variables, since they have a certain meaning for the compiler.

2. Which keywords are reserved but are not used in Java?

var

goto

try

$\alpha_j$   
const

const  
var  
correct

correct

Answer

Correct:

The keywords goto and const are reserved but are not used. If you try using them, you will get a compilation error.

3. Which of the following words is a literal expression and not a keyword?

**null**

var

final

this

Answer

Correct:

null is not a keyword, although in all development environments it is highlighted in the same way as a keyword. This is a literal expression (as are true and false).

4. Which keywords of those listed below are used to describe loops?

switch

**for**

**do**

**while**

Answer

Correct:

The keywords for, do, and while are used to describe loops.

5. Which keywords of those listed below are access modifiers in Java?

**public**

**protected**

default

**private**

correct Answer

Correct:

The keywords public, protected, and private are used to specify the visibility scope (access modifiers).

# Compiling and Running the Application. Console

## Introduction

In the previous lessons, you got familiar with the basics of the Java language. Now it is time to try your hand at writing programs. In this lesson, you will compile and run your first application in Java.

## Creating Your First Application

As your first application, you will need to print a line of text in the standard output stream – the console. You will use the standard text editor "**Notepad**" to write your first program.

Pay attention once again to the public class named First. Let's review it using several rules:

- According to the Java Code Convention, the names of all classes should start with a capital letter.
- According to the TitleCase notation, if the name uses two or more words, then each subsequent word is written without a space and is capitalized.
- One of the Java rules: the class name should match the file name.

The naming rules in Java will be described in more detail in the later lesson.

If you are not very confident with using the command line interface, please make sure that the .java file you created is in the same directory you work with in the command line (In the terminal, this is specified to the left of the > character). If it isn't, you can switch the current working directory in the command line using the "cd" (change directory) command or by using the absolute path to the .java file.

 Each time you change the source code, you need to recompile the application before launching it.

## Conclusion

In this lesson, you have written and compiled your first application using the standard text editor "Notepad".

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

3/3 points

1. Choose the file extension option for a file containing source code.

.java .class .css .cs

Answer

The .java extension is used for files containing source code.

2. Choose the file extension option for a file containing bytecode.

.java

.class

.css

.cs

Answer

Correct:

The .class extension is used for files containing bytecode.

3. How should you name classes correctly?

**The class name should match the file name**

**All class names should be capitalized**

**When using two or more words, each subsequent word is written without a space and is capitalized**

All class names should contain two or more words

Answer Correct:

There are certain rules (Java Code Convention) according to which the names of all classes in Java should start with a capital letter. If the name uses two or more words, then each subsequent word is written without a space and is also capitalized. This is called TitleCase notation. Besides, the class name should match the file name.

# Installing the IDE

## Introduction

In this lesson, you will study what an IDE is and what opportunities it gives you. You will also study the most popular IDEs for Java and will install the chosen version on your computer.

## What Is an IDE?

An **IDE** (Integrated Development Environment) is the environment that provides functions to create applications. On the surface, the integrated development environment looks like a code text editor and assures the following:

- Highlights code fragments
- Highlights errors (contains a base of language syntax)

Any modern IDE contains tools for compilation, execution, assembly automation, code debugging, etc.

Unlike many basic text editors, the IDEs can support full-fledged projects and not just process separate source code files. They can support the creation of a project that includes a group of files, as well as define specific parameters for the project. For instance, such parameters may include:



- Links to the necessary external libraries (third-party software products)
- Editor settings
- Support for versioning
- Debugging settings

The matter of choosing an IDE is very subjective. Each IDE has both a standard set of options and its own specific functionality.

*Click on each item to study the features of the development environment in more detail.*

### IDE Features



### Syntax

Provides support for various programming languages.

### Interpreter

Performs line-by-line analysis, processing, and program execution.

### Code completion

Provides automated code completion.

### Debugger

A tool that allows one to monitor program behavior during its execution and detect problems and errors.

### Compiler

A special program that transforms the source code of a program in a specific language (Java, C#) into a set of machine codes executed directly by the processor.

### Integration

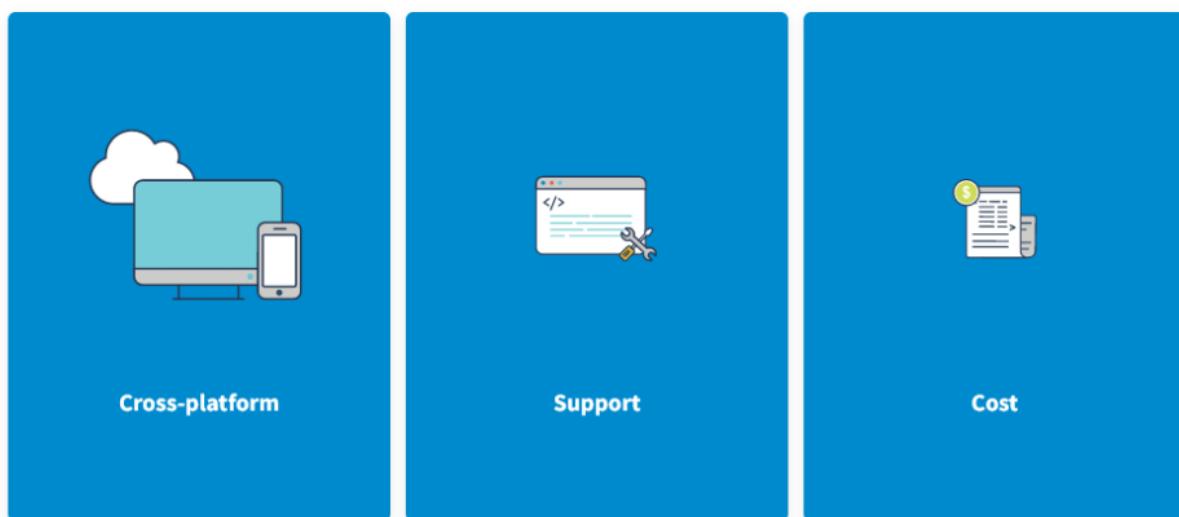
Provides integration with versioning control systems (SVN / Git) – the ability to store multiple versions of the same file and be able to go back to earlier versions, see any changes, and much more.

### Refactoring

Provides a process to improve the quality of code, without writing new functionality.

There are no strict rules for what can be considered as an IDE and what cannot. The more features of those listed above a program has, the closer it is to an IDE. What should you pay attention to when choosing an IDE?

*Click on each card to learn more about the selection criteria for an IDE.*



The integrated development environment should be able to operate on different devices and in different operating systems.

The IDE should be supported/maintained by its developers and regularly updated in line with modern technologies. This selection criterion is one of the most important ones.

There are both commercial and free development environments. It is recommended to first try using free IDE versions.

## IDE for Java

The **Java IDE** is software that contains all the necessary tools, libraries, and other resources for programming in Java. Clearly, the matter of choosing a development environment depends on personal preferences. Today, IntelliJ IDEA and Eclipse, as well as NetBeans, are the most popular options.

*Click on the tabs to learn more about the development environments.*

**IntelliJ IDEA**

**Eclipse IDE**

**NetBeans IDE**

**IntelliJ IDEA** is an integrated software development environment for many programming languages, specifically, for Java, JavaScript, Kotlin, Groovy, and Scala. It was developed by JetBrains Company. We recommend using this IDE as the main tool for developing applications in Java.

**Eclipse IDE** is a well-known integrated development environment from IBM. You can use different languages here (for example, C/C++, JavaScript/TypeScript, PHP, etc.). You can easily integrate support for several languages and other functions in any of the default packages. The Eclipse Marketplace provides almost unlimited customization and expansion of the environment through a variety of plugins and additional modules.

**NetBeans IDE** is a free integrated development environment with an open source code. It allows you to easily develop a wide range of different types of applications (desktop, mobile, web applications) and use HTML, JavaScript, CSS, PHP, and C/C++ technologies.

Note that it is recommended that programmers create a new Java project for each new Java application. The IDEs described above allow programmers to store several programs in a project, which is very convenient for writing first programs. It is quite easy to run the program: open the context menu in the source file and select "Run ...".

There is no need to compile Java source files in the described IDEs. This is because the development environments perform the so-called incremental compilation, that is, Java statements are compiled as you type.

## Conclusion

In this lesson, you explored the IDE for Java and tried installing the chosen IDE on your computer.

## Check Your Knowledge!

1. Choose the most complete description of the integrated development environment.

A tool to create and edit programs

**A system containing tools for development, debugging, assembly, refactoring, as well as a compiler and interpreters**

An environment for developing programs in several programming languages

A tool for editing a program's source code and for its execution

Answer

Correct:

The features of an IDE include code development, debugging, assembly, and refactoring. IDEs also include a compiler, interpreter, and code completion, etc.

2. Which of the IDEs listed below are most popular for development in Java?

**IntelliJ IDEA**

CodeLite

**NetBeans**

Code::Blocks

**Eclipse**

correct

Answer

Correct:

Today, IntelliJ IDEA, Eclipse, and NetBeans are the most popular IDEs for Java.

3. What are the functions of an IDE?

**Source code compilation**

**Code debugging**

Code reflection

**Source code refactoring**

**Code execution**

correct

Answer

Correct:

IDEs have a wide range of features.

# Compiling and Running the Application. IDE

## Introduction

In the previous lesson, you installed the IDE on your computer, but so far, you have not worked in it. In this lesson, you will have such a chance. You will make your first steps toward mastering the IDE, you will:

- Create your first application using the IDE
- Enter data into the application from the console
- Enter a line and numbers into the application

## Compiling and Running the Application. IDE

The following video series will give you a comprehensive overview of how to create the first Java application in the IDE. Do not hesitate to follow the instructions and get your first results.

The table below contains a description of some methods of the Scanner class, which was mentioned in the videos.

You can use them to read user input from the command line.

next()	Reads a "word"—a sequence of characters before a whitespace character
nextLine()	Reads a line—a sequence of characters before a line break (Several words may be read at once.)
nextInt()	Reads an integer
nextDouble()	Reads a number that contains a fraction

## Conclusion

In this lesson, you created your first Java application in the IDE and entered various data into this application.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

3/3 points

1. What is the result of compiling and running the following code?

```
public class Test {
    public static void main(String[] args) {
        println("Hello student!");
    }
}
```

Hello student!

"Hello student!"

Nothing will be printed

### Compilation error

Answer

Correct:

This code example is missing a field of the System.out. class that has to be specified before the method println("Hello student!"). A field of the System.out. class is usually used to return a line.

2. What is the result of compiling and running the following code?

```
public class Test {
    static public void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Hello world!

"Hello world!"

Nothing will be printed

Compilation error

Answer

Correct:

The code was written correctly, thus the output in the console will be: Hello world!

3. What does the icon of a red square in the IDE console mean?

Application error

**Application is working**

Application stopped working

Reboot the system

Answer

Correct:

A red square in the IDE console means that the application is working.

# Introduction

One of the Java features is the strict language typing. There are 8 primitive types in Java that you will study in the "Data types" module.

After studying this module, you will be able to:

- Create variables
- Use numeric and string literals
- Determine the place of data storage depending on their type
- Correctly use different operators
- Describe the process of typecasting
- Name various elements according to the recognized conventions

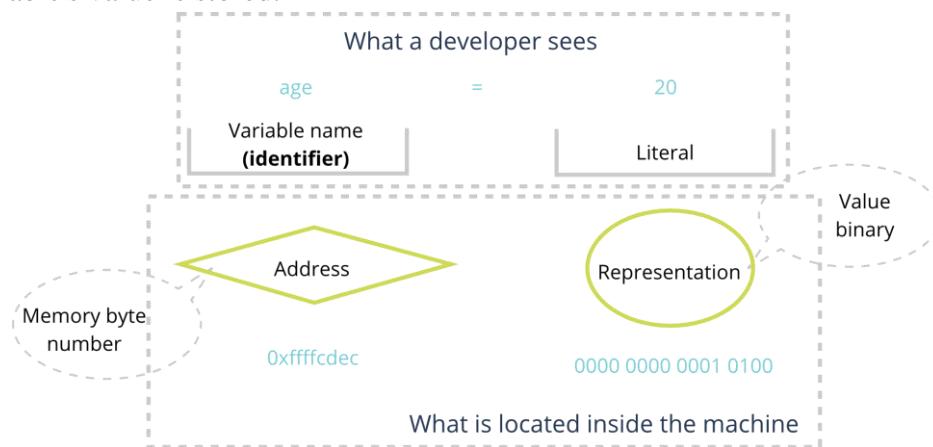
## Variables and Data Types in Java

### Introduction

In this lesson, you will study what a variable is and what syntax it has. You will also explore where data are stored in Java.

### Concept of the Variable

A **variable** is a named program object storing a value of the declared type that can change this value as the program is executed. Each variable has an **identifier** – the variable's name. The identifier is a reference to some memory address where the variable's value is stored.



In Java, variables can be located in the class body (be its element) or in the method body:

- A variable declared in the class body stores a zero value of the relevant type by default.
- A variable declared in the method body must be initialized before it can be used. These variables are local and are not initialized with zero values by default.

The visibility scope and lifetime of a variable are limited by the code block where it is declared. A **code block** means a part of code placed in curved brackets.

Explore how variables are initialized and described in Java.

*Click on the tabs to explore the syntax used to create variables.*

**Declaring a variable**

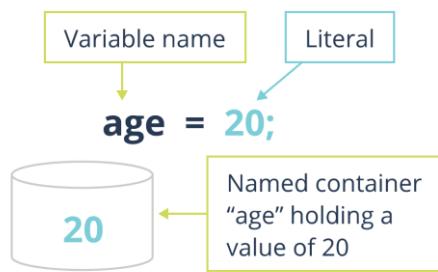
**Initializing a variable**

**Describing a variable**

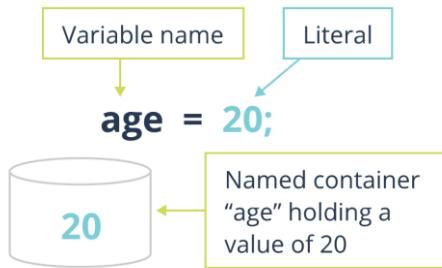
**Declaring a variable** means defining its type and name. For example:



You can **initialize** an earlier declared variable. Initialization includes the variable's identifier, assignment statement, and a value in the form of a literal expression. For example:



**Describing a variable** means declaring and initializing it in one line. For example:



You have seen that a variable stores a value of the declared type and can change this value during program execution. You have also explored the syntax for creating variables.

## Data Types and Their Storage Place

Depending on the types of variables, their storage places may be different. In this part of the lesson, you'll review this issue in more detail.

A **stack** is an area of memory to work with local variables as the program is executed. The stack is being continuously filled and cleaned. It is important for the stack to work quickly, since the running speed of the entire program depends on this. Thus, only small data whose size is known in advance are placed there. All the other data are placed in a **heap** – a large memory area that is not so quickly accessible but there you can place large complex objects.

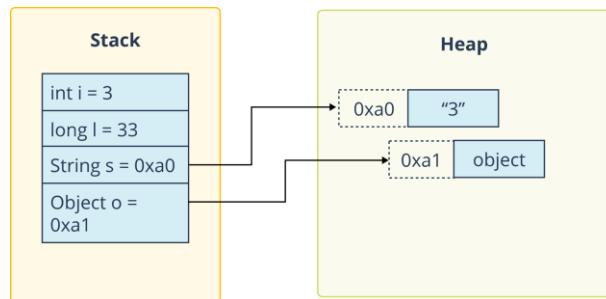
Therefore, two different types of data exist in Java:

- **primitive** – small types of data with a predefined size, mostly to work with numbers, values are stored in the stack directly.
- **reference** – for variables of this type, the stack only stores the address of the memory cell in the heap where the object related to this variable is stored.

Consider an example of code containing several variables. The first two variables are of the primitive types: **int** and **long**. For these types, the size is known in advance – 4 and 8 bytes, respectively; therefore, their values can be placed directly in the stack. The variables of the types **String** and **Object** are reference variables. These types do not define the size of objects related to them: a string value may be one or one thousand characters long. Thus, the data of these objects are stored in the heap, and the stack only stores their addresses.

```

public class Example {
public static void main(String[] args) {
int i = 3;
long l = 33L;
String s = "3";
Object o = new Object();
}
  
```



}

To sum up, data in Java can be stored on a stack or on a heap.

## Conclusion

In this lesson, you have found out that:

- A variable is a named program object storing a value of the declared type that can change this value as the program is executed.
- In Java, data can be stored in the stack or heap.
- Primitive data types are small, with a predefined size, used mostly to work with numbers.
- Reference data types – for such variables the stack only stores the address of the memory cell in the heap where the object related to this variable is stored.

## Check Your Knowledge!

1. Choose the correct characteristic of the variable scope.

The scope of a variable is limited by the stack volume

**The scope of a variable is limited by the code block where it is declared**

The scope of a variable is limited only for primitive data

The scope of a variable is not limited

Answer

Correct:

The visibility scope and lifetime of a variable are limited by the code block where it is declared.

2. What value does a variable declared in the method body store by default?

A zero value of the relevant type

The value is unknown. Without initialization, the variable's value is determined by the previous content of the memory allocated for this variable.

**None. This variable must be initialized.**

Answer

Correct:

In Java, local variables must be initialized before they are used.

3. The stack only stores the address of the memory cell located in the heap for:

A primitive variable

**A reference variable**

A variable of any type

Answer

Correct:

For reference variables, the stack only stores the address of the memory cell in the heap where the object related to this variable is stored.

# Primitives

## Introduction

In this lesson, you will study primitive data types and the specifics of their application. You will also explore the application of the keyword var.

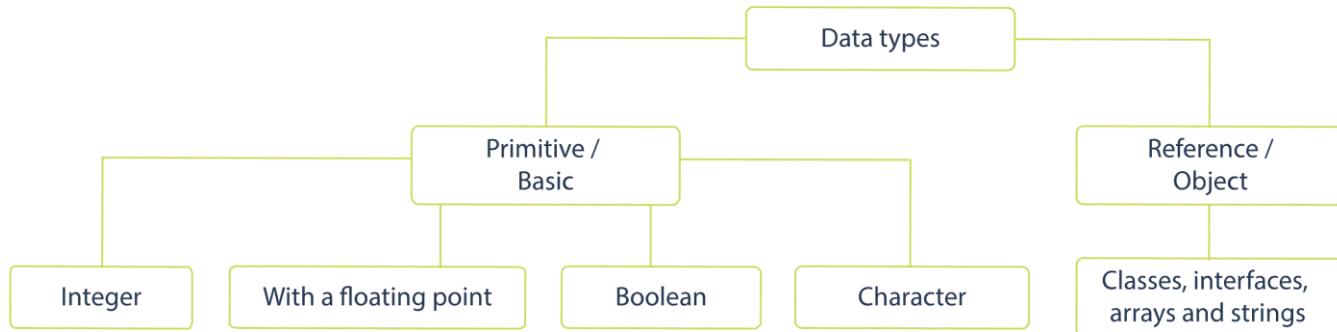
## Primitive Data Types

Although Java is an object-oriented programming language, not all data in it are objects. The language also uses primitive (or basic/predefined) data types. This was done to increase performance, since the values of primitive data types are placed in the stack memory. This memory is more readily accessible than the area of object storage.

- Any data processed by the program are part of some type.

Pursuant to standard [IEEE Std 1320.2-1998](#), a **data type (type)** is a set of values and operations with these values. You can also say that the **data type** is a fundamental concept in the programming theory. The data type not only determines the range of possible values and set of operations performed with these values but also the method of storing the values and performing operations with them.

*Click on the active elements to learn more.*



## Primitive data types

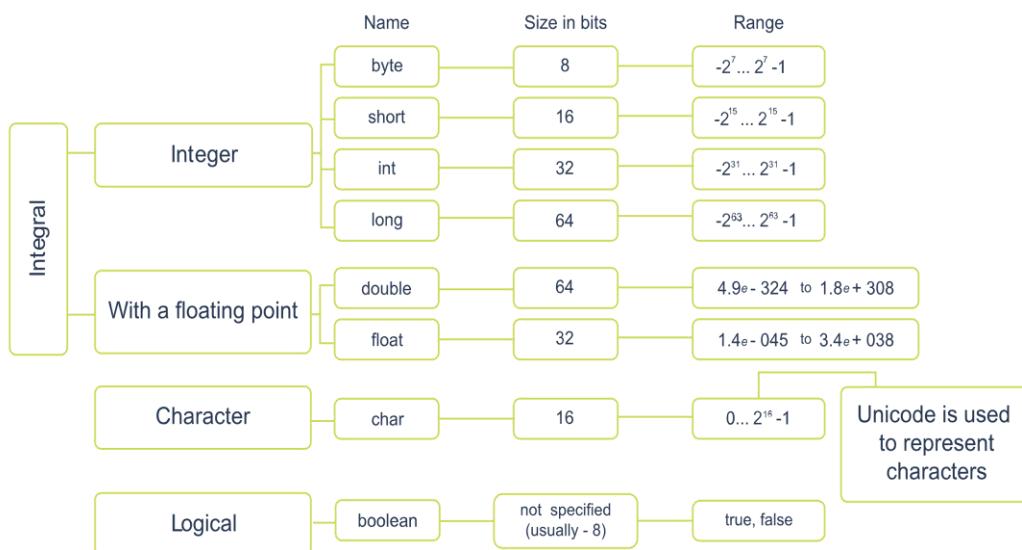
Primitive data types often use small and simple data. Depending on the type, they take one to eight bytes of memory.

## Reference data types

Reference data types are used to point at objects. In other words, an object's name is a reference to this object that contains the memory address where the object is stored. You cannot manipulate a reference, i.e., perform operations with it.

To support the object-oriented concept for each primitive data type, wrapper classes were created that encapsulate data of the primitive types into objects. Such objects are placed in the dynamic memory (heap).

Java has eight primitive data types for numeric, alphabetic, and logical values. The amount of memory taken by the values of primitive types is fixed and does not depend on the platform.



In Java, there are no unsigned data types. Each data type clearly defines a set of values and the method of their representation in the memory. Each type uniquely defines the set of operations with the values of this type. Now it is high time to find out what literals are.

## Literal Expressions

**Literal expressions** are clearly defined values of a certain type in the program code. You could say that literal expressions are a representation or reflection of a variable's value.

 A literal expression has a type in Java.

To represent integer values, various computing systems use an integer literal that has the type **int**.

For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

### ▼ Integer literals

byte var1

```
= -35;
short var2 = 0b1011;
int var3 = 0x51b;
int var4 = 071;
```

If you need to define a long literal of the type **long**, at the end specify the letter (suffix) L or l. For clarity, it is recommended to use an uppercase letter. For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
long var5 = 12345L;
long var6 = 0xffffL;
```

Starting from Java 7, for easier readability of long literals (large numbers), you can add an underscore ("\_") to separate digits. This style can also be used for numbers with a floating point. For example, instead of:

```
int var7 = 7000000;
```

You can write:

```
int var8 = 7_000_000;
```

Specifics of using the underscore symbol:

- It may only be used between digits.
- You can use any number of underscores between digits.
- The underscore symbol cannot be placed at the following places:
  - Before the first and after the last digits of the literal
  - Before the suffix or prefix

### ▼ Literal with a point

Floating-point literals describe real values and by default have the type **double**. This literal can be given in two forms:

- With a fixed point:
 

```
double var9 = 1.618;
```
- In exponential form (floating point):
 

```
double var10 = -0.112E-05;
```

When it is necessary to specify a literal of the type **float**, after the last digit use the character F or f as a suffix:

```
float var11 = -18.456F;
```

Pursuant to standard [IEEE 754](#), for real values special values were introduced to describe infinity (+Infinity and – Infinity) and the value NaN (Not a Number, uncertainty) that can be received as the result of execution of an incorrect operation. For example, a square root of a negative value.

### ▼ Boolean (logical) statements

The logical data type is used to program various alternative actions and program scenarios developing in different ways. This helps imitate real life. The logical type represents one bit of information, but its "size" is determined by the virtual machine (the number of bits to store the value).

 The boolean (logical) literals include **true** and **false**.

```
boolean flag = true;
```

### Character literals

The description of a character literal is always enclosed in single quotes, for example: 'a', '\n', '\141', '\u005a'. A character literal has two forms: the first in the form of a graphic image of the character in single quotation marks, and the second one in the form of 'ucode' according to which two bytes are allocated for each character:

- The first byte contains the code of the control character or alphabet.
- The second byte corresponds to the specification of the character in the standard of the ASCII code.

Unicode is mostly used for control characters that have no graphical image:

- '\n' - new line
- '\r' - transition to the beginning of the current line
- '\f' - new page
- '\t' - tabulation
- '\b' - backspace
- '\uxxxx' - hexadecimal Unicode character
- '\ddd' - octal character

Here are some examples of character literals:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
char ch1 = '3';
char ch2 = 'g';
char ch3 = '(';
char ch4 = '\u0034';
char ch5 = '\u002e';
char ch6 = '\t';
char ch7 = '\n';
char ch8 = '\r';
char ch9 = '\b';
```

### String literals

String literals are used to write text consisting of several characters in the program's source code. These literals are specified in double quotation marks.

A special area is allocated in the heap for exclusive storage of string literals called a **string pool**.

 String literals are not basic data, but objects of the class **String**.

```
String str = "Text";
System.out.println("Hello");
```

Note that a string literal may not end with the character '\0', since this is not a string in the format of the ASCII-code, but an object consisting of a set of characters. You can perform various operations with a string literal described in the methods of the class **String**. You will study the application of strings in more detail in the next lessons. Now just focus on the operation of string concatenation (consolidation), which is indicated by the + sign. This operation adds a string representation of another object at the end of the initial string object.

```
int number = 100;
String str = "Value";
System.out.print(str + " number " + number + "\n");
```

Thus, you have explored the concept and types of literals. You can find out more about primitive data types from the [official Oracle documentation](#).

## Keyword var

The keyword **var** was added to Java 10. It allows you to define a variable. The variable will have the same type as the literal it is initialized with. For example:



```
var x = 10;
var i = Integer.valueOf(2);
var str = "java";
```

If you want to define a variable using **var**, it needs to be initialized at declaration, i.e., you must specify its initial value. Otherwise, you will get a compilation error.

```
var x;
x = 10;
```

## Conclusion

In this lesson, you have studied primitive data types. You found out that:

- Integer literals include **byte**, **short**, **int**, and **long**. They are used to represent integer values.
- Floating-point literals describe real values and by default have the type **double**. They can also have the type **float**.
- The boolean (logical) literals include **true** and **false**. They are used to program various alternative actions.
- **Char** is a character literal. Its description must be enclosed in single quotes. A character literal has two forms: the first in the form of a graphic image of the character in single quotation marks, and the second one in the form of '\ucode' according to which two bytes are allocated for each character.
- String literals are used to write text consisting of several characters in the program's source code. These are objects of the class **String**.
- You can define a variable using the keyword **var**. To do this, you must initialize the variable when declaring it.

## Check Your Knowledge!

1. Choose primitive integer data types:

```
integer byte    short    long    data
```

Answer

Correct:

Integer literals include byte, short, int, and long.

2. Choose options for the correct use of the underscore symbol in the representation of literals:

```
double m1 = 5_000_000.75;
```

```
long m2 = _5_000_000;
```

```
int m = 0b_1010;
```

```
int m4 = 5_000_000;
```

```
int m5 = 0_10;
```

```
int m6 = 5_000_000_;
```

Answer

Correct:

The underscore symbol may only be used between digits. You can use any number of underscores between digits. The underscore symbol may not be used before the first and after the last digits of the literal, as well as before the suffix or prefix.

3. What is the size of the data type short in bits?

8      **16**      32      64

Answer

Correct:

The size of the data type short is 16 bits.

4. Choose the correct options to assign a value:

```
byte a = 0b0001_1110;
```

```
byte b = 1____14;
```

```
short s = 46_;
```

```
int i = _78;
```

Answer

Correct:

The '\_' symbol may not start or complete the declaration of a numeric literal.

# Operators

## Introduction

In this lesson, you will review the application of various operators in Java and explore their priority. Moreover, you will get familiar with methods of the class **Math**.

## Operators and Operands

In Java, **operators** are special commands informing the translator that you wish to perform an operation with the specified operands. A translator is a JDL tool transforming the source code of a Java program into a binary representation — machine commands. **Operands** are the data that are subject of the operation.

There are several kinds of operators:

Prefix unary

Operators specified before the operands and performing an operation with one operand.

Postfix unary

Operators specified after the operand and performing an operation with one operand.

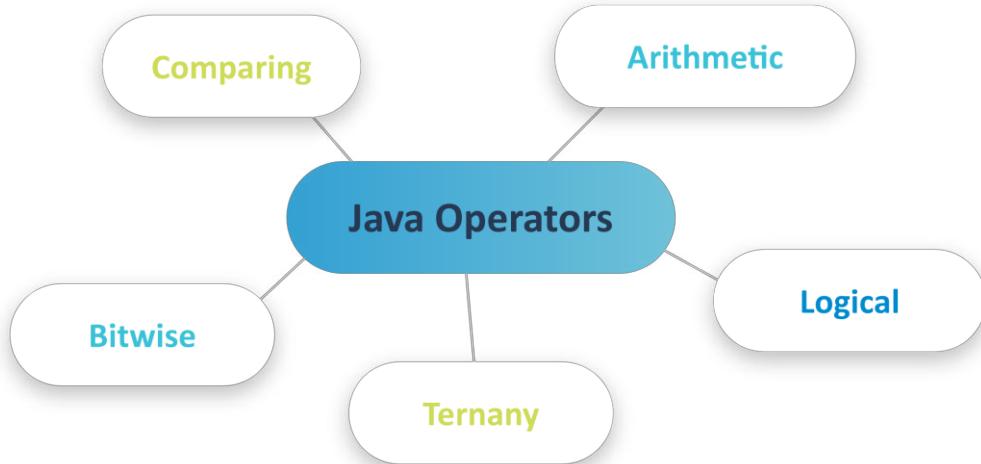
Infix binary

Operators specified between two operands.

Ternary

Operators working with three operands.

Consider the operators existing in Java. They can be split into several groups shown in the image:



Now review each type of operator separately and the specifics of their application.

*Click on the drop-down lists to learn more about operators in Java.*

### ▼ Arithmetic operators

Arithmetic operators are used in math expressions in the same way as they are used in algebra. For each arithmetic operator, there is a form in which the result is assigned to the first operand simultaneously with the specified operation. The Java language supports the following arithmetic operators:

Operator	Description	Type	Example
+	Adds the values on both sides of the operator.	Binary	$a + b$
-	Subtracts the right operand from the left one.	Binary	$a - b$
Changes the sign.		Prefix Unary	$-a$
*	Multiplies the values on both sides of the operator.	Binary	$a * b$
/	The operator divides the left operand by the right operand (or divides evenly for integers).	Binary	$a / b$
%	Divides the left operand by the right operand and returns the remainder.	Binary	$a \% b$
++	Increment – increases the value of the operand by 1. It has a prefix and a postfix form.	Prefix Unary Postfix Unary	$++a$ $a++$ $a$
--	Decrement – decreases the value of the operand by 1. It has a prefix and a postfix form.	Prefix Unary Postfix Unary	$--a$ $a--$ $a$

The following example demonstrates the work of arithmetic operators:

```
public class LearnArithmetic {
    public static void main(String args[]) {
        int a = 10;
        int b = 20;
        int c = 25;
        int d = 25;
        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("b / a = " + (b / a));
        System.out.println("b % a = " + (b % a));
        System.out.println("c % a = " + (c % a));
        System.out.println("a++ = " + (a++));
        System.out.println("a-- = " + (a--));
        System.out.println("d++ = " + (d++));
        System.out.println("++d = " + (++d));
        System.out.println("a += b = " + (a += b));
        System.out.println("a = " + (a));
        System.out.println("a -= b = " + (a -= b));
        System.out.println("a = " + (a));
    }
}
```

### Output:

$a + b = 30$

$a - b = -10$

$a * b = 200$

$b / a = 2$

$b \% a = 0$

$c \% a = 5$

$a++ = 10$

$a-- = 11$

```
d++ = 25
++d = 27
a += b = 30
a = 30
a -= b = 10
a = 10
```

### ▼ Bitwise operators

Bit (bitwise) operators deal with bits and perform operations bit by bit. In Java, bit operators are used only for integer types: int, long, short, char, and byte.

```
int a = 60; //In binary representation, it is 0000 0000 0000 0000 0000 0011 1100
int b = 13; //In binary representation, it is 0000 0000 0000 0000 0000 0000 1101
```

The Java language supports the following bitwise operators:

Operator	Description	Type	Example
&	The bitwise operator <b>and</b> will put 1 into the result bit if both operands have 1 in this position.	Binary	(a & b) will give 12 (... 0000_1100)
	The bitwise operator <b>or</b> will put 1 into the result bit if either operand has 1 in this position.	Binary	(a   b) will give 61 (... 0011_1101)
^	The bitwise operator <b>exclusive or</b> will put 1 into the result bit if one operand, but not both, has 1 in this position.	Binary	(a ^ b) will give 49 (... 0011_0001)
~	The bitwise operator <b>complement</b> "reflects" a bit.	Unary prefix	(~ a) will give -61 (1111_1111_1111_1111_1111_1100_0011)
<<	With the bitwise operator <b>left shift</b> , the value of the left operand is shifted to the left by the number of bits set by the right operand. The vacated bits on the right are filled with zeros.	Binary	a << 2 will give 240 (... 1111_0000) b << 2 will give 52 (... 0011_0100)
>>	With the bitwise operator <b>right shift</b> , the value of the left operand is shifted to the right by the number of bits set by the right operand. For positive numbers, the vacated leftmost bits are filled with zeros, and with ones for negative numbers.	Binary	a >> 2 will give 15 (... 0000_1111) b >> 2 will give 3 (... 0000_0011) (~ a >> 2) will give -16 (1111_1111_1111_1111_1111_1111_0000)
>>>	With the <b>zero-fill right shift</b> operator, the value of the left operand is shifted to the right by the number of bits set by the right operand, and the shifted values are always filled with zeros.	Binary	a >>> 2 will give 15 (... 0000_1111) (~ a >> 2) will give 1073741808 (0011_1111_1111_1111_1111_1111_1111_0000)

The following example demonstrates the work of bitwise operators:

```
int a = 60;
int b = 13;
```

```
System.out.println("a & b = " + (a & b)); // 12 = ... 0000 1100
System.out.println("a | b = " + (a | b)); // 61 = ... 0011 1101
System.out.println("a ^ b = " + (a ^ b)); // 49 = ... 0011 0001
System.out.println("~a = " + ~a); // -61 = 1111 1111 1111 1111 1111 1100 0011
System.out.println("a << 2 = " + (a << 2)); // 240 = ... 1111 0000
System.out.println("a >> 2 = " + (a >> 2)); // 15 = ... 0000 1111
System.out.println("a >>> 2 = " + (a >>> 2)); // 15 = ... 0000 1111
```

### Comparison operators

Often when working with numbers we use comparison operations. The result of comparison will always be "true" or "false". The Java language supports the following comparison operators:

Operator	Description	Type	Example
==	Checks whether the values of two operands are equal.	Binary	a == a
!=	Checks whether the values of two operands are not equal.	Binary	a != b
>	Checks whether the value of the left operand is larger than the value of the right operand.	Binary	a > b
<	Checks whether the value of the left operand is smaller than the value of the right operand.	Binary	a < b
>=	Checks whether the value of the left operand is larger than or equal to the value of the right operand.	Binary	a >= b
<=	Checks whether the value of the left operand is smaller than or equal to the value of the right operand.	Binary	a <= b

Note: The result of an operator will be "true" if the comparison operation gives a positive response, i.e., "yes".

The following example demonstrates the work of comparison operators:

```
public class LearnEquality {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        System.out.println("a == b = " + (a == b));
        System.out.println("a != b = " + (a != b));
        System.out.println("a > b = " + (a > b));
        System.out.println("a < b = " + (a < b));
        System.out.println("b >= a = " + (b >= a));
        System.out.println("b <= a = " + (b <= a));
    }
}
```

### Output:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

### Logical operators

When developing programs, often you must deal with logical operators that execute operations with logical values. The result of logical operations is always one of two possible values: "true" or "false". The Java language supports the following logical operators:

Operator	Description	Type	Example
&&	Logical "AND". If both operands have the value true, then the result will be true, otherwise – false.	Binary	a && b
	Logical "OR". If either of the two operands has the value true, then the result will be true, otherwise – false.	Binary	a    b
!	Logical "NOT". Changes the operand's value to the opposite one: from true to false and vice versa.	Prefix Unary	!(a && b) !a

The operator **instanceof** is used to test whether an object is an instance of the specified type. This operator also falls within the class of logical operators. It returns "true" if an object is an instance of the specified type. The syntax of the operator **instanceof** is as follows:

<reference to object> instanceof <type of class/interface>

The following example demonstrates the work of the operator **instanceof**:

```
public class LearnInstanceofMain {
    public static void main(String[] args) {
        String name = "EPAM";
        boolean result = name instanceof String;
        System.out.println(result);
    }
}
```

### Output:

true

You will study this operator in more detail in the upcoming lessons.

### ▼ Ternary operator

Ternary operator "?:" is an operator consisting of three operands and known as a "conditional operator". The goal of this operator is to make a decision about which value should be returned in the result of its execution. Syntax of the operator:

<condition> ? <expression\_1> : <expression\_2>

If the condition result is true, then the operator will return the value of expression\_1, otherwise the value of expression\_2. Characteristic property: both expressions should be of the same type. The following example demonstrates the work of the ternary operator:

```
public class LearnTernary {
    public static void main(String[] args){
        int experience = 5;
        int requirements = 10;
        String result = (experience > requirements) ? "Accept to project" : "Learn more" ;
        System.out.println(result);
    }
}
```

### Output:

Learn more

Thus, you have reviewed which operators are used in Java. As you may have noticed, there are a lot of them. What should a programmer do in a situation when one expression contains several operators? Which one should they use first? Check the next chapter of this lesson to find the answer.

## Priority of Operators in Java

The priority of operators affects the order of the expression computing and helps determine the grouping of operands in the expression. Some operators have a higher priority than others. For example, arithmetic operators: the multiplication operator has a higher priority than the addition operator. Consider an example of an expression:  $x = 7 + 3 * 2$ .

Here  $x$  will get a value of 13, not 20. Why so? The answer is simple: the "\*" operator has a higher priority than the "+" operator. Therefore, the multiplication "3 \* 2" will be executed first and then "7" will be added.



In an expression, the priority of operators is evaluated from left to right. The operators are shown in the table starting from the top and going from **highest to lowest priority**. Associativity of operators is the direction of execution of operators with the same priority.

### Priority of operators in Java

Postfix	$() [] .$ (dot)	Left to right
Unary	$++ -- ! ~$	Right to left
Multiplicative	$* / %$	Left to right
Additive	$+ -$	Left to right
Shift	$>> >>> <<$	Left to right
Relational	$> >= < <=$	Left to right
Equality	$== !=$	Left to right
Bitwise "AND" ("AND")	$\&$	Left to right
Bitwise exclusive "OR" ("XOR")	$^$	Left to right
Bitwise "OR" ("OR")	$ $	Left to right
Logical "AND" ("AND")	$\&\&$	Left to right
Logical "OR" ("OR")	$\ $	Left to right
Ternary (conditional)	$:$	Right to left
Assignment	$= += -= *= /= %= >>= <<= \&= ^=  =$	Right to left
Comma	$,$	Left to right

To work with various operators, it's useful to know the methods of the class **Math**. Study them in the next chapter.

## Methods of the Class Math

The class **Math** contains constants and methods to work with math functions. You can call these methods using the class name, a dot after the name, and then specifying parameters for the method in round brackets separated by a comma. For example:

```
int max = Math.max(3, 4);
```

The class **Math** contains approximated values of mathematical constants – the numbers "PI" and "E":

- PI // 3.14159265358979323846
- E // 2.7182818284590452354

Set of basic **math methods**:

- abs – returns the modulus of a number: Math.abs(-4) // 4;
- max – returns the larger of two numbers: Math.max(1, 2) // 2
- min – returns the smaller of two numbers: Math.min(1, 2) // 1

A set of **trigonometric functions** that accept angle values in radians:

- sin – returns the sine of an angle: Math.sin(0) // 0
- cos – returns the cosine of an angle: Math.cos(0) // 1
- tan – returns the tangent of an angle: Math.tan(0) // 0
- asin – returns the arcsine of an angle: Math.asin(0) // 0
- acos – returns the arccosine of an angle: Math.acos(1) // 0
- atan – returns the arctangent of an angle: Math.atan(0) // 0

There are also some convenient methods to **convert degrees to radians** and vice versa:

- toRadians: toRadians(1) // 0.017453292519943295
- toDegrees: toDegrees(1) // 57.29577951308232

Methods to work with **powers**:

- exp – returns  $e$  to the power of x: Math.exp(1) // 2.7182818284590452354
- log – returns the natural logarithm (base  $e$ ): Math.log(Math.E) // 1
- log10 – returns the base 10 logarithm of a value: Math.log10(100) // 2
- sqrt – returns the square root of a value: Math.sqrt(16) // 4
- cbrt – returns the cube root of a value: Math.cbrt(-8) // -2
- pow – returns the result of a value raised to some other value: Math.pow(2, 4) // 16

Major **rounding** methods:

- ceil – rounding up: Math.ceil(3.3) // 4
- floor – rounding down: Math.floor(5.7) // 5
- round – rounding pursuant to the standard math rules: Math.round(5.8) // 6

Method to **get a random value**:

- random – returns a random value of the type double within the limits of [0;1)

## Conclusion

In this lesson, you found out that:

- Arithmetic operators are used in math expressions in the same way as they are used in algebra.
- Bitwise operators execute operations bit by bit. In Java they are used only for integer types.
- The result of comparison and logical operations will always be "true" or "false".
- The goal of the ternary operator is to make a decision about which value should be returned in the result of its execution.

Moreover, you studied the order of priority of operators in Java, as well as explored the methods of the class **Math**.

## Check Your Knowledge!

1. What is the result of running the following code?

```
class Main {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
    }
}
```

1234554321

66666

**17777**

Answer

Correct:

The '+' operator performs addition notwithstanding the types of operands. Therefore, simple addition will be done.

2. What is the result of running the following code?

```
class PrintTest {
    public static void main(String[] args) {
        System.out.println("2 + 2 = " + 2 + 2);
    }
}
```

2 + 2 = 4

**2 + 2 = 22**

"2 + 2 = " 4

"2 + 2 = " 22

Answer

Correct:

The first operand is a string. A string will be the result of the entire operation. No arithmetic addition actions will be performed.

3. Which line codes will return true as a result for the following variables?

**int a = 10;**

**int b = 20;**

System.out.println(a > 20 && b > 10);

**System.out.println(a > 20 || b > 10);**

System.out.println(! (b > 10));

System.out.println(! (a > 20));

Answer

Correct:

a > 20 || b > 10 -- the result is "true", if at least one operand is "true". ! (a > 20) -- the result is "true", since the negation of the result is "false".

4. Which code lines will be compiled without errors for the following variables?

String s = "Hello";

**long l = 99;**

**double d = 1.11;**

**int i = 1;**

**int j = 0;**

j = i << s;

**j = i << j;**

j = i << d;

**j = i << l;**

Answer

Correct:

Only integers can be used in the right part of the shift operators.

5. What is the result of running the following code?

```
class TestOperationBinary {  
    public static void main(String[] args) {  
        System.out.println(010 | 4);  
    }  
}  
14  
0  
6  
12
```

Answer

Correct:

$1000 | 0100 = 1100$

6. Select the method of the class **Math** that returns the modulus of a number.

Answer

The method **abs** returns the modulus of a number.

7. Select the method of the class **Math** that performs rounding down.

Correct:

The method **floor** performs rounding down.

8. Select the method of the class **Math** that returns the square root of a value.

Correct:

The method **sqrt** returns the square root of a value.

In this lesson, you will explore what typecasting is and how it is used in Java. You will also review how compilation is optimized.

## What Is Typecasting?

A fixed number of bytes is allocated in the memory to store the values of any basic data type. This affects the results of execution of operators with various types of data. Therefore, to avoid an error you need to correctly define the type of a variable that will receive the result of the expression.

There are some rules that **determine the type of the resulting expression**.

*Click on the tabs to study these rules.*

Rule 1

Rule 2

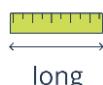
Rule 3

Rule 4



int

If an expression only has integer literals and/or variables up to the type **int**, inclusive, then the result of the expression will be **int**. In other words, even if the expression does not have variables of the type **int**, the result of the expression will be **int** anyway.



long

If an expression has a variable or literal of the type **long**, the result of the expression will be **long**.



float

If an expression has a variable or literal of the type **float**, the result of the expression will be **float**.



double

If an expression has a variable or literal of the type **double**, the result of the expression will be **double**.

If an expression contains data of different types, for the expression to be executed, all data have to be typecast to the same type.

There are two typecasting kinds for primitive types – implicit and explicit.



Implicit typecasting



Explicit typecasting

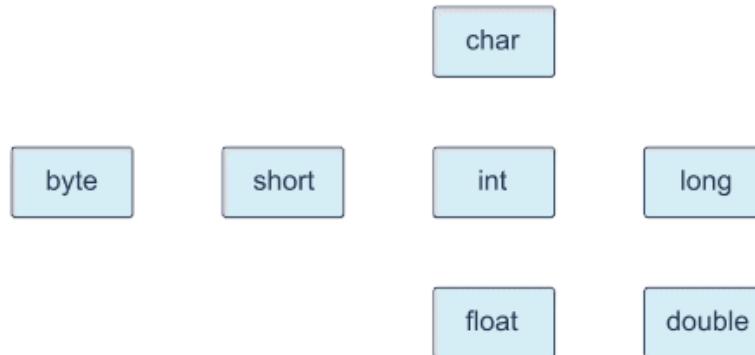
Only widening typecasting can be performed implicitly. This typecasting allows not to lose information about the value of a numeric expression. Java performs this typecasting automatically.

Narrowing typecasting is done explicitly. At that, the information about the value of the numeric expression, as well as precision and range, may be lost. This operation is performed by the developer.

As you may have already understood, implicit typecasting is done automatically, and explicit typecasting is done by the developer. Study these forms of typecasting in more detail in the next chapter of this lesson.

- byte → short, int, long, float, double
- short → int, long, float, double
- char → int, long, float, double
- int → long, float, double
- long → float, double
- float → double

In the following image, arrows show which kinds of typecasting can be performed automatically. Dashed arrows show automatic transformations with a possible loss of precision.



Therefore, as you can see from the image, most automatic transformations are performed without a loss of precision, but there may be cases when transformation precision may be lost. Dwell on this aspect.

*Click on the tabs to learn more.*

### Without a loss of precision

### With a loss of precision

**Widening transformations** are done automatically **without a loss of precision** – meaning transformations that widen the representation of an object in the memory. For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
byte b = 7;
int d = b;
```

In that case, the value of the variable of the type **byte** (with the memory volume of 1 byte) is widened to the type **int** (with a volume of 4 bytes).

There are transformations that can be done automatically between the data types with the same bitness or from a type with a greater bitness to a type with a lower bitness. For example, the following sequences of transformations, int into float, long into float, and long into double, are done without errors, but a loss of data can occur during the transformation. For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
int a = 2_147_483_647;
float b = a;
System.out.println(b);
```

You can widen casting explicitly, but this is not recommended as it is not necessary.

## Narrowing Typecasting

The narrowing typecasting of primitive types includes 22 specific transformations:

- short → byte or char
- char → byte or short
- int → byte, short, or char
- long → byte, short, char, or int
- float → byte, short, char, int, or long
- double → byte, short, char, int, long, or float

These transformations are performed explicitly and are described using the following form:

type variable/literal;

In the case of narrowing typecasting of one value to another integer type, all bits, except for the **n** junior bits (where **n** is the number of bits of the representation type resulting from typecasting), are truncated. At that, the following is possible:

- A loss of data about the value of the data point
- A change of sign of the resulting value

For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
int valueInt = 34_567;
short valueShort = (short) valueInt;
System.out.println(valueInt + " -> " + valueShort);
```

The narrowing transformation from the type **double** into the type **float** is governed by rounding rules IEEE 754. At that, the following can occur:

- A loss of precision
- A loss of range (for example, you can get a zero value of the type **float** from a nonzero value of the type **double**, and an unlimited value of the type **float** from a limited value of the type **double**)

For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
double valDouble = 1.0e-46;
float valFloat = (float) valDouble;
System.out.println(valDouble + " -> " + valFloat);
```

It is worth noting that there are conditions when no explicit typecasting is required meaning that in all such cases typecasting is done implicitly. Review such conditions.

*Click on the following tabs to learn more.*

#### ▼ The literal is included in the range of the variable type

When describing a variable: if the literal value is included in the range of the described type. For example:

```
byte b = 35;
```

#### ▼ The result of the expression is included in the range of the variable type

When describing a variable, if it is initialized using arithmetic operators for literals and read-only variables: if the value of the expression is included in the range of the described type. For example:

```
final byte b1 = 5;
byte b2 = b1 + 10;
```

#### ▼ Short form of operators

When

using the operations increment (++) and decrement (--), as well as a short form of operators: (+=, /=, \*/, etc.) For example:

```
byte b3 = 50;
int iVal = -100;
b3 += iVal--;
```

In this code, the value of the variable **iVal** is added to the variable **b3** and then is decreased by 1. The result is recorded in the variable **b3**. Because of the short form of the addition operation with an assignment, there is no need in explicit typecasting of the result to the type **byte**.

## Compilation Optimization

The compilation of Java applications is different from the compilation of applications in C-like programming languages. A static compiler transforms the source code directly into machine commands that can be executed on the target platform. The Java compiler transforms the source code into portable JVM bytecodes, that are "virtual machine instructions" for the JVM. At that, the Java compiler performs insignificant optimization unlike other static compilers that perform optimization during the program execution.

Consider several examples of compilation optimization.



The source code contains the following code line:

```
byte b = 2 + 3;
```

But it is not optimal to spend time during program execution calculating a sum of two invariant literals. Therefore, this is done at the stage of code compilation, and the bytecode receives the following:

```
byte b = 5;
```

The same applies to the finalized variables:

```
final int X = 7;
```

```
int y = 12 + X;
```

Since the value of the finalized variable does not change, the last bytecode line will be written as follows:

```
int y = 19;
```

## Conclusion

In this lesson, you have explored the rules that determine the type of the resulting expression. You have found out that:

- Widening typecasting can be performed implicitly. This type allows not to lose information about the value of a numeric expression. Java performs this typecasting automatically.
- Narrowing typecasting is done explicitly. At that, the information about the value of the numeric expression, as well as precision and range, may be lost. This operation is performed by the developer.

You have studied the peculiarities of performing these forms of typecasting and found out how compilation optimization is done in Java.

## Check Your Knowledge!

1. Which code lines will lead to a compilation error due to a wrong typecasting?

```
int i = 3;
byte b = 1;
byte b1 = 1 + 2;          // line 1
short s = 304111;        // line 2
short s1 = (short) 304111; // line 3
b = b1 + 1;              // line 4
b = (byte) (b1 + 1);     // line 5
b = -b;                  // line 6
b = (byte) -b;            // line 7
b1 *= 2;                 // line 8
b = i;                   // line 9
b = (byte) i;
b += i++;
float f = 1.1f;
b /= f;
```

Line 1

**Line 2**

Line 3

**Line 4**

Line 5

**Line 6**

Line 7

Line 8

**Line 9**

Correct:  
Page

Line 2: the maximum possible value for short is equal to 32767. Line 4: the addition operation will result in int. Besides, b1 was not declared as final. Line 6: the unary negation will result in a value of the type int. Line 9: the type int assigns the type byte without an explicit conversion.

2. Choose the correct expressions:

```
float f = 1 / 2;
int i = 1 / 3;
float f = 1.45;
double d = 555d;
```

correct

Answer

Correct:

The literal 1.45 is a literal of the type double and may not be assigned to the type float without explicit typecasting.

3. Will this code be compiled?

```
public class TestCompile {
    public static void main(String[] argv){
        long x = 5;
        long y = 2;
        byte b = (byte) x / y;
    }
}
```

Yes

No

Answer

Correct:

The code will not be compiled, since only the first operand of the division operation can be reduced to the type byte. For a successful compilation, the code shall be written as (byte) (x / y);

# Reference Types

## Introduction

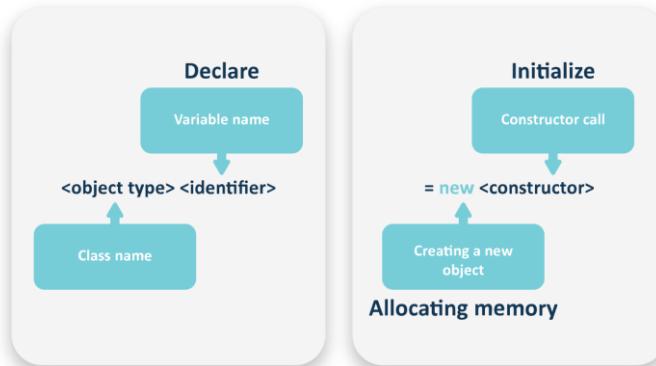
In this lesson, you will find out how objects and string literals are created. You will learn how storage of these data types is organized in Java.

## Creating Objects

In Java, objects are placed and stored in the memory area called the "heap". You can not manage this memory. Java takes care of allocation and distribution of memory, as well as its release.

The Java runtime system allocates memory to store an object when you request it using the operator **new**. If there are no references to the objects from the program, they will be destroyed (memory shall be released) by a special component of the virtual machine called **garbage collector**. In that case, developers say that the object is not accessible from the program.

Creating an object in Java looks as follows



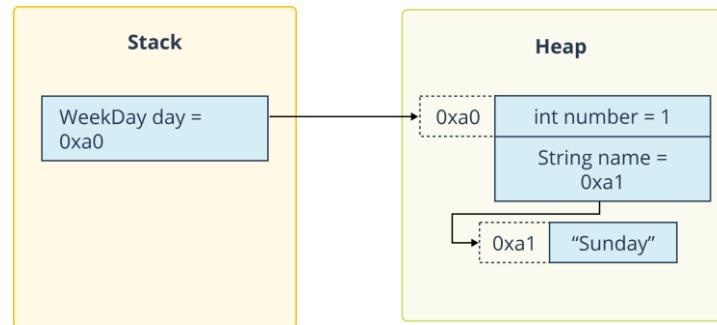
Usually, objects are a set of several fields and methods – special functions to work with them. At that, all field values, even primitives, are located in the object's memory area, in the heap. Field values of one object may refer to addresses in the heap of other objects as is shown in the following example.

```

class WeekDay{
    int number;
    String name;
}

public class Example {
    public static void main(String[] args) {
        WeekDay day = new WeekDay();
        day.number = 1;
        day.name = "Sunday";
    }
}

```



So, you have seen how objects are created, stored, and destroyed. Now, let's move on to the string literal.

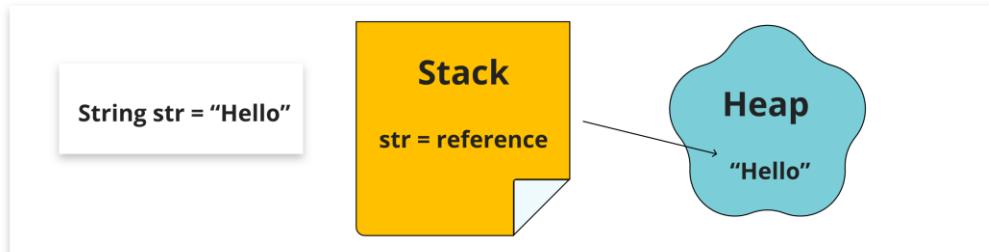
## String Literal

A **string literal** is a set of characters placed inside double quotation marks. All string literals are implemented as instances (objects) of the class **String**. For example:

```
String str1 = "Hello";
```

-  The compiler wraps all string literals in an object of the class **String** with the value of this literal.

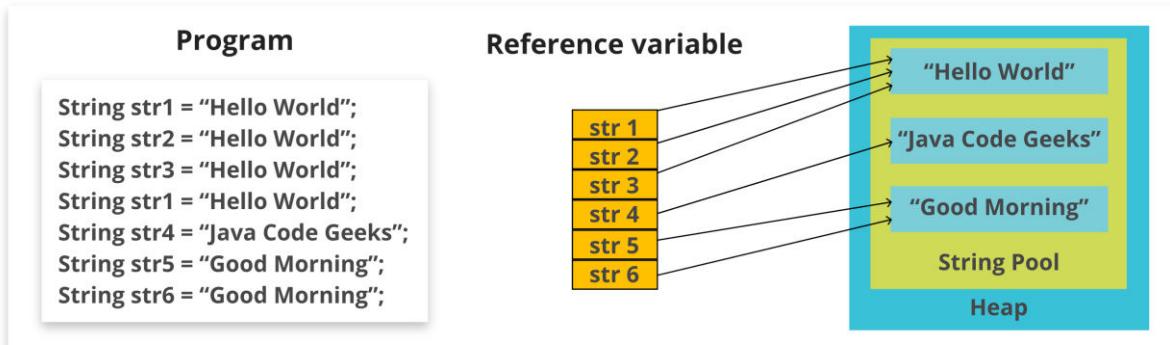
You already know that the memory used in computer programs can be of two types: stack and heap. The way they work is very different. The stack stores values of primitive type variables and the index values for reference types. The objects themselves to which these references apply are stored in the heap. The type **String** is a reference data type; therefore, string values are stored in the heap.



When you create an instance of the class **String** initializing it with a string literal (a value enclosed in double quotation marks), this object will be saved in the **string pool**. This is a specialized memory area in the heap that is a collection of references to the objects of the class **String** and is used to optimize memory.

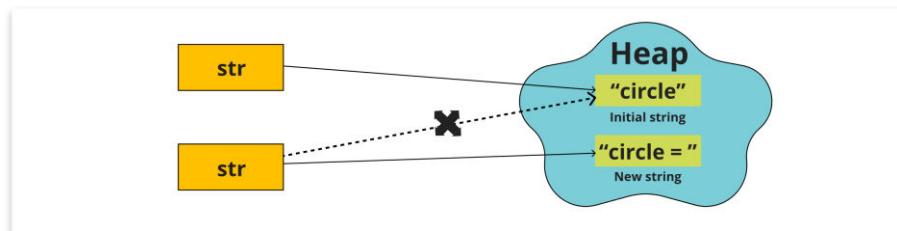
What else does it give? If you create another string object initialized with the same literal, then an attempt to add it to the string pool again will return a reference to the existing literal. This way values of literals in the memory will not be duplicated. A similar situation will occur if the literal can be calculated. For example, for the compiler the literals "Java" and "J+ava" are equivalent. If the literal you are searching for is not found in the pool (i.e., when creating a string using a truly new literal), then a new string will be added to the pool and a new reference will be returned.

The storage of the string pool is illustrated in the following image.



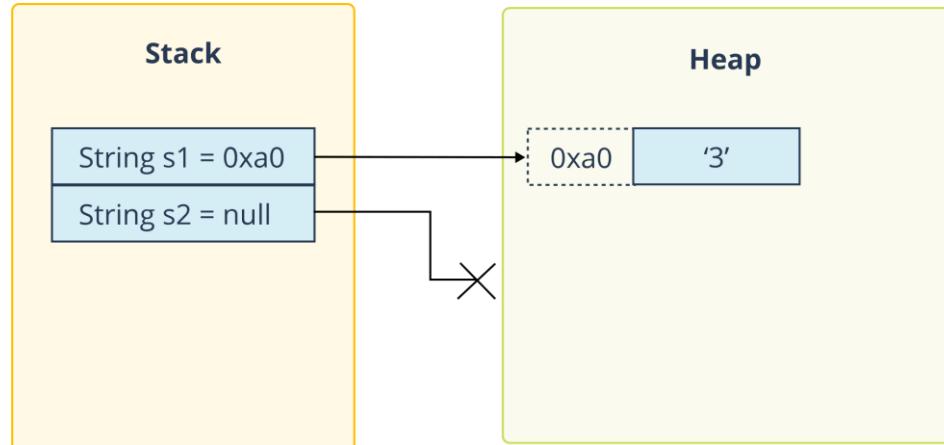
-  If two or more variables are initialized with one and the same literal, then when first accessing it a string will be created in the Heap called the "String Pool", and when accessing it later, a reference to the existing string will be returned.

This sharing is possible due to an important property called string immutability. The developers of Java decided that the efficiency of sharing outweighs the inefficiency of string editing by allocating sub-strings and concatenations. Each time the string content is edited, a new string (new object) is created with the content of the modified string.



When a reference variable is not linked to an object, this variable has a null/empty value. It is designated with the literal **null**. When accessing the fields or methods of a variable equal to **null**, you will get **NullPointerException**.

```
public class Example {
    public static void main(String[] args) {
        String s1 = "3";
        String s2 = null;
    }
}
```



Now you have explored the concept of a string literal and discovered that the compiler wraps string literals in a `String` class object with the value of the literal.

## Conclusion

In this lesson, you found out that:

- In Java, objects are placed and stored in the heap. The Java runtime system allocates memory to store an object when you request it using the operator `new`.
- A string literal is a set of characters placed inside double quotation marks. The compiler wraps all string literals in an object of the class `String` with the value of this literal. The type `String` is a reference data type; therefore, string values are stored in the heap.

## Check Your Knowledge!

1. What is a specialized memory area that is a collection of references to the objects of the class `String`?

Stack

Heap

**String pool**

Random access memory

Answer

Correct:

A string pool is a specialized memory area in the heap that is a collection of references to the objects of the class `String` and is used to optimize memory.

2. Where are objects placed and stored in Java?

Stack

**Heap**

String pool

Random access memory

Answer

Correct:

In Java, objects are placed and stored in the memory area called the "heap."

# Naming Convention in Java

## Introduction

In this lesson, you will study the naming convention for elements in Java and review examples of how you should name different elements.

## Notations

Any programs and applications work with a huge amount of data: local variables, object references, method parameters, class fields, etc. All these data must have names corresponding to certain rules (agreements). The naming agreements make the programs more understandable making their comprehension easier. They also give information on the identifier functions, for instance, whether it is a constant, package, or class. This helps understand code.

Java actively uses the **CamelCase** notation (writing styles) to name methods, variables, etc., as well as the **TitleCase** notation to name classes, interfaces, and other elements.

*Click on the cards to learn more about these notations.*



**CamelCase** is the practice of writing names consisting of several words where the words are written without spaces and separators; the first word starts with lower case and each subsequent word starts with a capital letter.

**For example:** iPhone, eBay, javaFundamentalsCourseContent, exampleTwo.

**TitleCase** is the practice of writing names consisting of several words where each word starts with a capital letter and there are no separators between the words.

**For example:** ImageSprite, TestExampleOne.

Further in this lesson, you will study Java agreements on the naming of various elements and will review examples. You will review the use of different language elements (interfaces, annotations, etc.) in more detail in the later in the course.

## Variables

All names of local variables, class fields, and method parameters should follow **CamelCase** notation.

The names of variables should be short but informative to describe their purpose. The choice of the variable name should be mnemonic, that is, it should aim to indicate its purpose to the user. Avoid single-character variable names, except for temporary "one-time" variables. For example:

- The variable names **i, j, k, m**, and **n** are often used to describe integer data.
- **c, d, and e** – for characters

```
public Long id;
public EmployeeDao employeeDao;
private Properties properties;
for (int i = 0; i < list.size(); i++) {
...
}
```

According to Java language specifications, identifiers (the names of variables, classes, and methods) can be constructed from letters, numbers, the underscore "\_" character, and the dollar sign "\$". Not only can Latin letters be used but also letters from any of the alphabets included in Unicode. Also, note that identifier names cannot start with a number and that the compiler does not allow identifiers with the same spelling as a keyword.

It is not recommended to start the variable name with the underscore symbol "\_" or dollar sign "\$", although both these symbols are permitted in the syntax and do not lead to compilation errors. Although they are used by Java itself in automatic naming.

Consider examples to see how to name variables.

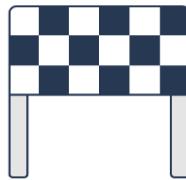
*Click on the title to see the example.*



- **Acceptable names:** firstMoney, flag, a\_string1, str1, str2, st, width, sc, seatsCount.
- **Unacceptable names:** field#, open^flag, 1searchIndex, open-flag, &string, my%Count, flag#true.
- And now check which acceptable variable names are correct and which are not:
- **Meaningful names:** firstMoney, flag, width, seatsCount.
- **Meaningless names:** a\_string1, str1, str2, st, sc.

## Read-Only Variables

There are no constants in the Java language, instead, the language uses immutable class fields. You will study classes in the following sections of the course. For now, it is enough to know that:



- Immutable variables are marked with the modifier **final**.
- Fields that are owned by a class and not by its instance are marked with the modifier **static**.

To identify immutable static fields, their names are specified in uppercase and the words are separated by the underscore symbol.

```
public static final String SECURITY_TOKEN = "...";
public static final int INITIAL_SIZE = 16;
public static final Integer MAX_SIZE = Integer.MAX;
```

## Methods

Since methods describe actions, their names should contain verbs, clearly describing the performed action. The method name may consist of one or several words required to clearly specify the action. First comes the verb. Method names follow the **CamelCase** notation.



```
public Long getId() {}
public void remove(Object o) {}
public Object update(Object o) {}
public Report findReportById(Long id) {}
public Report findReportByName(String name) {}
```

## Classes

Class names should be nouns and follow **TitleCase** notation. You should always try to use simple and self-explanatory class names. You should avoid short forms and acronyms, unless these acronyms are used wider than their full form, for example, URL or HTML.



```
public class Employee {}
public class Order {}
public class ImageSprite {}
```

## Interfaces

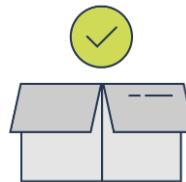
The names of interfaces, as a rule, should be adjectives and follow **TitleCase** notation. In some cases, interfaces may be named using nouns, for example, when they represent a family of classes, such as **List** and **Map**.



```
public interface Serializable {}
public interface Cloneable {}
public interface Iterable {}
public interface List {}
```

## Packages

The names of packages have a prefix that is always specified in lowercase and represents the domain name of the higher level: com, edu, gov, mil, net, org, or a two-letter country code pursuant to the standard [ISO 3166](#). The parts of the package names following the prefix can be different. They depend on the internal agreements about the names of the organization and include the names of divisions, department, projects, etc.



```
package com.company.attr;
package com.google.search.common;
```

## Parameters of the Generic Type

The names of parameters of the generic types should be a single capital letter:

- "T" is used to specify the type.
- "E" is used to specify the type of collection elements in the library JDK classes.
- "S" is used for service loaders.
- "K" and "V" are used for keys and values in **Map**.

```
public interface Map <K,V> {}
public interface List<E> extends Collection<E> {}
Iterator<E> iterator()
```

## Enums

The names of enum should be nouns and follow **TitleCase** notation. The names of enum elements the same as read-only variables, should consist of uppercase letters.

```
enum Direction {NORTH, EAST, SOUTH, WEST}
```

## Annotations

The names of annotations should follow **TitleCase** notation. They can be described by any part of speech: adjective, verb, or noun depending on the requirements.

```
public @interface FunctionalInterface {}
public @interface Deprecated {}
public @interface Documented {}
public @Async Documented {
public @Test Documented {}
```

## Conclusion

In this lesson, you found out that:

- The names of local variables, class fields, method parameters, and method names (should contain verbs) use **CamelCase**.
- The names of classes, interfaces, annotations, and lists use **TitleCase**. The names of classes and lists should be nouns, and the names of interfaces, as a rule, are adjectives.

- The names of variables with invariable values are specified in uppercase, and the words are separated with the underscore symbol.
- The names of packages have a prefix that is always specified in lowercase and represents the domain name of the higher level: com, edu, gov, mil, net, org, or a two-letter country code.
- The names of parameters of the generic types should be a single capital letter.

## Check Your Knowledge!

1. Choose the acceptable identifier names:

**firstMoney**

field#

**string1**

open-flag

**INDEX\_FOR\_SEARCH**

numberStudents&Juniors

Answer

Correct:

The names may not contain the # character or characters of arithmetic or logical operators.

2. Which of the below identifiers are acceptable?

2int;

int\_#;

**\_int;**

**\_2\_;**

**\$int;**

#int;

Answer

Correct:

The names of identifiers may not start with a number or contain the # character..

# Concept of the Code Block

## Introduction

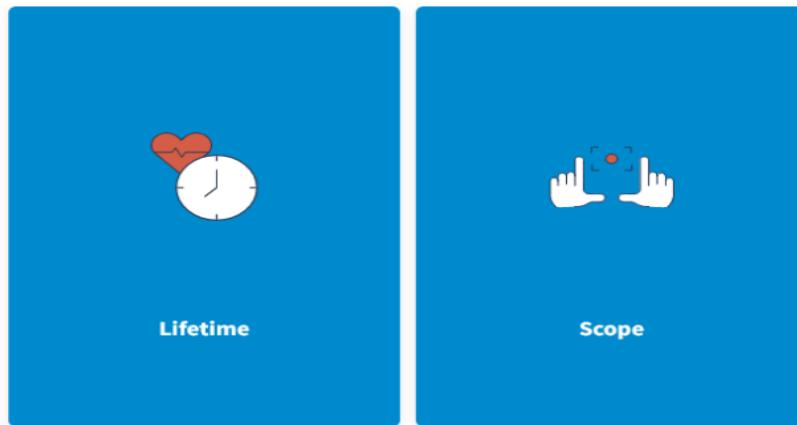
In this lesson, you will explore the concept of the code block and specifics of its implementation, scope, and life cycle of variables.

## Code Block and Scope of Variables

A **variable** is a named memory cell storing data (certain values). When developing, the developer has the opportunity to use any number of variables required to solve the task at hand. The main purpose of variables is to store interim program data and results.

Any variable declared in a program has two main properties.

*Click on the cards to learn more about the properties of variables.*



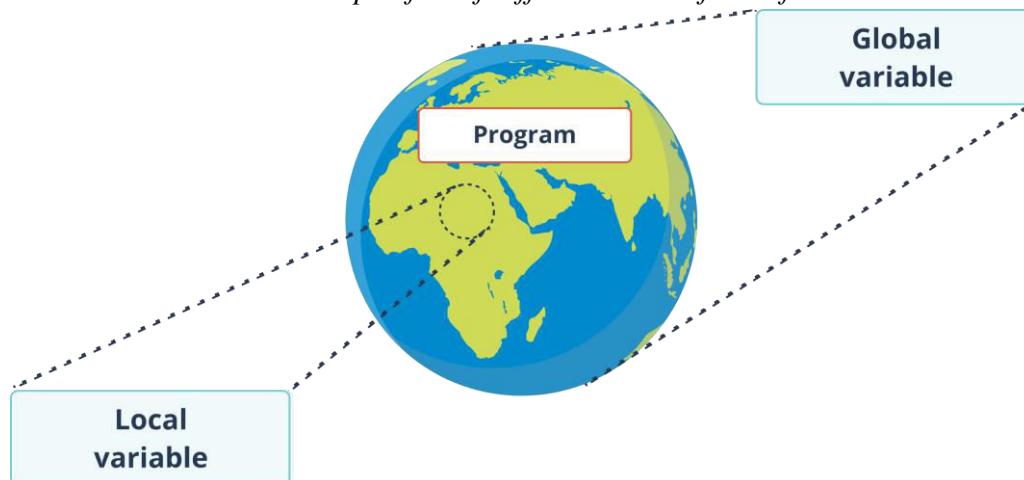
The time during which a variable exists.

The part of a program where a variable can be used.

The lifetime of a variable can be global and local. This differentiation depends on where in the program a variable is declared, for example:

- can be declared on the class level. In that case, they will be accessible in several methods.

*Click on the + signs to learn more about the specifics of differentiation of the lifetime.*



### Local variable

A **local variable** is a variable that is described in a block of code, and is created when the program execution enters a block and is destroyed as soon as the execution exits the block.

### Global variable

A **global variable** is a variable that is created when the program is launched for execution and lives until the program completes.

*A code block means a part of the program placed in curly brackets.*

The Java language is special in that variables cannot be declared outside of a code block. For example, the body of the method main, as well as the class body, is a code block. This means that the lifetime of variables is

related to the block where they are declared. In that case, the concept of a global variable will be considered based on its declaration in the class body but outside of the body of methods of this class. A local variable is treated as such when it is declared in the body of a class method.

*Click on the drop-down element to study the scope of variables.*

### Scope of a global variable

Since a simple program consists of one class, the scope of a global variable is the entire program.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
class Scope {
    static int x = -1;
    public static void main(String[] args) {
        System.out.println("x = " + x);
    }
}
```

### Scope of a local variable

The scope of a local variable begins at its declaration and extends to the end of the block. Characteristic property: local variables should be initialized before their first use.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
public static void main(String[] args) {
    int value = 100;
    {
        double valueD = 9.9;
        System.out.println("value = " + value);
        System.out.println("valueD = " + valueD);
    }
}
```

**The output from the program is:**

value = 100

valueD = 9.9

An example of a wrong use of a local variable:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
public static void main(String[] args) {
    short number = -100;
    {
        int valueInt = 2;
        System.out.println(number + ";" + valueInt);
    }
    System.out.println(number + ";" + valueInt);
}
```

*You cannot declare two or more variables with the same name (identifier) in the same code block.*

## Specifics of Application of code Blocks

Code blocks can be nested one within the other (the **membrane effect** is manifested):

- The scope of variables declared in the outer block passes into inner blocks.
- The scope of variables declared in the inner block does not extend to the outer block.

Pay attention to the specifics of application of code blocks.

*Click on the tabs to learn more.*

### NESTED CODEBLOCKS

Variables declared in the outer code block are also accessible in the nested/inner blocks.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
public static void main(String[] args) {
    int counter = 1;
    {
        int valueInt = 2 + counter++;
    }
}
```

```

        System.out.println("counter = " + counter + "; valueInt = " + valueInt);
    }
}

```

**The output from the program is:**

counter = 2; valueInt = 3

### INDEPENDENT CODEBLOCKS

You can declare two or more local variables with the same identifier in different code blocks.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```

public static void main(String[] args) {
{
    int x = 10;
    int y = -10;
    System.out.println("(" + x + ";" + y + ")");
}
{
    double x = 0.1;
    double y = -0.1;
    System.out.println("(" + x + ";" + y + ")");
}
}

```

**The output from the program is:**

(10; -10)

(0.1; -0.1)

## Conclusion

In this lesson, you found out that:

- A variable is a named memory cell storing data.
- A code block means a part of the program placed in braces.
- You cannot declare two or more variables with the same name in the same code block.
- Code blocks can be nested one within the other.

## Check Your Knowledge!

1. Which properties does any variable have?

**Lifetime**

Nesting

**Scope**

Variability

Answer Correct:

Any variable has two properties: lifetime and scope.

2. Which variable is created when the program execution enters a block and is destroyed as soon as the execution exits the block?

Local

Global

**Both variants (local and global)**

Answer

Correct:

Both variables (local and global) are created when program execution enters the block and is destroyed as soon as execution exits the block.

3. Is it true that the Java language has a feature that variables can be declared outside of a code block?

True

**False**

AnswerCorrect:

It is a feature of the Java language that variables cannot be declared outside of a code block.

# Conditional Statement

## Introduction

In the previous lesson, you were introduced to the concept of the code block. You found out that mostly code blocks are used in statements controlling the flow of program execution. In this lesson, you will start exploring these statements.

## Branching Statement if

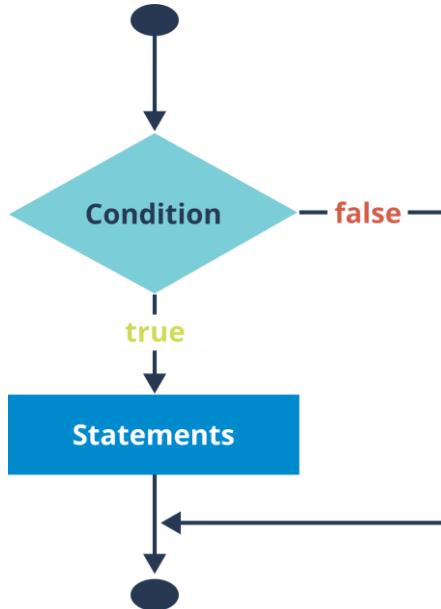
Often when trying to solve a problem, you need to introduce a condition for the execution of its solution. To do this, when writing programs in Java or other programming languages, you should use the branching statement **if**.

Consider a short form of the statement **if**. The statement **if-then** defines branching in the execution of program statements – it establishes a conditional expression whose result may only be "true" or "false":

- If the result is "true," the set of statements will be executed.
- If the result is "false," the set of statements will not be executed, it will be skipped.

Pay attention to the syntax of the short form of the statement **if**.

*Click on the drop-down elements to study the syntax and example.*



## SYNTAX

```
if (conditional expression) {
    /* statements */
}
```

## EXAMPLE

Please review a part of an algorithm of going to work:

```
if (it_is_raining) {
    take_umbrella
}
```

Or, if the value of x is negative:

```
if (x < 0) {
    x = 0;
}
```

## Branching Statements if-then-else

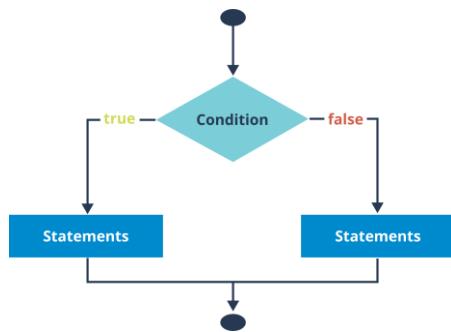
Explore the long form of the statement **if** the form **if-then-else**.

The statement **if-then-else** allows you to also define a set of statements that will be executed if the condition expression evaluates to "false":

- If the result is "true," one set of statements will be executed.
- If the result is "false," a different set of statements will be executed.

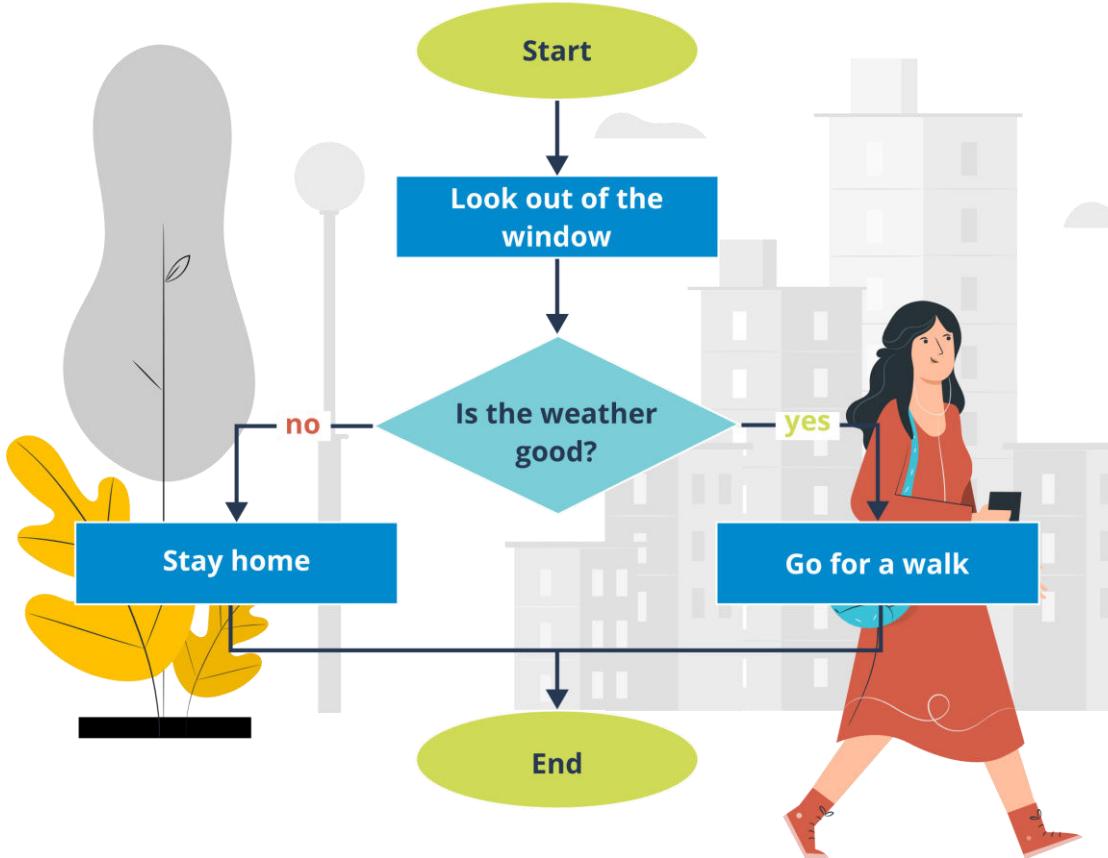
Pay attention to the syntax of the long form of the statement **if**.

*Click on the drop-down elements to study the syntax and example.*

**SYNTAX**

```

if (conditional expression) {
    /* statements */
} else {
    /* statements */
}
  
```

**EX****Branching algorithm “Go for a walk”**

Below is a part of an algorithm for a weekend:

```

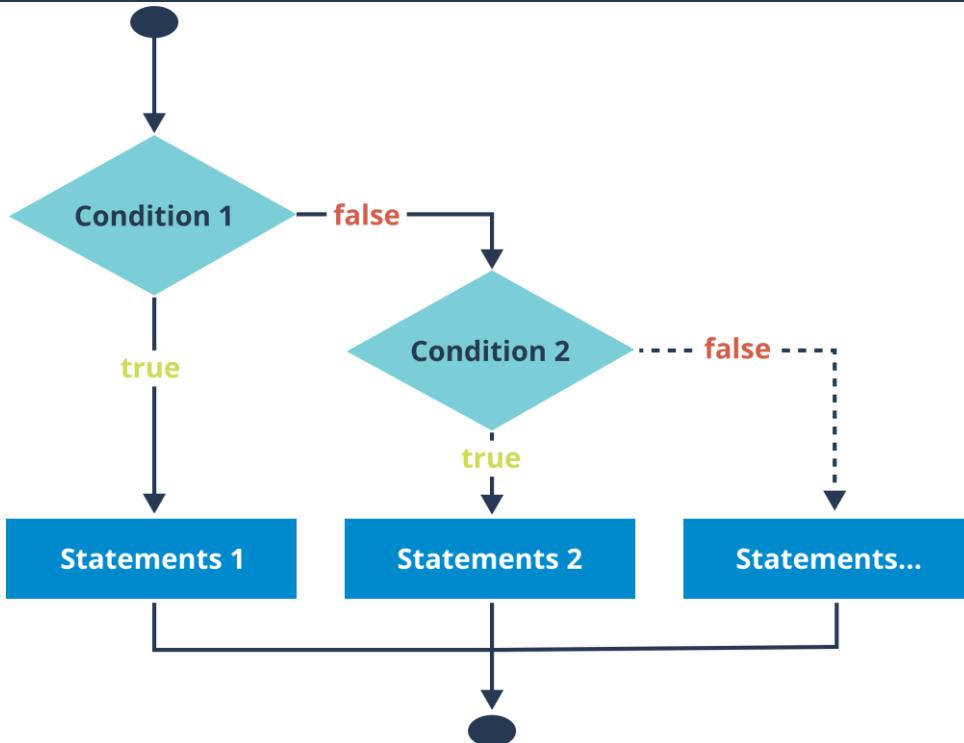
if (good_weather) {
    go_for_a_walk
} else {
    stayat_home
}
  
```

Or, determining the larger of the two values (a and b):

```

if (a > b) {
    max = a;
} else {
    max = b;
}
  
```

## Chains of Statements if—else-if—else



One statement "if-then-else" makes it possible to use another statement "if-then-else" and is used to combine **else** and **if** to check a whole range of mutually exclusive options. When using a chain of statements **if-then-else-if**, you should consider the following:

- The last specified **if** statement has an optional else branch.
- The sequence of **else-if** branches is not limited from the viewpoint of the statement.
- As soon as one of the **else-if** conditions returns "true," the rest are not checked.

Pay attention to the syntax of a chain of statements.

### SYNTAX

```

if (conditional expression1) {
    /* statements */
} else if (conditional expression2) {
    /* statements */
} else if (conditional expression3) {
    /* statements */
} else {
    /* statements */
}
  
```

### EXAMPLE

Pay attention to the example of determining the season by the current month:

```

if (month 1, 2 or 12) {
    it_is_winter
} else if (month 3, 4 or 5) {
    it_is_spring
} else if (month 6, 7 or 8) {
    it_is_summer

} else if (month 9, 10 or 11) {
    it_is_fall
} else {
    the_month_does_not_exist
}
  
```

Or, setting a discount for an airplane ticket:

```

int countDays = 75; // number of days before the flight
if (countDays >= 90) {
  
```

```

percent = 50;
} else if (countDays >= 60) {
    percent = 30;
} else if (countDays >= 30) {
    percent = 10;
} else {
    percent = 0;
}
}

```

## Conclusion

In this lesson, you have explored:

- The syntax of the short and long form of the if statement
- The syntax for a chain of statements
- An example of the construction if-then-else-if

## Check Your Knowledge!

1. Will this code be compiled without errors?

```

int i = 0;
if (i) {
    System.out.println("Hello");
}

```

Yes, it will compile without errors

### No, it won't compile

Answer

Correct:

The conditional expression in the if statement can only have a value of the type boolean or Boolean.

2. Will this code be compiled without errors?

```

boolean b = true;
boolean b2 = true;
if (b == b2) {
    System.out.println("So true");
}

```

### Without errors

With errors

Answer

Correct:

Values of the type boolean can be compared for equivalence.

3. Will this code be compiled without errors?

```

int i = 1;
int j = 2;
if (i==1 || j==2)
    System.out.println("OK");

```

Without errors correct

With errors

Answer

Correct:

Integer values can be compared for equivalence with literal expressions of their type. The results of comparison as boolean values can also become operands of a logical operator.

# Selection Statement

## Introduction

In this lesson, you will continue to study control flow statements and explore the statement **switch**.

## Statement switch

The statement **switch** is similar to the branching statement **if-else**. It transfers control to one of several statements depending on the value of an expression. But, unlike **if-then** and **if-then-else** statements, the **switch** statement can have a number of possible execution paths.

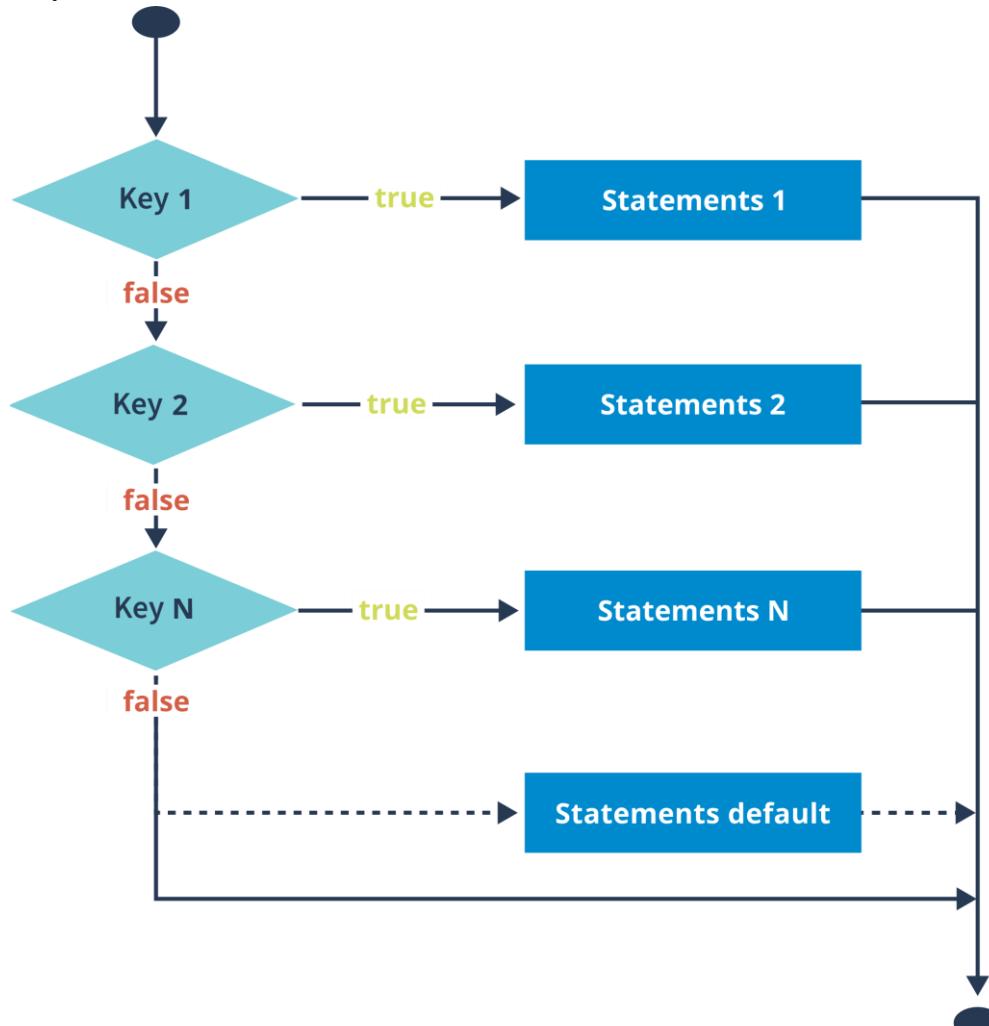
What are the nuances when working with the statement **switch**?

1. First, the value of the "expression" is calculated.
2. Then the value of the expression is successively compared with the values of keys in the branches of the statement **switch**.
3. When these values coincide, the program's execution is switched to the block of statements of the relevant branch.
4. Then the statement **switch** has to be exited. The statement **break** is used for this.
5. If no key has matched the expression value, then the statement is exited or the code block of the **default** branch is executed (if the branch is present, in other words, it is not mandatory).

It is also important to understand that the key values should:

- Have the same type as that of the "expression"
- Be a literal or final variables
- Have the type int, byte, short, char. It also works with an Enums and the String type, and a few special wrapper classes: Character, Byte, Short, and Integer

Pay attention to the syntax of the statement **switch**.



```

case key1:
    /* statements if the expression value matches key1 */
    break;
case key2:
    /* statements if the expression value matches key2 */
    break;
...
default:
    /* statements */
}

```

As you will see further, in some cases using the statement **switch** gives a more concise program code compared to the statement **if-then-else**. For example, if you have a structure with a chain of statements **if-then-else** implying a large number of conditional branching, they can be very bulky. Therefore, in cases when the chain of statements **if-then-else-if** checks the value of the same variable, there is a different solution – the statement **switch**.

In the next video, you can see an example of how to use this statement, as well as different execution options depending on the presence/absence of the statement **break** in the case branches.

Note that in Java 14, new forms of the statement **switch** were added that:

- Do not require the statement **break**
- Allow you to combine labels through commas for the same actions
- Apply a special expression syntax
- Allow you to use the statement in the form of a method

Pay attention to the following examples.

## SYNTAX OF EXPRESSIONS IN SWITCH

The statement **switch** can return a value. Needless to say, all **case subexpressions** have to return values of one type or those that can be reduced to one type. Instead of the ":" character and **return**, it is easier to use a special expression syntax. The colon is replaced with "->", and to the right of it there should be a value or expression that can return a value. When one of the **cases** is triggered, the value corresponding to it is calculated and the resulting value is returned as a result of the entire **switch** block.

```

public int defineLevel(String role) {
    return switch (role) {
        case "guest" -> 0;
        case "client" -> 1;
        case "moderator" -> 2;
        case "admin" -> 3;
        default -> {
            System.out.println("non-authentic role = " + role);
            yield -1;
        }
    };
}

```

## DIRECT ASSIGNMENT OF THE SWIITCH STATEMENT

When you have a code inside the **switch** block and you need to return a value from that code, you should use a new keyword **yield**. The returned value may be assigned to the result via direct assignment:

```

public int defineLevel(String role) {
    var result = switch (role) {
        case "guest" -> 0;
        case "client" -> 1;
        case "moderator" -> 2;
        case "admin" -> 3;
        default -> {
            System.out.println("non-authentic role = " + role);
            yield -1;
        }
    };
}

```

```

    return result;
}

```

## LIST OF KEYS IN THE STATEMENT SWITCH

Instead of several case branches, there is now a possibility to specify **one branch by combining keys separated by commas**:

```

int value = 1;
switch (value) {
    case 1, 2, 3, 4 -> System.out.println("1, 2, 3 or 4");
    case 777 -> System.out.println("Range: " + value);
    case 0 -> System.out.println("0");
    default -> System.out.println("Default");
}

```

## STATEMENT SWITCH AS A METHOD

The returned value of the **switch** statement can also be used in **case** expressions:

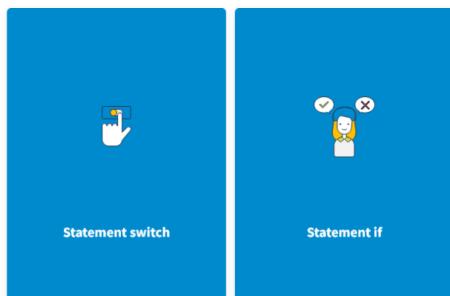
```

System.out.println(
    switch (value) {
        case 2, 3, 4 -> "2,3 or 4";
        case 777 -> "Range: " + value;
        case 0 -> "0";
        default -> "Default";
    });

```

Using the statement **switch** has several special features as compared to the statement **if**.

*Click on the cards to learn more about the features.*



- To switch to the statement branch, the statement only checks whether the expression value matches the branch key value.
- The expressions and keys may be enums, strings or integers, with the exception long.

- To switch to the branch, the conditional expression may have any complexity and produce only logical type values.
- The data participating in the calculation of the expression may be of any type.

Note that when the statement **switch** is used, then in two different branches the keys cannot have the same value. The keys may have the same values only when these branches are placed on different nesting levels (in case of nested statements **switch**).

You can learn more about the statement **switch** in the [official documentation](#).

## Conclusion

In this lesson, you found out that:

- The statement switch is similar to the branching statement if-then-else. It also allows you to organize branching of the program execution process.
- The statement switch may be applied only to a known number of possible situations.
- When the statement switch is used, then in two different branches the keys cannot have the same value.

## Check Your Knowledge!

1. Choose the correct statement.

The statement switch should always have a default branch

The statement switch should have only one case branch for each code segment

In the statement switch, the default branch is specified after all case branches

**A symbolic literal expression may be used as a key in the case branch correct**

Correct:

Q A symbolic literal expression may be used as a key in the case branch.

Q 2. What is the result of running the following code?

```

String day = new String("SAT");

```

```
switch (day) {  
    case "MON":  
    case "TUE":  
    case "WED":  
    case "THU": System.out.println("Time to work");  
        break;  
    case "FRI": System.out.println("Nearing weekend");  
        break;  
    case "SAT":  
    case "SUN": System.out.println("Weekend!");  
        break;  
    default: System.out.println("Invalid day?");  
}
```

Time to work

Nearing weekend

**Weekend!**

Invalid day?

Answer

Correct:

The case "SAT" does not have the statement break; therefore, the case "SUN" will also be executed, which will lead to an output of the phrase Weekend

# Loops

## Introduction

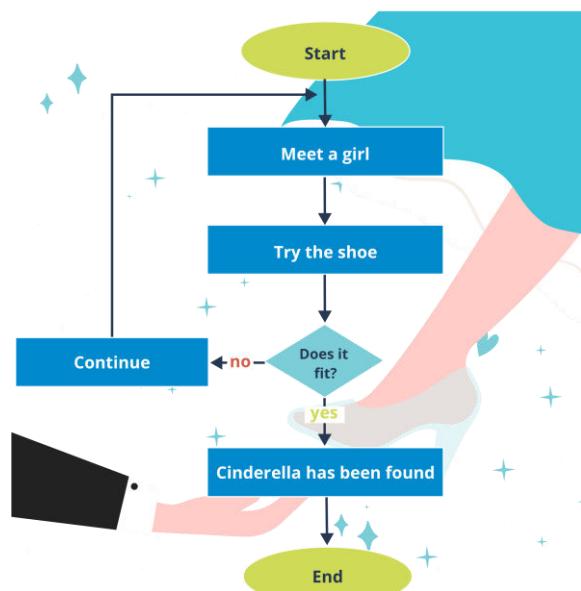
In this lesson, you will continue exploring control flow statements. You will study repetition statements (while, do-while), branching statements (continue and break), and the looping statement for. After learning about these statements, you will be able to implement the logic of a program.

## The while and do-while Statements

When implementing the logic of a program, very often you need to execute one and the same sequence of actions (statements) many times. At that, the number of repetitions might not be known in advance and depends on certain data (entered by the user, received interim results, etc.). To enable the repetition of actions, the Java programming language has **looping statements**. The sequence of statements that have to be executed a specified number of times is called an **iteration**.

There exist the following looping statements:

- **while** — with a pre-condition
- **do-while** — with a post-condition



The execution of a **while** loop can be described as follows:

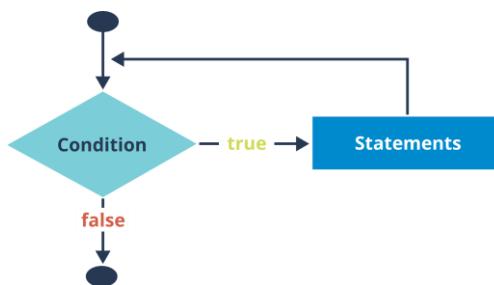
1. One or several variables used in the "conditional expression" are initialized before the looping statement (such variables are called *iterative*).
2. The "conditional expression" is calculated, and if its value is "true," the loop body is executed; otherwise, the program execution does not enter the loop body.
3. After that, in the body of the **while** statement you need to change the values of one or several variables that are part of the "conditional expressions" (the "conditional expression" should get the "false" value to end the loop).
4. If the value of the condition does not change, you will get an endless loop. To exit such a loop, you need to use the statement **break**.

### Syntax

### Example

While the conditional expression is "true", the statements of the loop body will be repeated.

```
while (conditional expression) {
  /*statements*/
}
```

**Example**

While the value of the iteration variable i does not exceed 5, the loop will be executed.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```

class TestWhile {
    public static void main(String[] args) {
        int i = 0;
        while (i < 5) {
            System.out.println("Iteration: " + i);
            i++;
        }
    }
}
  
```

**The output from the program is:**

Iteration: 0

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Now you are going to review the description, syntax, and example of application of the looping statement **do-while**.

The description of the **do-while** loop execution looks as follows:

1. One or several iterative variables are initialized before the looping statement.
2. The loop body is executed where the values of one or several iterative variables have to be changed.
3. The "conditional expression" is calculated and if its value is "true," the loop body is executed again. Otherwise, the loop is exited. Unlike the loop **while**, this loop will be executed at least once, while the body of the **while** loop might not be executed at all.

Syntax	Example
<b>Syntax</b>	<b>Example</b>

**Syntax**

```

do {
    /*statements*/
}while (conditional expression);
  
```

**Example**

An example of the **do-while** loop that will be executed only once:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```

class TestDoWhile {
    public static void main(String[] args) {
        int i = 0;
        do {
            System.out.println("Iteration: " + i);
            i++;
        } while (i > 5 && i < 10);
    }
}
  
```

```
}
```

**The output from the program is:**

Iteration: 0

It is worth noting that using the statements **while** and **do-while** is useful when it is necessary to repeat certain actions and the number of repetitions is unknown and depends on a certain condition. The main difference between the statements **while** and **do-while** is that the body of the loop **while** might not be executed at all, while the body of the loop **do-while** will always be executed at least once.

It is possible to create infinite loops, if the "conditional expressions" in these loops always have the value **true**. To exit such a loop, you can use the branching statement **break** that will be discussed further.

## Branching Statements break and continue

**Branching statements** are statements that make it possible to go from the point where the statement is used to a different part of the program. Branching statements:

- **break** is used to exit a loop or the statement **switch**.
- **continue** is used to go to the next loop iteration.

### ^ Statement break

The

statement **break** is used to exit the body of a loop and go to the statement following this loop. Moreover, it is used to complete the execution of a group of branching **case** statements in the statement **switch** and to go to the statement following this statement.

In other words, the statement **break** stops the execution of statements of the current code block and resumes the execution of the program after that block. For example, an endless loop is created and it is exited when the following situation occurs: the variable *i* receives a value exceeding 6.

```
public static void main(String[] args) {
    int i = 0;
    while (true) {
        if (i > 6) {
            break;
        }
        System.out.println(i++);
    }
}
```

**The output from the program is:**

```
0
1
2
4
5
6
```

### ^ Statement continue

The statement **continue** is most often used in looping statements. It assures a transition to the next loop iteration without completing the execution of the remaining statements of the current loop iteration. For example, you create a loop for 10 iterations that checks whether the variable *i* is a multiple of 4 or 8: if the variable *i* is not a multiple of these values, then its value is printed. Otherwise, this action is skipped for the current iteration.

```
public static void main(String[] args) {
    int i = 0;
    while (i++ < 10) {
        if (i == 4 || i == 8) {
            continue;
        }
        System.out.println(i);
    }
}
```

```

    }
}

```

**The output from the program is:**

```

1
2
3
5
6
7
9
10

```

## Statements break and continue with a Label

The statements **break** and **continue** in the Java language are special because they can be used with a label.

A **label** is a legal identifier that can be used to mark looping statements. In that case, the label is separated from the statement with a colon. The statements **break** and **continue** with a label can be used only when the program has nested loops.

The statement **break** with a label is used in nested loops and exits the loop that is marked with the specified label.

### Example

In the example, two loops are created, one is nested inside the other one. When the execution of the program reaches the statement “break outer;”, then both inner and outer loops are exited.

```

public static void main(String[] args) {
    int i = 0;
    outer: while (i < 5) {
        int j = 0;
        while (j < 2) {
            j++;
            System.out.println("i=" + i + ",j=" + j);
            break outer;
        }
    }
}

```

**The output from the program is:**

i = 0; j = 1

The statement **continue** with a label is used in nested loops. This statement breaks the execution of the current iteration of the nested loop and goes to the next iteration of the outer loop marked with the specified label.

*Click on the drop-down element to study the example.*

### Example

In the example, two loops are created, one is nested inside the other one. When the variable j reaches a value of 2, the nested loop is exited and the next iteration of the outer loop starts.

```

public static void main(String[] args) {
    int i = 0;
    outer: while (i++ < 3) {
        int j = 0;
        while (j++ < 5) {
            if (j == 2) {
                continue outer;
            }
            System.out.println("i = " + i + "; j = " + j);
        }
    }
}

```

**The output from the program is:**

i = 1; j = 1

```
i = 2; j = 1
i = 3; j = 1
```

The benefit of using **break** and **continue** is that, in case of a looped process, there is no need to execute extra iterations, if the target value has been found or the target result has been reached. This increases the speed of execution of the program, and the code structure is not deformed in case of an early termination of the loop process.

## Looping Statement for

The **for** statement is the most often used loop in programming. It is convenient to use when the number of iterations is known. Pay attention to the syntax, specifics, and use case for the **for** statement.

*Click on the drop-down elements to learn more.*

### Syntax

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
for (exp1; boolean exp2;exp3;) {
    /*statements*/
}
```

Example:

```
for (int i = 1; i <= 10; i++) {
    /*statements*/
}
```

 The variable **i** is an iteration variable (loop variable). Its value determines the condition for exiting the loop. After each iteration, the value of **i** increases by 1.

### Specifics

Note the following specifics of the **for** statement:



- The loop variable should be declared in the loop parameters if it is not needed outside of the statement.
- The rule for changing the loop variable starts working only after the first loop iteration.
- Completion conditions:
  1. If the expression's result is "true," then the program executes the loop body.
  2. If the expression's result is "false," then the program terminates the loop.
  3. If the first check of the condition results in a "false" value, the loop will not be executed at all.

### Example

Pay attention to the summation of the current value of the loop variable is performed in the loop.

```
int sum = 0;
for (int val = 1; val <= 10; val++) {
    sum += val;
}
System.out.println("Sum of numbers 1..10 = " + sum);
```

**The output from the program is:**

Sum of numbers 1..10 = 55

The syntax of the looping statement **for** is not limited by a loop with an only iteration variable. Both in the "initialization" expression and in the "change rule" expression, you can use a **comma** to separate several variables:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
for (exp1, exp2, ..., expN; boolexp; exp10, exp11, ..., expM;) {
    /*statements*/
}
```

**Example**

Pay attention to the following example. Two variables (*a* and *b*) are initialized in the loop parameters. The number of loop iterations depends on their values. After each iteration, the values of the *a* and *b* variables change. When their values cross, the loop will be terminated.

*Click on the drop-down element to study the example.*

**int** a, b;

```
for (a = 1, b = 4; a < b; a++, b--) {
    System.out.println("\n Iteration initialization");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("Iteration completion");
}
```

**The output from the program is:**

Iteration initialization

a = 1

b = 4

Iteration completion

Iteration initialization

a = 2

b = 3

Iteration completion

Pay attention to the examples of working with one and several iteration variables, as well as with and without the statements **break** and **continue**.

## Conclusion

In this lesson, you found out that:

- Using the statements **while** and **do-while** is useful when it is necessary to repeat certain actions and the number of repetitions is unknown and depends on a certain condition.
- Branching statements (**break**, **continue**) are statements that make it possible to go from the point where the statement is used to a different part of the program.
- It is convenient to use the **for** loop when the number of iterations is known.

## Check Your Knowledge!

1. Which of the below termination conditions are true for the **for** statement?

**If the expression's result is "true," then the program executes the loop body**

**If the expression's result is "false," then the program terminates the loop**

If the first check of the condition results in a "false" value, the loop will be executed once

correct

Answer

Correct:

One of the conditions contains a mistake: if the first check of the condition results in a "false" value, the loop will not be executed at all.

2. What is the result of running the following code?

```
int num = 10;
while (++num > 20) {
    num++;
}
System.out.println(num);
```

13

10

Answer

Correct:

The loop condition will compare `11>20`. In the result of comparison, the loop will not be entered and after the loop, the number 11 will be printed.

3. Is it true that the body of the statement **while** might never be executed, while the body of the statement **do-while** will always be executed at least once?

**True**

False

Answer

Correct:

The main difference between the statements while and do-while is that the body of the loop while might not be executed at all, while the body of the loop do-while will always be executed at least once.

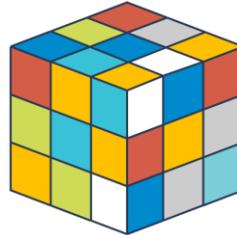
# What is an Array?

## Introduction

Nowadays, people deal with huge amounts of data. And to make it easier to work with, we need to be able to group data and ensure quick access to it. In this lesson, you will find out how to solve this problem in Java. You will study arrays, what they consist of, and how to create them.

## The Concept of Arrays

An **array** is a set of homogeneous data with a common name. Arrays help to group related data and may be used for different purposes. For example, in an array you can store data about monthly precipitation levels or the titles of books in your library. Arrays may be one-dimensional or multidimensional (but one-dimensional arrays are used more often).



The **key benefit of an array** is the possibility of quick access to its elements, which makes it easy and convenient to process data. For example, if you have an array of data about a student's marks in a specific subject, you can easily calculate the mean result. To do this, you need to iterate through the array elements in loops.

In Java, arrays are used in almost the same way as in other programming languages. However, there is one special feature: **in Java, arrays are implemented as objects**. This approach frees up the memory they take through a "garbage collector" when the arrays are no longer used. It also allows an array to store information about its length and prevent access to elements outside of its index space.

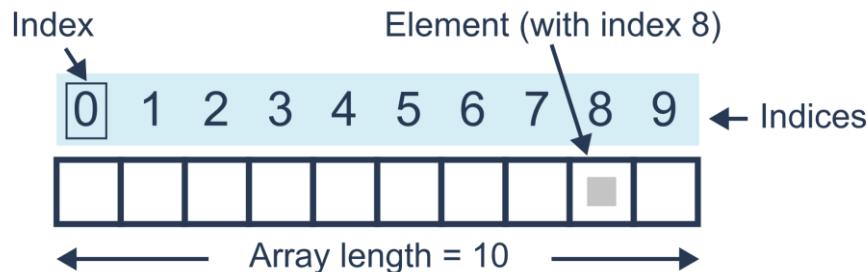
Review the different types of arrays and how they are used.

## One-Dimensional Arrays

An **array in Java** is a **container object** that has a fixed number of cells for storing data/values of one type. In other words, it is a numbered set of data with a common name. This is convenient. For example, instead of declaring 1000 variables of one type containing the last names of the company's developers, it is much easier to declare one array containing all the last names.

Graphically, you can represent an array as follows.

*Click on the active elements to learn more about the components of an array.*



### Index

Indexation of array elements **starts from zero**.

### Element

A piece of data in an array is called **an element**.

### Array length

The size of an array is defined and fixed when it is initialized. After an array has been initialized, you cannot change its length.

### Indices

Every piece of data has its own position — **the index**.

Thus, arrays consist of elements with indices. Further in this lesson, you will learn how to create arrays in Java.

To create an array, you need to:

1. Declare an array
2. Allocate memory for the array
3. Initialize the array

Explore each stage in detail.

## DECLARING AN ARRAY

Declaring an array means specifying the type of elements it contains and assigning it a name. This can be done in two ways:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

**First method:** <data type><identifier>[];

**Second method:** <data type>[] <identifier>;

The second method is recommended. Square brackets mean that you are dealing with a set of data and not a set of data names. Thus, it is logical to specify them next to the type. In this context, square brackets mean that the data belong to the **Arrays** class.

 Arrays can contain elements of primitive data types and reference types. When an array is declared, only a reference to the array is created; no memory is allocated for the array.

## ALLOCATING MEMORY FOR THE ARRAY

Allocating memory means reserving a fixed-size block of memory in the "heap" to store array elements. The key word **new** is used:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

<identifier> = **new** <data type>[size];

For example, you need to allocate 20 bytes (5 x 4 bytes) of memory and place references to this memory area in the array:

array = **new** int[5];

In other words, an array of five elements of the type **int** was created.

## INITIALIZING THE ARRAY

Initializing an array means writing data in the array's cells. Initialization can be done in two ways:

- By accessing each array element and specifying its index one by one
- By iterating through the elements using looping

Right after an array is initialized, it is filled with default values. Usually these are 0 for numeric types, false for boolean, and null for reference types.

Look at each initialization method in more detail.

**First method:** To access an array element, you need to specify the array's name and the index as a literal expression in square brackets — a nonnegative integer.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

int[] array = **new** int[3];

array[0] = 10;

array[1] = 20;

array[2] = 30;

**Second method:** To access an array element, you need to organize looping, where the number of iterations matches the index of array elements. Specify the name of the array in the body of the loop and the iteration variable in square brackets.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

int[] numberArray = **new** int[10];

int i = 0;

**while** (i < 10) {

    numberArray[i] = i;

```

    i++;
}
i = 0;
while (i < 10) {
    System.out.println((i+1) + "th array element = " + numberArray[i]);
    i++;
}

```

Initializing an array means writing data in the array's cells. Initialization can be done in two ways:

- By accessing each array element and specifying its index one by one
- By iterating through the elements using looping

Right after an array is initialized, it is filled with default values. Usually these are 0 for numeric types, false for boolean, and null for reference types.

Look at each initialization method in more detail.

**First method:** To access an array element, you need to specify the array's name and the index as a literal expression in square brackets — a nonnegative integer.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```

int[] array = new int[3];
array[0] = 10;
array[1] = 20;
array[2] = 30;

```

**Second method:** To access an array element, you need to organize looping, where the number of iterations matches the index of array elements. Specify the name of the array in the body of the loop and the iteration variable in square brackets.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```

int[] numberArray = new int[10];
int i = 0;
while (i < 10) {
    numberArray[i] = i;
    i++;
}
i = 0;
while (i < 10) {
    System.out.println((i+1) + "th array element = " + numberArray[i]);
    i++;
}

```

## Array Initializers

In Java, you can take advantage of a short form of the array definition. This form allows for consolidating the steps used during the standard array creation process. For instance, when you know the initial values of the array elements and the array is small, it may be initialized during declaration:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
<data type>[] <identifier> = {<literal expression>, ...};
```

In the case of using array initializers:

- The keyword **new** is absent.
- The initial values of the array elements are listed in curved brackets as literal expressions.
- The length (size) of the array is determined by the number of literal expressions listed.
- When a zero-dimension array is created, the curved brackets are left empty.

For example:

```
int[] color = {255, 126, 255};
```

The example above shows how to create an array with a length of three elements that receive the specified values.

The reference **color** receives the memory mapping address for this array.

This initialization method is only used together with the expression describing the reference to the array.

## Anonymous Array

You can create an array without specifying its name and length, that is, you can consolidate the process of memory allocation and initialization.

```
new <type>[] { <literal expression> , ... };
```



When using this method, an array will be created whose length depends on the number of literal expressions listed. A reference is returned that may be:

- Assigned to a variable
- Passed to a method as a parameter
- Returned from a method

Anonymous arrays are used when an array is created in one place and then needs to be transferred to another part of the program for processing. For example:

```
new int[] { 3, 5, 2, 8, 6 };
```

The example above demonstrates creating an array of integer-valued data with a length of five elements and getting a reference to it. Consider an example to see how feasible this approach is for initializing an array.

```
int[] daysInMonth;
daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
daysInMonth = new int[]{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

## Conclusion

This lesson introduced you to the concept of an array. You found out:

- An array is a set of homogeneous data under a common name in Java in the form of a container object.
- The data in an array is called elements, each of which has its own index and position.
- To create an array, you should declare it, allocate memory for it, and initialize it.
- If the array is small and the initial values of the array elements are known, it may be initialized during declaration.
- You can create an anonymous array, i.e., an array whose name and length are not specified.

## Check Your Knowledge!

1. When a programmer specifies the type of array elements and the name of the array, they are:

**declaring the array**

allocating memory for the array

initializing the array

creating the array

Answer

Correct:

Declaring an array means specifying the type of elements in it and assigning it a name.

2. Is it possible to initialize an array in Java when declaring it?

Yes, anonymous arrays are used for this.

**Yes, a short form of the array description is used for this.**

Yes, but this is not recommended.

When you know the initial values of the array elements and the array is small, it may be initialized during declaration. A short form of the array description is used for this.

3. Which expression is incorrect?

```
String[ ] names[ ];  
int numbers [] = new int[2] {10, 20};  
float[ ] f1[ ], f2;  
int[] scores = {1, 2, 3, 4};
```

Answer

Correct:

Only the declaration of the numbers array is incorrect. The length is declared twice: in square brackets and through the actual number of elements in curly brackets. It does not matter that the length values match.

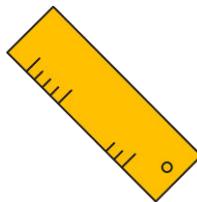
# Length and Copying of Arrays

## Introduction

In this lesson, you will find out how to determine the length of arrays, copy arrays, and whether you can change their length.

## The Length Property

You may have noticed that when you access an array, its length is not always evident. Thus, the length of an array (or the number of elements) is stored in its properties—the *length* field. You can access this field through the array's name. For example:



```
int[] numbers = {-9, 6, 0, -59};  
int num = numbers.length;  
System.out.println(num);
```

### Output:

4

*Use the length property to control access to an array's elements. This helps avoid going beyond its boundaries.*

If you try accessing an element of an array that goes beyond its range [0, *arrayName.length-1*], Java will throw the error **ArrayIndexOutOfBoundsException** when executing the program. For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
int[] arr4 = {11, 22, 33, 44, 55, 66};  
System.out.println(arr4[1]);  
System.out.println(arr4[2]);  
// ....  
System.out.println(arr4[6]);
```

You can find more information on arrays in the [official documentation](#).

## Conclusion

In this lesson, you:

- Found out that information about the amount of elements an array has is stored in the array's length property
- Copied part of one array into another array using the *System.arraycopy()* method
- Changed the length of an array by reinitializing the reference to it

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

1/1 point

1. It is necessary to iterate through all array elements and stop the iteration of arrays when the last element is reached. Which of the following lines of code can help with this?

array.length(); **array.length;** array.size;  
array.size();

 the array's length is stored in the length constant of the array class. This is the easiest way to determine its length.

# Looping Statements for and for-each. Working with Array Elements

## Introduction

In this lesson, you will explore structured statements that help process array elements. They are called **for** and **for-each** looping statements. You already explored the **for** statement in the previous module. Now you will review the syntax, specifics, and some application examples for the **for-each** statement, as well as the main pros and cons of each statement.

## The for-each Looping Statement

You already know that the **for** loop is the most used loop in programming. It is convenient to use when the number of iterations is known.

The **for** statement can be used to iterate over the array since:

- It is a compact way of describing the **while** statement
- Loop control is not part of the loop's body

Below are some examples of creating and using arrays.

*Click on the tabs to learn more.*

### EXAMPLE OF CREATING AN ARRAY

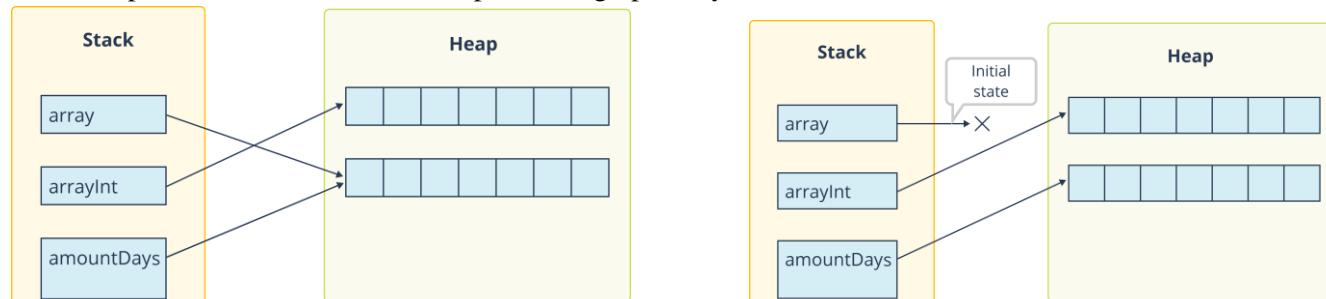
Creating an array implies the following:

- Three references are created for arrays of the **int** type, but only two of them are initialized: *arrayInt* and *amountDays*.
- A loop is organized to copy all elements of the array *amountDays* to the array *arrayInt*.
- Before copying the current element of the array *amountDays*, its value is changed if the value is less than 31.
- After the loop is completed, the *Array* reference is initialized. It will also refer to the *amountDays*.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
class ArrayMain {
    public static void main(String[] args) {
        int[] array;
        int[] arrayInt = new int[100];
        int[] amountDays = { 31, 28, 31, 30, 31, 30, 31, 31, 31, 30, 31, 30, 31 };
        System.out.println("April has " + amountDays[3] + " days");
        for (int i = 0; i < amountDays.length; i++) {
            if (amountDays[i] < 31) {
                amountDays[i] = 0;
            }
            arrayInt[i] = amountDays[i];
        }
        array = amountDays;
    }
}
```

The example described above can be represented graphically as follows:



## SYNTAX

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
for (<type> <iteration variable>: <array>) {
    /*statements*/
}
```

}

*The type of iteration variable should match the type of array elements.*

## SPECIFICS

During each loop iteration, a copy of the value of an array element is saved in its iterative variable (the loop is executed until all the elements of the dataset have been processed). However, it is possible to terminate the **for-each** loop early by using the **break** statement.



Advantages of the **for-each** loop:

- It simplifies the statement syntax.
- It rules out the possibility of going beyond the array when iterating through it.
- It is used exclusively to iterate through the values and not for some other operations (for example, to delete or edit array elements).

## EXAMPLE OF ARRAY OUTPUT

Below you can see a code fragment displaying the array in the console. Since this operation does not change the array elements, to iterate through it, you can use the **for-each** loop:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
int[] arrayInt = { 1, 2, 3, 4, 5, 6 };
for (int valueInt : arrayInt) {
    System.out.println(valueInt);
}
```

## EXAMPLE OF CHANGING ARRAY ELEMENTS

Below you can see a code fragment that involves changing the array elements. Since the **for-each** loop is used to iterate through the array, the array elements will not change. In this case, only the value of the *element* variable changes:

```
int[] arrayInt = { 1, 2, 3, 4, 5, 6 };
for (int element : arrayInt) {
    element *= element;
}
for (int valueInt : arrayInt) {
    System.out.println(valueInt);
}
```

### Output:

```
1
2
3
4
5
6
```

The changes work only for reference variable data types.



There are certain specifics when processing arrays using the **for-each** loop. You will see similar features when using the **for-each** loop with collections.

## The for-each loop and arrays

What are the advantages and disadvantages of the **for-each** loop as compared to the **for** loop?

*Click on the cards to learn more.*

### ADVANTAGES

The **for-each** loop has the following advantages:

- simplified representation,
- no need to introduce an additional loop variable or set its initial value and conditions for loop termination,
- no need to index the array.

### DISADVANTGES

The **for-each** loop has the following disadvantages:

- lack of flexibility when manipulating the iteration variable,
- the inability to change the value of the array elements (for primitive and immutable reference data types).

You have studied the syntax, specifics, and pros and cons of the **for-each** loop. In the next chapter, you will explore in detail how the **for** and **for-each** loops are used.

## Using the for and for-each Loops

The following example will help you understand how you should use the **for** and **for-each** loops.

**Problem:** Find the maximum value in the array and replace the negative elements in this array with this value.

What solution would you propose?



*Click on the drop-down elements to see the solution and the code.*

### SOLUTION

The solution is as follows:

- First, you need to iterate through the array and find the maximum value. Since you do not change the array in this case, it is better to use the **for-each**.
- Next, you need to change the elements in the array. It is correct to use the ordinary **for** loop since you need to access the array elements by index.

### CODE AND CONSOLE OUTPUT

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
int[] array = { 5, 10, 0, -5, 16, -2 };
int max = array[0];
for (int value : array) {
    if (max < value) {
        max = value;
    }
}
for (int i = 0; i < array.length; i++) {
    if (array[i] < 0) {
        array[i] = max;
    }
}
System.out.println("array[" + i + "] = " + array[i]);
```

**Output:**

```
array[0]= 5
array[1]= 10
array[2]= 0
array[3]= 16
array[4]= 16
array[5]= 16
```

Now that you have studied some use cases for the **for** and **for-each** loops, you can use them to solve your own problems.

**Conclusion**

In this lesson, you have explored the loop statements **for** and **for-each**, and found out that:

- The **for** loop is convenient when the number of iterations is known.
- The **for-each** loop is designed for strict sequential execution of loop statements for all elements of a dataset.

**Check Your Knowledge!**

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. What is the result of running the following code?

```
String[] strArray = new String[] {"One", "Two", "Three"};
```

```
strArray[2] = null;
```

```
for (String val : strArray)
```

```
System.out.print(val + ", ");
```

One, Two, Three,

One, null, Three,

**One, Two, null**

Compilation error

Answer

Correct:

Indexation of arrays starts from zero. The element at index 2 was replaced and will be displayed as the third one in the output line.

2. Which statements about the **for-each** loop are true?

The for-each loop may be used to initialize an array and change its elements.

**The for-each loop is not limited in terms of loop nesting.**

**The for-each loop cannot be used to iterate through several arrays in one loop.**

correct

Answer

Correct:

The for-each loop is not limited in terms of loop nesting, and it also cannot be used to iterate through several arrays in one loop.

## Two-Dimensional Arrays

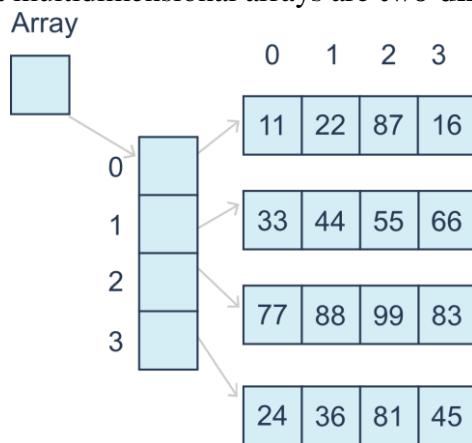
### Introduction

You have already found out about one-dimensional arrays, and it's now time to explore two-dimensional arrays. In this lesson, you will study the concepts of two-dimensional and multidimensional arrays, including how they are organized and the tools for creating them, and then you will delve into the subject of jagged arrays.

### The Concept of a Two-Dimensional Array

In Java, **multidimensional arrays** are arrays of arrays, that is, sets of one-dimensional arrays, references to which are stored in other one-dimensional arrays.

Multidimensional arrays may be two-, three-dimensional, etc. A separate set of square brackets is used to specify each additional dimension. The simplest multidimensional arrays are **two-dimensional** (see example on the picture).



### DESCRIPTION

To define two dimensions, it is necessary to specify two groups of empty square brackets when describing the reference type and two groups of square brackets with dimensions when allocating memory. For example:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

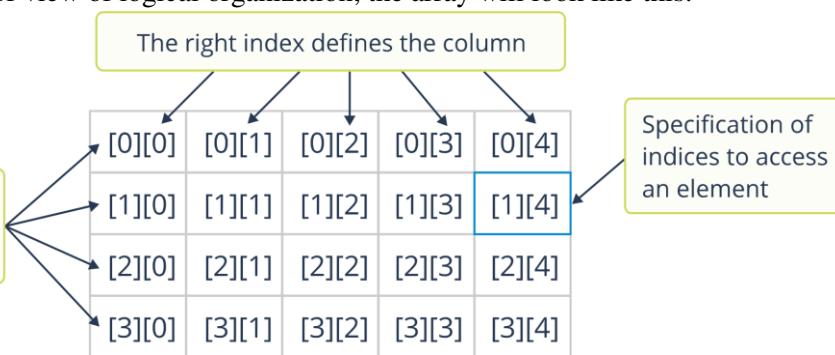
```
int[][] twoD = new int[4][5]
```

This is like describing a table where the first dimension determines the number of lines and the second the number of columns. Thus, to access an element of a two-dimensional array, two indices have to be specified: a line number and a column number.

Look at an example. Suppose the array **twoD** was set to describe the multiplication table. Then the value of the array element is the product of the indices of this element: the value of *twoD[2][4]* is 8, and the value of *twoD[3][7]* will be 21.

### LOGICAL ORGANIZATION

From the point of view of logical organization, the array will look like this:



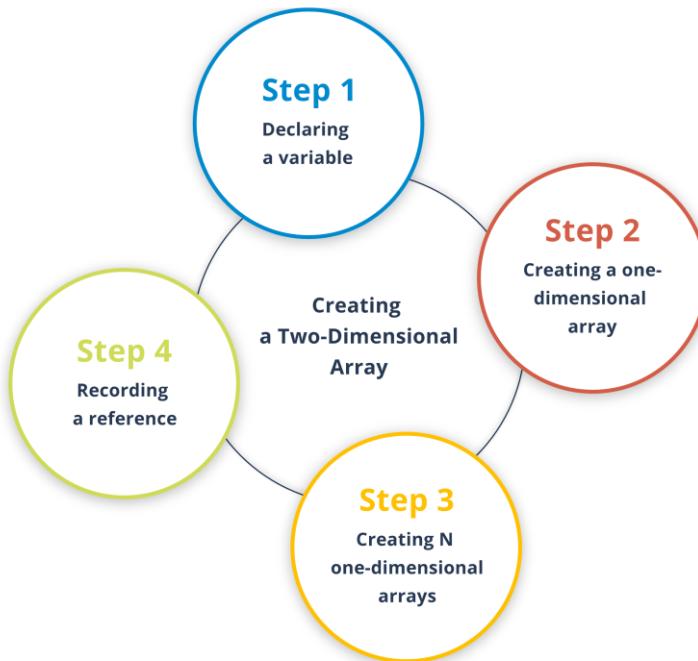
```
int[][] twoD = new int[4][5];
```

Thus, a two-dimensional array is the simplest type of multidimensional array whose description is similar to that of a table.

## Creating and Using a Two-Dimensional Array

What is the order of steps for creating a two-dimensional array? Pay attention to the actions performed by Java.

*Click on the + signs for more information.*



### Step 1. Declaring a variable

Java declares a variable that will contain a reference to an array of references. The elements of this array will refer to one-dimensional arrays of the type that was declared.

### Step 2. Creating a one-dimensional array

Java creates a one-dimensional array with dimension  $N$  specified by first index (the array of references is filled by default values, i.e., `null`).

### Step 3. Creating $N$ one-dimensional arrays

Java creates  $N$  one-dimensional arrays of the type that was declared, each containing  $M$  elements specified by second index (by default, each of these arrays is filled with null values).

### Step 4. Recording a reference

Java records references to each of these new arrays in the array created during Step 2.

Such behavior allows for creating only an array of references first and then initializing it if necessary.

The example below will help you better research the concept of a two-dimensional array and how it is used.

### Example

In this code fragment, when describing the array, only the first dimension is specified. Thus, the process looks like this:

- An array of arrays with a dimension of five elements is created.
- The array is displayed to the standard output.
- One-dimensional arrays are created, references to which are stored in the first array.
- A two-dimensional array is displayed in the console (for a better understanding of the mechanism for creating a two-dimensional array).

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```

int[][] multiplicationTable = new int[5][];
System.out.println("multiplicationTable = " + multiplicationTable );
for (int i = 0; i < multiplicationTable.length; i++) {
    System.out.println("multiplicationTable[" + i + "] = " + multiplicationTable[i]);
}
System.out.println("\nCreate array");
  
```

```

for (int i = 0; i < multiplicationTable.length; i++) {
    multiplicationTable[i] = new int[5];
    System.out.println("multiplicationTable[" + i + "] = " + multiplicationTable[i]);
}
System.out.println("\nInitialization array");
for (int i = 0; i < multiplicationTable.length; i++) {
    for (int j = 0; j < multiplicationTable[i].length; j++) {
        System.out.print(" " + multiplicationTable[i][j]);
    }
    System.out.println();
}

```

**Output:**

*Hover the mouse cursor over the info icon to see an explanation of the output.*

multiplicationTable	=	[[I@14991ad
multiplicationTable[0]	=	null
multiplicationTable[1]	=	null
multiplicationTable[2]	=	null
multiplicationTable[3]	=	null
multiplicationTable[4] = null		

Create array

multiplicationTable[0]	=	[I@d93b30
multiplicationTable[1]	=	[I@16d3586
multiplicationTable[2]	=	[I@154617c
multiplicationTable[3]	=	[I@a14482
multiplicationTable[4] = [I@140e19d		

Initialize array

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0 0 0 0 0				

It is important to understand that in multidimensional arrays, only the last level of indexation (the rightmost index) directly indicates the element of the declared data type of the array. The other indices contain only references to arrays.

**Common Pitfalls**

Pay special attention to some common pitfalls related to two-dimensional arrays.

*Click on the tabs to learn more.*

**An example of compilation error**

Review a description of a two-dimensional array that specifies only the first dimension (i.e., an array of references is created). When trying to write an integer value in the array of references, you get a compilation error. The compiler verifies the data type and cell type used to place the data:

*Hover the mouse cursor over the info icon to see an explanation of the error.*

`int[][] array2D = new int[10][];`

`array2D[0] = 10;`

In this case, you will have an error since this array element stores a reference to **an array of the type int** and not to a **value of the type int**. These are different data types.

Multidimensional arrays can be initialized using literal expressions. All you need to do is nest the description of some arrays into other arrays using nested curly brackets.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
int[][] products = { { 0, 0, 0, 0, 0 },
    { 0, 1, 2, 3, 4 },
    { 0, 2, 4, 6, 8 },
    { 0, 3, 6, 9, 12 },
    { 0, 4, 8, 12, 16 } };
```

Now that you have reviewed compilation and execution errors, you will find out about the specifics of processing a two-dimensional array.

### An example of runtime exception

Similarly, review a description of a two-dimensional array that specifies only the first dimension. When trying to write an integer value in it, you get a runtime exception. The compiler gives no error because, for the compiler, the data type and the cell type used with the data match. The compiler does not check for the presence of a cell, which is evaluated when the program is being executed.

*Hover the mouse cursor over the info icon to see an explanation of the error.*

```
int[][] array2D = new int[10][];
array2D[0][0] = 10;
```

Here you have an error since you did not create an array of values of the type **int**. A reference to a nonexistent cell causes this error.

Multidimensional arrays can be initialized using literal expressions. All you need to do is nest the description of some arrays into other arrays using nested curly brackets.

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
int[][] products = { { 0, 0, 0, 0, 0 },
    { 0, 1, 2, 3, 4 },
    { 0, 2, 4, 6, 8 },
    { 0, 3, 6, 9, 12 },
    { 0, 4, 8, 12, 16 } };
```

Now that you have reviewed compilation and execution errors, you will find out about the specifics of processing a two-dimensional array.

## Processing a Two-Dimensional Array

The problem below will help you understand how to process a two-dimensional array.

**Problem:** Find the sum of the elements of a two-dimensional array.



### What solution would you propose?

The solution is as follows:

1. The dimensions of the two-dimensional array should be 3 x 5. The short form should be used to define an array.
2. Nested loops are used to find the sum: external to iterate through the elements of the reference array and internal to iterate through the elements of the arrays of integers.
3. Both the array and the sum should be displayed.

During the second and third steps of working with the array, you will use the **for-each** loop since you do not need to change the array elements.

▼ **Code and console output**

```

int[][] array2D = { { 1, 2, 3, 4, 5 },
    { 5, 4, 3, 2, 1 },
    { 0, 2, 0, 4, 0 } };
sum = 0;
for (int[] row : array2D) {
    for (int element : row) {
        sum = sum + element;
    }
}
for (int[] row : array2D) {
    for (int element : row) {
        System.out.print(element + " ");
    }
    System.out.println();
}
System.out.println("sum = " + sum);

```

**Output:**

1 2 3 4 5  
 5 4 3 2 1  
 0 2 0 4 0  
 sum = 36

Also note the description of a three-dimensional array.

**int[][][] dim3D = new int[3][4][20];**

As a result, you get:

- A reference array with a dimension of 3
- Three one-dimensional reference arrays, each with a dimension of 4
- Twelve ( $3 \times 4$ ) one-dimensional arrays of the type **int**, each with a dimension of 20

It is worth noting that when accessing the value of *dim3D[1][1]*, you get the reference address. And only in case of total access to *dim3D[1][1][1]* you get the value of the array element of the type **int**.

So far, you have explored the concept of two-dimensional arrays, including how to create and organize them. You also studied common compilation and execution errors and processing features. Next, you will explore special kind of multidimensional array — "jagged" multidimensional arrays.

## Jagged Multidimensional Arrays

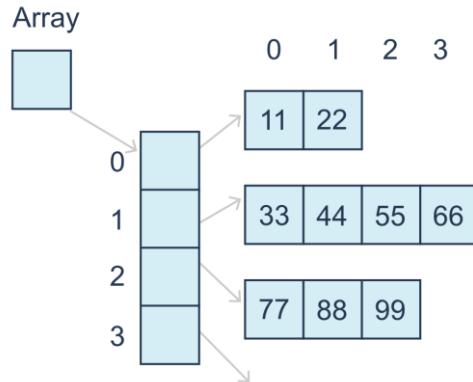
In Java, **multidimensional** arrays are arrays of arrays, that is, sets of one-dimensional arrays that are referenced in other one-dimensional arrays.

As you already know, multidimensional arrays are arrays of arrays. This means that it is possible to create a set of arrays of different lengths, that is, **jagged** arrays (as you can see on the picture). Please note that you can encounter various synonyms for jagged arrays, such as ragged, stepped, or unsymmetrical two-dimensional/multidimensional arrays.

When allocating memory for a multidimensional array, since it is enough to specify only the first (leftmost) dimension, you can separately allocate the memory that corresponds to the remaining dimensions of the array later.

Declaring an array like this has no benefits, but in some cases, it can be justified. Specifically, this allows you to set arrays of different lengths for each index of the array of references.

Examine carefully the various ways multidimensional arrays can be described.



### Method1: define initialization

If all the dimensions and values of the array elements are known, you can use definite initialization to describe it.

```
int[][] array = { { 11, 22 },
                  { 33, 44, 55, 66 },
                  { 77, 88, 99 },
                  {} };
```

### Method 2: memory allocation

If all the dimensions are known but the values of the array elements are not, first you may just reserve/allocate memory.

```
int[][] arr = new int [4][];
arr[0] = new int [5];
arr[1] = new int [4];
arr[2] = new int [3];
arr[3] = new int [2];
```

Take a look at an example of using a jagged multidimensional array. The code fragment shows all the options for manipulating such an array.

### Example

```
int[] numbers = { 1, 3, 5, 7, 9 };
int[][] array = new int[3][];
array[0] = numbers;
array[2] = new int[] { 2, 4, 6, 8 };
for (int[] row : array) {
    if (row != null) {
        for (int element : row) {
            System.out.print(element + " ");
        }
    } else {
        System.out.print(row);
    }
    System.out.println();
}
```

### Output:

```
1 3 5 7 9
null
2 4 6 8
```

For most applications, it is not recommended to use jagged arrays since this makes it difficult for other developers to read the code. However, in some cases, these arrays are quite acceptable and may boost a program's performance significantly. Thus, if you need to create a large two-dimensional array where not all elements are used,

a jagged array will save you a substantial amount of memory.

The following video presents an alternative view on the term "multidimensional array" and illustrates everything you have studied so far.

## Conclusion

In this lesson, you:

- Established that multidimensional arrays (arrays of arrays) are sets of one-dimensional arrays, references to which are stored in other one-dimensional arrays
- Declared, initialized, and used multidimensional arrays with various dimensions
- Created sets of arrays of different lengths, that is, jagged arrays

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. Which descriptions of a multidimensional array are correct?

`int[][] array1 = {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};`

`int[][] array2 = new array() {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};`

`int[][] array3 = {1, 2, 3}, {0}, {1, 2, 3, 4, 5};`

`int[][] array4 = new int[2][];`

correct

Answer

Correct:

`int[][] array2 = new array() {{1, 2, 3}, {}, {1, 2, 3, 4, 5}};` is wrong because to declare an array, you should use square brackets, not round brackets. `int[][] array3 = {1, 2, 3}, {0}, {1, 2, 3, 4, 5};` is wrong because the entire declaration of an array should be placed in one set of curly brackets.

2. What is the result of running the following code?

`String[] ejgStr = new String[][] { { null }, new String[] { "a", "b", "c" }, { new String() } }[0];`

`String[] ejgStr1 = null;`

`String[] ejgStr2 = { null };`

`System.out.println(ejgStr[0] + " " + ejgStr2[0] + " " + ejgStr1[0]);`

null and then the error NullPointerException

null null and then the error NullPointerException

**the error NullPointerException**

null null null

Answer

Correct:

The NullPointerException exception will appear when trying to index the array initialized by the literal expression null. The literal expression null cannot be indexed. Neither can you use it to access the fields and methods of any object.

## Methods of the Arrays Class

### Introduction

Now that you have studied one-dimensional and multidimensional arrays, it is time to explore the methods used to work with arrays. In this lesson, you will study the **Arrays** class and its methods.

### Arrays Class

There are many methods for manipulating arrays in the Java library. For example, a number of methods are determined in the **Object** and **System** classes in the *java.lang* package. There is also the special class **Arrays** (package *java.util*), which is most often used in practice. Review its main methods in more detail.

`toString()`

Returns a string representation of the contents of the specified array.

`copyOf(array, length)`

Copies the array with the specified length.

`equals(array1, array2)`

Compares arrays.

`sort(array)`

Sorts the elements of the specified array into ascending numerical order.

`binarySearch(array, element)`

Searches the specified array for the element value and returns its index if the element is found.

`fill(array, element)`

Assigns the element value to the array elements.

*All methods work with arrays of all primitive types.*

To use the methods of the **Arrays** class, this must be included (imported) in the program:

**import java.util.Arrays;**

To access a method, you should specify the class name with a dot in front of its name, for example:

**Arrays.fill(array, 5);**

Now that you are familiar with the **Arrays** class, it's time to review how the methods of this class work.

### Methods of the Arrays Class

Explore how each of the above methods works.

*Click on the drop-down elements to learn more about the methods.*

The `toString` method is used to get a string representation of the contents of the specified array. This should consist of square brackets containing the array elements separated by commas. It is convenient to use this method to display a one-dimensional array in the console.

Description:

**public static String `toString(<type>[] a)`**

For example:

```
int[] array = {9, 8, 7, 6, 5};
System.out.println(Arrays.toString(array));
```

**Output:**

[9, 8, 7, 6, 5]

If you try printing the array without using the `toString()` method of the **Arrays** class, the program will return not the contents of the array but the memory address where it is stored.

### copyOf

The `copyOf` method is used to receive a copy of the transferred array that specifies its new length.

Description:

**public static <type>[] copyOf(<type>[] original, int newLength) { }**

The `copyOf` method:

- Returns a copy of the transferred original array with the specified length *newLength*
- If *newLength* is larger than the original array, all missing elements should be filled with the **null** value.
- If *newLength* is smaller than the original array, an array will be returned where only the first *newLength* elements will be filled.

For example:

```
int[] array = {9, 8, 7, 6, 5};
System.out.println(Arrays.toString(array));
int[] newArray = Arrays.copyOf(array, 8);
System.out.println(Arrays.toString(newArray));
```

*Hover the mouse cursor over the info icon to see an explanation of the code.*

**Output:**

[9, 8, 7, 6, 5]

[9, 8, 7, 6, 5, 0, 0, 0]

### Equals

The `equals` method is used to compare two arrays of the same type.

Description:

**public static boolean `equals(<type>[] a, <type>[] a2) { }`**

Two arrays are considered equal (the same) if they contain the same number of elements and the relevant pairs of array elements are equal.

For example:

*Hover the mouse cursor over the info icon to see an explanation of the code.*

```
int[] arr1 = {1,2,3,4,5,6,7,8,9};
int[] arr2 = {1,2,3,4,5,6,7,8,9};
int[] arr3 = {1,2,5,5,5,5,8,9};
System.out.println(arr1 == arr2);
System.out.println(Arrays.equals(arr1, arr2));
System.out.println(Arrays.equals(arr1, arr3));
```

**Output:**

false

true

false

As you can see, comparing references to different objects will return "false" even if they have the same number of elements and are identical.

### **Sort**

The *sort* method is used to sort array elements into ascending numerical order (elements with equal values are placed next to each other). The sorting time depends on the initial ordering of the array elements in the required order. The more elements are arranged in the required order, the faster they are sorted.

Description:

```
public static void sort(<type>[] a) { }
```

For example:

```
int intArr[] = {55, 57, 61, 66, 18, 19, 27, 38, 10, 55, 15, 39, 51, 18, 83, 95};
```

```
Arrays.sort(intArr);
```

```
System.out.println("The sorted int array is:");
```

```
System.out.println(Arrays.toString(intArr));
```

### **Output:**

The sorted int array is:

```
[10, 15, 18, 18, 19, 27, 38, 39, 51, 55, 55, 57, 61, 66, 83, 95]
```

### **binarySearch**

The *binary Search* method is used to search an array. To do this, the binary search algorithm is used. If you are searching for several values, you will get the information on the location of only one such value. For this method to work properly, the **array must be sorted**.

Description:

```
public static int binarySearch(<type>[] arr, <type> key) { }
```

This method:

- Returns the index for which the element with the **key** value is contained in the **arr** array
- If the element is absent, a negative value will be returned, meaning that there is no such element.

For example:

```
int intArr[] = {10, 15, 18, 18, 19, 27, 38, 39, 51, 55, 55, 57, 61, 66, 83, 95};
```

```
int searchVal = 55;
```

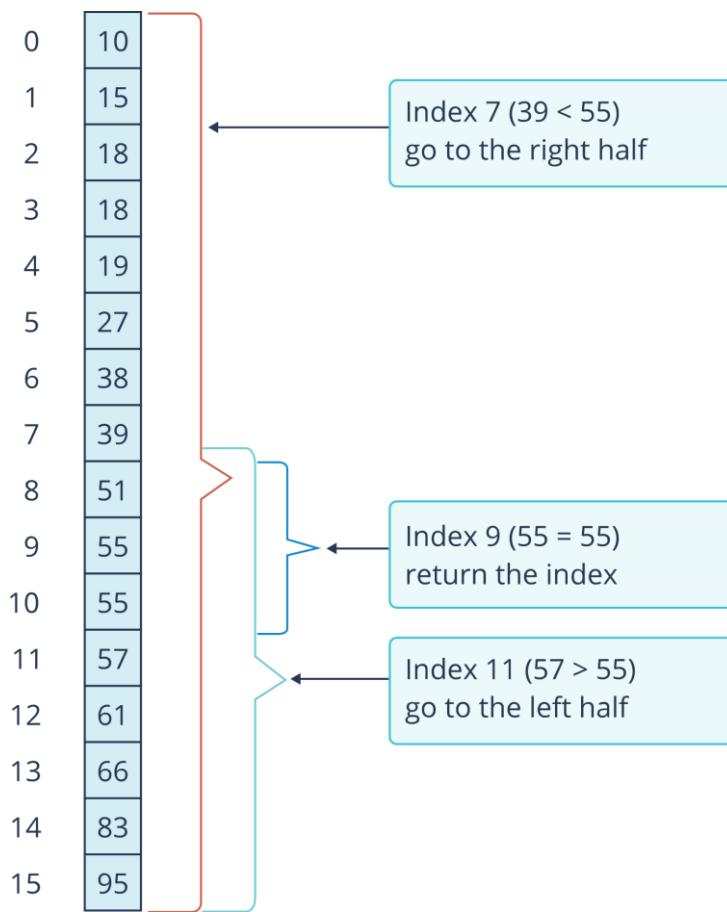
```
int retVal = Arrays.binarySearch(intArr, searchVal);
```

```
System.out.println("The index of element 55 is : " + retVal);
```

### **Output:**

The index of element 55 is : 9

### **Binary search algorithm**



1. Finds the index of the middle element between the indices of the left and right boundaries of the array.
2. The element received is checked for whether it is equal to the searched value:
  - If it is equal, its index is returned.
  - If it is not equal, one of the halves is searched.
3. The searched value and element are evaluated by the index found:
  - If the searched element has a lower value, the index preceding the middle value becomes the right boundary.
  - If the searched element has a higher value, the index following the middle value becomes the left boundary.
4. The search continues starting from step 1 and until the element is found or until the left and right boundaries cross. In the latter case, a negative value will be returned (search not successful).

## FILL

The *fill* method is used to fill the array with the assigned value. All or some array elements may receive the value. The method is used to fill the array with the set value.

Description:

```
public static void fill(<type>[] arr, <type> value) { }
```

In this method, no value is returned. All array elements receive the value *value*.

For example:

```
int[] array = new int[7];
```

```
Arrays.fill(array, -1);
```

```
System.out.println(Arrays.toString(array));
```

**Output:**

```
[-1, -1, -1, -1, -1, -1, -1]
```

In the previous lesson, you learned about multidimensional arrays. It is time to review the methods for manipulating them — *deepEquals* and *deepToString*.

Click on each tab to learn more.

The `deepToString` method returns a string representation of the specified multidimensional array. The string representation consists of a list of the array's elements enclosed in square brackets, depending on the nesting level.

Description:

**public static String deepToString(Object[] a)**

Example:

```
int [][] array = {{1, 2, 3},{4, 5, 6}};
System.out.println(Arrays.deepToString(array));
```

**Output:**

[[1, 2, 3], [4, 5, 6]]

The `deepEquals` method returns **true** if the two specified arrays are completely equal. Unlike the `equals` method, this method can be used with multidimensional arrays. Two multidimensional arrays are considered equal if one of the following conditions is met:

- Both are **null**.
- They refer to arrays that contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal.

Description:

**public static boolean deepEquals(Object[] a1, Object[] a2) { }**

Example:

```
int[][] array = { { 1, 2, 3 }, { 4, 5, 6 }, { 7 } };
int[][] anotherArray = { { 1, 2, 3 }, { 4, 5, 6 }, { 7 } };
System.out.println(Arrays.equals(array, anotherArray));
System.out.println(Arrays.deepEquals(array, anotherArray));
```

**Output:**

false

true

You can find more information on the methods of the **Arrays** class in the [official documentation](#).

## Conclusion

In this lesson, you studied the **Arrays** class and the following methods:

- `toString(array)` — returning a string representation of an array.
- `deepToString(array)` — returning a string representation of a multidimensional array.
- `copyOf(array, length)` — copying an array with a specific length.
- `equals(array1, array2)` — comparing arrays.
- `deepEquals(array1, array2)` — comparing multidimensional arrays.
- `sort(array)` — sorting the elements of a specific array into ascending numerical order.
- `binarySearch(array, element)` — searching a specific array for the element value and returning its index if the element is found.
- `fill(array, element)` — filling array elements with the value element.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

3/3 points

1. What is the result of running the following code?

```
int[] array = new int[] {3, 4, 2, 1};
Arrays.sort(array);
System.out.println(Arrays.toString(array));
 [3,4,2,1]
 [1,2,3,4]
```

[3,4]

a compilation error

Answer

Correct:

The sort method is used to sort the elements of a specific array into non-descending numerical order.

2. What is the result of running the following code?

```
int[] array = {1, 2, 3, 4};
```

```
System.out.println(Arrays.toString(array));
```

**[1,2,3,4]**

an unpredictable output

an array

a compilation error

Answer

Correct:

The toString method is used to return a string representation of the array.

3. What is the result of running the following code?

```
int size = 4;
```

```
int[] testArr = new int [size];
```

```
Arrays.fill(testArr, 1);
```

```
System.out.println(Arrays.toString(testArr));
```

**[1,2,3,4]**

**[1,2,3]**

**[1,1,1]**

**[1,1,1,1]**

Answer

Correct:

The fill method is used to fill the array with the required value.

## Introduction to OOP

### Introduction

This lesson will provide an overview of object-oriented programming (OOP).

### Concept of OOP

As computers become more and more integral in all areas of society, software systems are becoming simpler for users but more complex in terms of their internal architecture.

Programming has become a team matter:



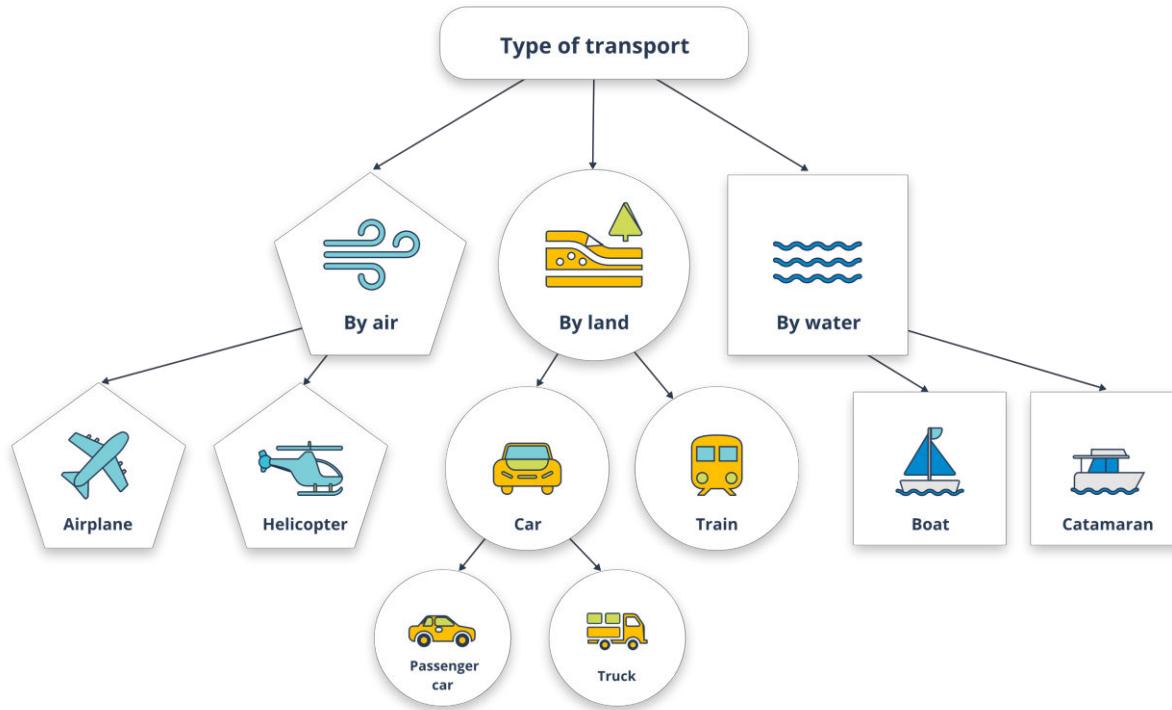
- A small project now means a project carried out by a team of 5–10 specialists in 6 months or more.
- A large project means a project that lasts for years and can be executed by hundreds of developers from around the world.

Modern information technologies and methodologies such object-oriented programming (OOP) and functional

programming (FP) have become the main methods for creating complex software products.

First, let's look at OOP in more detail.

**OOP** is an approach to software development based on the representing of a program as a collection of objects, where each object is an instance of some class, forming hierarchical structures. For example, a hierarchical structure of transportation would look like the following:

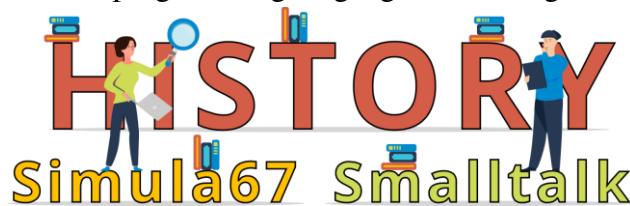


This approach reflects the real world.

## History of OOP Development

Let's look at the two key stages that led to the rise of OOP.

The foundation for OOP was laid in the early 1960s. The first programming language to work with objects was **Simula 67**, which was developed to create simulations of marine vessel explosions. The authors grouped marine vessels into various categories and assigned classes to them. Thus, Simula not only introduced the concept of class but also presented a working model of OOP. Simula 67 was an innovative system that later became the basis for the development of a large number of other programming languages, including Pascal and Lisp.



OOP gained popularity in the second half of the 1980s. The term itself was first used by Xerox PARC in the programming language **Smalltalk** to denote the process of using objects as a basis for computing. The developers designed their language to be dynamic: Objects could be modified, created, or deleted, distinguishing it from the static systems that were popular at the time. This programming language was also the first to use the concept of *inheritance*.

According to many experts, the object-oriented approach will remain the primary paradigm in programming development in the decades to come.

## Conclusion

In this lesson you:

- Discovered what object-oriented programming is
- Reviewed a short history of OOP development as it relates to the programming languages Simula 67 and Smalltalk

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. When was the foundation of OOP laid?

1950s

**1960s**

1970s

1980s

Answer

Correct:

The foundation of OOP was laid in the early 1960s with the development of the Simula 67 programming language.

2. What was special about the programming language Smalltalk?

Smalltalk became the basis for the development of languages such as Pascal and Lisp.

**In Smalltalk, objects can be modified, created, or deleted.**

Smalltalk was designed as a static language.

**Smalltalk was the first language to use the concept of inheritance.**

correct

Answer

Correct:

The Smalltalk language was designed as a dynamic language: Objects can be modified, created, or deleted. Smalltalk was also the first programming language to use the concept of inheritance.

## Classes and Objects

### Introduction

In this lesson, you will explore the concepts of *classes* and *objects* in Java. You will also review the principles of building classes that are used in object-oriented programming (OOP).

### Classes and Objects

Classes may be viewed as certain descriptions, schemes, or drawings according to which objects are created. In one class, you can create an unlimited number of objects, each with the same behavior and set of features.

For example, if you have a schematic (class) of a bicycle, you can create different bicycles (objects) with the same number of wheels and a similar rotating pedal mechanism and brake system.

*Click on the + signs for more information.*



### Class

A **class** is a template describing multiple similar objects in the real world or in a virtual world. This set of objects should have the same set of characteristics (fields/properties) and the same set of actions that can be performed with these objects by changing their internal state. For example, changing the positions of a bicycle's pedals sets it in motion. In other words, the mechanism of pedal rotation is the method of the class.

## Object

An **object** is an instance of a class. For example, each bicycle has its own color, weight, and different components. These are its features/characteristics. For each created object, these characteristics have their own values that are specific to the object.

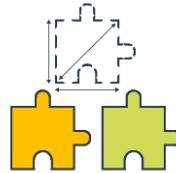
When creating an object, you consider mechanisms to teach the object to perform various operations. You can teach a bicycle to analyze its surroundings, make decisions, and perform certain actions communicated by the bicyclist. Now imagine that you have created a description of that object as a template. Using this template, and with the help of the program, you have made 500–600 such "bicycles." Each "bicycle" has a set of parameters: a certain number of wheels, a handlebar and two pedals, and the ability to switch gears. By placing them in certain circumstances, you can set them in motion. And here you should mention the fundamental difference between a class and an object:



A class is a description, while an object is a materialization.

In other words, a **class** is an abstract type of data type that determines the form and behavior of an **object** with the help of fields and methods:

- Fields are properties (data, attributes) that represent the content and the state of an object.
- Methods are functions that determine an object's behavior. They describe operations performed with data.



The briefest description of an object is offered by Grady Booch: "An object has a state, behavior, and identity."

Let's sum up what classes are and also try to answer several questions:

- What groups of classes exist?
- How can you create a class?
- What do classes of different groups contain?
- What has to be considered when designing classes?

Thus, similar objects are created based on the same model. By creating a sample, drawing, or type, you create a class (a drawing of a bicycle), and then the program creates objects of the class (a virtual bicycle based on this drawing).

## Describing a Class

When defining a class, you declare its specific form and behavior. To do this, you specify the fields it contains as well as the methods used to operate these fields. The simplest classes may contain only methods or only fields, but most real classes contain both.

The syntax of class description (Components placed in square brackets are optional.):

*Click on the + signs for a more detailed description of classes.*

**[Access\_Modifier] class Identifier [Type\_Parameters]  
[Super\_Class] [Super\_Interfaces] {Class\_body}**

### Access\_Modifier

**Access\_Modifier** establishes the rules for using a class (visibility scope).

### Identifier

**Identifier** is the name of a class (that is, the data type).

### Type\_Parameters

**Type\_Parameters** determine the generality of the class data.

## Super\_Class

**Super\_Class** defines a class's parent class.

## Super\_Interfaces

**Super\_Interfaces** determine a class's additional behavior.

## Class\_body

**Class\_body** determines the fields and methods of a class.

Most of the aspects of declaring a class will be discussed later in the course.

A correctly constructed class should determine one and only one logical entity. For example, a class that stores a user's personal data usually does not contain data about the equity market, average precipitation, sunspot cycles, or other irrelevant information. Thus, a correctly constructed class should group logically related information. If unrelated information is placed inside a class, the code structure is quickly broken.

Take a look at the following example of a class description.

*Click on the dropdown element to study the example.*

*Hover the mouse cursor over the info icon to see an explanation of the code.*

### example

```
package com.epam.oop;
public class Car {
    private String model;
    private int maxSpeed;
    private int year;
    public Car (String model, int year, int maxSpeed) {
        this.model = model;
        this.year = year;
        this.maxSpeed = maxSpeed;
    }
    public int getMaxSpeed() {
        return maxSpeed;
    }
}
```

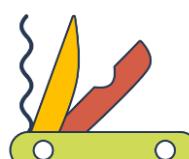
Thus, when declaring a class, you should declare its specific form and behavior. To do this, specify the fields it contains and the methods used to operate these fields.

## OOP Properties and Principles

An introduction to the five major properties of object-oriented programming (OOP) will help us better understand classes and objects in Java.

*Click on the tabs to learn more.*

**Everything is an object.** An object can store and transform information. Generally, any element of a problem being solved (e.g., a house, dog, service, or chemical reaction) can represent an object. An object can be imagined as a Swiss Army knife: It is a set of different knives and "openers" (storage), but at the same time, you can cut or open things (transformation).



**A program is a set of objects that tell each other what to do.** To access one object, another object "sends it a message." A "response message" is also possible. For example, a program can be imagined as a set of three objects: a writer, a pen, and a sheet of paper. The writer "sends a message" to the pen, which in turn "sends a message" to the sheet of paper. As a result, you see a text (sending a message from the sheet to the writer).



consists of doors, rooms, windows, wiring, and heating. A door, in turn, consists of a door panel, a doorknob, a lock, and hinges. Wiring consists of wires, sockets, and a switchboard.



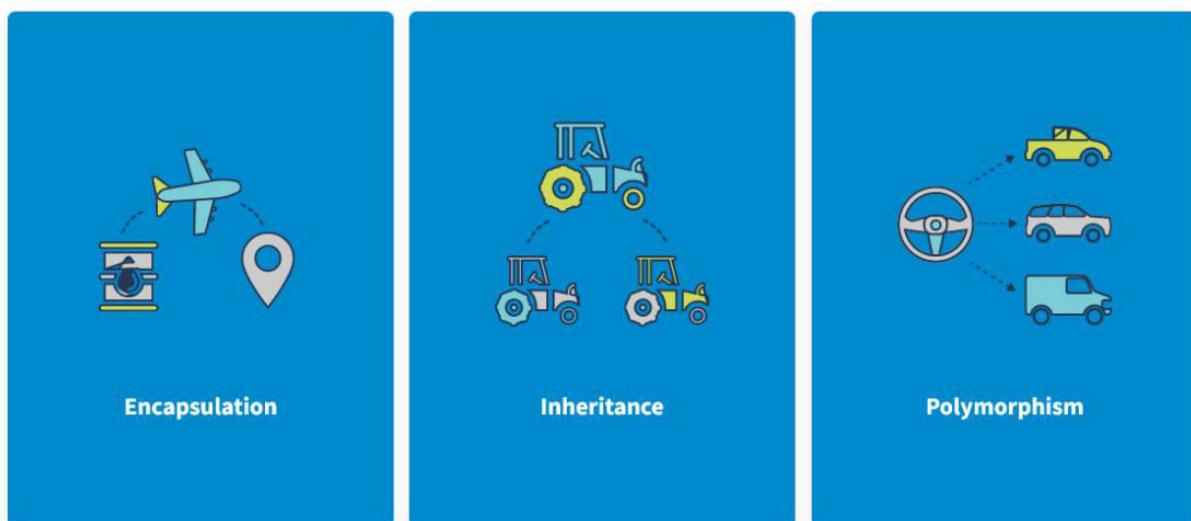
**Each object has its type.** Sometimes a type is also called a class. A class (type) determines which messages objects can send to each other. For example, an battery can send current to a lamp, while it cannot send an impulse or a physical effort.



**All objects of one type can receive identical messages.** For example, let's say you have two objects: a green paper cup and a transparent plastic cup. They are different in form, color, and material. But you can drink from both if they are not empty. In this case, the cup is the object type.



OOP involves three main principles: encapsulation, inheritance, and polymorphism.



**Encapsulation** is a property that allows data—and the methods used to work with it—to be consolidated in a class while hiding the implementation details from users.

**Inheritance** is a property that allows a new class to be created based on an existing one, and in which the properties of the parent class will be assigned to the child class.

**Polymorphism** is the ability to provide a single interface to entities of different types.

Thus,

OOP has a number of characteristics, including three major principles of class-building: encapsulation, inheritance, and polymorphism.

## Conclusion

In this lesson, you have seen that:

- A class is an abstract data type that determines the form and behavior of an object.
- An object has a state, behavior, and identity.
- The creation of classes is based on three major OOP principles: encapsulation, inheritance, and polymorphism.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

4/4 points

1. From a programming standpoint, what is a **class**?

The definition of the structure of some entity

The definition of the behavior of some entity

**The definition of the structure and behavior of some entity**

The definition of interactions between entities of one type

Answer

Correct:

From a programming standpoint, a class defines the structure and behavior of some entity.

2. What is an **object** of a class?

A class description

**A class instance**

A class similarity

A class representative

Answer

Correct:

A class instance is called an object.

3. Which of the properties below refer to object-oriented programming?

**Everything is an object.**

**A program is a set of objects that tell each other what to do.**

Separate objects can have their own "memory" consisting of other objects.

**Each object has its type.**

**All objects of one type can receive identical messages.**

correct

Answer

Correct:

One of the above items is not completely correct. In fact, every object has its own "memory" consisting of other objects.

4. Which of the items listed below is a property of classes that allows for the use of class objects with the same interface without any information on the type of implementation or the internal structure of the object?

Encapsulation

Inheritance

**Polymorphism**

Abstraction

Answer

Correct:

Polymorphism is a property of classes that allows for the use of class objects with the same interface without any information on the type of implementation or the internal structure of the object.

## Fields

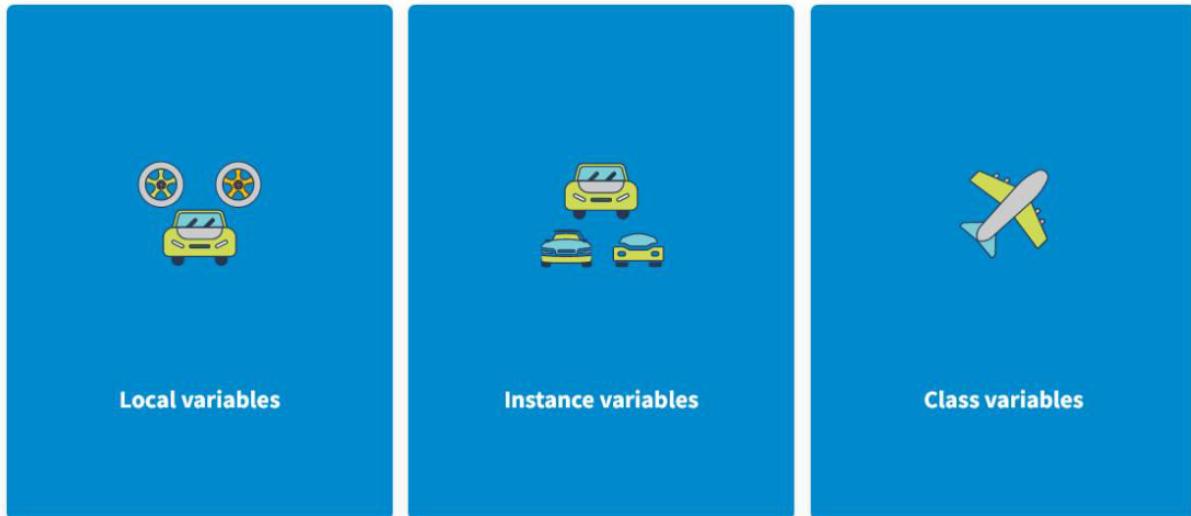
### Introduction

You are now familiar with the specifics of classes and objects, as well as the topic of fields in the class description. In this lesson, you will explore fields in more detail.

### The Concept of a Field

You already know that when declaring a class, you should declare its specific form and behavior. To do this, you specify the fields it contains as well as the methods used to operate these fields. So, what exactly is a field? A **field** is a declaration or description of a variable.

*Click on the cards to learn more about each type of variable.*



**Local variables** are defined inside methods, constructors, or blocks. They are declared and initialized in these constructions and destroyed upon exiting them.

**Instance variables** are defined within a class but outside of methods. They are initialized when a class object is declared. These variables are available inside a method of the instance, constructor, or block of the initialization.

**Class variables** (static variables) are declared in the class body outside of a method using the keyword **static**. They are initialized when a class is formed and available in all methods, constructors, and blocks of the initialization.

The keyword **static** can be placed in front of the class field. Find out what this means and how to use this field in the following video.

## Conclusion

In this lesson you discovered:

- The concept of a *class field*
- Which variables a class can contain
- What a static class variable is

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. Which variables can a class contain?

**Local variables**

**Instance variables**

**Class variables**

Field variables

Answer

Correct:

A class can contain local, instance, and class variables.

2. How are instance variables declared?

Inside methods, constructors, or blocks

**Within a class but outside of methods** correct

Within a class inside methods

In the class body, outside of any method using the keyword **static**

Answer

Correct:

Instance variables are defined within a class but outside of methods. They are initialized when a class object is declared.

## Methods

### Introduction

In this lesson, you will review methods and the rules for describing them, the specifics of calling methods, how to pass arguments to methods, and much more. Let's get started!

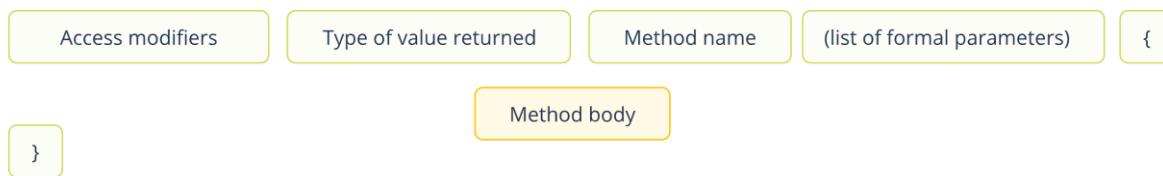
### Methods

**Methods** are functions that are declared within a class. They are used to execute actions with class data (fields) and give access to this data. They contain declarations of parameters, local variables, and Java statements that are

executed when a method is called. Methods can also return the result of their work to the code calling them. They can also accept parameters (arguments) that are values provided by the code that called the method.

A description of a method looks like the following:

*Click on the + signs for a more detailed description of each component of a method.*



## Access modifiers

Access rule (optional code element).

## Type of value returned

Methods can return the result of their work or not return this result (a mandatory code element).

## Method name

Method identifier (mandatory code element).

## Method body

Statements that solve the problem (method body can be empty).

## (List of formal parameters)

The data to be processed is passed via a list (The data itself may be absent, but the round brackets are mandatory).

{  
A method body is opened (block starting).  
}

A method body is closed (block completion).

*Click on the tabs to learn more.*

## METHOD NAME

The name of a method should:



- Start with a lowercase letter (except for constructors, which are special methods used to create objects)
- Contain the name of the action performed (verb)
- Be informative—A method can consist of several words, but the second and each subsequent word should be capitalized (camelCase notation)

For example: updateName, calculateLogarithm

## TYPE OF VALUE RETURNED

The following should be specified as the type of value returned:

- An indicator of the type **void** if the method is not supposed to return the result of its work to the calling method.
- A specific data type if the method is supposed to return the result of its work to the calling method.

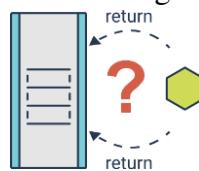


## RETURNING RESULT

To return a value from the method, use the statement **return**. Using the statement **return** means immediately stopping the execution of a method and returning (usually with a result).

- It is good practice to use one statement **return** in one method (usually at the end of the method body).
- It is also possible to use several **return** statements in one method under certain conditions.

The **return** statement may also be used in a method with the **void** type of the returned value. In that case, the **return** statement does not have an operand and simply means returning from the method.



## METHOD PARAMETERS

A formal parameter is a declaration of a variable that will be used in the method body. Through a formal parameter, methods receive data for processing.

The number of parameters (which are separated by commas) depends on how many variables you need to pass.



Let's look at two examples of method descriptions: returning a result and without a result.

### RETURNING A RESULT

his method finds the sum of two numbers and returns it.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public int sum(int a, int b) {
    int result = 0;
    result = a + b;
    return result;
}
```

### WITHOUT RETURNING A RESULT

This method resembles the countdown at the start of a race (does not return a result).

*Hover your cursor over the info icon to see an explanation of the code.*

```
void start(int number) {
    if (number > 10 || number <= 0) {
        System.out.println("Start failed!");
    } else {
        while (number > 0) {
            System.out.println(number--);
        }
        System.out.println("Start!");
    }
}
```

So far, you have reviewed what a method is and the rules to follow when describing a method.

## Calling a Method

**Calling a method** means transferring control to the method for execution.

Pay close attention to the syntax used to call a method:

Method name

(list of actual parameters)

The method parameters (arguments) are separated with commas.

Arguments can be represented by:

- A literal statement
- An expression
- A variable

- Calling a method returning a value

Pay close attention to the details of the method call.

*Click on the dropdown element to learn more.*

## RECEIVING RESULTS FROM A METHOD

- The result of a method is placed at the point where the method was called.
- If a method returns a value, this value should be accepted (for example, in a variable).

*Hover your cursor over the info icon to see an explanation of the code.*

```
public static String getWelcome() {
    return "Welcome!";
}
public static void main(String[] args) {
    String str = getWelcome();
    System.out.println(str);
}
```

## CALLING A METHOD WITHOUT ARGUMENTS

If the description of a method does not have a list of formal parameters, round brackets are specified as empty when calling the method.

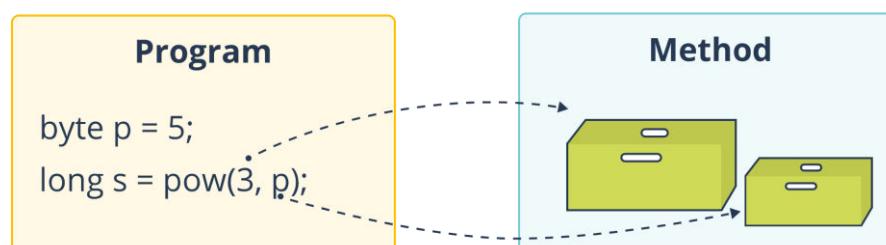
```
public static String getWelcome() {
    return "Welcome!";
}
public static void main(String[] args) {
    System.out.println(getWelcome());
}
```

## CALLING A METHOD WITH ARGUMENTS

- If the description of a method has a list of formal parameters, then when calling the method, you should specify the arguments in round brackets.
- The number of arguments must match the number of formal parameters.
- The argument type must match the type of the corresponding formal parameter or be compatible with it.
- If the argument and formal parameter types do not match and the implicit (expanding) cast does not fit, and no explicit cast has been performed, you will get an error.

For example, calling the method using `pow(5, 3)` will produce an error. This is because the second argument is a literal expression of the type **int** that cannot be passed without an explicit cast a parameter to the **byte** type.

```
long pow(int x, byte p) {
    long result = x;
    int counter = 1;
    while (counter++ < p) {
        result = result * x;
    }
    return result;
}
```



As you have seen, calling a method means transferring control to the method for execution. You have also explored the syntax used to call a method and other important details that must be considered in this process.

## Ways to Pass Arguments to Methods

To do its job, a method may require some data.

- A value passed to a method is called an **argument**.

- The variable that receives the argument is called a **formal parameter** or simply a **parameter**.

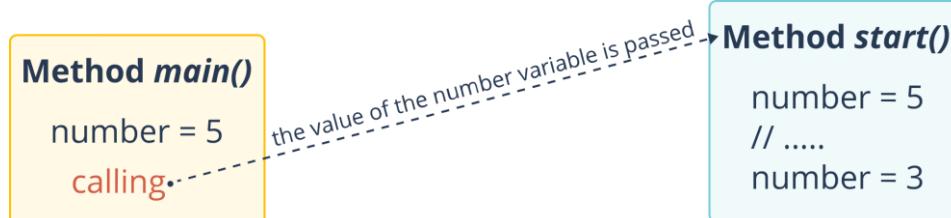
Parameters are declared in round brackets after the method name. The syntax for declaring parameters is the same as for variables. The scope of parameters is the method body. Multiple parameters can be defined in a method, and they are separated with commas.

*Click on the tabs to learn more.*

### BY VALUE

This is how values of **primitive** types are passed to a method.

The data passed to a method is accepted by the parameter. The change in the parameter's value in the method body does not change the initial data.



Two methods are shown in the example: *start()* and *main()*. The method *start()* is called from the method *main()* and receives a copy of the value of the **number** variable. Since the body of the method *start()* is a block of code independent of the body of the method *main()*, the name **number** may be used as the method's parameter. Changing the value of the parameter **number** in the method *start()* does not affect the variable **number** in the method *main()* since these are different variables.

```

public static void start(int number) {
    System.out.println("Old value of \"number\" into \"start\" method is:" + number);
    number = 3;
    System.out.println("New value of \"number\" into \"start\" method is:" + number);
}

public static void main(String[] args) {
    int number = 5;
    System.out.println("Old value of \"number\" into \"main\" method is:" + number);
    start(number);
    System.out.println("New value of \"number\" into \"main\" method is:" + number);
}
  
```

### Output

Old value of "number" into "main" method is:5

Old value of "number" into "start" method is:5

New value of "number" into "start" method is:3

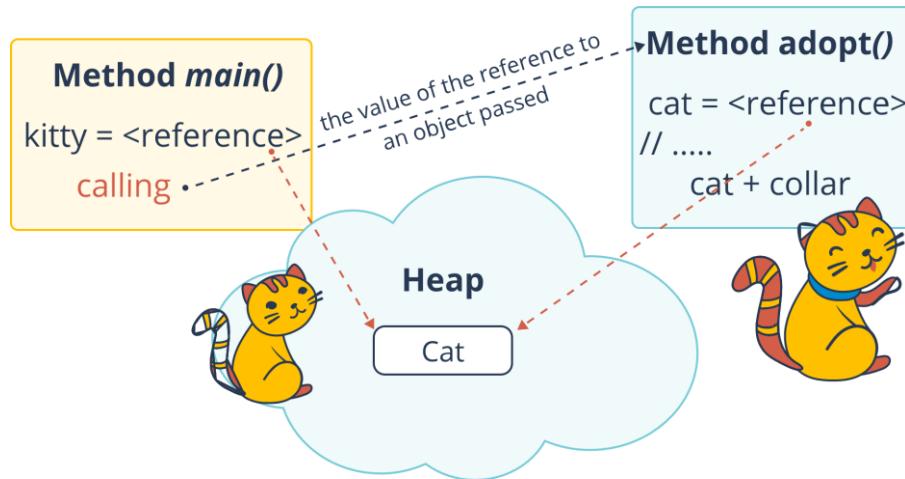
New value of "number" into "main" method is:5

Now that we have studied how to pass data to a method, it is time to explore methods with a variable number of parameters.

### BY REFERENCE VALUE

This is how values of **reference** types are passed to a method.

A copy of the reference value to an object is passed to a method and not the object itself: Changes made with the object's content in the method will be accessible/visible from the part of the program where the object was created.



Two methods are shown in the example: `adopt()` and `main()`. The method `adopt()` is called from the method `main()` and receives a copy of the value of the reference to an object of the type **Cat**. As a result, both methods have variables that refer to the same object. Thus, changes made with the cat in the method `adopt()` (for example, putting a collar on the cat) will also be visible in the method `main()`.

```

class Cat {
    private boolean collarStatus = false;
    public boolean isCollarStatus() {
        return collarStatus;
    }
    public void setCollarStatus(boolean status) {
        collarStatus = status;
    }
}

public class Main {
    public static void adopt(Cat cat) {
        System.out.println("Old value of \"collarStatus\" in \"adopt\" method is:" + cat.isCollarStatus());
        cat.setCollarStatus(true);
        System.out.println("New value of \"collarStatus\" in \"adopt\" method is:" + cat.isCollarStatus());
    }
    public static void main(String[] args) {
        Cat kitty = new Cat();
        System.out.println("Old value of \"collarStatus\" in \"main\" method is:" + kitty.isCollarStatus());
        adopt(kitty);
        System.out.println("New value of \"collarStatus\" in \"main\" method is:" + kitty.isCollarStatus());
    }
}

```

## Methods with a Variable Number of Arguments

Java allows for describing a method that can be called with a different number of arguments of a certain type. These methods are called **variable arity methods** (varargs). Variable arity methods are used when the same action needs to be executed under a different number of values of one type.

The syntax used to describe variable arity methods is shown below.

*Hover your cursor over the info icon to see an explanation of the code.*

<type of returned value> <method name>(<data type> ... <parameter name>) { }

Let's consider several important questions related to method calls.

Click on the dropdown element to learn more.

## Methods with a Variable Number of Arguments

Java allows for describing a method that can be called with a different number of arguments of a certain type. These methods are called **variable arity methods** (varargs). Variable arity methods are used when the same action needs to be executed under a different number of values of one type.

The syntax used to describe variable arity methods is shown below.

*Hover your cursor over the info icon to see an explanation of the code.*

<type of returned value> <method name>(<data type> ... <parameter name>) { }

Let's consider several important questions related to method calls.

*Click on the dropdown element to learn more.*

## WHAT NO OF PARAMETERS ARE POSSIBLE?

- A method may be called without parameters.
- A method may be called with any number of parameters of the same type.

## HOW CAN I SPECIFY PARAMETER?

- You can list parameters of the same type by separating them with commas.
- Create a one-dimensional array of the same type as the variable arity parameter and specify a reference to this parameter as the method parameter.
- If a method needs data of a different type along with the variable arity parameter, this parameter is specified last in the list of method parameters.

## HOW CAN I IMPLEMENT VARIABLE ARITY PARAMETER?

Inside the method body, a variable arity parameter (varargs) is treated as an array.

```
public class VarArg {
    public int calcSum(int... values) {
        int res = 0;
        for (int x : values) {
            res += x;
        }
        return res;
    }
}

public class TestArgVar {
    public static void main(String[] args) {
        VarArg tstvarg = new VarArg();
        System.out.println(tstvarg.calcSum());
        System.out.println(tstvarg.calcSum(3));
        System.out.println(tstvarg.calcSum(55, 66));
        System.out.println(tstvarg.calcSum(77, 55, 33, 11, 99));
    }
}
```

### Output:

0

3

121

275

Note the technical aspect: Pay close attention to the specifics of using methods with a variable number of arguments. A method can be declared as **final**. Take a look at the specifics of this action.

## Conclusion

In this lesson, you have seen that:

- Methods are used to perform operations with class data and give access to this data.

- When describing a method, certain rules must be followed.
- Calling a method means transferring control to the method for execution.
- Java allows for describing a method that may be called with a varying number of arguments of a certain type. These methods are called variable arity methods.

## Check Your Knowledge!

1. Can you change initial data if it was passed to a method by value?

Yes

No

Answer

Correct:

Data can't be changed if it was passed to a method by value.

2. Which set of arguments can you use to call the following variable arity method?

`public int sum (int... args) { }`

Without arguments

One argument of the type int

Multiple arguments of the type int

correct

Answer

Correct:

All the options are correct.

3. Which method descriptions will be compiled without errors?

`void doIt () { }`

`String doIt() { return "Hello!"; }`

`int doIt() { return 5; }`

`null doIt() { }`

correct

Answer

Correct:

`doThat (String s, int i) { }` is incorrect since in Java a value returned by a method should be a data type or void.

4. Which method descriptions will be compiled without errors?

`doThat (String s, int i) { }`

`void doThat (String s) {}`

`byte doThat () { return 1; }`

`double doThat (int i1, int i2) { return 1.4; }`

correct

Answer

Correct:

`doThat (String s, int i) { }` is incorrect since in Java a value returned by a method should be a data type or void.

## Introduction

In this lesson, you will continue exploring methods. You will learn about the keyword **this**, methods for accessing closed class fields, and static methods.

### The Keyword **this**

As you already know, you can create several objects of the same type. Each object will contain its own copy of fields with its own values. However, methods are not duplicated, which raises a question: How can a method understand with which set of field values (objects) it is being executed at a given time? In other words, a method needs information on the object that called it.

The keyword **this** is a reference to the current object inside a class, that is, the object for which a method or constructor is being executed. The word **this** is used to access any member of the current object inside the instance method or constructor.

For example, suppose you have the class **Car**. This class has a field with the name **speed**, and the method **setSpeed()** has a parameter with the name **speed**. Inside this method, the parameter overlaps with the field's visibility scope. To signify a reference to the current object, the reference **this** is added before it (accessing the current object's field).

```
public class Car {
    private String model;
    private int year;
    private int speed;
    public void setSpeed(int speed) {
        this.speed = speed;
    }
}
```



The most common reason for using the keyword **this** is the need to differentiate between the fields and local variables/parameters if they have identical names.

### Methods for Accessing Closed Class Fields

There are special methods for accessing closed class fields:

A **getter** is a method for receiving the value of a field. The method name starts with the verb **get** and then follows the name of the field in camelCase style.

A **setter** is a method used to set the value of a field. The method name starts with the verb **set** and then follows the name of the field in camelCase style.

Consider the following examples.

*Click on the tabs to see the examples.*

### USING THE KEYWORD THIS

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Student {
    private String firstName;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    // ...
}
```

### WITHOUT USING THE KEYWORD THIS

*Hover your cursor over the info icon to see an explanation of the code.*

```
public void setFirstName(String name) {
    firstName = name;
}
```

In short, two methods are used to access closed class fields: getters and setters. They can be used with or without the keyword **this**.

## Static Methods

In addition to regular methods, classes can also have static methods. Let's take a look at what is special about these methods and how to use them.

## Conclusion

To sum up, in this lesson you discovered that:

- The keyword **this** is a reference to the current object inside a class, that is, the object for which a method or constructor is being executed.
- There are special methods for accessing closed class fields: A **getter** is a method used to receive the value of a field, while a **setter** is a method used to set the value of a field.
- Static methods are distinguished by their special behavior, status, tools, and terms of use.

## Check Your Knowledge!

1. Which statements about the keyword *this* are true?

**It can be used as a reference to an instance inside a class.**

**The keyword this can be passed as an argument when calling a method.**

**It can be passed as an argument when calling a constructor.**

The keyword this is a reference to an instance that must be initialized explicitly in the class methods.

correct

Answer

Correct:

The keyword this can be used as a reference to an instance inside a class and can be passed as an argument when calling a method or constructor.

2. Which method is used to get a field value?

**Getter correct**

Setter

Answer

Correct:

A getter is a method used to receive the value of a field.

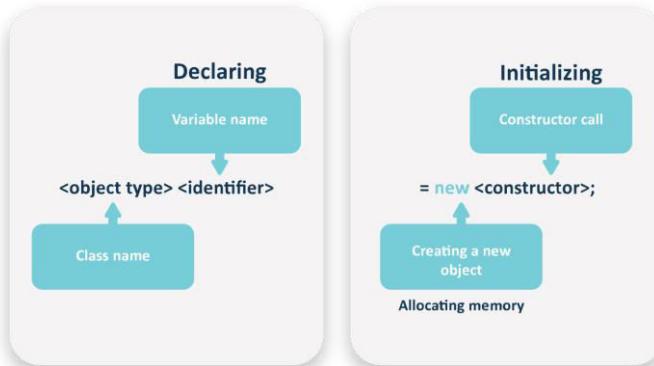
# Creating Objects

## Introduction

In this lesson, you will study how to create objects in Java and explore the specifics of using the literal expression `null` when creating objects.

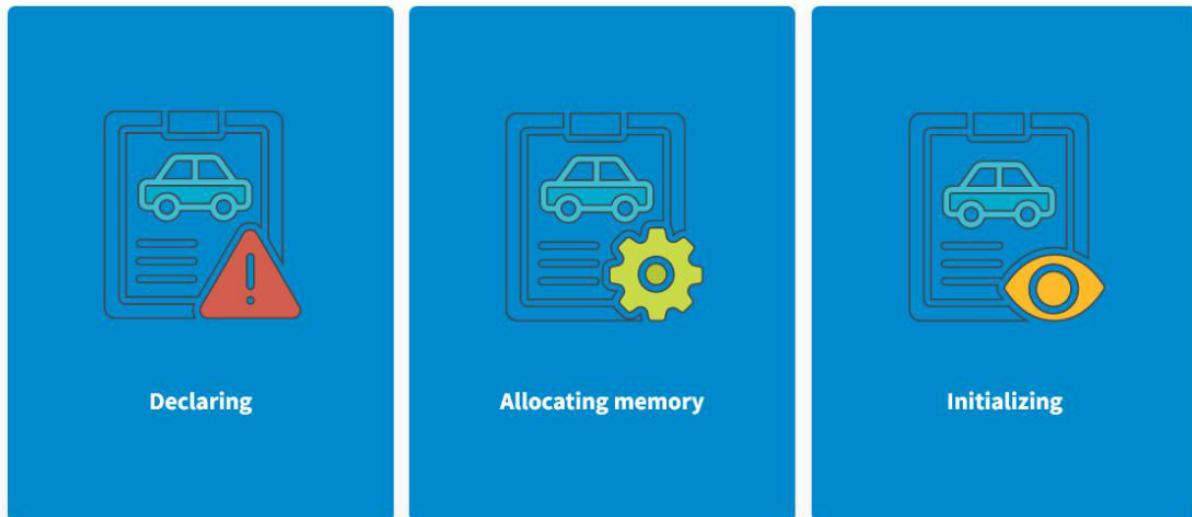
## Creating Objects

As was mentioned in one of the previous lessons, the Java runtime system allocates memory to store an object when you request it using the operator `new`. Thus, it is necessary to call a constructor after the operator—a special method for initializing the object (which will be discussed later). Creating an object in Java looks like the following:



As you can see in the image, creating an object involves three steps: declaring, allocating memory, and initializing. Let's review each of these steps in detail.

*Click on each card to learn more.*



Linking the name of a variable with the object type—that is, creating a reference variable to store the memory address where the object is placed.

A request to allocate memory to store the object (which is always done using the operator/keyword `new`).

Calling a constructor whose task it is to initialize the object being created (which comes paired with `new`).

Now is a great time to review an example of describing the class `Car` and creating objects with it. The class contains a declaration of the field `carModel` and the method `getCarModel()` to access it. When the keyword `new` is used, the system allocates the necessary amount of memory for the new object in a "heap." Then it calls a constructor to initialize the object, or set its fields to the initial values. After that, the fields and methods of the object become accessible through the received reference to the object.

```
public class Car {
    private String carModel;
    public String getCarModel() {
        return carModel;
    }
}
```

```

}
public class Demo {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car();
        Car car3 = new Car();
        String name = car1.getCarModel();
        System.out.println(name);
    }
}
  
```

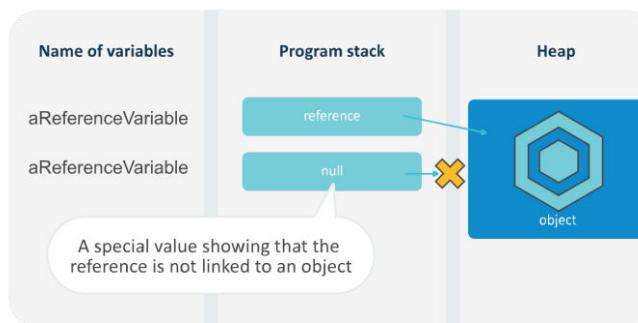
So far, you have analyzed an example of creating an object. But what will happen if the reference variable is not linked to an object during the declaration process? Let's take a closer look at this.

## Using the Literal Expression **null**

If a reference variable is not linked to an object, it has a null/empty value:

- The null value is determined by the literal expression **null**.
- The literal expression **null** can always be assigned/ascribed or reduced to any reference type.

Visually, you can represent this as follows:



Let's take a look at the actions that can be performed with the literal expression **null**.

### REDUCING

- The literal expression **null** can be assigned to a variable of any reference type.
- The literal expression **null** is compatible with any reference type.

For example:

```

String str1 = null;
Car car1 = null;
String str2 = (String)null;
Car car2 = (Car)null;
  
```

### OPERATIONS

- The literal expression **null** can be compared with the values of reference variables by using the operators **==** and **!=**.
- The literal expression **null** cannot be used with other relation operators like **<** or **>**.

For example:

```

System.out.println(null == null);
System.out.println(car1 == null);
System.out.println(car1 != car2);
  
```

### CALLING METHODS

When calling an instance method using a reference variable with the value **null**, an execution error will be thrown - **NullPointerException**!

For example:

```

car1 = null;
car1.getCarModel();
  
```

## Conclusion

In this lesson, you have seen that:

- Objects in Java are stored in a memory area called a "heap", which you cannot manage by yourself.
- Creating an object involves three steps: declaring, allocating memory, and initializing.
- If a reference variable is not linked to an object, it has a null value defined by the literal statement **null**.

In addition, you explored the actions that can be performed with the literal expression **null**.

## Check Your Knowledge!

1. Choose the actual characteristics of the **null** literal.

**Null can be cast to any reference type.**

**Null** can be used with any relational operators.

**When invoking an instance method using a reference variable with the null value, a runtime error will be thrown.**

correct

Answer

Correct:

Great job! The null literal can be cast to any reference type. It can be compared with the values of reference variables using the == and != operators. When you call an instance method through a reference variable with a null value, the runtime error NullPointerException will occur.

2. What will be the result of executing the following code?

```
public class Employee {
    String name;
    int age;
    public Employee() {}
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee();
        System.out.println(e.name);
    }
}
```

Nothing will be printed

**null**

0

A compilation error

Answer

Correct:

The console output will be the value null since the field name has not been initialized with any value. However, the class field will receive the default value (null) for the reference type.

3. What will be the result of executing the following code?

```
public class Employee {
    String name;
    int age;
    public Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee("Alex", 30);
        System.out.println(e.name);
    }
}
```

```
 }  
}  
Nothing will be printed  
null
```

**Alex**

A compilation error

Answer

Correct:

With the help of a constructor, the name field was initialized by the string Alex, and the age field with a value of 30.

# Constructors

## Introduction

This lesson will provide an in-depth overview of class constructors. You will find out what they are used for, what types of constructors exist, as well as the specifics of applying them.

## What Are Constructors?

**Constructors** are special methods that initialize fields with initial values and implicitly return an object of the class where they are defined. Constructors accept data used to initialize an object.

Constructors are used to initialize an object when it is created. As a rule, they are used to specify initial values for an object's fields specified in a class or to perform any other actions required to create a fully formed object.

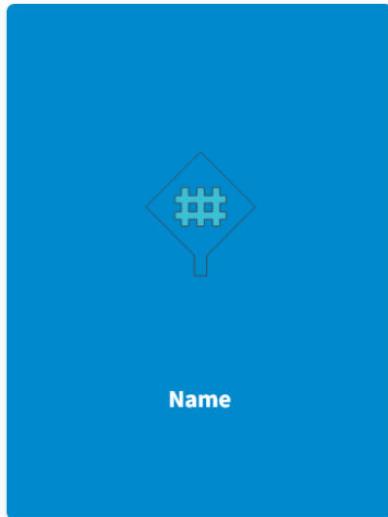
Java provides constructors automatically. Thus, all classes have constructors, regardless of whether you define them in the class body explicitly. A constructor initializes all fields of an instance/object with its default values:

- For most data types, the default value is zero.
- For the Boolean type, the default value is **false**.
- For the reference type, the value is **null**.

What properties are characteristic of constructors?



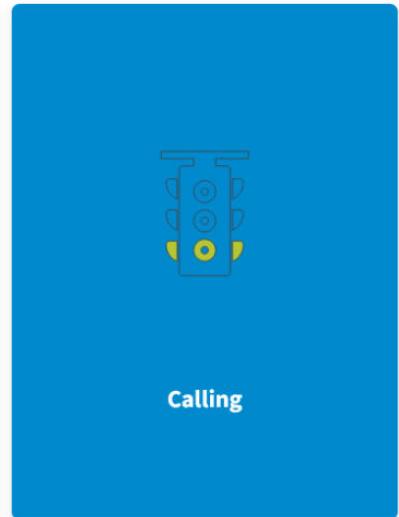
*Click on the cards to see the features of constructors*



Name



Returned value



Calling

A constructor's name matches the class name, including the letter case. In terms of syntax, a constructor is similar to a method without a returned value.

A constructor has no explicitly specified type of returned value since it always returns a reference of its own class.

Outside of a class, a constructor can only be called using the operator **new**.

Note that if you specify a type of returned value when describing a constructor, Java will classify it as a class method, not a constructor—for example, **void Car(String carModel) { }**

Look at the following example of using a constructor.

**O** Click on the dropdown element to study the example.

**H** Hover your cursor over the info icon to see an explanation of the code.

```
public class Car {
    private String carModel;
```

```

public Car(String carModel) {
    this.carModel = carModel;
}
public String getCarModel() {
    return carModel;
}
}

public class DemoConstructor {
    public static void main(String[] args) {
        Car car1 = new Car("Audi");
        Car car2 = new Car("BMW");
        Car car3 = new Car("Bentley");
        System.out.println(car1.getCarModel());
        System.out.println(car2.getCarModel());
        System.out.println(car3.getCarModel());
    }
}

```

**Output:**

Audi

BMW

Bentley

Thus, constructors are special methods defined in a class. They initialize and return the object of the class where they are defined. All classes have constructors.

**Types of Constructors**

There are two types of constructors: without parameters (constructors by default) and with parameters (user constructors).

*Click on the dropdown lists to learn more.*

**CONSTRUCTORS WITHOUT PARAMETERS**

A **constructor without parameters** is a constructor that has no parameters and is used to initialize the fields of an instance with zero values or certain predefined/initial values. If a class contains no explicit description of a constructor, Java will create a constructor without parameters that initializes the fields of an instance with zero values of its type.

For example, the class **Car** does not have a constructor description, so Java will add a constructor of the type **public Car() { }**



*Hover your cursor over the info icon to see an explanation of the code.*

```

public class Car {
    private String modelCar;
    public String getModelCar() {
        return modelCar;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        System.out.println(car.getModelCar());
    }
}

```

**Output:**

Null

## Constructors with parameters

In some cases, it is enough to have a default constructor in a class. However, a constructor often has to accept one or several parameters to initialize an object. Parameters are passed to a constructor the same way they are passed to a method. It is enough to declare them in round brackets after the name of the constructor.



One class can have several constructors with different sets of parameters for different initial states of an object (different object representation). That is, constructors, just like methods, can be overloaded. For example, in the following class description, you can observe several constructors:

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    private String model;
    private String brand;
    public Car() {}
    public Car(String model) {
        this.model = model;
    }
    public Car(String model, String brand) {
        this.model = model;
        this.brand = brand;
    }
    // getters and setters
}
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Camry");
        Car car3 = new Car("Camry", "Toyota");
        System.out.println(car1.getModel() + " " + car1.getBrand());
        System.out.println(car2.getModel() + " " + car2.getBrand());
        System.out.println(car3.getModel() + " " + car3.getBrand());
    }
}
```

### Output:

null null  
Camry null  
Camry Toyota

If a class contains an explicitly described constructor with parameters, Java **will not create** a default constructor. This means that calling such a constructor will lead to a compilation error. To use it, you have to describe this constructor explicitly as a constructor without parameters!

 You can use any access modifier in the description of the constructor. If a constructor in a class has **private** access, it may not be accessed from anywhere outside of the class.

 Take a look at the example below, which demonstrates the essence of constructor visibility.



## EXAMPLE

In the description of the class **Car**, the constructor with two parameters is private. If you try to create an object using this constructor, you will get a compilation error.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    private String model;
    private String brand;
    public Car() {}
    public Car(String model) {
        this.model = model;
    }
    private Car(String model, String brand) {
        this.model = model;
        this.brand = brand;
    }
    // getters and setters
}
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Camry");
        Car car3 = new Car("Camry", "Toyota");
    }
}
```

Private constructors may be called from other constructors or static methods of the same class.

Any constructor can refer to another constructor in the same class using the keyword **this**:

- This is done without using the operator **new**.
- Instead of the class name, use the keyword **this**.
- Calling another constructor should be the first action in the body of that constructor.



Let's look at two examples demonstrating how one constructor is called from another one.

*Click on the dropdown list to study the examples*

### Calling a constructor

In the description of the class **Car**, a constructor without parameters and a constructor with one parameter call a constructor with two parameters. This syntax prevents duplication of the code initializing the instance fields.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
```

```

private String model;
private String brand;
public Car() {
    this("Camry", "Toyota");
    System.out.println("Init");
}
public Car(String model) {
    this(model, "Toyota");
}
public Car(String model, String brand) {
    this.model = model;
    this.brand = brand;
}
}

```

## Accessing a constructor

Below you can see an example of incorrect access to one constructor from another constructor in the same class:

*Hover your cursor over the info icon to see an explanation of the code.*

```

public class Car {
    private String model;
    private String brand;
    public Car() {
        System.out.println("Init");
        this("Camry", "Toyota");
    }
    public Car(String model) {
        this(model, "Toyota");
    }
    public Car(String model, String brand) {
        this.model = model;
        this.brand = brand;
    }
}

```

## Conclusion

In this lesson, you have seen that:

- Constructors are special methods defined in a class that initialize and return objects of the class where they are defined.
- All classes have constructors.
- There are two types of constructors: without parameters (default constructors) and with parameters (user-defined constructors).

## Check Your Knowledge!

1. Which of the following statements about defining and using overloaded constructors are true?

**Overloaded constructors should be defined using different lists of parameters.**

Overloaded constructors can only be defined by changing the access modifier.

**Overloaded constructors can have different access modifiers.**

**A constructor can call another constructor using the keyword this.**

A constructor can call another constructor using the name of its class.

correct

Answer

Correct:

Overloaded constructors can be defined using different lists of parameters, and they can also have different access modifiers. In addition, a constructor can call another constructor using the keyword this.

2. Does the following class have a constructor with parameters?

```

public class Employee {
    String name;
}

```

```

int age;
public void Employee(String newName, int newAge) {
    name = newName;
    age = newAge;
}
}

```

**Yes****No**

Answer

Correct:

The declaration **public void** Employee(String newName, **int** newAge) is a declaration of a usual method because it has a value to be returned, but constructors do not have returned values.

3. Does the following class have a constructor?

```

public class Employee {
    String name;
    int age;
}

```

**Yes****No**

Answer

Correct:

If a class has no declared constructor, the compiler will always generate a default constructor without parameters.

4. Does the following class have a default constructor?

```

public class Employee {
    String name;
    int age;
    public Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

```

**Yes****No**

Answer

Correct:

The class has an explicitly declared constructor; a default constructor will not be generated.

5. What will be the result of executing the following code?

```

public class Employee {
    String name;
    int age;
    public Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

```

**public class** Main {

**public static void** main(**String**[] args) {

    Employee e = **new** Employee();

    System.out.println(e.name);

}

115 }

Nothing will be printed

null

Alex

**A compilation error**

Answer

Correct:

In the method main(), a constructor is being called without parameters of the class Employee. While in the class, the constructor is declared with parameters. The compiler does not find a constructor without parameters and throws an error. There would be no error if the class were declared without constructors altogether. In that case, the compiler would create a default constructor without parameters.

# Initializers

## Introduction

In this lesson, you will become familiar with the process of initializing classes and objects and learn what initializers exist and how to use them in real life. You will also study the specifics of initializing fields of the type **final**, as well as the process of destructing objects.

## Initializers

**Initializers** are special class structures used to initialize class fields (static) and instance fields. Class initialization starts with loading its description into random access memory.

The initialization process involves several steps:

- Automatic initialization of class fields
- Explicit initializers of class fields
- Static initializers
- Dynamic initializers/constructors

Study each step of initialization in detail.

### Automatic initialization of class fields

The first step of initializing a class is **automatic initialization** with default values. For example, the default value for the type **char** is the symbol '\u0000'. Look at an example of a class description with several static fields of various types. You will leave them without explicit initialization and display their values in the console.

```
public class InitDemo1 {
    private static char ch;
    private static boolean bb;
    private static byte by;
    private static int ii;
    private static float ff;
    private static String str;
    private static int[] array;
    public static void main(String[] args) {
        System.out.println("char: " + ch);
        System.out.println("boolean: " + bb);
        //...
    }
}
```

### Output:

char:  
boolean: false  
byte: 0  
int: 0  
float: 0.0  
String: null  
Array: null

If you only declare static fields, they will be initialized with default values, and you can use them.

### Explicit initializers of class fields

The second step involves **explicit initializers** of class fields with their initial values. By the way, each class field can be initialized explicitly with a value if this initialization can be written as one line. Study an example of a class

description with several static fields of various types. You will initialize them when declaring with literal expressions of the appropriate type and display their values in the console.

```
public class InitDemo2 {
    private static char ch = 'A';
    private static boolean bb = true;
    private static byte by = -56;
    private static int ii = 1000;
    private static float ff = 1.25e-2F;
    private static String str = "Data";
    private static int[] array = {1, 2, 3, 4};
    public static void main(String[] arg) {
        System.out.println("char: " + ch);
        System.out.println("boolean: " + bb);
        //...
    }
}
```

### Output:

char: A  
 boolean: true  
 byte: -56  
 int: 1000  
 float: 1.25e-2  
 String: Data  
 Array: [1, 2, 3, 4]

The Java compiler automatically generates the class initialization method (an internal method with the name <clinit>) for each class:

The method is guaranteed to be called only once when the class is first used or mentioned.

Initialization expressions for class fields are inserted into the class initialization method in the order in which they appear in the source code. As a result, previously declared fields of this class can be used in the initialization expression of a class field.

```
public class InitDemo3 {
    //...
    private static byte by = 17;
    private static int ii = 24 * by;

    //...
}
```

In the initialization expression of a class field, you can access a static method. The advantage is that you can reuse this method's code as needed.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class InitDemo4 {
    private static int ii = initSt();

    //...
    private static int initSt() {
        System.out.println("Init ii value");
        return 1000;
    }
    //...
    public static void main(String[] arg) {
        System.out.println("Main");
        System.out.println("int: " + ii);
    }
}
```

}

**Output:**

Init ii value

Main

int: 1000

Please note: If a class is initialized with an entry point into a Java program, the method **main()** will be executed after the class fields are initialized.

**Static initializers**

The third step is **static initializers**. They are used when a certain logic is required, and it is assumed that they will only be used once (for example, handling errors or loops to fill complex data sets).

Constraints:

- The operator **return** cannot be used inside static initializers.
- The keyword **this** cannot be used inside static initializers.
- You cannot refer to an instance field from static initializers.

The Java compiler places the code of a static initializer in the method **cinit()** after the class fields are initialized with statements. In the next example, a static initializer is used to initialize an array of symbols with a letter of the Latin alphabet (This has to be done only once; thus, there is no need to describe the method.):

*Hover your cursor over the info icon to see an explanation of the code.*

```
import java.util.Arrays;
public class InitDemo5 {
    private static char[] alph;
    static {
        alph = new char[26];
        int i = 0;
        for (char c = 'a'; i < alph.length; c++, i++) {
            alph[i] = c;
        }
    }
    public static void main(String[] args) {
        System.out.print(Arrays.toString(alph));
    }
}
```

Static class initialization is now complete, and the class can be used to create objects.

It is necessary to point out the following properties:

- A class can have any number of static initializers.
- They can occur anywhere in the class body.
- The runtime system assures that static initializers are called in the order in which they appear in the source code.
- A static initializer is executed once when the class is initialized or loaded for the first time.

**Dynamic initializers**

The fourth step involves **dynamic initializers**, which can serve as an alternative to class constructors for initializing instance fields. They are simply described in the class body, outside of any other blocks.

- They can be used to separate blocks of code between several constructors.
- The Java compiler inserts the code of the dynamic initializers into each constructor (at the beginning of its body) in the order in which they are described in the class body.

Dynamic initializers have the following syntax:

```
{ /* Any code needed for initialization */ }
```

Review a description of the class **Car** with several constructors:

- In each constructor, the value of the class field **numOfCars** is increased by 1 (the objects of this class are counted)—as shown in the example. In this case, the code is duplicated.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    private static int numOfCars;
    //...
    public Car() {
        //...
        numOfCars++;
    }
    public Car(String model) {
        //...
        numOfCars++;
    }
}
```

To avoid duplication, you should move the common code from the constructors to the dynamic initializer, as shown in the example.

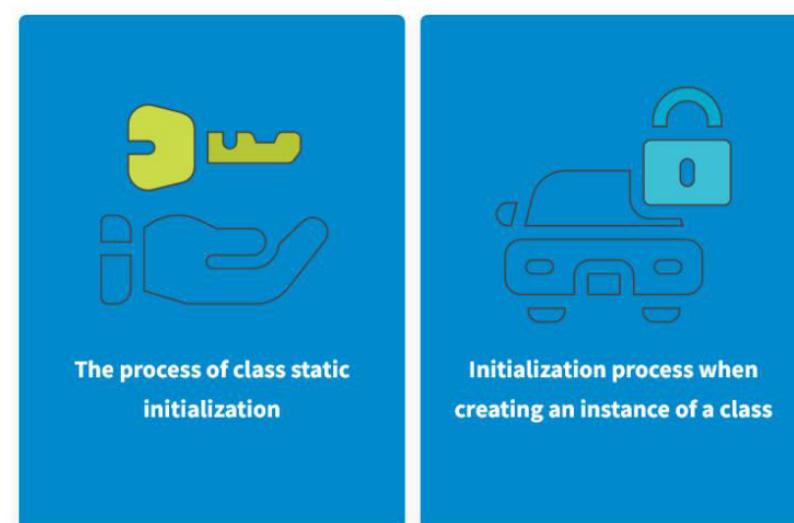
*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    private static int numOfCars;
    //...
    {
        numOfCars++;
    }
    public Car() {
        //...
    }
    public Car(String model) {
        //...
    }
}
```

Dynamic initializers are only executed when a class object is created. Thus, dynamic initializers can be used to set the initial state of an object.

Note that there are two options for the initialization process.

*Click on the cards to learn more.*



1. Initialization of static fields with default values.
  2. Initialization of static fields with statements.
  3. Execution of static initializers.
  4. If a startup command specifies a class to run, its **main** method is executed. This method becomes an entry point to the program.
1. Recursive call and execution of the superclass constructors—parent classes.
  2. Initializing the fields of the class instance with default values or initial values.
  3. Execution of dynamic initializers.
  4. Execution of the class constructor body.

Pay special attention to the example of the class **InitDemo6**, where different initializers for class fields and instance fields are described to demonstrate the process of executing them.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class InitDemo6 {
    private int a = 5;
    private static int b = 100;
    {
        a = -5;
        System.out.println("Dynamic initialization section");
    }
    public InitDemo6() {
        a = 10;
        b = 10;
        System.out.println("Constructor");
    }
    static {
        b = -5;
        System.out.println("Static initialization section");
    }
    public static void main(String[] arg) {
        System.out.println("Main");
        InitDemo6 obj = new InitDemo6();
        System.out.println("a=" + obj.a + " b=" + b);
    }
}
```

### Output:

Static initialization section

Main

Dynamic initialization section

Constructor

a=10 b=10

So far, you have learned that initializers are special class structures used to initialize static class fields and instance fields. You have also studied the main steps in initialization, the process of class initialization, and the process of initialization when creating a class instance. Now let's move on to the next topic regarding the initialization of final fields.

## initialization of final Fields

Fields marked by the keyword **final** are used to describe read-only properties of a class or object. As you know, after initialization, you cannot change the value of a **final** variable. Thus, this field should be initialized in one of three ways:

- In the same line where it is declared
- In each constructor
- In one of the dynamic initializers

This is because a **final** field can only be initialized once.

Below you will find some examples of correct and incorrect initialization of **final** fields.  
*Click on each tab to see the examples.*

## Correct initialization

This example shows correct initialization of fields.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class InitDemo7 {
    private final int XX = 50;
    private final int ZZ;
    private final int YY;
    {
        ZZ = 20;
        System.out.println("Dynamic section");
    }
    public InitDemo7() {
        YY = 30;
        System.out.println("Constructor");
    }
    public static void main(String[] arg) {
        System.out.println("Main");
        InitDemo7 obj = new InitDemo7();
    }
}
```

In the example, you can see the specifics of initializing **final** fields.

## Incorrect initialization

The next example shows incorrect initialization of a **final** field. Here, initialization is specified in several places. According to the initialization rule, after the value is assigned once (here the field *XX* is initialized at declaration—an explicit initializer), it can be changed neither in the dynamic initializer nor in the constructor.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class InitDemo8 {
    private final int XX = 50;
    {
        XX = 20;
        System.out.println("Dynamic section");
    }
    public InitDemo8() {
        XX = 30;
        System.out.println("Constructor");
    }
    public static void main(String[] arg) {
        System.out.println("Main");
        InitDemo8 obj = new InitDemo8();
    }
}
```

In the example, you can see the specifics of initializing **final** fields.

## Initialization error

This example demonstrates an initialization error—the **final** field must be initialized explicitly. Unlike with other fields, no default value is set for the **final** field.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class InitDemo9 {
    private final int XX;
    {
        System.out.println("Dynamic section");
    }
    public InitDemo9() {
```

```

        System.out.println("Constructor");
    }
public static void main(String[] arg) {
    System.out.println("Main");
    InitDemo9 obj = new InitDemo9();
}
}

```

As you have seen, an instance **final** field must be initialized, but only in one place: where it is declared, in the dynamic initializer, or in the constructor.

## Destructuring Objects

In Java, there are no special methods or statements for deleting class objects—destructors. Thus, the developer does not control this process. Unused objects are destroyed by a special JVM component called a *garbage collector*.

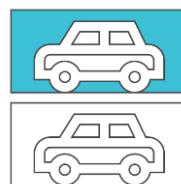
Below you can explore the process of destructuring objects and garbage collection.

*Click on each dropdown list to learn more.*

### Memory distribution

As you know, when using the **new** operator, memory for the objects created is allocated dynamically from the pool of free random-access memory (RAM). RAM is limited, and sooner or later free memory is exhausted. This may make it impossible to execute the **new** operator.

For this reason, one of the main functions of any dynamic memory allocation system is timely cleaning of unused objects.



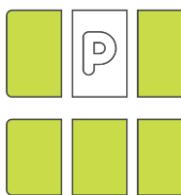
### Memory deallocation

In many programming languages, previously allocated memory is deallocated manually. For example, the language C++ uses the operator **delete** for this purpose. But Java uses an approach called garbage collection.

Garbage collection deallocates memory automatically and works as follows: If there are no references to an object from the program stack, this object is classified as unnecessary, and the memory it occupies is returned to the pool of free memory. This deallocated memory can be allocated to other objects later.

Garbage collection is only done periodically while the program is being executed. It is not performed as soon as unused objects are detected. Usually, to avoid a decline in performance, garbage collection is performed only if two conditions are satisfied: if there are objects that have to be deleted and if there is a need to deallocate the memory taken up by them.

Keep in mind that garbage collection takes time. The Java runtime system is guided by the principle of expediency when performing this operation. Therefore, you can never know for sure when garbage will be collected.



### The **finalize** method

In Java, you can define a finalizer method—*finalize()*—that will be called before deleting an object from the memory for good. This allows you to make sure the object can be deleted without consequences.

To add the finalizer to a class, it is enough to redefine the method *finalize()* inherited from the class **Object**. The Java runtime system will call this method before actually deleting the object. In the body of the method *finalize()*, you should indicate actions that have to be performed before actually deleting the object.

*Hover your cursor over the info icon to see an explanation of the code.*



```
protected void finalize() {  
}
```

Please note:

- The keyword **protected** prevents access to the method from outside of this class.
- The method is called when the garbage collector decides to deallocate the memory taken up by this object.
- If at the time the object is beyond the scope of visibility, the method will not be called. For example, if the program completes before garbage collection is launched, the method will not be called.

With all this in mind, the performance of the program must not depend on whether the method *finalize()* is called. Other tools are used to release the resources the program no longer needs.

As you have seen, unused objects are destroyed by a special JVM component called a garbage collector.

## Conclusion

In this lesson, you have found out that:

- Initializers are special class structures used to initialize class fields (static) and instance fields.
- A **final** instance field must be initialized, but only in one place: where it is declared, in a dynamic initializer, or in a constructor.
- Unused objects are destroyed by a special JVM component called a garbage collector.

## Check Your Knowledge!

1. Which statements about initializers are true?

**Initializers of one type are executed in the order in which they are described in the code.**

Dynamic initializers are launched once, when a class is first loaded to memory.

Static initializers are launched every time a class instance is created.

Answer

Correct:

Initializers of the same type are executed in the order in which they are described in the code.

2. Will the following code be compiled?

```
public class Boo {  
    int i;  
    static {  
        i = 5;  
    }  
}
```

Yes

No

Answer

Correct:

Non-static class fields cannot be accessed from a static initializer.

3. Will the following code be compiled?

```
public class Boo {  
    static int i;  
    {  
        i = 5;  
    }  
}
```

Answer

Correct:

Static class fields can be accessed from a dynamic initializer.

4. Will the following code be compiled?

```
public class Boo {
```

```
    final byte b;
```

```
{
```

```
    b = 10;
```

```
}
```

```
}
```

**Yes**

No

Answer

Correct:

Initializing the final field is mandatory.

5. Will the following code be compiled?

```
public class Boo {
```

```
    final byte b;
```

```
}
```

**Yes**

**No**

Answer

Correct:

A field of the type final cannot be assigned a default value. It either needs to be initialized explicitly by assigning a value or initialized in a dynamic initializer or constructor—but only once!

# Packages

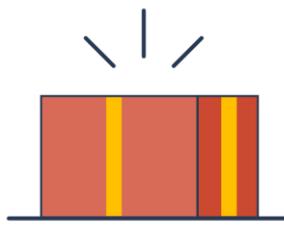
## Introduction

In this lesson, you will explore why packages exist in Java and how to optimize work with packages.

## The Concept of Packages

In Java, **packages** are used to prevent conflicts with names; control access; make searching easier; and find and use classes, interfaces, lists, and annotations.

An ordinary compilation unit that has no package declaration (but has at least one other kind of declaration) is part of an unnamed package. The operator **package** placed at the beginning of the source program file defines a named package. In other words, it defines a namespace for the classes contained in this file. The action of the package **operator** designates the location of a file relative to the project root. For example:



`package com.epam.eun.entity;`

Thus, the program file will be placed in a subdirectory with the name **com.epam.eun.entity**. When this package is accessed from another package, the package name should be attached to the class name **com.epam.eun.entity.Student**.

In projects, packages are named as follows:

*Click on the + signs to see a more detailed description of each component.*

com. epam. eun. entity. Student

### com.epam

This is the reverse of the software manufacturer or customer's internet domain name—for example, for epam.com, you will get com.epam.

### eun

This is the project name (which is usually abbreviated)—for example, eun.

### entity.Student

These packages define the application.

The general form of the file containing Java source code may be as follows:

- A single operator package (optional but highly recommended)
- Any number of import operators (optional)
- A single public class (optional)
- Any number of package classes (optional and not recommended)

When using classes, the complete package name of the package this class belongs to should be specified before the class name and separated by a period.

## Arrangement of packages

Below you can find a non-exhaustive list of packages of a real application. From the package names, you can determine which classes they contain without looking inside. When creating a package, you should always be guided

by a simple rule: Use a simple name that reflects the meaning, behavior logic, and functionality of the classes consolidated in it.

```
com.epam.eun
com.epam.eun.administration.type
com.epam.eun.administration.dbhelper
com.epam.eun.common.type
com.epam.eun.common.dbhelper.annboard
com.epam.eun.common.dbhelper.course
com.epam.eun.common.dbhelper.guestbook
com.epam.eun.common.dbhelper.learnresult
com.epam.eun.common.dbhelper.message
com.epam.eun.common.dbhelper.news
com.epam.eun.common.dbhelper.prepinfo
com.epam.eun.common.dbhelper.statistic
com.epam.eun.common.dbhelper.subjectmark
com.epam.eun.common.dbhelper.subject
com.epam.eun.common.dbhelper.test
com.epam.eun.common.dbhelper.user
com.epam.eun.common.menu
com.epam.eun.common.object
com.epam.eun.common.servlet
com.epam.eun.common.tool
com.epam.eun.consultation.type
com.epam.eun.consultation.dbhelper
com.epam.eun.consultation.object
com.epam.eun.core.type
com.epam.eun.core.dbhelper
com.epam.eun.core.exception
com.epam.eun.core.filter
com.epam.eun.core.manager
com.epam.eun.core.taglib
```

### application of packages

Each class is added to the package specified during compilation. For example:

```
package com.epam.eun.object;
public class CommonObject {
    // more code
}
```

In short, any Java class belongs to a certain package. The operator **package** placed at the beginning of the source program file defines a named package.

### import

A class starts from the indication that it belongs to a package; for example, the class **CommonObject** belongs to the package **com.epam.eun.object**. This means that the file **CommonObject.java** is located in the directory **object**, which is located in the directory **eun**, and so on. You cannot change the name of a package without changing the name of the directory where its classes are stored. To access a class from another package, the package name is specified before the name of that class: **com.epam.eun.object.CommonObject**.

The keyword **import** will help avoid such long names when creating class objects.



For example:

```
import com.epam.eun.object.CommonObject;
```

```
import com.epam.eun.object.*;
```

In the second option, the entire package is imported, which means all the classes of the imported package can be accessed. It is not recursive, though, so it will not import sub-packages or classes in them. In practical programming, it is recommended to use the individual class **import** so that it is possible to quickly determine the location of the class being used when analyzing code.

You can access a class from another package using one more method, but it is not recommended.

*Click on the dropdown element to study the example.*

## Using a complete package name during inheritance

This entry is used when a class needs access to classes with identical names.

```
package com.epam.eun.usermng;
public class UserStatistic extends com.epam.eun.object.CommonObject {
    // more code
}
```

When importing a class from another package, it is always best to specify the complete path and the name of the imported class. In large projects, this makes it possible to easily find a class definition when it is necessary to see the class's source code.

*Click on the dropdown element to study the example.*

## Applying a complete path to a class during import

```
package com.epam.eun.action;
import com.epam.eun.object.CommonObject;
import com.epam.eun.usermng.UserStatistic;
public class CreatorStatistic extends CommonObject {
    public UserStatistic statistic;
}
```

Please, note:

- If a package does not exist, it needs to be created before the first compilation.
- If a package is not specified, a class will be added to an unnamed package; thus, no unnamed directory is created.

However, in real projects, classes are not created outside of packages, and there is no reason to deviate from this rule.

Take a look at the following example, which shows how and why the word **import** is used.

## Static import

Java has language tool that expand the capabilities of the keyword **import**. It is called **static import**. For this purpose, the directive **import** is used with the modifier **static**.

What is the purpose of static import?

*Click on the dropdown element to study the example.*

## Calling static methods and constants

When calling static methods and accessing static constants, you use the class name as the prefix, which makes the code more complex and difficult to comprehend.

```
package by.epam.learn.demo;
public class ImportMain {
    public static void main(String[ ] args) {
        System.out.println(2 * Math.PI * 3);
        System.out.println(Math.floor(Math.cos(Math.PI / 3)));
    }
}
```

## Static import of methods and constants

Static constants and static methods may be used without specifying the class they belong to if you use static import, as shown below.

```
package by.epam.learn.demo;
import static java.lang.Math.*;
public class ImportLuxMain {
    public static void main(String[ ] args) {
```

```

        System.out.println(2 * PI * 3);
        System.out.println(floor(cos(PI / 3)));
    }
}

```

If you need to access only one static constant or method, the import should be performed as follows:

```

import static java.lang.Math.E; //for one constant
import static java.lang.Math.cos; //for one method

```

## Conclusion

In this lesson, you have found out that:

- In Java, **packages** are used to prevent conflicts with names; control access; make searching easier; and find and use classes, interfaces, lists, and annotations.
- The keyword **import** is used to get access to classes and interfaces from other packages and avoid long names when creating class objects.
- Static constants and static methods of a class can be used without specifying the class. This helps avoid code complexity and promotes quick comprehension.

## Check Your Knowledge!

1. Which part of the following package name refers to the name of the project? **com.epam.job.entity.Employee**  
**com.epam**

**job**

**entity.Employee**

**Answer**

**Correct:**

The name of the project in this package is job.

2. True or false? The operator **import** significantly reduces the amount of source code entered.

**True**

**False**

**Answer**

**Correct:**

The operator import reduces the amount of entered source code.

Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.

## Introduction

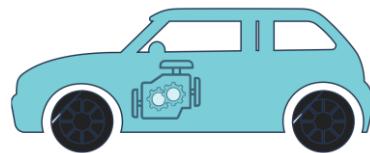
In this lesson, you will explore one of the key principles of OOP—encapsulation. You will also review access-level modifiers and find out how they are related to encapsulation.

### What Is Encapsulation?

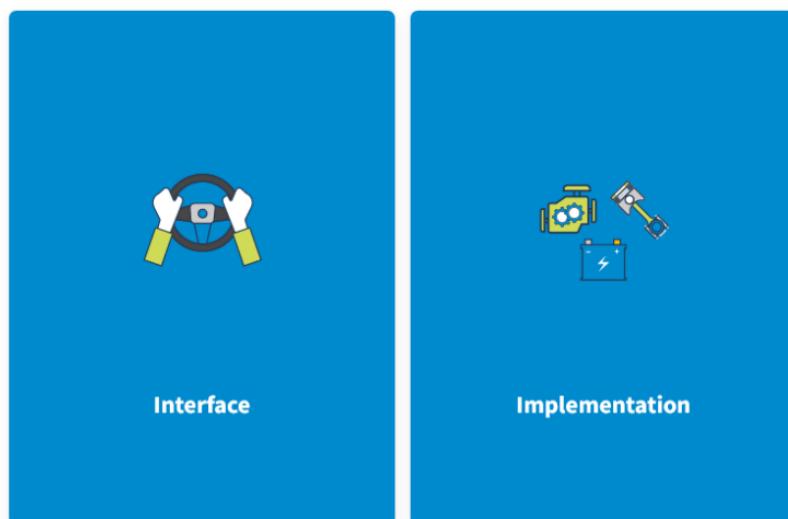
**Encapsulation** is the consolidation of data and methods for working with data in one package, or "capsule," with the possibility of concealing it from the external environment (other objects).

As you already know, data characterizing an object are called **instance variables**, and operations performed with these data are called **instance methods**. In a certain object (class instance), these fields have certain values. The set of field values defines the object's current **state**. Applying any method to an object can change its state.

**What does encapsulation do?** Mainly, it forbids direct access to the fields of this class instance from other classes. Different parts of the program, as well as external programs, can access the object's data only by using the methods of this object. This means that you can change the way data is stored in a class while retaining the methods used to process this data. Thus, other objects will be able to cooperate with the objects of this class just as they did before the changes.



In practice, this means that every class has two sides—**interface** and **implementation**.



The interface includes everything related to the interaction of this object with any other objects.

The implementation means concealing all details not related to the process of the objects' interaction from other objects.

You can observe the principle of encapsulation in everyday life. Let's say you—an individual—are "an instance of the class Human". Everything related to your appearance—your height, build, and eye color—as well as general information such as your first and last name are all known to those around you. However, your internal organs are hidden inside your body, so other objects do not have direct access to them. The same is true for personal data that you would not want to lose or disclose—for example, your ATM card PIN.

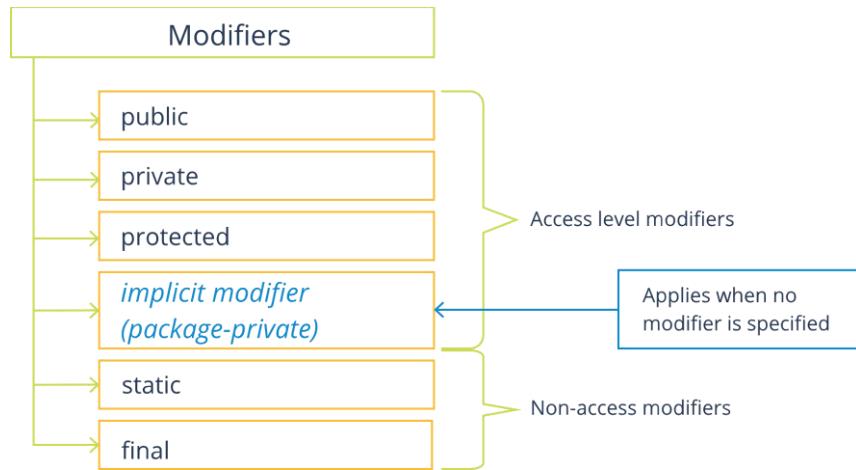
In Java, **access modifiers** are responsible for implementing the principle of encapsulation. These are keywords that regulate the level of access to data (classes, fields, methods and constructors). You will learn more about these later in this lesson.

## Access Modifiers

Access modifiers determine how other classes can use this class, field, or method. All modifiers can be split into two groups:

1. Access modifiers
2. Non-access modifiers

Some of the modifiers in Java are shown in the picture below.



There are two levels of access control: the top level and the member level.

The **top level** includes:

- **public**
- **package-private**

The **member level** includes:

- **public**
- **protected**
- **package-private**
- **private**

Let's take a closer look at access modifiers and how to use each one.

### Public

The modifier **public** may be used for a class, method, constructor, or field:

- It allows access to a class from any Java program.

- All code can access the field, constructor or method.

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    //...
    public int speed;
    //...
    public int getSpeed() {
        return speed;
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        Car fastCar = new Car();
        fastCar.speed = 100;
        System.out.println(fastCar.getSpeed());
    }
}
```

## Private

The modifier **private** can be used only for a class member: a method, constructor, field, or nested class/interface. Thus, the class member can only be accessed in the body of the same class.

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    //...
    private int speed;
    //...
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        this.speed = speed;
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        Car fastCar = new Car();
        fastCar.speed = 100;
        fastCar.setSpeed(100);
        System.out.println(fastCar.getSpeed());
    }
}
```

## Protected

The modifier **protected** can be used for a class member: a method, constructor, field, or nested class/interface. This means that the class member may be accessed only in the body of that same class, a class from the same package, or a successor class, even if it is located in a different package.

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {
    //...
    protected int speed;
    //...
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        this.speed = speed;
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        Car fastCar = new Car();
        fastCar.speed = 100;
        System.out.println(fastCar.getSpeed());
    }
}
```

## Package-private

If the access modifier is not specified, the class or class member will be visible and accessible only in the body of the same class or a class from the same package. In this case, the modifier **package-private** will be set by default.

Let's consider an example of how to use it.

*Hover your cursor over the info icon to see an explanation of the code.*

```
package com.epam.eun.entity;
public class Car {
    //...
    int speed;
    //...
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        this.speed = speed;
    }
    //...
}

package com.epam.eun.main;
public class Main {
    public static void main(String[] args) {
        Car fastCar = new Car();
        fastCar.speed = 100;
        fastCar.setSpeed(100);
        System.out.println(fastCar.getSpeed());
    }
}
```

## Access Modifiers and Encapsulation

Why is it not recommended to use only the modifier **public**? As an example, let's take the field **yearManufactureCar**, which determines the year when a car was produced. If other classes have direct access to this field (according to the modifier **public**), it is very likely that when the program is run, an incorrect value will be set for this field—for example, a year when cars were not yet invented. These data changes are undesirable. It is

recommended to limit direct access to data as much as possible to protect it from undesired access from the outside (both to receive the value and change it). Using the modifier **private** guarantees that the data will not be distorted or changed incorrectly. Note that the principle of **encapsulation** corresponds to the idea of concealing data inside a certain visibility scope.

Now study the recommendations on how to implement encapsulation.

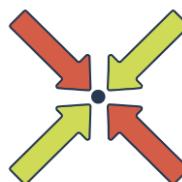
### Limiting access

Use the most limited access level that makes sense for a specific member. That is, do not use **public** without a good reason for doing so.



### Minimizing open data

Avoid declaring open fields. Use them only for read-only fields. As a rule, open fields lead to a specific implementation and limit flexibility when modifying code.



### Special access methods

To access **private** class fields, use special methods called **getters and setters**, which were explored in the lesson "The keyword this" in the "Classes" module.



## Conclusion

In this lesson, you have seen that:

- Encapsulation prohibits direct access to the fields of a given class instance from other classes.
- In Java, access modifiers are responsible for implementing the principle of encapsulation. These are keywords that regulate access to data and code.

You also reviewed several examples that demonstrate how to use the access modifiers **public**, **private**, **protected**, and **package-private** and reviewed some recommendations for using encapsulation.

### Check Your Knowledge!

1. Specify the possible access levels for the top-level class.

**public**

**package-private**

**private**

**protected**

**static**

**correct**

**Answer**

**Correct:**

The top level includes public and package-private access.

2. Which classes can access protected elements in a class?

**Subclasses in that same package**

**Unrelated classes in that same package**

**Subclasses in another package**

Unrelated classes in another package

None of the above

correct

Answer

Correct:

Protected access means that a class member may be accessed only in the body of that same class or in a class from the same package or successor class, even if it is located in a different package.

3. Which classes can access package-private elements in a class?

**Subclasses in that same package**

**Unrelated classes in that same package**

Subclasses in another package

Unrelated classes in another package

None of the above

correct

Answer

Correct:

By default, a class or class member will be visible and accessible only for classes in that same package.

4. Choose the statements that correctly characterize access modifiers.

Static **private** members of a class are only accessible using static methods of that class.

**Static public members of a class are accessible using all methods of that class.**

**Protected members of a class are accessible by successors located in a different package.**

A field that is a class member declared without an access modifier is accessible by classes of different packages.

correct

Answer

Correct:

Static private fields in a class are accessible in all methods of that class, as are static public fields. Protected fields in a class can be accessed from subclasses located in a different package. Class fields declared without an access modifier (package-private) are accessible in the classes of that same package.

# The Modifiers static and final

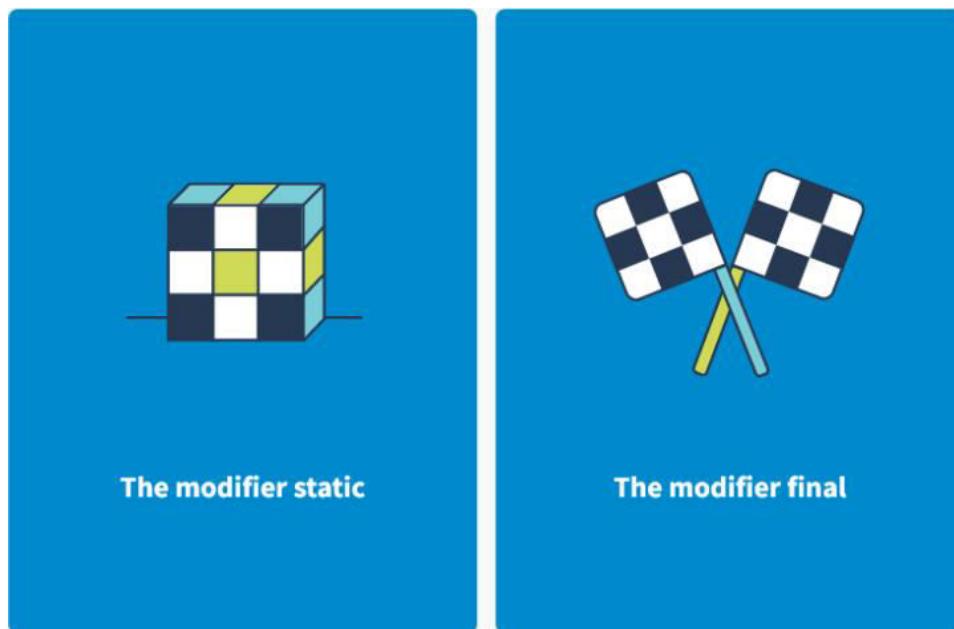
## Introduction

In this lesson, you will study **static** and **final** modifiers.

## The Modifiers static and final

Java offers a range of modifiers not only for providing access but also for implementing other functional capabilities—for example, static and final modifiers.

*Click on the cards to learn more about these modifiers.*



The modifier **static** is used to create methods and variables (fields) belonging to a class and not to an object.

The modifier **final** is used to define a final implementation of classes and methods, as well as invariability of variables and fields.

## static

The data field or method declared in a class as **static** is common for all objects of that class and is called a **class field** or a **class method**. In other words, they belong to a class and not to a class instance.

Static fields (class fields) have a number of features compared to instance fields:

### Feature 1

Static fields should be used without creating a class instance.

### Feature 2

If one object changes the value of such a field, all objects will see this change.

Static methods (class methods) are used to work with static class fields or only with data specified in their parameters. Just like class fields, **static methods have a number of features**:

- Static methods are not polymorphic; that is, the method version to be executed is determined at compile time.
- Static methods can be called from instance methods directly, just like static class fields.

- Static methods are not linked to a class instance and thus cannot use the keywords **this** or **super** to access a specific object.
- Static methods cannot access instance fields and methods directly. To access these, you need to create or receive a reference to the object.

To **access** static fields and methods, it is enough to specify before them the name of the class where they are defined. Of course, you can access a static method also using a reference to an object since any class instance has access to the static members of the class. However, this access will be logically incorrect, make the code more difficult to understand, and lead to the corresponding warning—though it will not cause a compilation error



Consider a few examples of using the keyword **static**.

Example 1:

This example describes the class **StaticDemo**. In this class, the two class fields **a** and **b** and the class method **callme()** are defined. In the method **main()** of the class **Demo1**, it is demonstrated how to correctly access static elements of the class **StaticDemo**, i.e., without creating an object that is not needed to interact with the fields and methods of the class **StaticDemo**.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}
public class Demo1 {
    public static void main(String[] s) {
        StaticDemo.callme();
        System.out.print("b = " + StaticDemo.b);
    }
}
```

Example 2:

Let's say we want to keep track of how many **Car** objects are created. This example shows a declaration of the class **Car**. The private field **numOfCars** is defined in it to calculate the number of objects of this class. When objects are created, the value of the field **numOfCars** increases by 1. By accessing the method **getNumOfCars()** you can get the value of the field **numOfCars**. In the method **main()** of the class **Demo2**, three objects of the class **Car** are created, and then the value of the field **numOfCars** is printed to console.

```
public class Car {
    private int numOfCars;
    //other fields
    public Car() {
        numOfCars++;
    }
    //other methods
    public int getNumOfCars() {
        return numOfCars;
    }
}
```

```

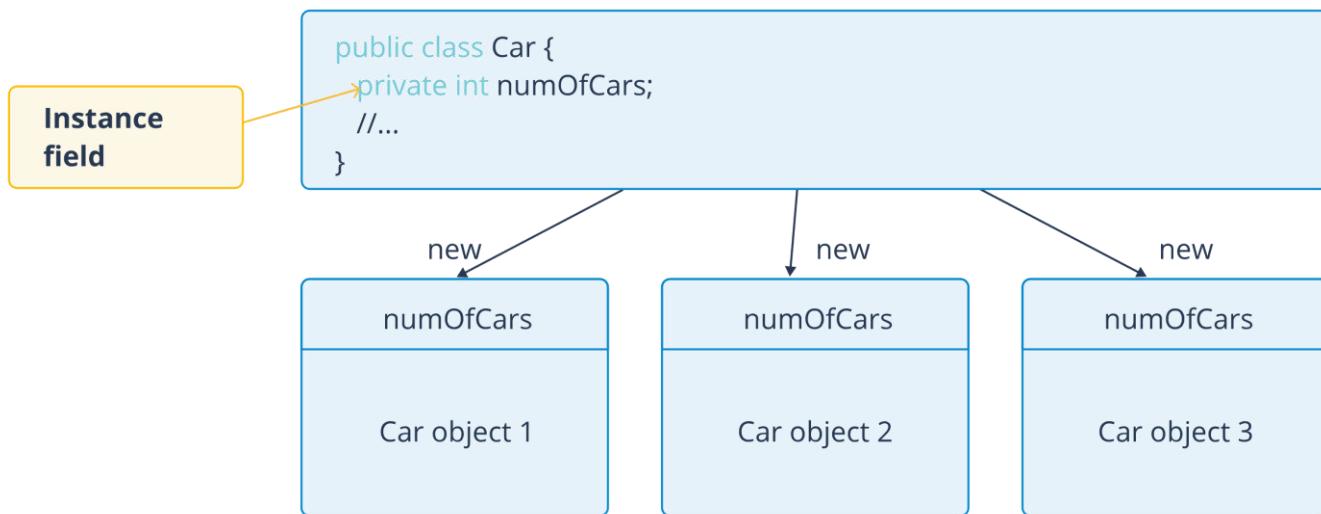
}
public class Demo2 {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car();
        Car car3 = new Car();
        System.out.println(car3.getNumOfCars());
    }
}
  
```

**Output:**

1

As a result, instead of the expected value of 3, we got 1. This happened because the field ***numOfCars*** is an instance field. Each object has its own copy of this field with the initial value of 0 and that takes a value of 1 when the object is created. In this example, it would be more correct to make the field ***numOfCars*** static. All objects would then have this in common.

This example can be represented graphically as follows:

**Example 3:**

This example is a modification of the previous example. In the class **Car**, the field ***numOfCars*** is declared as static. Thus, the method used to get its value—***getNumOfCars()***—is also static.

*Hover your cursor over the info icon to see an explanation of the code.*

```

public class Car {
    private static int numOfCars;
    //other fields
    public Car() {
        numOfCars++;
    }
    //other methods
    public static int getNumOfCars() {
        return numOfCars;
    }
}
public class Demo3 {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car();
    }
}
  
```

```

Car car3 = new Car();
System.out.println(Car.getNumCars());
}
}

```

**Output:**

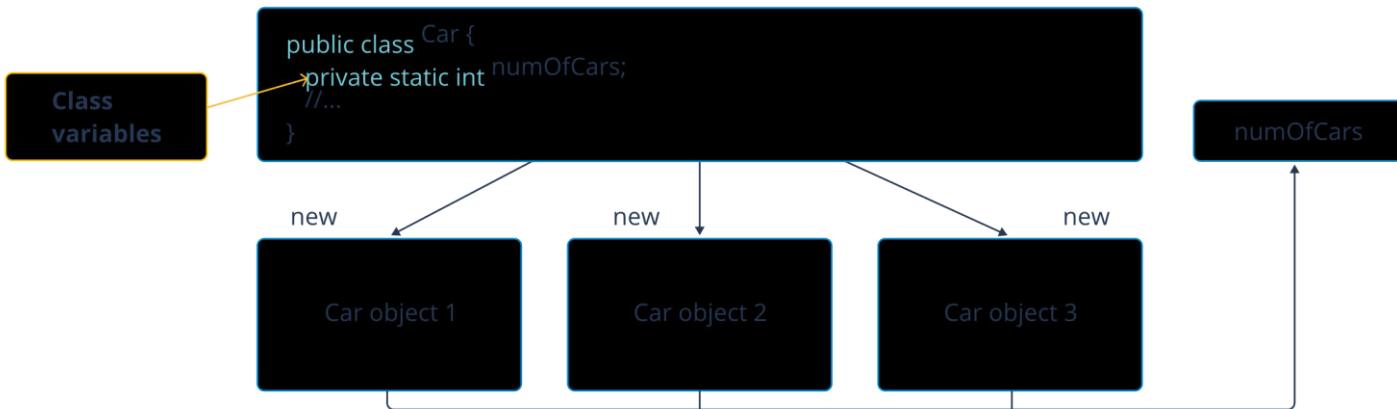
3

Now, when three objects of the class **Car** are created in the method *main()* of the class **Demo2**, and then the value of the field **numOfCars** is printed to console, you get "3" as a result. This happens because all objects have the field **numOfCars** in common and they all share access to it.

This example can be represented graphically as follows:

Let's review what you've learned so far about static fields and methods.

*Click on each tab to learn more.*



**Recommendation:** Methods using only static class data should be declared as static so that you can use them without creating a class instance.

Let's review what you've learned so far about static fields and methods.

**Static methods**

The definition of a class method as static depends on the data used by it:

- All necessary data is passed explicitly (in parameters).
- Only static fields are used.

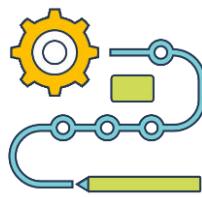
**Example.** The class **CustomMath** has no instance fields. Methods receive data through their parameters; that is, class instances are stateless. Because of this, methods should be declared as static.

```

public class CustomMath {
    public static int percent;
    public static int add(int x, int y) {
        return x + y + percent;
    }
    public static int multiply(int x, int y) {
        return x * y;
    }
}

```

```
}
```



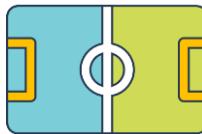
## Static fields

Static class fields:

- Are created when the class is first accessed
- Are common to all objects of the class
- Exist independently of class instances
- Can be accessed before a class instance is created

**Example.** Let's define the class **DemoStatic**, which contains only static elements. The method *main()* of the class **DemoMain** shows interaction with fields and methods of the class **DemoStatic** without creating an instance, as there is no need for this.

```
public class DemoStatic {
    public static int x;
    public static int y;
    public static int lengthVector() {
        return (int) Math.sqrt(x*x + y*y);
    }
}
public class DemoMain {
    public static void main(String[] args) {
        DemoStatic.x = 3;
        DemoStatic.y = 4;
        System.out.println("length = " + DemoStatic.lengthVector());
    }
}
```



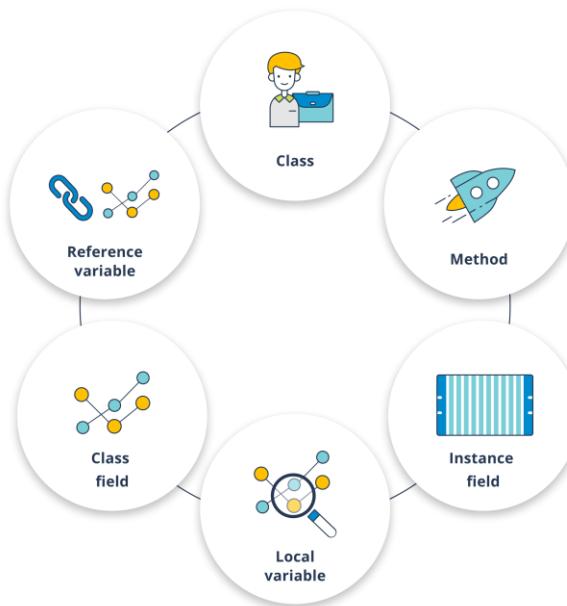
Limitations for static methods:

- Static methods can directly invoke only static methods.
- Static methods can directly access only static fields or their own parameters.
- Static methods cannot be accessed using the references **this** and **super**.
- Static methods can be overloaded with non-static ones and vice versa.

## final

The keyword **final** is a non-access modifier that applies to classes, methods, and variables, which makes them unchangeable (impossible to inherit or override). It can be used in a variety of contexts.

*Click on each + sign to learn more about each feature*



## Class

A class declared with **final** cannot have successors

## Method

A method declared with **final** cannot be overridden, i.e., its body cannot be changed in the successor.

## Instance field

An instance field declared as **final** is only initialized once: when declaring it, in the constructor, or in the initializer block.

## Local variable

A local variable or method parameter declared with **final** cannot be changed.

## Class field

For class fields, the modifier **final** is often used with the modifier **static** to make them constant.

## Reference variable

To a reference variable declared as **final** cannot be assigned another object. However, the data inside the object can be changed.

Note the naming convention for constants. The names of constants are written in uppercase, and the words are separated with underscores ("\_"), e.g., DISCOUNT, MAX\_RANGE, MIN\_WIDTH.

Now let's look at a few examples of how to apply the modifier **final**.

## 14 An incorrect attempt to modify a reference field

The example describes the class **TestFinal**. The class has the final field **lastName** of the type **String** (reference) and the method **setLastName()**, which is trying to change the value of the field.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class TestFinal {
    private final String lastName;
    public TestFinal(String lastName) {
        this.lastName = lastName;
    }
    public String getLastname() {
        return lastName;
    }
    public void setLastName(String name) {
        this.lastName = name;
    }
}
```

This attempt will lead to a compilation error since you cannot change a reference to another object for a field declared as **final**.

### Changing the state of an object, not a reference to an object

In the example, in the class **Person** the field **friends** is defined as final (a reference to a collection of friends, which is initially empty). There are also methods for receiving and adding friends to the collection.

*Hover your cursor over the info icon to see an explanation of the code.*

```
import java.util.ArrayList;
public class Person {
    private final ArrayList friends = new ArrayList<>();
    public ArrayList getFriends() {
        return friends;
    }
    public void addFriend(Person friend) {
        friends.add(friend);
    }
}
public class Main {
    public static void main(String[] args) {
        Person man1 = new Person();
        Person man2 = new Person();
        Person man3 = new Person();
        man3.addFriend(man1);
        man3.addFriend(man2);
        System.out.println(man3.getFriends());
    }
}
```

Since the method *addFriend()* does not change the reference and only changes the state of the collection, no error will occur.

## Conclusion

In this lesson, you have seen that:

- The modifier **static** is used to create methods and variables belonging to a class, not an object.
- The modifier **final** is used to define a final implementation of classes and methods, as well as invariability of variables and fields.

You also studied the specifics of using these modifiers—including their limitations—with the help of a few examples.

## Check Your Knowledge!

1. What is the result of running the following code?

```
public class JavaCourse {
```

```

        String courseName = "Java";
    }
public class University {
    public static void main(String[] args) {
        JavaCourse courses[] = { new JavaCourse(), new JavaCourse() };
        courses[0].courseName = "MegaCourse";
        for (JavaCourse c : courses) {
            c = new JavaCourse();
        }
        for (JavaCourse c : courses) {
            System.out.println(c.courseName);
        }
    }
}

```

Java  
Java

**MegaCourse**

**Java**

MegaCourse

MegaCourse

Nothing will be printed

Answer

Correct:

Both array objects are initialized by the string Java. Later, in the object with a zero index, the value is replaced with "MegaCourse", after which the program prints the value.

2. What is the result of running the following code?

```

public class Boo {
    static int i;
    public static void main(String[] args) {
        System.out.print(i);
    }
}

```

Nothing will be printed

null

**0**

A compilation error

Answer

Correct:

In the absence of explicit initialization, a static field will receive a default value. For the type int, this will be 0. The static method main has access to the static field of its class, which means the output will be 0.

3. Will the following code be compiled?

```

public class Emp {
    final static public int MIN_AGE = 20;
    static final int MAX_AGE = 70;
    final static int AVR_AGE = 70;
}

```

**Yes**

No

Answer

Correct:

The compilation will be error-free since rearranging modifiers when declaring fields does not play any role here.

However, by convention, the sequence public static final should be considered correct.

4. What is the result of running the following code?

```

public class Foo {
    static final byte FOO_MAX = 10;
}

```

```
}
```

```
public class Boo {
```

```
    public static void main(String[] args) {
```

```
        fooMaxChange();
```

```
    }
```

```
    static void fooMaxChange(){
```

```
        Foo.FOO_MAX *= 2;
```

```
        System.out.print(Foo.FOO_MAX);
```

```
    }
```

```
}
```

```
10
```

```
0
```

```
20
```

**A compilation error**

Answer

Correct:

A compilation error in the line `Foo.FOO_MAX *= 2;` occurs as a result of an attempt to change the value of a final field.

# Inheritance. Part 1

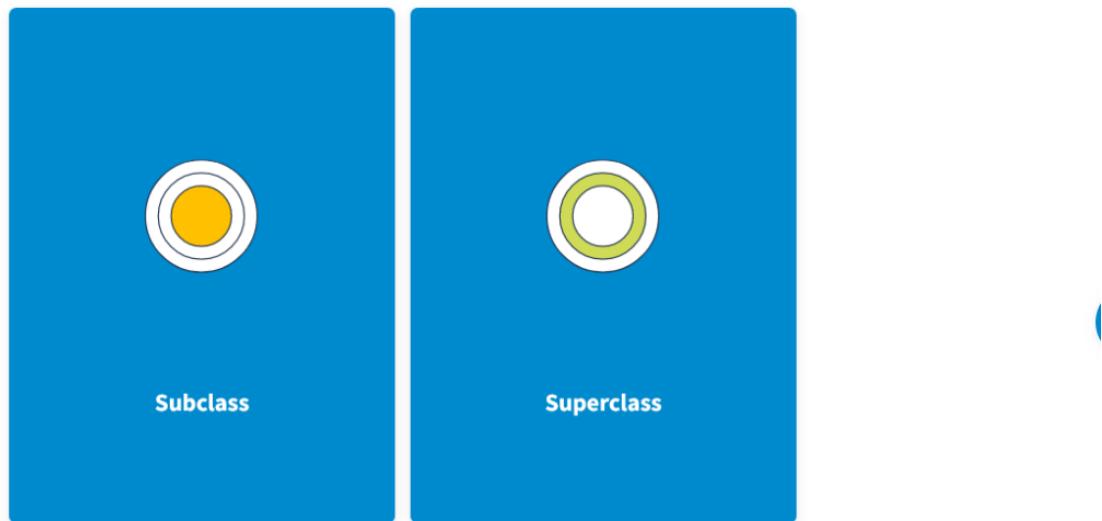
## Introduction

In this lesson, you will explore one of the fundamental principles of OOP—inheritance. You will study the various types of inheritance and how to use it with different access rights. You will also discover superclasses and subclasses and find out the different capabilities of subclasses.

## The Concept of Inheritance

**Inheritance** is a property that allows you to create a new class using an existing one with a partial or complete import of characteristics. Classes can then be arranged in a hierarchy that shows relationships of the type "is-a" (is a type/is a subtype). For example, "Car" is a subtype of "Vehicle."

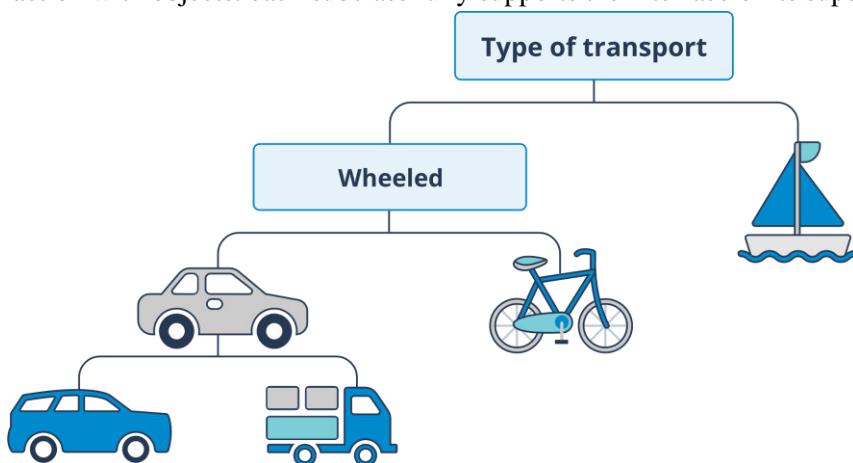
Java uses the terms subclass and superclass for classes that are located in one inheritance branch.



**A subclass** is a class that is inherited from or extends another class. A subclass can also be called a derivative or a child class.

**A superclass** is a class from which inheritance occurs. It can also be called a base or a parent class.

A subclass completely matches the specifications of the superclass. However, a subclass may have additional functionality in terms of its interface of interaction with objects: each subclass fully supports the interface of its superclass, but not vice versa.



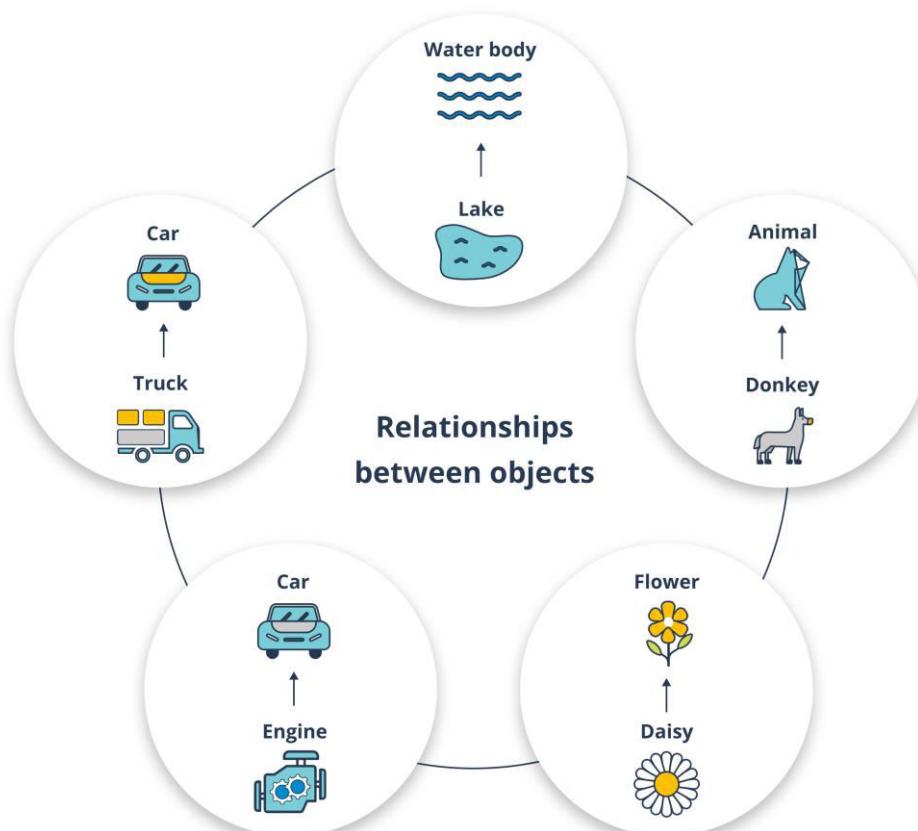
Let's consider an example.

A superclass car has the property body type.

A subclass electric car has additional properties: charging time and cell capacity.

Instead of inheritance, there is a different relationship between the classes—**composition**, when one class references objects of other classes. This is how complex objects are created. Composition involves a "has-a" relationship between classes (i.e., "has in its structure").

Now let's check how well you understand inheritance. Think about whether the relationships below are examples of inheritance.



Since the successor receives a set of properties from the parent, one of the benefits of inheritance is **reuse**.

*Click on the arrow to learn more about reuse and see examples.*

After defining the behavior (methods) in the superclass, this behavior is automatically inherited by all subclasses. Thus, you only write a method once, and then it can be used by all subclasses.

For example, if the class **Car** has a defined method `move()`, it will automatically be inherited by the subclass **ElectricCar**.



After defining a set of data (fields/properties) in a superclass, the same properties will be inherited by all subclasses. The class and its child class share a common set of properties.

For example, if the class **Car** has a field *weight*, it will also be inherited by the subclass **ElectricCar**.



A subclass only needs to implement the differences between itself and the superclass.

For example, for the subclass **ElectricCar**, we need to add the field *chargeTime*.



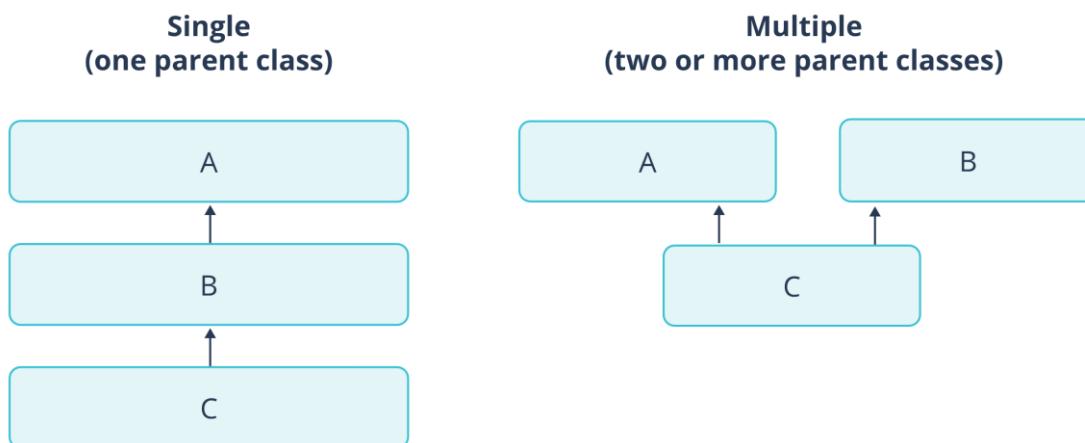
Let's study the syntax for describing a subclass:

*Hover your cursor over the info icon to see an explanation of the code.*

```
<modifiers> class <subclass name> extends <superclass name> {
    subclass_body
}
```

## Types of Inheritance

Two types of inheritance are determined by the number of parent classes—single and multiple.



As you already know, the object saves its state in the fields that are declared in the class. Imagine a situation in which you can define a new class that extends several classes. When you create an object of that class, it will inherit fields from all superclasses. But what if methods or constructors of different superclasses initialize the same field? Which method or constructor will take precedence?

To avoid problems of inheriting the same **Inheritance and Access Rights**

Take note of the rules for inheriting class members, which are determined by the rules for accessing them.

*Click on the tabs to learn more.*

field from several classes, Java uses only single inheritance.

In Java, except for the Object class, which has no superclass, every class has one and only one direct superclass (single inheritance).

## Public



A subclass inherits **public** elements of its superclass; that is, they are transferred to the visibility scope of the subclass.

## Protected



A subclass inherits **protected** elements of its superclass regardless of whether they are defined in one package or in different ones. This means they are transferred to the visibility scope of the subclass.

## Package-private



A subclass inherits **public** elements of its superclass; that is, they are transferred to the visibility scope of the subclass.

## Private



A subclass does not inherit **private** elements of its superclass; that is, they are not transferred to the visibility scope of the subclass. The subclass only interacts with such elements through the inherited methods.

Let's consider an example of describing inheritance for the classes **Car** and **ElectricCar**, leaving the body of the subclass **ElectricCar** without implementation. Remember that the compiler will add a default constructor to the class

**ElectricCar.** According to the rules of inheritance, you should also make sure that the class **ElectricCar** can access the methods *setName()* and *show()*, which are inherited from the class **Car**.

*Click Show more to see the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Car {  
    private String name;  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void show() {  
        System.out.println("Name: " + name);  
    }  
}  
public class ElectricCar extends Car {}  
public class Program {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.setName("Wheels");  
        car.show();  
        ElectricCar electricCar = new ElectricCar();  
        electricCar.setName("Lightning");  
        electricCar.show();  
    }  
}
```

## Output

Name: Wheels  
Name: Lightning

## Capabilities of Subclasses

In the example above, the subclass **ElectricCar** used inherited methods without changes. But more often, a successor implements behaviors that differentiate it from the superclass.

What can a subclass do with the elements inherited from its superclass?

*Click on the headings to learn more.*

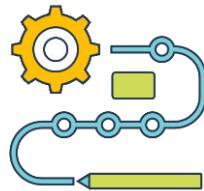
### In relation to fields

- Inherited fields can be used exactly like any other fields of the subclass.
- In the subclass, you can declare a field with the same name as in the superclass, thus hiding the inherited field (**Note: this is not recommended at all**).
- You can declare new fields in the subclass that are absent in the superclass.



## In relation to methods

- Inherited methods can be used just as they are.
- You can write a new instance method in the subclass that will have the same signature as one of the instance methods of the superclass, thus overriding it.
- You can write a new class method (static) in the subclass that will have the same signature as one of the methods of the superclass, thus hiding it.
- You can declare new methods in the subclass that are absent in the superclass.



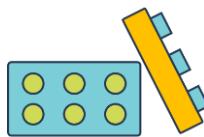
## In relation to constructors

Superclass constructors are not inherited. They can only be accessed. A subclass constructor can access a superclass constructor by using the keyword super. The following syntax is used for this purpose:

```
super(<arguments list>);
```

Usually, a call to a superclass constructor in the body of a subclass constructor is used to initialize private fields in the superclass. These fields are not transferred to the visibility scope of the subclass; however, these properties have to be initialized when creating an object of the subclass.

So far, you have studied the capabilities of subclasses in relation to their inherited elements. You can find more information on inheritance in [the official documentation](#). Next, you will delve into the specifics of using the keyword **super**.



Superclass constructors are not inherited. They can only be accessed. A subclass constructor can access a superclass constructor by using the keyword super. The following syntax is used for this purpose:

```
super(<arguments list>);
```

Usually, a call to a superclass constructor in the body of a subclass constructor is used to initialize private fields in the superclass. These fields are not transferred to the visibility scope of the subclass; however, these properties have to be initialized when creating an object of the subclass.

So far, you have studied the capabilities of subclasses in relation to their inherited elements. You can find more information on inheritance in [the official documentation](#). Next, you will delve into the specifics of using the keyword **super**.

The keyword **super** is a predefined reference to an object of the superclass in the body of the subclass. This keyword may be used in subclass constructors for explicit access to superclass constructors. Its main purpose is to initialize private fields in the superclass.

Consider the example of creating a subclass object with an explicit call to a superclass constructor.

*Click Show more to see the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Base {
    private int a, b;
    Base(int a, int b) {
        this.a = a;
        this.b = b;
    }
}
public class Derived extends Base {
    private int c;
    Derived(int a, int b, int c) {
        super(a, b);
        this.c = c;
    }
}
public class Demo {
    public static void main(String[] args) {
        Derived obj = new Derived(1, 2, 3);
    }
}
```

## Conclusion

In this lesson, you have seen that:

- Inheritance allows you to create a new class (subclass) based on an existing class (superclass) with a complete or partial import of its characteristics. The main benefit of inheritance is reuse.
- In Java, classes can have one and only one direct superclass (single inheritance).
- A subclass inherits **public** and **protected** elements from its superclass. **Package-private** elements of a superclass are inherited only if they are defined in the same package. A subclass does not inherit **private** elements from a superclass.
- Subclasses have various use cases for the elements they inherit—fields and methods.
- The keyword **super** may be used in subclass constructors to explicitly access superclass constructors and initialize private fields in the superclass.

## Check Your Knowledge!

1. Which elements of the class are inherited by its subclass?

**Package-private elements, if the base and derivative classes are located in the same package**

**Protected elements, in spite of the packages where the base and derivative classes are defined**

Private elements

Constructors of the base class

**Public elements**

correct

Answer

Correct:

A subclass inherits a superclass's public and protected elements. Package-private elements are inherited only if they are defined in the same package. A subclass does not inherit a superclass's private elements.

2. Which of the following statements are true?

A class may be used as its own superclass.

In a class constructor, you can call constructors through this() and super() at the same time.

**Static methods can be declared in subclasses with the same signature as in the base class.**

**Static methods can be overloaded in subclasses.**

correct

Answer

Correct:

A class cannot inherit itself. In a class constructor, you cannot use super() and this() since the call to the constructor should always be the first operator in a constructor. The compiler will not prevent you from declaring static methods in subclasses; however, the early binding mechanism will be used when calling them. You cannot apply @Override annotation to static methods. Static methods can be overloaded in subclasses. Access to these methods depends on the reference type and access attribute.

3. What type of relationship does the following code demonstrate?

```
public class A {
    void job() {
        System.out.println("Class A");
    }
}

public class B extends A {
    void job() {
        System.out.println("Class B");
    }
}
```

was-a

has-a

have-a

**is-a**

Answer

Correct:

Inheritance allows you to categorize classes in hierarchies that show relationships of the type "is-a" (is a type/subtype).

4. What is the result of running the following code?

```
public class A extends Object {
    public void job() {
        System.out.println("Class A");
    }
}

public class B extends A {
    public void job() {
        System.out.println("Class B");
    }
}

public static void main(String[] args) {
    A b = new B();
    b.job();
}
```

Class A

**Class B**

null

A compilation error

Answer

Correct:

The Java virtual machine calls the appropriate method for the object that is referred to in the variable. It does not call the method that is defined by the type of variable.

5. What is the result of running the following code?

```
public class A {
    public void job() {
        System.out.println("Class A");
    }
}

public class B extends A {
```

```
public void job(int i) {  
    System.out.println("Class B");  
}  
public static void main(String [] args) {  
    B b = new B();  
    b.job();  
}
```

**Class A**

Class B

null

A compilation error

Answer

Correct:

The version of the overloaded method is always determined at the stage of code compilation.

# Inheritance. Part 2

## Introduction

In this lesson, you will see the capabilities of subclasses in more detail. This will include:

- What constructor chains are and how to access them
- Why it is not recommended to hide fields
- How to implement method overriding
- How to break inheritance

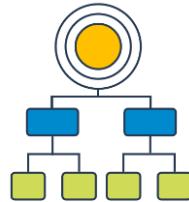
## Constructor Chains

When creating a subclass object, constructors of all its superclasses are always invoked. For example, class C is inherited from class B, and class B from class A; thus, classes A and B are superclasses for class C.

This sort of invoking is called **constructor chaining**. This is logical since a subclass shares all the properties declared in its superclasses. Therefore, when you create a subclass object, it is necessary to initialize all its properties. As you know, class constructors do this.

There are several rules for invoking constructor chaining. Let's study them.

### Rule 1



When creating a subclass object, constructors of all the superclasses in its inheritance chain are invoked in the order of their inheritance, **from superclass to subclass**. This means that the chain of inheritance is followed from the topmost superclass. Then constructors are executed one by one, following the hierarchy from top to bottom.

#### Example

In this example, only default constructors are declared in the classes **Cat** and **BritishCat**. The constructor in the class **BritishCat** does not explicitly call the constructor in the class **Cat**. As you can see, the constructor in the class **Cat** is executed.

```

public class Cat {
    Cat() {
        System.out.println("Cat constructor");
    }
}

public class BritishCat extends Cat {
    BritishCat() {
        System.out.println("British constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        BritishCat Cat = new BritishCat();
    }
}
  
```

}

## Output

Cat constructor  
British constructor

## Rule 2



An explicit invocation (**super**) to a superclass constructor should be the first operator in the body of the subclass constructor.

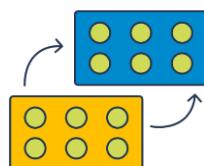
In this example, the constructors in the classes **Cat** and **BritishCat** are only declared with parameters. The constructor in the class **BritishCat** explicitly calls the constructor in the class **Cat** as its first action.

```
public class Cat {  
    Cat(String name) {  
        System.out.println("Cat constructor – name " + name);  
    }  
}  
  
public class BritishCat extends Cat {  
    BritishCat(String name) {  
        super(name);  
        System.out.println("British constructor");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        BritishCat Cat = new BritishCat("Mulberry");  
    }  
}
```

## Output

Cat constructor – name Mulberry  
British constructor

## Rule 3



If a subclass constructor does not invoke explicitly any of the superclass constructors, the default superclass constructor should be invoked automatically.

## Example

In this example, two constructors are declared in the class **Cat**: a default constructor and one with parameters. And the class **BritishCat** only has a constructor with parameters. The constructor in the class **BritishCat** does not

explicitly call either constructor of the class **Cat**. As you can see, the default constructor in the class **Cat** will be executed.

```
public class Cat {
    Cat() {
        System.out.println("Cat default constructor");
    }
    Cat(String name) {
        System.out.println("Cat constructor " + name);
    }
}
public class BritishCat extends Cat {
    BritishCat(String name) {
        System.out.println("British constructor");
    }
}
public class Main {
    public static void main(String[] arg) {
        BritishCat Cat = new BritishCat("Mulberry");
    }
}
```

## Output

Cat default constructor  
British constructor

## Rule 4



If a superclass has no default constructor and the subclass constructor does not invoke another superclass constructor explicitly, the Java compiler will throw an error.

Let's consider some examples of implementing constructor chaining when creating a subclass object. We will use the class **Cat** as a superclass and the class **BritishCat** as a subclass. The constructors in these classes use text output only to demonstrate how to invoke constructor chaining.

Note: You should not use any data output or input in constructors.

### example

In this example, only constructors with parameters are declared in the classes **Cat** and **BritishCat**. The constructor in the class **BritishCat** does not explicitly call the constructor in the class **Cat**. This means that the default constructor in the class **Cat** should be called automatically. The compiler checks the chain of constructors and does not find a default constructor in the class **Cat**; thus, it throws an error.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Cat {
    Cat(String name) {
        System.out.println("Cat constructor " + name);
    }
}
```

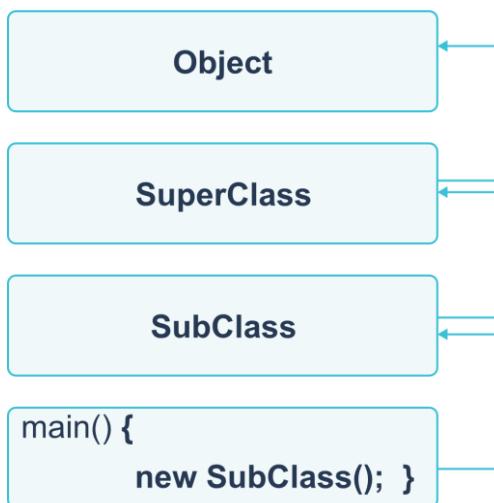
```

}
public class BritishCat extends Cat {
    BritishCat(String name) {
        System.out.println("British constructor");
    }
}
public class Main {
    public static void main(String[] args) {
        BritishCat Cat = new BritishCat("Mulberry");
    }
}

```

Now let's analyze the process of initializing a class when creating its object. As you already know, the first step is to invoke the constructor chain. In other words, when creating a subclass object, the chain of constructors of superclasses is unfolded and executed.

For example



```

class SuperClass { }
class SubClass extends SuperClass { }
public class Main{
    public static void main(String[] args) {
        SubClass c = new SubClass();
    }
}

```

The class **Object** is a superclass for any Java class. It will be explored later in this course.

So far, you have studied how subclasses behave when working with constructors. Now let's look at how they behave when working with fields. As you remember, if you declare a field in a subclass with the same name as in its superclass, the field will be hidden. Let's see why it is not recommended to do this.

## Hiding/Shadowing Fields

If a subclass field has the same name as the inherited superclass field, it hides/shadows the superclass field even if they are of different types.

*It is not recommended to use field hiding/shadowing since this makes code difficult to read.*

To access a superclass field, the keyword **super** is used in the subclass. This is a second way of using the keyword **super**—as a reference to a superclass object in the subclass body. It has the following syntax:

**super.<instance field/instance method>;**

Let's take a look at an example of field shadowing. The class **Vehicle** has the field *maxSpeed*. The scope of this field will be extended to the subclass **Car** because the field access level is set as **protected**. However, the class **Car** declares its own field with the name *maxSpeed*. The visibility of the inherited field is shadowed. To resolve this situation in the method *showSpeed()* in the subclass **Car**, the keyword **super** is used.

*Click Expand to see the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Vehicle {
    protected int maxSpeed = 230;
}
public class Car extends Vehicle {
    protected int maxSpeed = 300;
    public void showSpeed() {
        System.out.println(super.maxSpeed);
        System.out.println(maxSpeed);
    }
}
public class Demo {
    public static void main(String[] args) {
        Car car = new Car();
        car.showSpeed();
    }
}
```

## Output

230  
300

In terms of syntax, the issue of accessing both fields has been resolved. However, in this case the class **Car** has two fields storing the maximum speed with different values. The question is: which one is true? This is why it is best not to apply field shadowing. Now let's look at how to override methods.

## Overriding Methods

Inheritance and overriding of methods are used to implement the differences in the behavior of a superclass and subclass. During inheritance, subclasses add new capabilities in the form of fields and methods or by overriding inherited methods.

*Click on the tabs to study the rules.*

### Overriding

A subclass instance method that has the same signature (name, number, and types of parameters) and type of returned value as the instance method in the superclass overrides/replaces this superclass method. In other words, overriding a method means implementing it is different for the subclass.

The ability of a subclass to override methods allows this class to inherit from a class whose behavior is "close enough" to its own so as to change this behavior if necessary.



## The annotation @override

When overriding a method, you can use the annotation **@Override**, which specifies for the compiler that this is an overridden superclass method. This annotation will help avoid further errors when using polymorphism. If the compiler detects an absence of this method in all superclasses, it will generate an error warning.



## Cancelling an override

To cancel the opportunity to override a method, the modifier **final** is used when declaring it. It is used in cases when the behavior implemented in the superclass should not be changed in subclasses.

For example, methods called from constructors should usually be declared as **final**. If the constructor calls a non-**final** method, the subclass can override this method with unexpected or undesired consequences.



## Specifics

When overriding a method, you can apply a wider access modifier to the method. For example, if the inherited superclass method has the access type **protected**, it can be replaced with **public** when overriding the method.



Let's consider some examples of overriding methods in different classes.

### Complete replacement of behaviour

In this example, the method *show()* is described in the superclass **Base** and is inherited by the subclass **Derived**. The subclass **Derived** replaces the body of that method, adjusting it to its own needs. As a result, when the method *show()* is called at objects of these classes, the appropriate invoked method is determined by the object type.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Base {
    public void show() {
        System.out.println("Base");
    }
}
public class Derived extends Base {
    @Override
    public void show() {
        System.out.println("Derived");
    }
}
```

```

    }
}

public class Demo1 {
    public static void main(String[] args) {
        Base base = new Base();
        Derived obj = new Derived();
        base.show();
        obj.show();
    }
}

```

## Output

Base  
Derived

## Supplementing behaviour

A subclass does not always need to completely replace the body of the inherited superclass method. Sometimes you only need to supplement it. To do this, when overriding the method, you can call the superclass method using the keyword **super**.

It is worth noting that if the chain of inheritance has more than two classes, the call through the keyword **super** always refers to the nearest/direct superclass.

In the following example, the method *show()* declared in the superclass **Base** and inherited by the subclass **Derived** is overridden as follows: First, the method *show()* of the superclass is called, and then additional actions are implemented. As a result, when the method *show()* is called at the object of the class **Derived**, all messages are printed, including the message of the method *show()* of the superclass.

*Hover your cursor over the info icon to see an explanation of the code.*

```

public class Base {
    public void show() {
        System.out.println("Information from Base");
    }
}

public class Derived extends Base {
    @Override
    public void show() {
        super.show();
        System.out.println("Information from Derived");
    }
}

public class Demo2 {
    public static void main(String[] args) {
        Derived obj = new Derived();
        obj.show();
    }
}

```

## Output

Information from Base  
Information from Derived

## Changing the access modifier

As you already know, elements with **protected** access are inherited by a subclass regardless of whether the superclass and its subclass are defined within one package or in different ones. However, if the superclass and subclass are located in different packages, the visibility scope of **protected** elements should only be limited by the subclass body.

Outside of it, they will not be accessible. To make inherited **protected** methods accessible for use beyond the subclass body, when overriding this method, you can replace the access modifier with **public**.

In the following example, three classes in different packages are described. The superclass **Vehicle** is in the package *com.model.basic\_transport* with a **protected** field *maxSpeed* and a **protected** instance method *showSpeed()*. When overriding the method *showSpeed()*, the subclass **Bicycle** in the package *com.model.derived\_transport* changes the access modifier to **public**. In the third package, the class **Demo** *com.model.test\_transport* creates objects of the classes **Vehicle** and **Bicycle**. Then the field *maxSpeed* is accessed, and the method *showSpeed()* is called at these objects.

*Hover your cursor over the info icon to see an explanation of the code.*

```
package com.model.basic_transport;
public class Vehicle {
    protected int maxSpeed = 230;
    protected void showSpeed() {
        System.out.println(maxSpeed);
    }
}
package com.model.derived_transport;
import com.model.basic_transport.Vehicle;
public class Bicycle extends Vehicle {
    public void showSpeed() {
        System.out.println(maxSpeed);
    }
}
package com.model.test_transport;
import com.model.basic_transport.Vehicle;
import com.model.derived_transport.Bicycle;
public class Demo {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        Bicycle bicycle = new Bicycle();
        System.out.println(vehicle.maxSpeed);
        System.out.println(bicycle.maxSpeed);
        vehicle.showSpeed();
        bicycle.showSpeed();
    }
}
```

We got a compilation error in every case, except for when we called the method *showSpeed()* at the object of class **Bicycle**. This happened because **protected** elements of one class are not accessible from classes located in other packages.

## Breaking Inheritance

The keyword **final** can be used to break the inheritance chain. In other words, you can prohibit class inheritance and create a sealed (final) class. The following syntax is used for this:

```
final class <class name> { }
```

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
public final class Cat {
    ...
}
public class BritishCat extends Cat {
    ...
}
```

## Conclusion

In this lesson, you have seen that:

- When creating a subclass object, the constructors of all its superclasses are invoked. This is called constructor chaining. You also studied the rules for invoking superclasses as well as some examples of how to do this.
- Methods are inherited to implement differences in the behavior of a superclass and a subclass.
- It is not recommended to use field hiding since this makes code difficult to read.
- To disable a chain of inheritance, the keyword **final** is used.

### Check Your Knowledge!

1. Which of the following options can be used independently in place of the comment "line 0" for a successful code compilation and execution?

```
class A { }
class B extends A { }
class C extends B { }
class Main {
    public static void main(String[] args) {
        //line 0
    }
}
A obj = new B();
B obj = new A();
A obj = new C();
B obj = new C();
C obj = new A();
C obj = new B();
correct
```

Answer

Correct:

A reference to a superclass can be assigned a subclass object. However, a reference to a child class can be assigned an object of the parent class only using explicit typecasting.

2. What is the result of running the following code?

```
public class Boy {
    private int age = 19;
    private String name = "Michael";
    public static void main(String[] args) {
        Boy b = new Boy();
        b.toString();
    }
    public String toString() {
        return name + " is " + age + " years old";
    }
}
```

Michael is 19 years old

**Nothing will be printed**

null

A compilation error

Answer

Correct:

Nothing will be printed to console because there is no operator `System.out.print(b)` or `System.out.print(b.toString())`, which is basically the same.

3. What will be printed to console after compiling and executing code?

```
class Person {
    String version = "Person";
}
class Student extends Person {
    String version = "Student";
```

```
public class Main {
    public static void main(String[] args) {
        Person person = new Student ();
        System.out.println(person.version);
    }
}
```

Compilation error

**Person**

Student

Runtime error

Answer

Correct:

In Java, fields are not polymorphic. This means that it depends on the reference and not the object type of the field accessed from outside of the subclass body. In this case, the class Student has two fields with the name version (its own and inherited). The field is accessed through a reference to the superclass Person; thus, the inherited field will be used.

4. What is the result of running the following code?

```
public class Person {
    String name;
}

public class SuperMan extends Person {
    String skills;
    SuperMan(String skills, String name) {
        super();
        this.skills = skills;
    }
    public String showInfo() {
        return name + " has " + skills;
    }
}

public class Main {
    public static void main(String[] args) {
        SuperMan superMan = new SuperMan("super vision", "Clark Kent");
        System.out.print(superMan.showInfo());
    }
}
```

Clark Kent has super vision

Clark Kent has null

**null has super vision**

A compilation error

Answer

Correct:

The field skills will be initialized with the value passed to the constructor of the class SuperMan. The field name of the superclass Person received a default value because the value has not been passed from the constructor SuperMan to the superclass constructor.

# Polymorphism

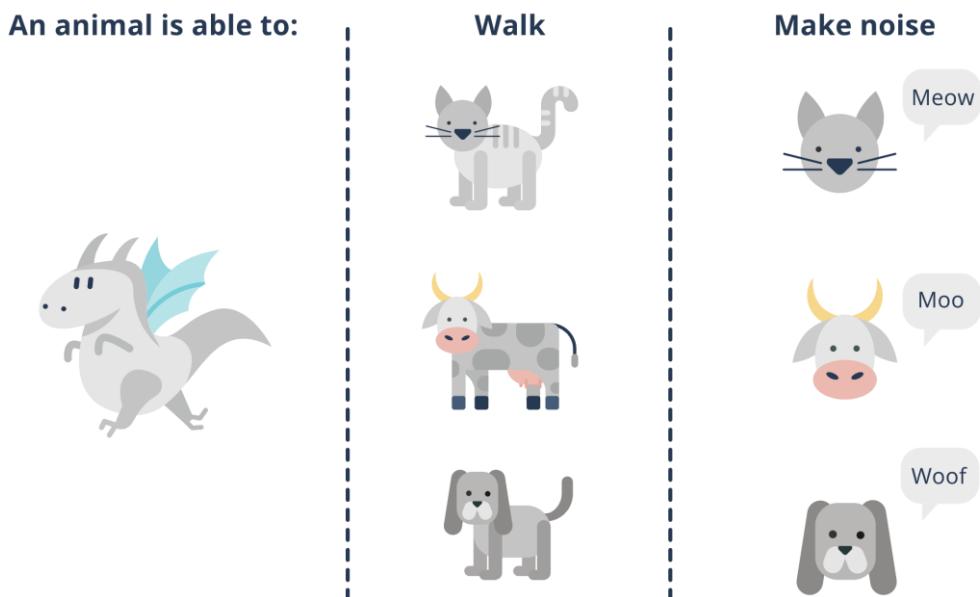
## Introduction

In this lesson, you will be introduced to the concept of polymorphism and the advantages of using it. You will also explore early and late binding and when it is used. Lastly, you will study what object typecasting is and why you should use **instanceof**.

## Polymorphism

Polymorphism is one of the fundamental notions in object-oriented programming, along with inheritance and encapsulation. In programming languages and type theory, polymorphism involves providing a single interface to entities (objects) of different types or using a single symbol to represent multiple different types.

For example, suppose we have a hierarchy of animals. The superclass "Animal" declares a behavior interface—walk, make noises, eat, etc. All its successors ("Cat," "Dog," "Cow," etc.) share this interface, although they all implement it differently.



You can say that an "Animal" can meow, bark, or make other noises. In other words, an "Animal" can take different forms.

The most widely used polymorphism in OOP is the use of a reference to a superclass when manipulating a subclass object.

Consider an example of polymorphism that applies to OOP. Imagine you are designing a program. You need to implement a functionality that allows you to draw or represent different shapes, but you don't know exactly which ones.

## Step 1

Let's create the class **Shape** with the method *draw()* to display the shapes we are going to draw.

```
public class Shape {
    public void draw() {
        System.out.println("Shape");
    }
}
```

## Step 2

We get a task to draw a square. To do this, we define the class **Shape** to extend the class **Shape** and override its method *draw()* so that it will draw a square.

```
public class Square extends Shape {
    @Override
    public void draw() {
        System.out.println("Square");
    }
}
```

## Step 3

We will describe the program launch as follows:

```
public class Main {
    public static void main(String[] args) {
        Shape myShape = new Square();
        myShape.draw();
    }
}
```

## Step 4

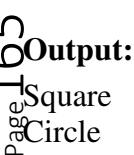
Sometime later, the customer decides it would be good to draw other shapes, such as a circle or a triangle, but we do not want to rewrite a bunch of code. Therefore, we create classes that inherit the **Shape** class, **Circle** (circle) and **Triangle** (triangle):

```
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
}
public class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Triangle");
    }
}
```

## Step 5

We will change the code for launching the program as follows:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = { new Square(), new Circle(), new Triangle() };
        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}
```

**Output:**  


```

graph TD
    Root[Square] --> Circle[Circle]
    Root --> Triangle[Triangle]
  
```

## Triangle

Thus, thanks to the principle of inheritance, subclasses are a variety of their superclass (relation "is-a"), and they all have in common the method `draw()`. And this means the following: A square is a shape, and so is a triangle. Therefore, we can access objects of subclasses (for example, `Square`) through the superclass type (for example, `Shape`).

Using a reference to a superclass, you can access objects of any of its subclasses. Subclasses can define their own unique behavior and, at the same time, share the functionality of their superclasses.

## Early and Late Binding

Polymorphism can differ by when implementation is chosen:

- Statically—during compilation
- Dynamically—during execution

In Java, the mechanism for choosing implementation—defining the called method—is called **binding**. There are two types of method binding.

### Static early binding

This is performed during compilation. It becomes clear which exact method will be executed based on the call expression after the compilation:

- It is used to resolve calling overloaded methods.
- It is based on the type of reference variable, not the object type (calling static methods).

Static polymorphism is executed faster since there are no dynamic overheads, but additional support from the compiler is required. Dynamic polymorphism is more flexible but is performed slower since a dynamically bound library can work with objects without knowing their complete type.

Let's consider some examples of static and dynamic binding using families of different objects.



### EXAMPLE

For static class methods, dynamic polymorphism cannot be applied since calling static elements is done by the reference type, not the object type, through which it is accessed. The version of the called static method is always determined at the compilation stage—early binding.

In the example, the superclass **Insurance** has the following elements:

- The static field **LOW**
- A static method `getCategory()` that returns the name of the superclass
- An instance method `getPremium()` that returns the value of the field **LOW**

The class **CarInsurance** is a subclass **Insurance** and has:

- Its own static field **HIGH**
- The static method `getCategory()`, which is overridden and returns the name of the subclass

- The instance method `getPremium()`, which is overridden and returns the value of the field **HIGH**

In the class **Main**, a reference to the superclass `current` is declared and initialized by the subclass object. Then, using the reference to the object and the subclass name, the static method `getCategory()` is called.

```
public class Insurance {
    public static final int LOW = 100;
    public int getPremium() {
        return LOW;
    }
    public static String getCategory() {
        return "Insurance";
    }
}
public class CarInsurance extends Insurance {
    public static final int HIGH = 200;
    @Override
    public int getPremium() {
        return HIGH;
    }
    public static String getCategory() {
        return "CarInsurance";
    }
}
public class Main {
    public static void main(String[] args) {
        Insurance current = new CarInsurance();
        System.out.println("category: " + current.getCategory());
        System.out.println("category: " + CarInsurance.getCategory());
    }
}
```

### Output:

```
category: Insurance
category: CarInsurance
```

As you can see, calling an option of the static method depends on the type of reference to the object, not the object type. In the given case, the way the method `getCategory()` is called depends on the reference type of the object `current`, meaning the type **Insurance**.

### Dynamic late binding

This is performed during program execution. The compiler has no opportunity to determine which method with identical names should be invoked:

- It happens when calling an overridden subclass method through a reference to the superclass.
- It is based on the object type, not the reference type.

Static polymorphism is executed faster since there are no dynamic overheads, but additional support from the compiler is required. Dynamic polymorphism is more flexible but is performed slower since a dynamically bound library can work with objects without knowing their complete type.

Let's consider some examples of static and dynamic binding using families of different objects.



## EXAMPLE of late binding

- The example describes three classes: **Figure** (as a superclass), **Rectangle**, and **Triangle** (as subclasses).
- The class **Figure** has two fields declared to store the dimensions of figures (*dim1* and *dim2*) as well as an implemented method to calculate the area of a figure *area()*.
- The shape of the figure is not known for the class **Figure**; therefore, the method returns 0.
- The subclasses **Rectangle** and **Triangle** override the method *area()* to calculate their area. The constructors of these classes pass their dimensions for storage to the fields *dim1* and *dim2*: **Rectangle**—width and height—and **Triangle**—base length and height.
- The class **FindAreas** creates three objects—one each for the classes **Figure**, **Rectangle**, and **Triangle**. It also creates a reference of the type **Figure**—*figref*. This reference accesses created objects one by one, and the method *area()* is called at it.

```

public class Figure {
    protected double dim1;
    protected double dim2;
    Figure(double dim1, double dim2) {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }
    public double area() {
        System.out.print("Area of the figure not determined ");
        return 0.0;
    }
}
public class Rectangle extends Figure {
    Rectangle(double dim1, double dim2) {
        super(dim1, dim2);
    }
    public double area() {
        System.out.print("Area of the quadrangle ");
        return dim1 * dim2;
    }
}
public class Triangle extends Figure {
    Triangle(double dim1, double dim2) {
        super(dim1, dim2);
    }
    public double area() {
        System.out.print("Area of the rectangle ");
        return dim1 * dim2 / 2;
    }
}
public class FindAreas {
    public static void main(String[] args) {
        Figure f = new Figure(10.0, 5.0);
        Rectangle r = new Rectangle(9.0, 5.0);
        Triangle t = new Triangle(10.0, 8.0);
        Figure figref;
        figref = r;
        System.out.println( figref.area() );
        figref = t;
        System.out.println( figref.area() );
        figref = f;
        System.out.println( figref.area() );
    }
}

```

**Output:**

Area of the quadrangle 45.0  
 Area of the triangle 40.0  
 Area of the figure not determined 0.0

What you can see: The syntax of calling the method is the same in the three cases (*figref.area()*), while the result is different. This is because the variant of the method to be invoked is chosen during program execution and depends on the object type, not the type of reference to the object. The objects in the three cases are different; therefore, the method *area()* is implemented differently.

**Example of early binding**

For static class methods, dynamic polymorphism cannot be applied since calling static elements is done by the reference type, not the object type, through which it is accessed. The version of the called static method is always determined at the compilation stage—early binding.

In the example, the superclass **Insurance** has the following elements:

- The static field **LOW**
- A static method *getCategory()* that returns the name of the superclass
- An instance method *getPremium()* that returns the value of the field **LOW**

The class **CarInsurance** is a subclass **Insurance** and has:

- Its own static field **HIGH**
- The static method *getCategory()*, which is overridden and returns the name of the subclass
- The instance method *getPremium()*, which is overridden and returns the value of the field **HIGH**

In the class **Main**, a reference to the superclass *current* is declared and initialized by the subclass object. Then, using the reference to the object and the subclass name, the static method *getCategory()* is called.

```
public class Insurance {
    public static final int LOW = 100;
    public int getPremium() {
        return LOW;
    }
    public static String getCategory() {
        return "Insurance";
    }
}
public class CarInsurance extends Insurance {
    public static final int HIGH = 200;
    @Override
    public int getPremium() {
        return HIGH;
    }
    public static String getCategory() {
        return "CarInsurance";
    }
}
public class Main {
    public static void main(String[] args) {
        Insurance current = new CarInsurance();
        System.out.println("category: " + current.getCategory());
        System.out.println("category: " + CarInsurance.getCategory());
    }
}
```

**Output:**

```
category: Insurance
category: CarInsurance
```

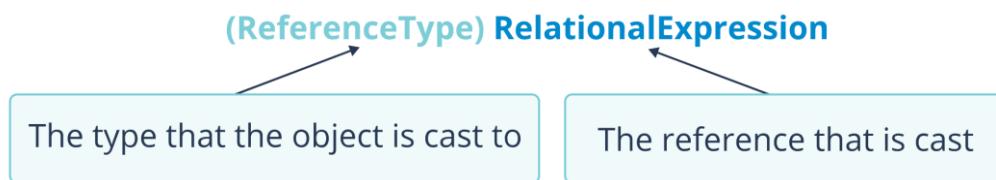
As you can see, calling an option of the static method depends on the type of reference to the object, not the object type. In the given case, the way the method `getCategory()` is called depends on the reference type of the object *current*, meaning the type **Insurance**.

## Object Typecasting. The Operator instanceof

In the context of polymorphism, sometimes it is necessary to get access to all methods of an instance of a successor class when operating through a reference to the parent class. This is called **object typecasting**. It is possible to typecast:

- To the type of the object's class
- To one of its subclasses, superclasses, or interfaces

The syntax for object typecasting looks like the following:



For example:

```
Object obj = "abracadabra";
String str = (String)obj;
```

When trying to cast an object to a class not included in the inheritance hierarchy of the source object, a **ClassCastException** exception will be thrown.

Let's look at an example that describes the superclass **Cat** with the method `move()`. Two of its subclasses are also declared: **BritishCat** and **PersianCat**, which override this method. In the class **Main**, we will create the reference `myCat` of the type **Cat** and initialize it with the object of the class **BritishCat**. If we then cast the reference `Cat` to the type **BritishCat**, everything will be executed correctly since the object is the same. However, now we are looking not just at a cat, but at a British cat. However, if we cast the reference `Cat` to the type **PersianCat**, we will get an execution error—**ClassCastException**. This happens because the types **BritishCat** and **PersianCat** are located in different inheritance branches.

*Click Show More to see the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Cat {
    public void move() { System.out.println("Cat move"); }
}
public class BritishCat extends Cat {
    @Override
    public void move() { System.out.println("British cat move"); }
}
public class PersianCat extends Cat {
    @Override
    public void move() { System.out.println("Persian cat move"); }
}
public class Main {
    public static void main(String[] args) {
        Cat cat = new BritishCat();
```

```

        BritishCat cat2 = (BritishCat)cat;
        PersianCat cat3 = (PersianCat)cat;
    }
}

```

To avoid such an error, you should check the casting correctness using the operator **instanceof**. This operator has the following syntax.

```
<RelationalExpression> instanceof <ReferenceType>
```

The operator **instanceof** returns **true** if the value *RelationalExpression* is not **null** and can be cast to *ReferenceType* without throwing **ClassCastException** (i.e., if the object is an object of this class, one of its subclasses, or one of its superclasses). Otherwise, the result will be **false**.

The result of using this operator in relation to the reference **null** is always **false** because you cannot assign **null** to any type.

In the above example, we had to check this possibility before casting:

```

if(cat instanceof PersianCat) {
    System.out.println("Persian cat!");
    PersianCat cat3 = (PersianCat) cat;
} else {
    System.out.println("Not Persian cat!");
}

```

## Benefits of Polymorphism

How is polymorphism useful?

- Since superclasses and subclasses form a hierarchy as their specialization increases, this allows subclasses to define their own behavior and maintain interface consistency with the superclass. This is possible due to the common form of methods.
- The type of object reference defines the variety of objects you can refer to using this reference.
- Using a reference to a superclass, you call only inherited methods of the superclass; that is, you cannot call the subclass's own methods using an object reference to a superclass.

Moreover, polymorphism has the following benefits.



### Reuse

If there is code that works with a supertype, it can also be used to work with the objects of subtypes. This allows you to substitute object implementation. Testing is based on this.

The compiler performs checks related to the use of data types. This protects you from runtime errors due to incorrect use of various operations with different data types that are part of some family.

## More modular structure

When designing program architecture, the developer can highlight components that they can later replace with other components, thus allowing the code to remain functional.

## Extending flexibility

You can add new data types and apply them to the existing functional code.

You can find more information on polymorphism in the [official documentation](#).

## Conclusion

In this lesson, you have found out that:

- Polymorphism involves providing a single interface to entities of different types or using a single symbol to represent multiple different types.
- The most widely used polymorphism in OOP is the use of a reference to a superclass when manipulating a subclass object.
- Static binding is performed during compilation, and dynamic binding is used during program execution. Static polymorphism is executed faster since there are no dynamic overheads, but additional support from the compiler is required. Dynamic polymorphism is more flexible but is performed slower since a dynamically bound library can work with objects without knowing their complete type.
- Object typecasting involves receiving access to all methods of a successor class instance when operating it through a reference to the parent class. You can check the correctness of this casting using the operator **instanceof**.

## Check Your Knowledge!

1. What is the result of compiling and running the following code?

```
class ParentClass {
    void parentMethod(int i) {
        System.out.println("parentMethod ParentClass" + i);
    }
}
class ChildClass extends ParentClass{
    public void parentMethod(int i) {
        System.out.println("parentMethod ChildClass" + i);
    }
    public void childMethod(int i) {
        System.out.println("childMethod ChildClass" + i);
    }
    public static void main(String args[]) {
        ParentClass quest = new ChildClass(); //1
        quest.parentMethod(1); //2
        quest.childMethod(1); //3
    }
}
parentMethod ParentClass 1
childMethod ChildClass 1
parentMethod ChildClass 1
parentMethod ChildClass 1
Compilation error in line 1
Compilation error in line 2
Compilation error in line 3
```

**Compilation error in line 3**

Answer

Correct:

There will be no error in line 1 since a safe upcasting is performed. A polymorphic method is being called in the second line. A compilation error will occur when trying to call a method that belongs only to the subclass using a reference to the superclass through which it cannot be accessed.

2. What will be printed to console as the result of compiling and running the following code?

```
class Base {
    public void print() {
        System.out.println("Base");
    }
}
class SubClass extends Base {
    public void print() {
        System.out.println("SubClass");
    }
}
public class Main {
    public static void main(String[] args) {
        Base object = new SubClass();
        object.print();
    }
}
```

Compilation error

Base

**SubClass**

Runtime error

Answer

Correct:

When calling a method here, the mechanism of late binding will be used. Therefore, the method will be called not based on the type of reference variable but based on a specific object, which means that the class B method will be called.

3. What will be printed to console as the result of compiling and running the following code?

```
class A {
    String version = "1.0 A";
    String testMethod() {
        return "A";
    }
}
class B extends A {
    String version = "1.0 B";
    String testMethod() {
        return "B";
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        System.out.println(a.version + a.testMethod());
    }
}
```

A compilation error

**1.0 AB**

1.0 AA

1.0 BB

1.0 BA

Answer

Correct:

When calling a method here, the mechanism of late binding will be used. Therefore, the method will be called not based on the type of reference variable but based on a specific object, which means that the class B method will be called. In Java, fields are not polymorphic, so the field will be used based on the reference variable type.

# Method Overloading

## Introduction

When using polymorphism, some methods can be overloaded. In this lesson, you will review what to do in such situations.

## Method Overloading and How to Resolve It

If one class has two methods with the same name, their parameter lists should be different. These methods are considered **overloaded**.

Java distinguishes these methods by their **signature**—this includes **the name of the method and the number and types of parameters**. This allows two or more methods to be defined using the same name while having different parameters within one class.

This happens due to the need to perform similar actions with data of different types and amounts:

- The type of value returned is not considered when resolving overloaded methods.
- Static methods can be overloaded with non-static ones and vice versa (there are no limitations).

Here is an example of declaring overloaded methods *test()* that differ by type or number of parameters:

```
public void test(String s) {
    s = "abcd";
    System.out.println("test(String)");
}
public void test(double dd) {
    System.out.println("test(double)");
}
public double test(int i, double f) {
    System.out.println("test(int, double)");
    return i * f;
}
```



When calling, the available method is called, and the overloading is resolved. This is performed during compilation. The compiler selects the method of execution based on the method's signature and the type of arguments transferred. This process includes the following steps:

Search for the type and candidate methods to be applied, i.e., methods that can be correctly called according to the set arguments.

### STEP 3

Select the method from the candidates that will actually be performed during execution.

Selecting the method to be executed from among the candidates also involves several steps—first, the number and then the types and sequence of data types. This consists of the following steps:

1. The exact match of the number and types of method arguments and parameters must be determined.
2. If there is no exact match, typecasting is used for the arguments:
  - For primitive types—widening
  - For object/reference—to the nearest supertype
3. If typecasting does not work, the mechanism of autoboxing, or unboxing, is used between primitive types and their corresponding class wrappers.
4. If the method has not been found in the previous steps, methods of variable arity are checked (if they are indeed present among the candidates).

Let's look at a few examples of how to resolve overloading at different steps.

*Click on the headings to see the examples.*

## EX 1 AT THE FIRST STEP

Two methods with the name *doJob()* are described in this example:

- The first one with the primitive type of parameter **byte**
- The second one with the reference **Byte**

When the methods are called, the arguments also have the type **byte** and **Byte**. Therefore, we have an exact match of the types of arguments and parameters.

```
public class Main {
    static void doJob(byte b) { System.out.println("byte"); }
    static void doJob(Byte b) { System.out.println("Byte"); }
    public static void main(String[] args) {
        byte b = 5;
        Byte bb = b;
        doJob(b);
        doJob(bb);
    }
}
```

### Output:

byte  
Byte

## EX 2 AT THE SECOND STEP

In this example, three methods with the name *doJob()* are described: with the parameters of the primitive types **byte**, **int**, and **double**. When the methods are called, the arguments have the type **short**, **long**, and **double**. In this case, two types of arguments do not match the types of parameters. Since the arguments fall within the primitive data types, the compiler will first execute widening typecasting and check the result for a match with the parameters. The nearest corresponding type for **short** is **int**, for **long** is **double**.

```
public class Main {
    static void doJob(byte b) { System.out.println("byte"); }
    static void doJob(int i) { System.out.println("int"); }
}
```

```

static void doJob(double d) { System.out.println("double"); }
public static void main(String[] args) {
    short s = 10;
    long x = s;
    double dd = s;
    doJob(s);
    doJob(x);
    doJob(dd);
}
}

```

**Output:**

int  
double  
double

**EX 3 AT THE FIRST STEP FOR REFERENCE TYPES**

In this example, two methods with the name *doJob()* are described, with reference parameters of the type **String** and **Object**. When the methods are called, they receive a reference to a string that is defined by the types **String** and **Object**. Therefore, here we have an exact match of the types of arguments and parameters.

```

public class Main {
    static void doJob(String s) { System.out.println("String"); }
    static void doJob(Object o) { System.out.println("Object"); }
    public static void main(String[] args) {
        String str = "abcd";
        Object obj = str;
        doJob(str);
        doJob(obj);
    }
}

```

**Output:**

String  
Object

When resolving overloading for reference types, it's not the object type that plays the main role but the type of reference to the object.

**EX 4 AT THE SECOND AND THIRD STEPS**

Two methods with the name *doJob()* are described in this example:

- The first one with the primitive type parameter **int**
- The second one with the reference type **Double**

When the methods are called, the arguments have the type **byte** and **Byte**. Therefore, for the first call, the resolution happens at the second step with the typecasting **byte** → **int**. For the second call, when typecasting the reference to a supertype, no relevant method has been found. Next, the compiler performs unboxing of the argument of the type **Byte** into the primitive type **byte**. Then, using widening typecasting, it finds a method with the parameter type **int**.

```

public class Main {
    static void doJob(int i) { System.out.println("int"); }
    static void doJob(Double d) { System.out.println("Double"); }
    public static void main(String[] args) {
        byte b = 5;
        Byte bb = b;
        doJob(b);
    }
}

```

```

        doJob(bb);
    }
}

```

**Output:**

```
int
int
```

**EX 5 AT THE FOURTH STEP**

In this example, three methods with the name `doJob()` are described, with one and two parameters of the reference type **String**, as well as with a variable arity parameter of the same type **String**. Since variable arity methods are checked last for a match, this method will only be performed during the third call when three arguments are specified for the method.

```

public class Main {
    static void doJob(String s) {
        System.out.println("String");
    }
    static void doJob(String s1, String s2) {
        System.out.println("String, String");
    }
    static void doJob(String s1, String... str) {
        System.out.println("String, String...");
    }
    public static void main(String[] args) {
        doJob("hi");
        doJob("hi", "hi");
        doJob("hi", "hi", "hi");
    }
}

```

**Output:**

```
String
String, String
String, String...
```

**EX 6 A COMPILATION ERROR AT THE FOURTH STEP**

In this example, two methods with the name `doJob()` are described:

- The first with one variable arity parameter of the type **String**
- The second with two parameters of the type **String**
- The third with one mandatory parameter of the type **String** and a second variable arity parameter of the type **String**

If we call a method with one argument, we get uncertainty—two methods of the described ones match the call (the first and third). Or, if we call a method with three arguments, we will also get uncertainty—two methods of the ones described match the call (the first and third). Thus, the compiler will throw an error.

```

public class Main {
    static void doJob(String... ss) {
        System.out.println("String...");
    }
    static void doJob(String s1, String s2) {
        System.out.println("String, String");
    }
    static void doJob(String s1, String... str) {
        System.out.println("String, String...");
    }
}

```

```

}
public static void main(String[] args) {
    doJob("hi");
    doJob("hi", "hi", "hi");
}
}

```

## EX 7: AT DIFFERENT STEPS

Two methods with the name `doJob()` are described in this example:

- The first one with two parameters of the types **Object**
- Another one with the first parameter of the type **String** and the second parameter of variable arity of the **Object**

When calling a method with two arguments of the type **String**, the method with two parameters of the type **Object** will be executed (resolution occurs at the second step). When calling a method with arguments of the type **Object** and **String**, the method with two parameters of the type **Object** will be executed (resolution occurs at the second step). Only when calling a method with three arguments of the type **String** will the method with the variable arity parameter of the type **Object** be executed (resolution occurs at the fourth step).

```

public class Main {
    static void doJob(Object obj1, Object obj2) {
        System.out.println("Object, Object ");
    }
    static void doJob(String str, Object... oo) {
        System.out.println("String, Object... ");
    }
    public static void main(String[] args) {
        doJob("hi", "hi");
        doJob(new Object(), "hi");
        doJob("hi", "hi", "hi");
    }
}

```

### Output:

Object, Object  
Object, Object  
String, Object...

### Conclusion

In this lesson, you have seen that:

- If one class has two methods with the same name, their parameter lists should be different. These methods are overloaded.
- When calling, the appropriate method is called so the overloading is resolved. This is performed during compilation. The compiler selects the execution method based on the method's signature and the type of arguments passed. This process includes three steps.

### Check Your Knowledge!

1.What is the result of running the following code?

```

public class Employee {
    Employee(byte b) {
        System.out.println("Good day employee!");
    }
    Employee(long i) {
        System.out.println("Good night employee!");
    }
}

```

```
public class Main {
    public static void main(String[] args) {
        Employee e = new Employee(5);
    }
}
```

**Good night employee!**

null

Good day employee!

A compilation error

A runtime error

Answer

Correct:

Since the argument is an int literal when the constructor is called, and there is no constructor with an int parameter, Java implicitly can only perform widening casts on the argument. As a result, the constructor with a parameter of long type will be executed and the string "Good night employee!" will be displayed.

2. What is the result of running the following code?

```
public class Main {
    static void doJob(Integer num) {
        System.out.println("Integer");
    }
    static void doJob(Object obj) {
        System.out.println("Object");
    }
    static void doJob(int number) {
        System.out.println("int");
    }
    public static void main(String[] args) {
        Object object = 10; //Line1
        doJob(object);
    }
}
```

Integer

Int

**Object**

A compilation error in line //Line1

A runtime error

Answer

Correct:

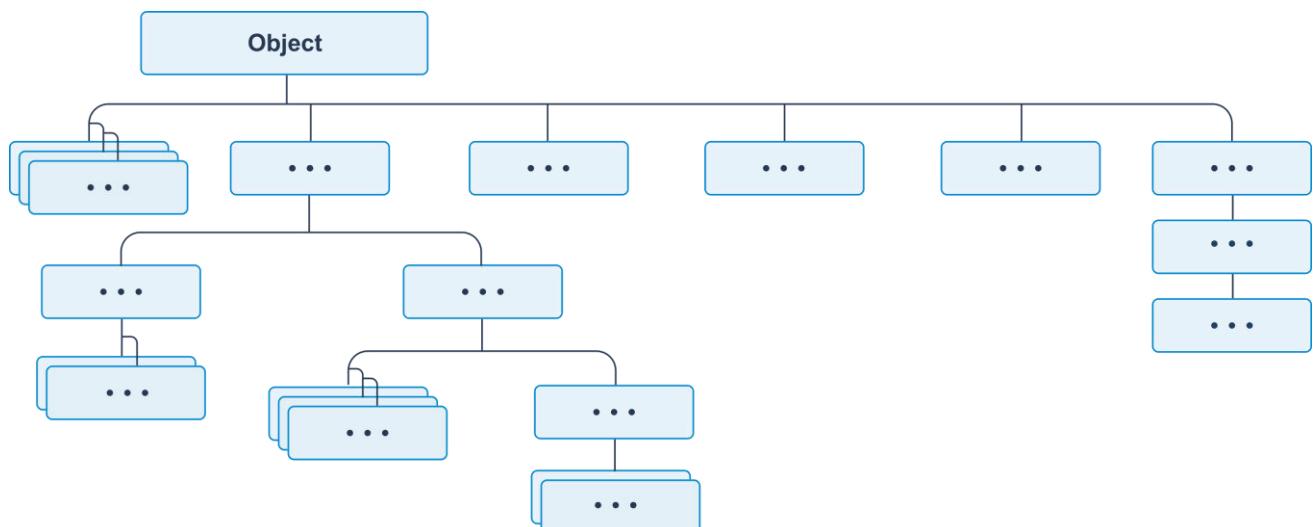
There will be no error in line //Line1 since the number will be autoboxed into an object of the type Integer. The reference to it is assigned to the variable of the type Object. Overloading will then be resolved in the second step of this process.

# The Class Object

## Introduction

In this lesson, you will become familiar with the class **Object**, which is at the top of the inheritance hierarchy for all classes in Java. You will find out what methods the class **Object** has and study some of them in detail. Lastly, you will explore some examples of using methods and the rules for overriding them.

## The Class Object and Its Methods



At the core of the hierarchy of all classes lies the class **java.lang.Object**, the topmost class in Java. **java.lang.Object** is the progenitor (superclass) of all objects and the root class from which all other classes are derived. The methods defined in **Object** are crucial since they appear in each instance of each class everywhere in Java.

Note the following features:

- Each class has the class **Object** as its superclass (implicitly).
- All objects, including arrays, inherit the methods of this class.
- Only simple types cannot be objects: numbers, characters, and logical expressions.

The class **Object** has the following methods:

`equals(Object obj)`

A method that returns the result of comparing two objects.

`hashCode()`

○ A method that returns a unique identifier of an object().

`toString()`

└─ A method that returns a string representation of an object.

`clone()`

A method that returns a copy of an object.

`getClass()`

Returns an object of the type `Class` (a description of the object class).

`wait()`

Puts the thread in standby mode.

`notify()`

Resumes one of the threads that called `wait()` on the same object.

`notifyAll()`

Resumes (notifies) all the threads that launched the method `wait()` at the same object.

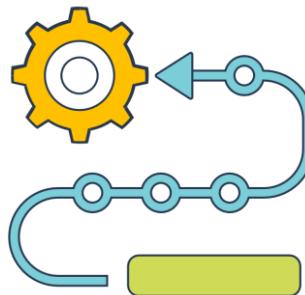
`finalize()`

Performs finalization before the object is destroyed by the garbage collector.

## The Method `toString()`

The method `toString()`, which returns a string representation of an object, is one of the most important methods in the class **Object**. For the class **Object**, this method returns a string that consists of the following:

- A class name
- A character "@"
- An unsigned hexadecimal representation of the object's hash code



If you look at the implementation of this method, we can see the following:

```
getClass().getName() + "@" + Integer.toHexString(hashCode())
```



The method `toString()` should return a short but informative representation of an object. This method should be overridden in all user classes.

The majority of the `toString()` methods return a string that includes the class name followed by the values of its fields in square brackets.

Now let's take a look at how to implement the method `toString()`.

*Click on the arrow to see the description.*

Below you can see how to implement the method `toString()` for the class **Employee**:

```
public String toString() {
    return "Employee [name=" + name
        + ", salary=" + salary
        + ", hireDay=" + hireDay
        + "]";
}
```

Now the string representation of the object of the class **Employee** is returned as follows:

```
Employee [name=..., salary=..., hireDay=...]
```

This method can be improved. Without strict coding of the class name in the method `toString()`, let's call the method `getClass().getName()`, and we will get a string containing the class name:

```
public String toString() {
    return getClass().getName()
        + " [name=" + name
        + ", salary=" + salary
        + ", hireDay=" + hireDay
        + "]";
}
```

In this implementation, the method `toString()` also works with subclasses. A developer creating a subclass should define the method `toString()` and add fields of the subclass. If the superclass uses the call `getClass().getName()`, the subclass simply calls the method `super.toString()`. Below you can find an example of the method `toString()` for the class **Manager**—the successor of the class **Employee**.

```
class Manager extends Employee {
    // ...
    public String toString() {
        return super.toString()
            + " [bonus=" + bonus
            + "]";
    }
}
```

Now the string representation of an object of the class **Manager** is returned as follows:

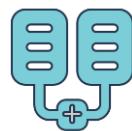
```
Manager [name=..., salary=..., hireDay=...] [bonus=...]
```

It is strongly recommended to override the method `toString()` in each class created.

Pay special attention to some important aspects of working with the method `toString()`.

### COMBINATION WITH A STRING

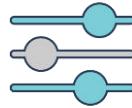
If an object is combined with a string using the operation `+`, the Java compiler automatically calls the method `toString()` to get its current state.



```
Manager boss = new Manager ("John", 1000, 2020, 04, 01, 200)
```

```
String message = "The current position is = " + boss;
```

### PATAMETER FOR PRINTLN



If an object was passed as a parameter to one of the output methods (for example, `println()`), the method `toString()` will be called automatically for the object.

```
Manager boss = new Manager ("John", 1000, 2020, 04, 01)
System.out.println(boss); // It is the same as System.out.println(boss.toString())
```

## LOGGING TOOL

The method `toString()` is a great logging tool. Many classes from the standard library define the method `toString()` to give useful information about the object's state.

```
System.out.println("The current position is " + boss);
```

However, a little later, we will demonstrate that there is an even better solution:

```
Logger.global.info("The current position is " + boss);
```



In short, one of the most important methods of the class **Object** is `toString()`, which returns a string representation of an object.

## The Method `equals()`

The method `equals()` in the class **Object** checks whether two objects are equivalent. The implementation of this method uses the equality comparison operator (`==`).

Note that for primitive data types, the comparison operator for equality gives a correct result, but for objects, it does not; because the method `equals()` checks whether the references to an object are equal, that is, whether the objects being compared are the same. In terms of verification, these actions are justified: Any object will be equivalent to itself.



```
public boolean equals(Object obj) {
    return (this == obj);
}
```

For some classes, nothing else is required. However, in some cases, objects of the same type that have the **same state** should be considered equivalent. Look at the following example to see what can happen if you do not override the method `equals()`:

## EXAMPLE

```
class Point {
    protected int x;
    protected int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class Demo7 {
```

```

public static void main(String[] arg) {
    Point point1 = new Point(5, -5);
    Point point2 = point1;
    Point point3 = new Point(5, -5);
    Point point4 = new Point(5, 5);
    System.out.println(point1.equals(point2));
    System.out.println(point1.equals(point3));
    System.out.println(point1.equals(point4));
}
}

```

## Output:

true  
false  
false

When overriding the method `equals()`, the equivalence relation specified by the Java language:

*Click on the + signs to see more information.*



## Reflection

If you take any value of the reference  $x$  not equal to null, the result of calling  $x.equals(x)$  should be **true**.

## 184 Symmetry

If we take any values of the references  $x$  and  $y$  not equal to null, the result of calling  $x.equals(y)$  should be the same as the result of calling  $y.equals(x)$ .

## Transitivity

If we take any values of the references **x**, **y**, and **z** not equal to **null**, and the results of calling **x.equals(y)** and **y.equals(z)** are **true**, then the result of calling **x.equals(z)** should also be **true**.

## Consistency

If we take any values of the references **x** and **y** not equal to null, then when calling **x.equals(y)** multiple times, the returned value (**true** or **false**) should always be the same. A situation when we have no information on the changes in the objects used to compare will be an exception.

## Comparison with null

If we take any value of the reference **x** not equal to null, the result of calling **x.equals(null)** should be **false**.

Let's solidify what you have learned so far by considering the following example, which shows the overridden method **equals()**.

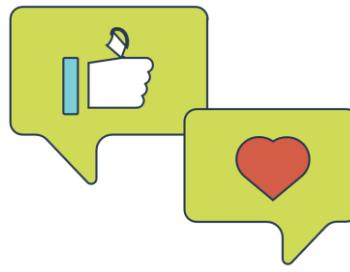
*Click on the dropdown list to study the example.*

```
class Point {
    protected int x;
    protected int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (this.getClass() != obj.getClass()) return false;
        Point other = (Point) obj;
        return this.x == other.x && this.y == other.y;
    }
}
public class Demo7 {
    public static void main(String[] args) {
        Point point1 = new Point(5, -5);
        Point point2 = point1;
        Point point3 = new Point(5, -5);
        Point point4 = new Point(5, 5);
        System.out.println(point1.equals(point2));
        System.out.println(point1.equals(point3));
        System.out.println(point1.equals(point4));
    }
}
```

## Output:

```
true
true
false
```

As you have seen, the method **equals()** in the class **Object** checks whether two objects are equivalent. Now you will explore some basic recommendations for creating this method.



In this example, you will review some basic recommendations for creating the method `equals()`.

### STEP 1: CHECKING REFERENCES FOR EQUIVALENCE

First, you must check to see if the references **this** and **obj** are identical. The expression below is used to optimize the check: It is a lot faster to check the equivalence of references than to compare fields of objects.

```
if (this == obj) return true;
```

### STEP 2 CHECKING THE NULL REFERENCE

Next, you have to find out if the reference **obj** is **null**. If it is, the value **false** must be returned.

```
if (obj == null) return false;
```

### STEP 3 COMPARING CLASSES

It is necessary to compare the classes **this** and **obj**. If the verification semantics can change in the subclass, you need to use the method `getClass()`.

```
if (this.getClass() != obj.getClass()) return false;
```

### STEP 4 USING INSTANCE OF

If the verification principle remains valid for all subclasses, you should use the operation **instanceof**.

```
if (!(obj instanceof NameClass)) return false;
```

### STEP 5 TRANSFORMATION INTO VARIABLE

It is necessary to transform the object **obj** into a variable of the necessary class.

```
NameClass other = (NameClass) obj;
```

### STEP 6: COMPARING FIELDS

Finally, you need to compare all the fields. For fields of primitive types, use the comparison operator `(==)` for equality, while for object fields, use the method `equals()`. If all the fields of the two objects match, the value **true** is returned; otherwise, **false** is returned.

```
return field1 == other.field1 && field2.equals(other.field2) && ...;
```

Important! If the method `equals()` is overridden in a subclass, you need to include the call `super.equals(other)` in it.

Take a look at the following code fragment:

```
public class Point {
    ...
    public boolean equals(Point other) {
        return this.x == other.x && this.y == other.y;
    }
}
```

Executing this code will cause an error. However, beginning with Java SE 5.0, there is a way to prevent this from happening: You specify that the development method aims to replace the relevant superclass method by using the descriptor `@Override`:

```
@Override
public boolean equals(Object obj)
```

If you define a new method incidentally, the compiler will throw an error. Suppose the class **Point** has the following code string:

```
@Override
public boolean equals(Point obj)
```

Since this method does not override any method defined in the superclass **Object**, an error will be thrown.

## The Method `hashCode()`

The method **`hashCode()`** returns the value of the object's hash code. This method is defined in the class **Object**. Therefore, each object has a hash code set by default.

*Click on the + signs to learn more about each component.*



### Hash function

A **hash function** is any function that can be used to map arbitrary-sized digital data to fixed-size digital data.

### Hash value

A **hash value**—also known as a hash code, hash sum, or simply a hash—is the result of executing a hash function.

A hash of object in Java is an integer generated as a result of applying a hash function at a specific object. The hash code can be treated as a cipher: If `x` and `y` are different objects, the results of calling `x.hashCode()` and `y.hashCode()` will likely be different.

The method **`hashCode()`** is required to work with collections—for example, **HashMap**.

In the class **Object**, method `hashCode()` calculates the hash code by converting the address of the object's location in the memory into an integer.

Let's take a look at an example.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Demo8 {
    public static void main(String[] args) {
        Point point1 = new Point(5, -5);
        Point point2 = point1;
        Point point3 = new Point(5, -5);
        Point point4 = new Point(5, 5);
        System.out.println(point1.hashCode());
        System.out.println(point2.hashCode());
        System.out.println(point3.hashCode());
        System.out.println(point4.hashCode())
    }
}
```

### Output:

1484678  
1484678  
22052786  
32487478

The method *hashCode()* should always be overridden when the method *equals(Object)* has been overridden.

When overriding the method *hashCode()*, three rules should always be followed:

- When executing a Java application, if the method used to calculate the hash code is called at the same object several times, it should return the same value, unless the object has changed.
- When two objects are equal (i.e., the result of calling *equals(Object)* is true), calling the method used to calculate the hash code for each of the two objects should return **exactly the same** result.
- When two objects are not equal (i.e., the result of calling *equals(Object)* is false), calling the method used to calculate the hash code for each of the two objects should return **different** results.

The method *hashCode()* should return an integer (which can be negative)—a unique identifier that, in most cases, depends only on the values of the objects' properties. To ensure that different objects have unique hash codes, it is enough to combine the hash codes of the instance fields.

The following video will help bring together everything you have learned about the method *equals(Object)* and provide some basic recommendations for calculating *hashCode()*.

*Watch the video to learn more.*

### The methods *equals* and *hashCode*

Take a look at an example of overriding the method *hashCode()*.

*Click on the dropdown list to study the example.*

```
public class Student {
    private String name;
    private long phone;
    private int age;
    // ...
    @Override
    public int hashCode() {
        int result = 17;
```

```

        result = 31 * result + name.hashCode();
        result = 31 * result + (int) (phone ^ (phone >>> 32));
        result = 31 * result + age;
        return result;
    }
}

public class Demo9 {
    public static void main(String[] arg) {
        Student stud1 = new Student("Peter", 5558956L, 20);
        Student stud2 = new Student("Ivan", 9876543L, 18);
        Student stud3 = new Student("Dasha", 5558956L, 20);
        Student stud4 = new Student("Ivan", 9876543L, 18);
        System.out.println(stud1.hashCode());
        System.out.println(stud2.hashCode());
        System.out.println(stud3.hashCode());
        System.out.println(stud4.hashCode());
    }
}

```

**Output:**

1160475683  
-1786389060  
-1015000986  
-1786389060



The methods *equals()* and *hashCode()* should be compatible: if *x.equals(y)* returns true, then the results of executing *x.hashCode()* and *y.hashCode()* should also match.

**Conclusion**

In this lesson, you have seen that:

- Every class has the class Object as its superclass, and the class Object includes certain methods.
- The method *toString()* returns a string representation of an object.
- The method *equals()* checks whether two objects are equivalent.
- The method *hashCode()* returns the value of the object's hash code.

**Check Your Knowledge!**

1. Which of the following are methods in the class Object?

**toString()**

**equals()**

**println()**

**hashCode()**

**notifyAll()**

correct

Answer

Correct:

The methods of the class Object include *toString()*, *equals()*, *hashCode()*, and *notifyAll()*.

2. Which of the following methods overrides the method *equals()*?

**boolean equals(Object o) {return true; }**

**public boolean equals(Boo b) {return true; }**

**public boolean equals(Object o) {return true; }      correct**

**public void equals(Boo b) { }**

Answer

Correct:

The method **public boolean equals(Object o) {return true; }** overrides the method *equals()*.

3. Which of the following methods overrides the method hashCode()?

**int hashCode() {return 5; }**  
**public void hashCode() { }**  
**public int hashCode(Boo b) {return 5; }**  
**public int hashCode() {return 5; }correct**

Answer

Correct:

The method **public int hashCode() {return 5; }** overrides the method hashCode().

# Abstract Classes

## Introduction

In object-oriented programming, abstraction is one of the key concepts on which encapsulation, inheritance, and polymorphism are based. The main goal of abstraction is to reduce program complexity by hiding unnecessary details from the user. In this lesson, you will study the notion of abstraction in programming in detail, explore the abstraction mechanism, and learn to create abstract data types (classes) and methods.

## The Notion of Abstraction

**Abstraction** involves viewing a complex entity as a single system and performing actions with it without delving into the details of its internal structure and functioning.

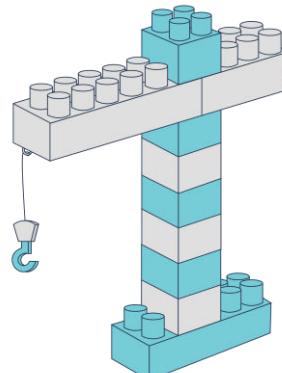
We can consider the following situation as an example. Imagine that a driver is driving a car. At that moment, the driver is not thinking about the chemical composition of the car's paint, how the sprockets interact in the car's transmission, or how the shape of the car's body affects speed (well, maybe except for when the driver is in bumper-to-bumper traffic and has nothing else to do). However, the driver regularly uses the steering wheel, pedals, and turn signal.

Therefore, **abstraction** is a way to single out a set of significant properties of an object while excluding those that are not significant. Abstractions allow us to decompose a domain into a set of concepts and interactions between them.



*Click on the dropdown to study the example.*

In the given image, a mechanism is built from a child's constructor set. Nevertheless, you can easily recognize the entity—a crane. This means that you understand its functionality and can interact with it without delving into the details of the materials used to build it.



There is also such a concept as the **level of abstraction**, which can be high or low.

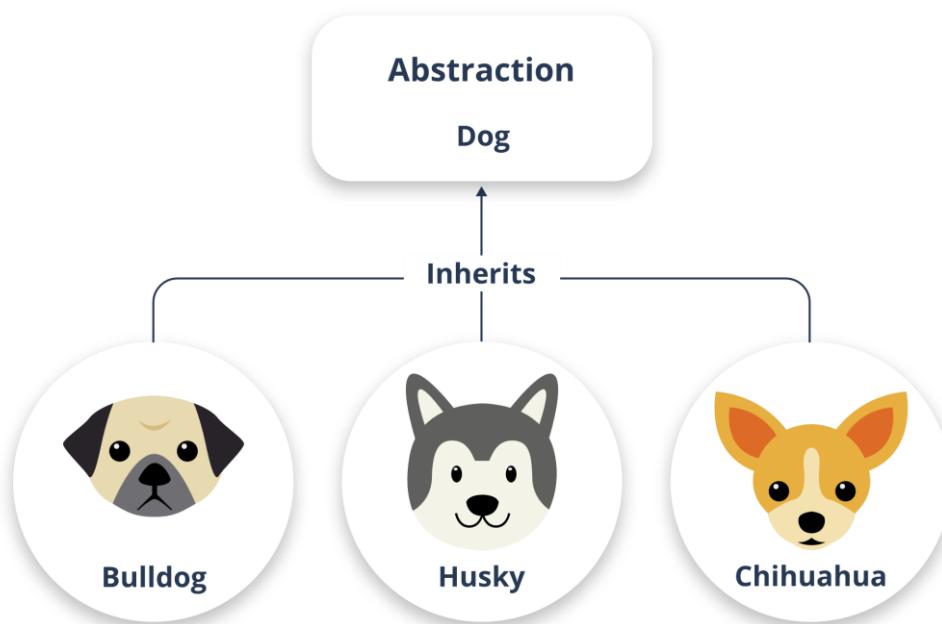
A high level of abstraction gives just an approximated description of an object and does not allow it to model its reliable behavior.

A low level of abstraction makes a model too complex, overloaded with details, and thus unfit for use.

For example, most children see a cat as something soft and fluffy.

For example, after disassembling your car down to the last screw, you forget how to put it back together.

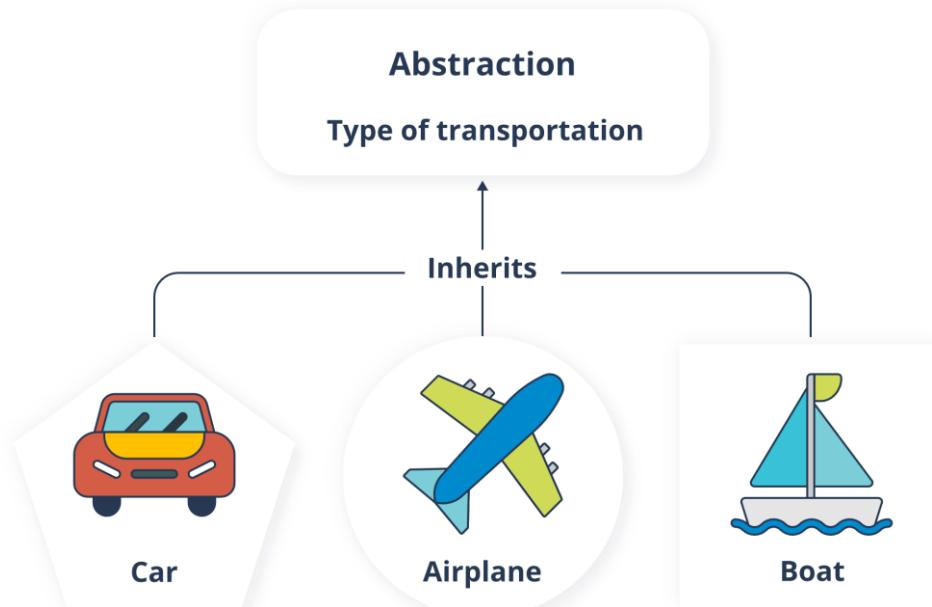
In OOP, abstraction is manifested in the description of classes. In other words, classes are mechanisms of grouping and generalization. Thus, grouping is achieved by the fact that similar objects are assigned to the same class, and generalization is achieved by the hierarchy of classes. It is important to understand that abstraction does not represent the entire object but only a set of its significant properties.



Let's consider what abstract data types provide:

1. **Grouping** related operations and data
2. **Simplification** due to building a higher level of abstraction
3. **An opportunity to model** real entities
4. **Isolation of complexity** or simplification due to hiding implementation details
5. **Better** code readability and understanding
6. **Limiting** the impact of changes
7. **Locality** of code changes

All this reduces program complexity.



Therefore, abstraction is a way to single out a set of an object's significant properties and exclude the insignificant ones from the review. In short, abstraction helps reduce program complexity.

## Abstract Classes and Methods

Let's take a look at the basics of abstract data types (classes) and methods.

In the Java language, an **abstract class** is a class declared by the keyword **abstract**. It can have declared fields as well as methods with implementation and without it (abstract methods). It is important to understand that you cannot instantiate abstract classes but that they can be superclasses. You may say that this is a class that defines a generalized form used by all its subclasses, which means that this class generally determines how they behave.

Consider the following example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
abstract class Vehicle {
    protected void move() {
        System.out.println("Vehicle move");
    }
}
```

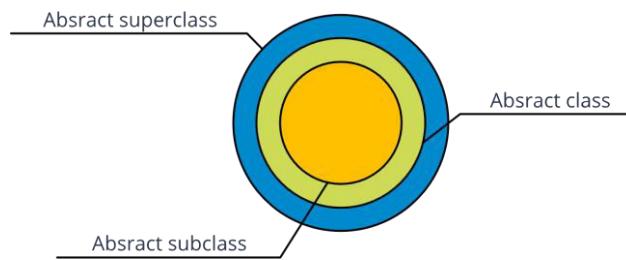
An **abstract method** is a method that contains the keyword **abstract** in its description and does not have a body (a semicolon is placed after the list of parameters). For example, for the mentioned earlier class **Vehicle**, the method **move()** can be described as follows:

```
protected abstract void move();
```



**Abstract methods are not implemented in an abstract class** since an abstract class might not know how this method should work. And each inheritor (subclass) should decide on its own how to implement this method.

*Click on the + signs to see more information.*



## Abstract superclasses

If an abstract class is a superclass, its subclass ensures that all the abstract methods in its parent class are implemented.

## Abstract classes

If a class includes abstract methods, it must be declared as an abstract class.

## Abstract subclasses

If a subclass does not ensure implementation of all the abstract methods in its parent class, it must also be declared as an abstract class.

## Examples of Implementing Abstract Classes

Below are some examples of implementing abstract classes.

*Click on the dropdown to study the example.*

In this example, the class **Animal** is declared as an abstract superclass with an abstract method *move()*. Then the class **Reptiles** is declared as its subclass without any code. As a result, we get a compilation error since the class **Reptiles** inherited the abstract method *move()* and provided no implementation for it. This means that it must be *declared explicitly* as an abstract class.

*Hover your cursor over the info icon to see an explanation of the code.*

```
abstract class Animal {  
    public abstract void move();  
}
```

```
class Reptiles extends Animal {  
}
```

Correct:

*Hover your cursor over the info icon to see an explanation of the code.*

```
abstract class Animal {  
    public abstract void move();  
}
```

```
abstract class Reptiles extends Animal {  
}
```

In this example, the class **Animal** is declared as an abstract superclass with the abstract method *move()*. Then its subclass **Reptiles** is also declared as an abstract class without any code. And then, we declared a inheritor of the

class **Reptiles**—the subclass **Boa** (as a regular class implementing the method *move()*). In the class **Main**, we create a reference of the type **Animal** and initialize it with an object of the subclass **Boa**, where we later call the method *move()*. Everything is working correctly—dynamic polymorphism.

*Hover your cursor over the info icon to see an explanation of the code.*

```
abstract class Animal {
    public abstract void move();
}

abstract class Reptiles extends Animal { }

class Boa extends Reptiles {
    @Override
    public void move() {
        System.out.println("Boa move");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Boa();
        animal.move();
    }
}
```



A reference to an abstract class can be initialized with an object of its subclass if it is not abstract. You can only call methods of the abstract superclass using this reference.

In this example, an abstract class **GraphicObject** is declared with two fields, *x* and *y*, an abstract method for drawing shapes *draw()*, and a method to move the shapes *moveTo()* with implementation. Next, its subclass **Circle** is declared, which contains an empty implementation of the method *draw()*, meaning that the class is not abstract and we can create objects for it. In the class **Runner**, a reference of the type **GraphicObject** is created that is initialized by an object of the type **Circle**. At this object, the method *draw()* is called. Everything is working correctly.

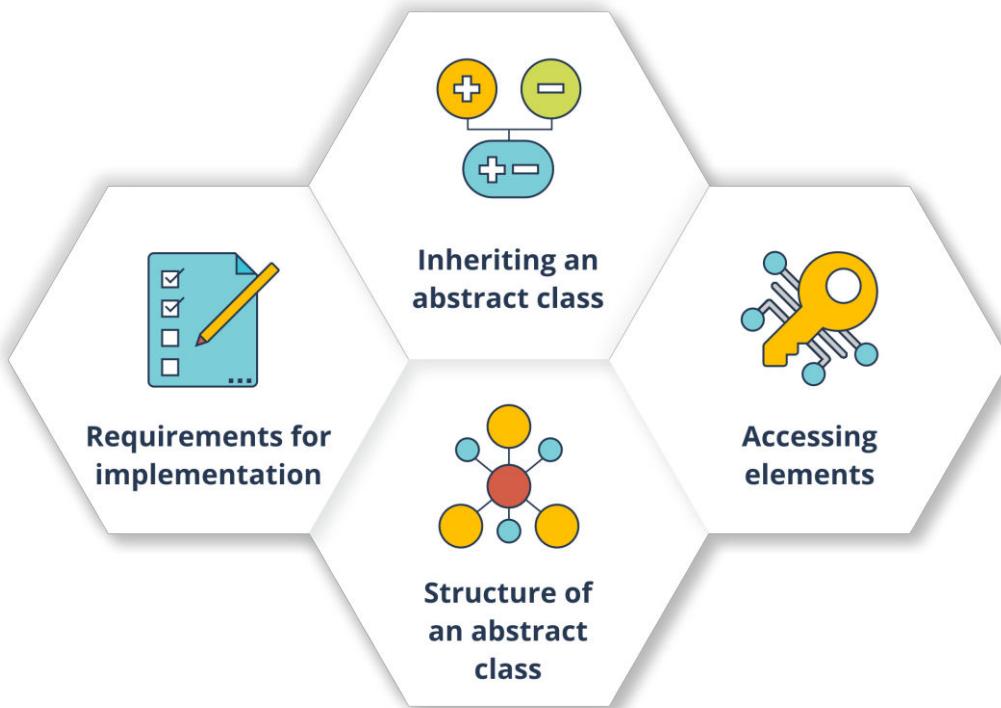
```
public abstract class GraphicObject {
    public abstract void draw();
    public void moveTo(int x, int y) { }
}

public class Circle extends GraphicObject {
    @Override
    public void draw() {
        // implementation drawing a circle
    }
}

public class Runner {
    public static void main(String[] args) {
        GraphicObject mng;
        // mng = new GraphicObject();
        // object can not be created!
        mng = new Circle();
        mng.draw();
        mng.moveTo(10, 10);
    }
}
```

Let's summarize what you have learned about abstract classes and methods.

Click on the + signs to see more information.



## Requirements for implementation

When inheriting an abstract class, if the subclass represents a specific entity, it should implement the abstract methods of its superclass. A subclass can override abstract methods of its superclass with the same or less limited visibility.

## Inheriting an abstract class

An abstract class can extend another abstract class, but it should not necessarily implement the abstract methods of its superclass.

## Accessing elements

Abstract methods and other elements of an abstract class can be declared with any visibility except **private**. It makes no sense to create private abstract methods since no one can implement them.

## Structure of an abstract class

Abstract classes can have fields, abstract methods, and fully implemented methods.

## Conclusion

In this lesson, you have seen that:

- Abstraction is a general concept that can be observed in the real world and in OOP languages.
- Any objects in the real world or classes that conceal internal details help to provide abstraction.
- Abstractions significantly simplify the processing of program code, reducing complexity and breaking code into smaller parts.

## Check Your Knowledge!

1. Which of the following statements about abstract classes are true?

An abstract class must contain abstract methods.

A inheritor of an abstract class that does not implement all of its abstract methods should be declared as an abstract class.

You cannot create an object in an abstract class.

A reference to an abstract class cannot be used polymorphically.

correct

Answer

Correct:

A inheritor of an abstract class that does not implement all of its abstract methods should be declared as an abstract class. You cannot create an object in an abstract class.

2. Which statements about abstract methods are true?

An abstract method is a method that does not have a body and should be declared with the keyword *abstract*.

An abstract method can be declared with any access except *private*.

An abstract method can have a body, but it should be indicated with the keyword *abstract*.

A class can have a maximum of one abstract method.

correct

Answer

Correct:

An abstract method is a method that does not have a body and should be declared with the keyword *abstract*. It can be declared with any access except *private*.

3. Which of the following statements about implementing abstract classes are true?

If a subclass represents a specific/real entity, it should implement all the abstract methods in its superclass.

When overriding an abstract method, you cannot change its visibility.

One abstract class can be inherited from another without implementing the abstract methods in the superclass.

One abstract class can be inherited from another and should implement the abstract methods in the superclass.

correct

Answer

Correct:

If a subclass represents a specific/real entity, it should implement all the abstract methods in its superclass. One abstract class can be inherited from another without implementing the abstract methods in the superclass.

4. Will the following code be compiled?

```
public abstract class Book {
    public static final String type;
    public abstract String getType();
}
```

Yes

No correct

Answer

Correct:

The field of the final type should be initialized.

# Concept of a Nested Class

## Introduction

In this lesson, you will find out what nested classes are and why they are used. You will also study the different types of nested classes.

## Nested Classes



A **nested class** is a class that is described inside another class. A nested class is always connected to an enclosing/limiting/outer class that limits and determines its scope of action. If a nested class is declared within the scope of action of an outer class, it will be a member of the class. It is also possible to declare a nested class within a block, thus making it a local class.

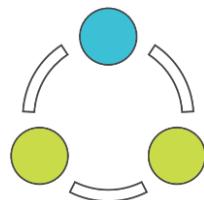
Nested classes are used for several reasons.

*Click on the arrows to study the reasons for using nested classes.*

### Grouping classes logically

A logical grouping of classes that are used only in one place:

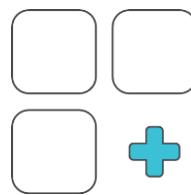
- If the current class is used by only one other class, it makes sense to enclose it in that class and to keep them together.
- Enclosing such "auxiliary" classes makes the structure of classes more streamlined and organized.



### Enhancing encapsulation

Let's consider an example of two upper-level classes—**A** and **B**:

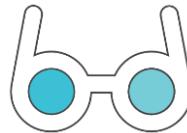
- Class **B** needs access to the elements in class **A**. If there were no such need, the elements of class **A** would be declared as **private**.
- By hiding class **B** in class **A**, the elements in class **A** can be declared as **private** and that class **B** can access them directly.
- Class **B** itself can also be hidden from the surrounding environment.



## Improving code readability

Nested classes can also make code more readable and easier to maintain:

- Nested classes in small top-level classes place code closer to where it is used.

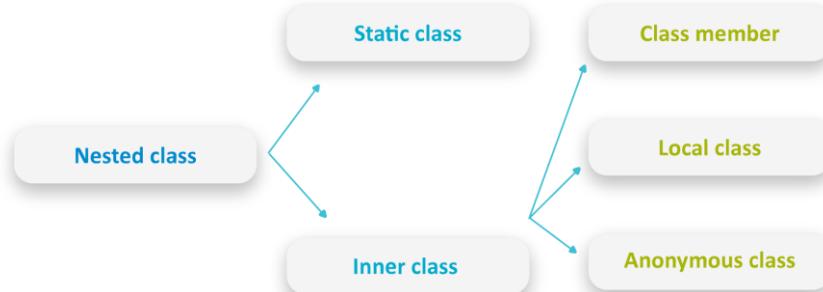


Therefore, a nested class is an element of the class that encloses it. Using such classes improves code readability and enhances encapsulation. They can also be used to group classes in a logical way.

## Classifying Nested Classes

There are two types of nested classes. One is declared using the access modifier **static**; these classes are called **static nested classes**. Classes without this modifier are called **inner classes**. These classifications are shown in the diagram below.

*Click on the active elements to see the description of classes.*



### Nested class

A nested class is an element of the enclosing (outer) class. It can be declared with any access level—**private**, **public**, **protected**, or **package-private**—depending on where it is described.

### Static class

A static class is a class described inside another class with the modifier **static**; it has **direct access only to static** elements (members) of the class that encloses it.

An inner class is a non-static nested class that has **direct** access to the elements of the class that encloses it, even when they are declared as **private**.

## Class member

A class member is simply a class that is defined in the body of the class that encloses it. This class is most often referred to as the inner class.

## Local class

A local class is a class defined in a block of some class. For instance, a method body can act as a block.

## Anonymous class

An anonymous class is a class without a name. If it is necessary to create a sole class object, there is no need to assign a name to this class.

You can learn more about nested classes in [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- A nested class is a class that is described inside another class.
- Nested classes are used to group classes logically, enhance encapsulation, and make code easier to read and maintain.
- One category of nested classes is declared using the access modifier static; these classes are called static nested classes. Classes that lack this modifier are called inner classes. Inner classes are divided into local classes, anonymous classes, and class members.

## Check Your Knowledge!

1. Which of the following modifiers can be applied to nested classes?

final abstract

public

private

protected

local

correct

Answer

Correct:

Nested classes have all the usual properties of ordinary classes; therefore, they can be declared with the modifiers final, abstract, and public. Nested classes can also limit their visibility with the help of the modifiers private and protected.

2. Which of the following are types of nested classes?

Static nested classes

Local classes

Multiclasses

Anonymous classes

Inner classes

Interwoven classes

correct

Answer

Correct:

Nested classes can be static, inner, local, and anonymous. There is also the category class members.

3. Which of the following statements about inner classes are true?

They are described inside another class without the modifier static.

They are described inside another class with the modifier static.

They are members of another class and can be declared with any access.

They are declared only inside a block of another class.

They are nested classes described only with public access.

correct

Answer

Correct:

An inner class is a non-static nested class with direct access to the elements of the class that encloses it, even when they are declared as private.

## Inner Classes

### Introduction

In this lesson, you will continue exploring nested classes. You will take a closer look at inner classes, find out why and how they are used, and learn the rules for working with these classes.

### Inner Classes

An **inner class** is a non-static nested class described in the body of another class. An inner class has access to all the fields and methods of the enclosing class and can access them directly. Sometimes an inner class is used to serve its enclosing class.

Unlike top-level classes, an inner class can be private. As soon as you declare an inner class as private, the objects lying beyond this class will no longer have access to it.



To create an inner class, simply describe it inside another class. An example of the syntax is given below:

```
[public] class OuterClass {
    //...
    <access> class InnerClass {
        // ...
    }
}
```

For example, if an **outer class** is a **screen** that represents a pixel matrix (TV dots), an **inner class** is a **pixel** describing a single dot (its coordinates, color, etc.).



To create an inner class, simply describe it inside another class. An example of the syntax is given below:

```
[public] class OuterClass {  
    //...  
    <access> class InnerClass {  
        // ...  
    }  
}
```

For example, if an **outer class** is a **screen** that represents a pixel matrix (TV dots), an **inner class** is a **pixel** describing a single dot (its coordinates, color, etc.).

Inner classes are used when their objects cannot exist or are not needed outside the body of the enclosing class.

Let's take a closer look at the specifics of inner classes, including how to use them.

*Click on the arrows to study each aspect.*

### Relation to an object of an enclosing class

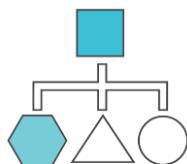


- Just like instance methods and fields, an inner class is related to an instance of its enclosing class.
- An instance of an enclosing class can be related to several instances of its inner class.
- An instance of an inner class is only related to one instance of its enclosing class.
- **Static context**



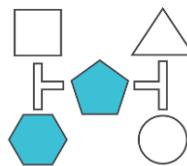
- Inner class can define static elements inside itself since Java 16.

### Access to elements of the enclosing class



- The instance of the inner class has direct access to the fields and methods of its enclosing class's instance.

## Access to elements of the inner class



- To use an instance of an inner class, an instance of its enclosing class must be created first.

Below are the rules for describing and using inner classes and their objects.

*Click on the headings to explore the rules.*

### INVOKING ELEMENTS OF INNER AND OUTER CLASSES

In this example, we describe the outer class **Ship**, where the following elements are described:

- The private instance field **x**
- The instance method *testing()*
- The inner class **Engine** with the **protected** access

The inner class contains a description of the method *test()*, in whose code we directly invoke the field **x** of the outer class.

To invoke the method of the inner class, the outer class first creates its own object: the body of the method *testing()*.

*Hover your cursor over the info icon to see an explanation of the code.*

```
class Ship {
    private int x = 10;
    protected class Engine {
        public void test() {
            x = 20;
        }
    }
    public void testing() {
        Engine eng = new Engine();
        eng.test();
```

To get an instance of an inner class **outside the body of its enclosing class**, it is necessary to:

- Create an instance of the outer class
- Create an instance of the inner class within the object of the outer class with the following **syntax**:

<outer class name> <name of outer class's object> = new <constructor of outer class>;

<outer class name>.⟨inner class name⟩ <name of inner class's object> =  
  <name of outer class's object>.new <constructor of inner class>;

For example:

```
Ship ship = new Ship();
Ship.Engine engine = ship.new Engine();
}
```

To get an instance of an inner class **outside the body of its enclosing class**, it is necessary to:

- Create an instance of the outer class
- Create an instance of the inner class within the object of the outer class with the following **syntax**:

```
<outer class name> <name of outer class's object> = new <constructor of outer class>;
```

```
<outer class name>.⟨inner class name⟩ <name of inner class's object> =
<name of outer class's object>.new <constructor of inner class>;
```

For example:

```
Ship ship = new Ship();
Ship.Engine engine = ship.new Engine();
```

Getting and using an object of an inner class outside the body of its outer class is only possible if the inner class has the necessary visibility scope (i.e., it is not hidden in the description of the outer class).

Let's take a look at an example of using an inner class to serve its outer class. In other words, this is an example of transferring or hiding how operations are implemented from the object of an outer class in its inner class.

In the example:

- The outer class **Outer** is described containing the array of integers **nums**, which is initialized by a constructor of this class, along with the method *analyze()*.
- Using an object of the inner class **Inner**, the method *analyze()* prints information about the minimal and maximal array elements to console, as well as information about the average value of its elements.
- The operations used to search for information are described in the methods of the inner class.

```
public class Outer {
    private int[] nums;
    Outer(int[] n) {
        nums = n;
    }
    public void analyze() {
        Inner inOb = new Inner();
        System.out.println("Min: " + inOb.getMin());
        System.out.println("Max: " + inOb.getMax());
        System.out.println("Avg: " + inOb.getAvg());
    }
    class Inner {
        public int getMin() {
            int min = nums[0];
            for (int i = 1; i < nums.length; i++) {
                if (nums[i] < min)
                    min = nums[i];
            }
            return min;
        }
        public int getMax() {
            int max = nums[0];
            for (int i = 1; i < nums.length; i++) {
                if (nums[i] > max)
                    max = nums[i];
            }
            return max;
        }
        public double getAvg() {
            double avg = 0.0;
            for (int element: nums) {
```

```

        avg += element;
    }
    return avg / nums.length;
}
}

public class DemoInnerClass {
    public static void main(String[] args) {
        int[] x = {1, 2, 3, 5, 6, 7, 8, 9};
        Outer outOb = new Outer(x);
        outOb.analyze();
    }
}

public void analyze() {
    Inner inOb = new Inner();
    System.out.println("Min: " + inOb.getMin());
    System.out.println("Max: " + inOb.getMax());
    System.out.println("Avg: " + inOb.getAvg());
}

class Inner {
    public int getMin() {
        int min = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < min)
                min = nums[i];
        }
        return min;
    }

    public int getMax() {
        int max = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > max)
                max = nums[i];
        }
        return max;
    }

    public double getAvg() {
        double avg = 0.0;
        for (int element: nums) {
            avg += element;
        }
        return avg / nums.length;
    }
}

public class DemoInnerClass {
    public static void main(String[] args) {
        int[] x = {1, 2, 3, 5, 6, 7, 8, 9};
        Outer outOb = new Outer(x);
        outOb.analyze();
    }
}

```

## Output

Min: 1  
 Max: 9  
 Avg: 5.0

You can learn more about inner classes in [Oracle's official documentation](#). For more examples of using inner classes, check out [this article](#).

In this lesson, you have seen that:

- An inner class is a non-static nested class described in the body of another class. An inner class has access to all the fields and methods of the enclosing class and can access them directly.
- These classes are used when their objects cannot exist or are not needed outside the body of the enclosing class.

You also studied the specifics of and rules for describing and using inner classes and their corresponding objects.

## Check Your Knowledge!

1. Given the following code, which example of creating an instance of an inner class is correct?

```
class Owner {  
    class Inner { }  
}  
new Owner.Inner();  
Owner.new Inner();  
new Owner.new Inner();  
new Owner().new Inner();  
correct  
Owner.Inner();  
Answer  
Correct:
```

To create an object of an inner class, first you need to create an object of the external class.

## Local Classes

### Introduction

In this lesson, you will continue exploring inner classes by focusing on local classes. You will become familiar with the specifics of using local classes and study a few examples.

### Local Inner Classes

**Local classes** – are inner classes declared in the block of an outer class where a method's body is the common block.  
Syntax:

```
class Outer {
    // ....
    void method() {
        class LocalClass {    }
        // ....
    }
}
```



Local classes are usually used to conceal the implementation of actions of their outer class.

Let's take a closer look at local classes and how to use them.

*Click on the cards for more information about local classes.*

#### VISIBILITY SCOPE

- Local classes are only visible within the block where they are declared.
- An instance of a local class can be created inside the block but below its description.

By default, local classes have **package-private** access. They cannot be declared as **private**, **public**, **protected**, or **static**.

## STATIC CONTEXT

Local classes could not contain static elements (fields, methods and classes) inside them to Java 16. The exception was fields of type **static final**. These restrictions have now been lifted.

## ACCESS TO THE ELEMENTS OF ENCLOSING CLASS

A local class has access to all elements of its enclosing class.

## ACCESS TO LOCAL VARIABLES

A local class has access to local variables of the enclosing block that are declared as **final** (for Java 7 and earlier) and **effectively final** (for Java 8 and later).

**Effectively final** refers to a variable or parameter whose value never changes after initialization.

Next, you will consider the rules for describing and using local classes, as well as some examples of applying them.

## EXAMPLES OF ACCESS TO ELEMENTS OF THE OUTER CLASS

In this example, we describe the outer class **Ship** with the private field **x** and the method **work()**. This method contains a description of the local variable **y**, which is only initialized and does not change its value. Therefore, Java interprets this variable as **effectively final**. The method **work()** also contains a description of the local class **LocalClass** with the **test()** method. This method demonstrates that the local class has direct access both to the private field of the outer class **x** and to the local variable **y**, despite the fact that the visibility scope of the class is limited only by the method **work()**.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Ship {
    private int x = 10;
    void work() { // start method
        int y = 10;
        class LocalClass { // local class
            public void test() {
                x = 20;
                System.out.println(x + " " + y);
            }
        }
    } // end method
}
```

## INCORRECT USAGE OF LOCAL VARIABLES

In this example, we describe the outer class **Ship** with the private field **x** and the method **work()**. This method contains a description of the local variable **y**, which is initialized and then changes its value. Therefore, the variable **y** is no longer **effectively final**. The method **work()** also contains a description of a local class **LocalClass** with the **test()** method. This method demonstrates that the local class has direct access to the private field of the outer class **x**, whereas it can no longer access the local variable **y**.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Ship {
    private int x = 10;
    void work() {
        int y = 10;
        class LocalClass {
            public void test() {
                x = 20;
                System.out.println(x + " " + y);
            }
        }
        y++;
    }
}
```

## HIDING AN IMPLEMENTATION

In this example:

- The interface **Read** is described—with one abstract method, **readLabel()**.
- The class **DemoLocal** is described and contains:
  - The field **y** of the type **int**
  - The method **dest()** with the immutable parameter **s**. All this method does is create an object of the local class **PD** described in its body.

The local class **PD** implements the interface **Read** and overrides its method **readLabel()** as follows: To the value of its field **label**, which is initialized with a constructor when creating an object, it adds the value of the field **y** of the outer class and adds the value of the parameter **s** of the **dest()** method.

In the method **main()**, an object of the outer class **DemoLocal** is created, where the method **dest()** is invoked with the argument "QQQQQ". This method returns an object of the inner class **PD** through a reference to the interface **Read**. Then, the method **readLabel()** is invoked on this reference, and the result is printed to console.

Therefore, in the inner class, we concealed how a mark is formed—which we can get later by calling the interface **method**.

Hover your cursor over the info icon to see an explanation of the code.

```
public interface Read {
    String readLabel();
}
public class DemoLocal {
    private int y = 33;
    public Read dest(final String s) {
        class PD implements Read {
            private String label;
            PD(String st) {
                label = st;
            }
            public String readLabel() {
                return label + y + s;
            }
        }
        return new PD("qqqqq");
    }
    public static void main(String[] str) {
        DemoLocal demoLocal = new DemoLocal();
        Read read = demoLocal.dest("QQQQQ");
        System.out.println(read.readLabel());
    }
}
```

### Output:

qqqqq33QQQQQ

For more about local classes, see [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- Local classes are inner classes declared in the block of an outer class where a method's body is the common block.
- Local classes are usually used to conceal the implementation of actions of the outer class.

You also studied the specifics of describing and using local classes and analyzed different examples.

## Check Your Knowledge!

1. What is the result of running the following code?

```
public class Outer {
    private String x = "Outer";
    void doStuff() {
        class Inner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
            }
        }
        Inner mi = new Inner();
        mi.seeOuter();
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.doStuff();
    }
}
```

Outer x is Outer correct  
Outer x is null

A compilation error

A runtime error

Answer

Correct:

An object of an outer class with the field x = "Outer" invokes its own method—an object of a local class is created, and the local class reads the field x of the class Outer directly.

2. What is the result of running the following code?

```
public class Outer {
    private String x = "Outer";
    void doStuff() {
        class Inner {
            Inner mi = new Inner();
            mi.seeOuter();
            public void seeOuter() {
                System.out.println("Outer x is " + x);
            }
        }
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.doStuff();
    }
}
```

Outer x is Outer

Outer x is null

A compilation error correct

A runtime error

Answer

Correct:

In the class Inner, the method mi.seeOuter() is invoked at a place for declaring fields and methods, throwing a compilation error.

## Anonymous Classes

### Introduction

In this lesson, you will study one more type of inner nested classes—anonymous classes. You will review the specifics of and reasons for using them and also learn how to describe them.

### Anonymous Inner Classes



An **anonymous class** is a class without a name. If it is necessary to create a sole class object, then there is no need to assign a name to this class. An anonymous class does not use constructors. Instead of constructors, it is possible to use dynamic initialization sections.

The Oracle documentation gives a good *[recommendation](#)*: "Use anonymous classes if you need to use a local class only once."

The place where an anonymous class is described refers to expressions for creating objects, meaning that this class is declared at the same time an object of some type is created. As such, the following syntax is used:

```
new <class/interface> ( [<list of arguments>] ) {
    <body of anonymous class>
};
```

 Anonymous classes are used to create an interface object or an abstract class, i.e., for a sole implementation of abstract methods.

Now let's look at the specifics of describing and using anonymous classes.

*Scroll the horizontal slider to learn more.*

## Visibility scope

Anonymous classes are only visible within the expression where they are declared.

## Access modifier

By default, anonymous classes have **package-private** access. They cannot be declared as **private**, **public**, **protected**, or **static**.

## Absence of static context

Anonymous classes cannot have static elements inside them—fields, methods, or classes. Fields of the **static final** type are an exception.

## Elements of an anonymous class

Anonymous classes can have their own fields, methods, and dynamic initialization sections, but they cannot declare constructors.

## Access to elements of the enclosing class

Anonymous classes have direct access to all the elements in their enclosing class.

## Access to local variables

An anonymous class has access to local variables of the **final** type or the **effectively final** block where the expression with the class is described.

Next, look at the rules for describing and using anonymous classes and some examples of how to apply them.

*Click on the headings to learn more.*

### Using an anonymous class to implement an interface

In this example, we have the interface **CustomTest** with the abstract method *test()*. We have the outer class **Ship** with the method *doJob()*, where the reference *tst* to the interface **CustomTest** is created. This reference is initialized by an interface object, for which an anonymous class is described implementing the method *test()*. Then on the reference *tst*, we invoke the *test()* method. Thus, the method *test()* of the interface is implemented only for one object next to the place where it is used.

*Hover your cursor over the info icon to see an explanation of the code.*

```
213 public interface CustomTest {
    void test();
}
214 public class Ship {
    void doJob() {
```

```

CustomTest tst = new CustomTest () {
    public void test() {
        System.out.print("TEST");
    }
};
tst.test();
}
}

```

#### ▼ Access to elements of an outer class

In this example:

- We have described the interface **CustomTest** using the abstract method *test()*.
- An outer class **Ship** is described with the private field *x* and the method *doJob()*, which includes:
  - A description of the local variable *y* of the **final** type
  - The creation of the reference *tst* of the type **CustomTest**
  - Invocation of the *test()* method on that reference

The reference *tst* is assigned to an object that is an instance of an anonymous class. The anonymous class is declared as implementation of the **CustomTest** interface. To make it possible we need to provide implementation of the abstract method *test()* right inside declaration of the anonymous class. Besides, the anonymous class has its own field *z*, and a dynamic initialization block is described.

The method *test()* demonstrates access to the private field *x* of the class **Ship** and the local variable *y* of the method *doJob()*, despite the fact that this is a method of the anonymous class, which is part of the expression used to create an object in a method of a different class.

*Hover your cursor over the info icon to see an explanation of the code.*

```

public interface CustomTest {
    void test();
}

public class Ship {
    private int x = 10;
    void doJob() {
        final int y = 20;
        CustomTest tst = new CustomTest() {
            private int z = 10;
            { System.out.print("Init block"); }
            public void test() {
                System.out.print(x + " " + z + " " + y );
            }
        };
        tst.test();
    }
}

```

#### ▼ Anonymous classes and the interface Comparator

You already know that the **Comparator** interface makes it possible to define a rule for comparing data so it can be organized. This interface has one abstract method, *compare()*, to compare two objects. To implement the **Comparator** interface, it is convenient to use an anonymous class because:

- Only one method is implemented.
- Only one object of this type is required.
- The implementation code will be placed close to where it is used.

Let's look at an example. Suppose we are describing an array of strings that will be organized in reverse alphabetical order using the static method `sort()` of the class **Arrays** and will then be printed to console. This method needs an object of the type **Comparator**, which is created as the second method argument and is implemented by the anonymous class. Here we do not even need to create a reference of the **Comparator** type since it would only be used as an argument of the method `sort()` and nowhere else.

*Hover your cursor over the info icon to see an explanation of the code.*

```
String[] arr = {"java", "scala", "fortran", "ada", "modula"};
Arrays.sort(arr, new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        String str1 = (String) o1;
        String str2 = (String) o2;
        return str2.compareTo(str1);
    }
});
System.out.println(Arrays.toString(arr));
```

### Output:

[scala, modula, java, fortran, ada]

For more about anonymous classes, see [the official Oracle documentation](#).

## Conclusion

In this lesson, you have seen that:

- An anonymous class is a class without a name. In addition, if it is necessary to create a sole class object, there is no need to assign a name to this class.
- Anonymous classes are used to create an interface object or an abstract class, i.e., for a sole implementation of abstract methods.

You have also studied the rules for describing and using anonymous classes and analyzed different examples.

## Check Your Knowledge!

1. Which of the following statements about anonymous classes are true?

They have package-private access.

They can be abstract.

They can be marked as public.

They have access to private elements of the enclosing class.

They can be marked as static.

correct

Answer

Correct:

By default, anonymous classes have package-private access. Anonymous classes have direct access to all the elements of their enclosing class.

2. What is the result of running the following code?

```
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}
class Food {
    Popcorn p = new Popcorn() {
        public void sizzle() {
            System.out.println("anonymous sizzling popcorn");
        }
        public void pop() {
```

```

        System.out.println("anonymous popcorn");
    }
};

public static void main(String[] args) {
    Food food = new Food();
    food.p.pop();
    food.p.sizzle();
}
}

popcorn
anonymous sizzling popcorn
A runtime error
anonymous
anonymous sizzling popcorn
A compilation error
correct
Answer
Correct:
```

Only overridden methods can be invoked from an anonymous class through a reference. Methods declared in an anonymous class cannot be accessed from outside. Therefore, the invocation food.p.sizzle(); is not possible.

3. What is the result of running the following code?

```

interface Cookable {
    public void cook();
}

public class Food {
    Cookable c = new Cookable() {
        public void cook() {
            System.out.println("anonymous cookable implementer");
        }
    };
}

public static void main(String[] args) {
    Food food = new Food();
    food.c.cook();
}
```

anonymous cookable implementer correct

null

A compilation error

A runtime error

Answer

Correct:

The method cook() of the Cookable interface is implemented by the anonymous class when the field c of the Food class is created. Therefore, invoking this method on the object c will not cause any errors.

## Static Nested Classes

### Introduction

In this lesson, you will study static nested classes in more detail. You will explore the specifics of describing and applying them as well as analyze some examples of how they are used.

### Static Nested Classes

A **static nested class** is a class described inside another class with the modifier **static**. A static nested class cooperates with its enclosing class (and other classes) in the same way as any other top-level class. In other words, **behavior-wise**, this is a top-level class that was nested inside another top-level class to make it easier to pack.

A static nested class has the following syntax:

```
[public] class OuterClass {  
    //...  
    <access> static class StaticNestedClass {  
        // ...  
    }  
}
```

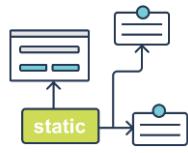
On the following cards, you can view the specifics of describing and using static nested classes.

*Click on each card to learn more.*



Just like class methods and fields, a static nested class is related to its enclosing class.

- Just like static class methods, a static nested class cannot directly access the fields and methods of an instance defined in its enclosing class.
- A static nested class can use the fields and methods of an instance of its enclosing class only through a reference to its object.



A static nested class has direct access only to the static fields and methods of its enclosing class.

Below you can review the rules for describing and using static classes, as well as some examples of how they are applied.

*Click on the headings to learn more.*

#### ▼ Access to elements of an outer class

In this example, we have the outer class **Ship**, which contains the public instance field **x** and the private class field (static) **y**, as well as the static nested class **Boat**.

In the class **Boat**, the **test()** method is described and demonstrates access to the elements of the outer class **Ship**. Specifically, it shows that the static nested class has no direct access to the **x** field of the outer class despite the fact that the field is described as public.

Since an instance of the nested static class is not related to any of the enclosing class instances, the only possibility to reach non-static members of the enclosing class within the inner class is to create or somehow get an instance of the enclosing class. In the example below, we create an instance of the **Ship** class inside the "Boat.test()" method to get access to the **x** field of the **Ship** class.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Ship {
    public int x = 10;
    private static int y = 10;
    public static class Boat {
        public void test() {
            x = 20;
            Ship sh = new Ship();
            sh.x = 20;
            y = 20;
        }
    }
}
```

## Using an object of a static class outside its enclosing class

Static nested classes, just like top-level classes, can be used outside the body of the enclosing class. However, to invoke them, the name of the enclosing class needs to be specified:

<name of outer class>.<name of static class>

To get an instance of a static nested class **outside the body** of its enclosing class, use the following syntax:

```
<name of outer class>.<name of static class> <name of object of static class> =  
new <name of outer class>.<constructor of static class>;
```

For example:

```
Ship.Boat boat = new Ship.Boat();  
boat.test();
```



When using a reference to a static class, you must specify that it is part of an outer class (just as in the case of static class elements).

Let's consider an example of using a static nested class within the following problem.

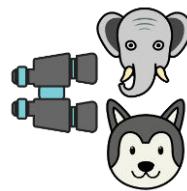
**Problem:** Find the minimum and maximum numbers in an array.

*Click on the arrow to see the steps of the solution.*

### Problem solution option No. 1

Usually, two methods are created—one to find the maximal element and one for the minimal one.

**Drawback:** When invoking both methods, the array is checked twice. Therefore, it is more efficient to check the array only once, determining the maximum and minimum values at the same time. However, in this case, the method should return two values, which the syntax does not allow

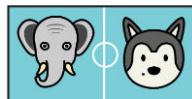


### Problem solution option No. 2

Define a class **Pair** that contains two fields and is used to encapsulate the result of the method's work.

**Drawback:** The name of the **Pair** class is too common, and when executing a large project, another developer might decide to use this name for a class that will encapsulate strings instead of numerical data, for example.

**Eliminating the drawback:** This potential issue can be resolved by making the **Pair** class a nested class that is defined inside the class that performs operations on the array.



## Implementation

A class **ArrayOperation** is described with a static class **Pair** and a static method to search for the maximum and minimum values *searchMinMax()* in the passed array. The search result returns a method that wraps these values into an object of the class **Pair**. In this example, we also have the class **DemoNestedMain**, where an array of real values is created. Then a reference to the static nested class **ArrayOperation.Pair** is described and initialized by an object returned by the method *searchMinMax()*. By using getters of the nested class, we get the minimum and maximum values of the array.



## Example

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class ArrayOperation {
    public static class Pair {
        private double min;
        private double max;
        public Pair(double f, double s) {
            min = f;
            max = s;
        }
        public double getMin() {
            return min;
        }
        public double getMax() {
            return max;
        }
    }
    public static Pair searchMinMax(double[] values) {
        double min = values[0];
        double max = values[0];
        for (double v: values) {
            if (min > v) {
                min = v;
            }
            if (max < v) {
                max = v;
            }
        }
        return new Pair(min, max);
    }
}
public class DemoNestedMain {
    public static void main(String[] args) {
        double[] array = new double[20];
        for (int i = 0; i < array.length; i++) {
            array[i] = 100 * Math.random();
        }
    }
}
```

```

        ArrayOperation.Pair pair = ArrayOperation.searchMinMax(array);
        System.out.println("min = " + pair.getMin());
        System.out.println("max = " + pair.getMax());
    }
}

```

For more about static nested classes, see [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- A static nested class is a class described inside another class with the modifier static.
- Behavior-wise, a static nested class is a top-level class nested inside another top-level class to make packing easier.

You also explored the specifics of describing and using static nested classes and analyzed some examples of applying them.

## Check Your Knowledge!

1. Which of the following statements about static nested classes are true?

A static class has no direct access to the elements of an instance its enclosing class.

The fields and methods of a static class should contain the modifier static.

If the external class Outer contains the static class Nested, then to create an object of the nested class, developer should use new Outer.Nested().

A static class should extend its enclosing class.

A static class can implement interfaces.

A static class can be inherited only from another static class.

A static class cannot be inherited from an inner class.

A static class can be inherited only from top-level classes.

correct

Answer

Correct:

A static class has no direct access to the elements of an instance its enclosing class. A static class can implement interfaces. A static class cannot be inherited from an inner class. If the external class Outer contains the static class Nested, then to create an object of the nested class, we should use new Outer.Nested().

2. What is the result of running the following code?

```

public class Outer {
    private int x = 7;
    static class CustomStatic {
        private int x = 8;
        public void seeOuter() {
            System.out.println(" x is " + x);
        }
    }
    public static void main(String[] args) {
        CustomStatic obStatic = new CustomStatic();
        obStatic.seeOuter();
    }
}

```

x is 7

x is 8 correct

A compilation error

A runtime error

Answer

Correct:

An instance field x of a static nested class hides the visibility of an instance field x of the enclosing class. So, in the line of code: "System.out.println(" x is " + x);" there is a call to the field x of the static class, and no error occurs.

3. Which option used instead of //insert\_code\_here will print "spooky" to console?

```

public class Tour {
    public static void main(String[] args) {

```

```

// insert_code_here
    s.go();
}
}

public class Cathedral {
    static class Sanctum {
        void go() {
            System.out.println("spooky");
        }
    }
}

Sanctum s = new Sanctum();
Sanctum s = new Cathedral.Sanctum();
Cathedral.Sanctum s = new Cathedral.Sanctum();
correct
Cathedral.Sanctum s = Cathedral.new Sanctum();
Answer
Correct:
```

An object in a static nested class is created using the name of its enclosing class as a mark of being part of it. To create an object of a static class, it is not necessary to create an object of the external class.

4. What is the result of running the following code?

```

public class Outer {
    private int x = 10;
    public void makeStatic() {
        Nested obSt = new Nested();
        obSt.seeOuter();
    }
    static class Nested {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
        }
    }
    public static void main(String[] args) {
        new Outer().makeStatic();
    }
}
```

Outer x is 10

Outer x is 0

A compilation error correct

A runtime error

Answer

Correct:

A static nested class has no direct access to non-static fields of its enclosing class. A compilation error will be thrown in the line

```
System.out.println("Outer x is " + x);
```

## Specifics of Nested Classes

### Introduction

This lesson will help you organize what you have learned so far about the visibility of nested classes.

### Visibility Scope of Nested Classes

Now that you have studied the various types of nested classes, let's summarize the relationship between a nested class and its enclosing class.

#### What can you see and access from a nested class inside an enclosing class?

*Click on the tabs to learn more.*

**From an inner class**

**From a static nested class**

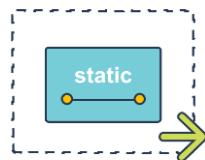
**From a local class**

**From an anonymous class**

- All (even **private**) fields and methods of the enclosing class (instance and static)
- **public** and **protected** fields and methods of a superclass of the enclosing class (instance and static, i.e., those that are visible in the enclosing class)



- All static fields (even **private**) and methods of the enclosing class
- **public** and **protected** static fields and methods of the superclass of the enclosing class (those that are visible in the enclosing class)



- All fields (even **private**) and methods of the enclosing class—instance and static
- **public** and **protected** fields and methods of the superclass of the enclosing class—instance and static (those that are visible in the enclosing class)
- Local variables of the block (**final** and **effectively final**) where the local class is described



- All fields (even **private**) and methods of the enclosing class—instance and static
- **public** and **protected** fields and methods of the superclass of the enclosing class—instance and static (those that are visible in the enclosing class)
- Parameters of the type **final** or **effectively final** of the expression where the anonymous class is described



**What is the visibility scope of a nested class, and what determines it?**

*Click on the active elements to learn more.*



Static  
class



Local  
class



Anonymous  
class

**For an inner class**

According to the access modifier

**For a static class**

According to the access modifier

**For a local class**

Only in the scope of the block (method) where it is defined: from the place of description until the end of the block

**For an anonymous class**

Only at the place where it is defined—in other words, in the expression that creates an object of an anonymous class

**Which data types can be extended by nested classes? Which data types can act as superclasses or interfaces?**

*Click on the arrows to learn more.*

**For an inner class**

Top-level classes and interfaces

- Other inner classes from the same outer class

**For a static class** Top-level classes and interfaces

- Other static nested classes from the same outer class

**For a local class** Top-level classes and interfaces

- Other inner classes from the same outer class
- Other local classes defined in the same block

**For an anonymous class**

Any data types

**Which data types can extend nested classes? For which data types can nested classes act as superclasses?**

*Scroll the elements to learn more.*

**Inner class**

Can be a supertype for another inner class from the same outer class or its subclasses

**Static class**

Can be a supertype for any class, both nested and top-level

**Local class**

Can be a supertype for other local classes defined in the same block

### **Anonymous class**

Cannot be a supertype

### **What elements can be described in nested classes?**

#### **Inner**

class

Can contain descriptions not only of instance fields and methods, but also static elements as well, beginning from Java 16.

#### **Static**

class

Can contain descriptions of any elements, just like a top-level class

#### **Local**

class

Can contain descriptions not only of instance fields and methods, but also static elements as well, beginning from Java 16.

#### **Anonymous**

class

Can contain descriptions of only instance fields and methods, with the exception of fields of the type **final static**, as well as dynamic initialization sections

To learn more, see [Oracle's official documentation](#).

## Conclusion

This lesson has served to organize what you learned previously about nested classes. All the properties of nested classes are summarized in the following table:

Type	Related to an object of the outer class?	Place of description	Visibility
Static nested class	No	As a member of a different class	Depends on the access modifier
Inner class	Yes	As a member of a different class	Depends on the access modifier
Local class	Yes (if not defined in a static method)	Inside a method	From the place of description until the end of the method
Anonymous class	Yes (if not defined in a static method)	In the expression where the object is defined	No

# The Class String

## Introduction

In this lesson, you will find out what strings are and why they are needed in Java. You will also review some of the basic operations that are performed with strings.

## Strings in Java

A **string** is a sequence of characters. In Java, strings are implemented with the help of three classes of the module [java.base](#) of the [java.lang](#) package: **String**, **StringBuilder**, and **StringBuffer**.

The `java.lang` package is connected automatically. All three classes are declared as **final**. This means classes cannot be extended. To format and process strings, the **Formatter**, **Pattern**, and **Matcher** classes are used.

## The Class String

Most of the data that processed by real applications is represented by text. Therefore, **String** is one of the classes that is used most often in Java.

There are several things about the class **String** to keep in mind:

- It is generally accessible (i.e., there is no need to import a package).
- It is final, which means it cannot be subclassed.
- It implements the interfaces `Serializable`, `Comparable<String>`, and `CharSequence`.
- An object of the class **String** is also a string and cannot be changed—i.e., it is immutable.

You can create objects of the class **String** in different ways. You already touched upon this topic in Module 2 when you studied string literals. The following video will help you recall what you studied and introduce you to other methods used to create strings.

You can perform different operations with strings. The class **String** has a number of methods for this.

*Click on the dropdown element to study the methods.*

#### ▼ Methods of the class String

##### Method prototype

Method prototype	Purpose of the method
char <i>charAt(int pos)</i>	Returns the char value at the specified index
String <i>concat(String s) or "+"</i>	Concatenates the specified string to the end of this string
boolean <i>endsWith(String suffix)</i>	Tests if this string ends with the specified suffix
boolean <i>equals(Object ob)</i>	Compares this string to the specified object
boolean <i>equalsIgnoreCase(String s)</i>	Compares this String to another String, ignoring case considerations
static String <i>format(String format, Object... args)</i>	Returns a formatted string using the specified format string and arguments
void <i>getChars(&lt;parameters&gt;)</i>	Copies characters from this string into the destination character array
int <i>indexOf(int ch)</i>	Returns the index within this string of the first occurrence of the specified character
String <i>intern()</i>	Returns a canonical representation for the string object
int <i>lastIndexOf(char c)</i>	Returns the index within this string of the last occurrence of the specified character
int <i>length()</i>	Returns the length of this string
boolean <i>matches(String regex)</i>	Tells whether or not this string matches the given <a href="#">regular expression</a>
String <i>repeat(int count)</i>	Returns a string whose value is the concatenation of this string repeated count times
String <i>replace(char oldChar, char newChar)</i>	Returns a string resulting from replacing all occurrences of oldChar in this string with newChar
String <i>replaceAll(String regex, String replacement)</i>	Replaces each substring of this string that matches the given regular expression with the given replacement
String <i>split(String regex)</i>	Splits this string around matches of the given regular expression
boolean <i>startsWith(String prefix)</i>	Tests if this string starts with the specified prefix
String <i>substring(int beginIndex)</i>	Returns a new string that is a substring of this string
String <i>substring(int beginIndex, int endIndex)</i>	Returns a new string that is a substring of this string
String <i>toLowerCase()</i>	Converts all the characters in this String to lower case using the rules of the default locale
String <i>toUpperCase()</i>	Converts all the characters in this String to upper case using the rules of the default locale
String <i>trim()</i>	Returns a copy of the string, omitting leading and trailing whitespace
static String <i>valueOf(&lt;value&gt;)</i>	Returns the string representation of the specified base type value

You can find out more about the methods of the class **String** in the [official documentation](#).

Now it's time to move on to the main operations performed with strings:

- Joining
- Comparing
- Formatting

#### • Joining Strings

- **Joining, or concatenation,** is an operation used to join strings in the result of which you get one string received from adding the second string at the end of the first one.



The result of joining should be placed in a variable so that it can be used later.

- In Java, there are two ways to join strings: using the method ***concat()*** and the overloaded operators "+" and "+=".

*Click on the tabs to see the examples.*

**Method concat()**

**Overloaded operators**

The method ***String concat(String str)*** does not make changes in a string. It creates a new string (a new String object) as a result of joining the current string and the string passed to the method as a parameter.

```
String str1 = "Learning ";
String str2 = "java!";
String str3 = str1.concat(str2);
System.out.println(str3);
```

**The output to console is:**

Learning java!



If concatenation only needs to be performed once, it is better to use the method ***concat()***; in other cases, it is recommended to use the operator "+" / "+=" or methods of the classes **StringBuilder/StringBuffer**.

If a non-string value (a primitive or an object of another type) is concatenated, this value is transformed into a string. This transformation is done by explicitly invoking the method ***toString()*** at the object "Class Object". This method is also often invoked when returning objects.

*Click on the dropdown element to study the example.*

**Example**

```
public class Student {
    private String name;
    public Student(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + "]";
    }
}
public class Main {
    public static void main(String[] args) {
        Student john = new Student("John Smith");
        System.out.println("First student: " + john);
    }
}
```

**The output to console is:**

First student: Student [name=John Smith]

Now that you have explored the most common methods of the **String** class, it's time to take a look at some examples of how to use them.

Click on each dropdown list to learn more.

### String length

Each string has a length. To find the length, you need to use the method ***length()***, which returns the number of characters in the string.

```
String str0 = "";
String str1 = "Learning ";
String str2 = "java!";
String str3 = str1 + str2;

System.out.println("str0 - " + str0.length() + " symbols");
System.out.println("str1 - " + str1.length() + " symbols");
System.out.println("str2 - " + str2.length() + " symbols");
System.out.println("str3 - " + str3.length() + " symbols");
```

**The output to console is:**

str0 - 0 symbols  
 str1 - 9 symbols  
 str2 - 5 symbols  
 str3 - 14 symbols

### ^ Working with indices in a string

Each character in a string has its own **index**—the position number. If you know the value of the index, you can return the respective string character. To do this, you need to use the method ***charAt(int index)***.

It is important to remember that:

- Indexation starts from 0.
- The index of the last character is one unit smaller than the length of the string.



```
String str = "Software And Hardware!";
char aChar0 = str.charAt(0);
System.out.println(aChar0);
System.out.println(str.charAt(9));
System.out.println(str.charAt( str.length() - 1));
```

**The output to console is:**

S  
 A  
 !

If the index specified as the parameter is not included in the string boundaries (negative or is not smaller than the length of this string), the exception ***StringIndexOutOfBoundsException*** will be thrown.

When you need to extract an array of characters simultaneously, you should use the method ***getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)***, where:

- **srcBegin** is the first index in the string; it is required to start the extraction of characters.
- **srcEnd** is the last index in the string for which characters will be extracted.
- **dst** is the array where the extracted characters will be placed.
- **dstBegin** is the index in the array **dst**, starting from which we need to add the characters extracted from the string.

```
String str = "Software And Hardware!";
int start = 9;
int end = 12;
char[] dst = new char[end - start];
str.getChars(start, end, dst, 0);
System.out.println(dst);
```

**The output to console is:**

And

▼ **Returning a substring**

To return a substring from a string, we need to use the method ***substring(int beginIndex)*** or ***substring(int beginIndex, int endIndex)***.

*Hover your cursor over the info icon for an explanation of the code.*

```
String str = "Software And Hardware!";
String substr1 = str.substring(13);
System.out.println(substr1);
System.out.println(str.substring(0, 8));
System.out.println(str.substring(13, 17));
```



**The output to console is:**

Hardware!

Software

Hard

▼ **Locating a character in a string**

To find a character (i.e., determine its index) in a string, the following methods of the class **String** are used:

- ***indexOf(int ch)***
- ***indexOf(int ch, int fromIndex)***
- ***lastIndexOf(int ch)***
- ***lastIndexOf(int ch, int fromIndex)***

*Hover your cursor over the info icon for an explanation of the code.*



```
String str = "Software And Hardware!";
int i1 = str.indexOf('a');
int i2 = str.lastIndexOf('a');
System.out.println(i1);
System.out.println(i2);
System.out.println(str.indexOf('x'));
```

**The output to console is:**

5  
18  
-1

If the character being searched for is not found in the source string, the methods will return the value "-1".

#### ▼ Locating a substring in a string

To find a substring (i.e., to determine the index of the first occurrence) in a string, use the following methods of the class **String**:

- ***indexOf(String str)***
- ***indexOf(String str, int fromIndex)***
- ***lastIndexOf(String str)***
- ***lastIndexOf(String str, int fromIndex)***

*Hover your cursor over the info icon for an explanation of the code.*

```
String str = "String in java is a sequence of characters java";
int i1 = str.indexOf("java");
int i2 = str.lastIndexOf("java");
System.out.println(i1);
System.out.println(i2);
System.out.println(str.indexOf("java", 20));
```

**The output to console is:**

10  
43  
43

Similar to the previous methods, if the substring being searched for is not found in the source string, the methods will return the value "-1".

#### ^ Replacing strings

232 To replace any character or substring with another character or substring, it is necessary to use one of the following methods:



- `replace(char old, char new)`
- `replaceAll(String old, String new)`

```
String str = "Welcome";
System.out.println(str.replace('e', 'A'));
System.out.println(str.replaceAll("el", "AB"));
```

**The output to console is:**

WA~~l~~comA  
WA~~B~~come

Both methods replace all occurrences and return a new object of the class **String**, and the source string under the str reference does not change!

#### ▼ Transforming strings

Thus, you have studied the methods for joining strings, as well as methods of the class **String**. Now it's time for comparing strings.

## Comparing Strings

There is often a need to compare strings, or check them for equivalence. This is a very important and useful operation that involves several approaches:

`boolean equals(Object anObject)`

This method is overridden from the class **Object** and compares strings with regard to the case.

`boolean equalsIgnoreCase(String anotherString)`

This method compares strings without regard for the case.

Pay attention to the following example.

*Click on the dropdown list to see the example.*

Example

`regionMatches()`

This method compares certain substrings within two strings.

This method has two forms:

Form 1: **boolean regionMatches(int toffset, String other, int ooffset, int len)**

Form 2: **boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)**, where:

- **ignoreCase** is a flag specifying that the letter case does not need to be considered when comparing (if the flag has a **true** value, the case is not considered).
- **toffset** is an index from which the comparison starts in the string where we call this method.

- **other** is a string with which the calling string is compared.
- **oofset** is the index.
- **len** is the number of characters to be compared in the two strings.

Take a look at the following example.

*Click on the dropdown list to see the example.*

**Example**

```
String str1 = "Learn Java";
String str2 = "Cool avatar!";
boolean result = str1.regionMatches(7, str2, 5, 3);
System.out.println(result);
```

**The output to console is:**

true

The methods **int compareTo(String anotherString)** and **int compareToIgnoreCase(String str)** allow to compare two strings. Also, these methods can tell us which of the two strings is larger in lexicographical order and which is smaller:

- If the method returns a value above zero, the initial string is larger.
- If the method returns a value below zero, the new string is larger.

"In lexicographical order" means, for example, that string "A" is smaller than string "B" because the letter A comes before B in the alphabet. If the first characters in the strings are identical, the next characters are considered.

The given methods are used to sort strings.

*Click on the dropdown list to see the example.*

*Hover your cursor over the info icon for an explanation of the code.*

```
String hello = "Hello";
String welcome = "Hello World!";
String ss = welcome.substring(0, welcome.indexOf(" "));
boolean flag = (hello == ss);
System.out.println(hello + " == " + ss + " -> " + flag);
boolean flag1 = hello.equals(ss);
System.out.println(hello + " equals " + ss + " -> " + flag1);
```

**The output to console is:**

Hello == Hello -> false  
Hello equals Hello -> true

Thus, you have studied when to use the operation of string comparison and what methods are used for this. Let's move to the formatting.

## Formatting Strings

The **String** class allows developers to create formatted strings. To do this, the static **String format(String format, Object... args)** method is used. The first parameter in the method is the template string, where special characters are placed at the position in which the values need to be substituted: **%s**, **%d**, etc. (control sequences). After the template string, you need to transmit the parameters, whose values will be substituted for the characters **%s** and **%d**.

Study the following example carefully.

*Click on the dropdown list to see the example.*

## Example

Thus, you have studied when to use the operation of string comparison and what methods are used for this. Let's move to the formatting.

## Formatting Strings

The **String** class allows developers to create formatted strings. To do this, the static **String format(String format, Object... args)** method is used. The first parameter in the method is the template string, where special characters are placed at the position in which the values need to be substituted: %s, %d, etc. (control sequences). After the template string, you need to transmit the parameters, whose values will be substituted for the characters %s and %d.

Study the following example carefully.

*Click on the dropdown list to see the example.*

### The output to console is:

We are printing a double variable (0.700000), a string ("Java"), and an integer variable (10).

Below are some of the most common control sequences.

*Click on the cards to learn more.*



%d – integer (int, long, ...)  
 %f – real number (float, double)



Character / string

%s	–	string
%c	–	character
%% – character %		



Date / boolean

%t

%b – boolean value

date/time

It is important to note that almost every control sequence has additional settings.

So, you have explored the specifics of string formatting, as well as the list of the most common options of control sequences.

## Conclusion

In this lesson, you have seen that:

- A string is a sequence of characters.
- Strings can be concatenated, compared, and formatted.

## Check Your Knowledge!

1. Which of the listed operations are correct for the given string definitions?

String s = **new** String("Java");

String t = **new** String();

String r = null;

r = s + t + r;

r = s + t + 'r';

r = s & t & r;

r = s && t && r;

correct

Answer

Correct:

r = s + t + r;—The operation is correct because three strings are being concatenated. r = s + t + 'r';—The operation is correct because two strings and a character are being concatenated.

2. What is the result of compiling the following code?

```
public class Test {
    public static void main(String[] args) {
        String str1 = new String("Paul");
        String str2 = new String("Paul");
        System.out.println(str1 == str2);
    }
}
```

true

false correct

A compilation error

A runtime error

Answer

Correct:

The result is false since you are comparing references to objects located in different areas of the memory. You can get a result of true if you compare the objects' content using the equals method.

## The Classes **StringBuilder** and **StringBuffer**

### Introduction

In this lesson, you will explore how to solve the problem of inefficient memory use in Java using the classes **StringBuffer** and **StringBuilder**.

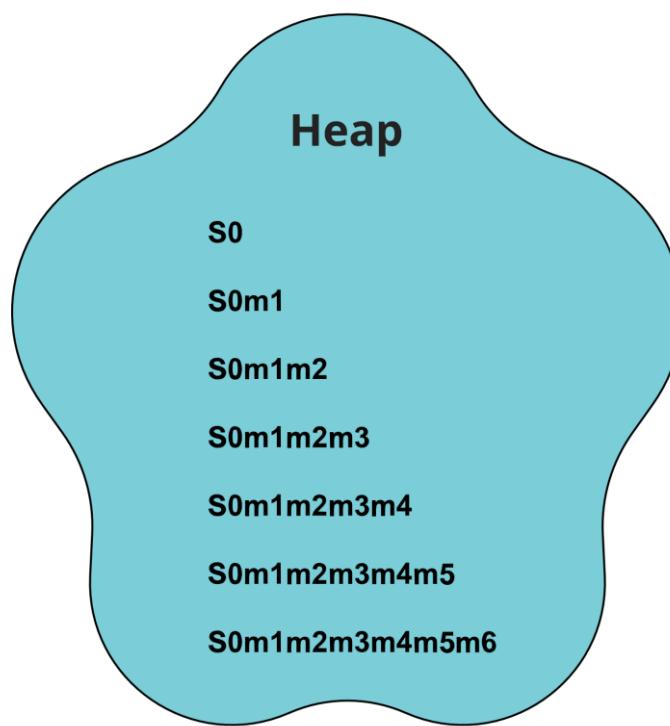
### The Classes **StringBuilder** and **StringBuffer**

In Java, strings (objects of the class **String**) are immutable. To perform an operation to change a string, a new string is created, which negatively affects the performance of applications.

*Click on the dropdown list to see the example.*

```
String str = "S0";
for (int i = 1; i <= 6; i++) {
    str += "m" + i;
}
```

Objects to be created in Heap memory:



To solve the issue of ineffective memory use, the classes **StringBuffer** and **StringBuilder** were added to Java. Basically, this is an expandable string where changes can be made without compromising performance.

*Click on the tabs to study the specifics of using these classes.*

#### Differences

#### Application

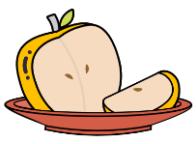
#### Examples

#### Transformations

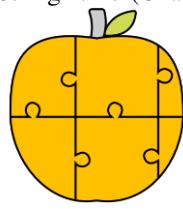
The classes **StringBuffer** and **StringBuilder** are very similar. They have identical constructors and the same methods that are used in the same way. The main difference is that **StringBuffer** is thread-safe and synchronized. This means that it is better to use it in multi-thread applications, where several threads can have access to an object. If a single-thread application is being developed, it is better to use the class **StringBuilder**. It is not thread-safe, but it works faster than **StringBuffer**.



The classes **StringBuffer** and **StringBuilder** are very similar. They have identical constructors and the same methods that are used in the same way. The main difference is that **StringBuffer** is thread-safe and synchronized. This means that it is better to use it in multi-thread applications, where several threads can have access to an object. If a single-thread application is being developed, it is better to use the class **StringBuilder**. It is not thread-safe, but it works faster than **StringBuffer**. Hover your cursor over the info icon for an explanation of the code.

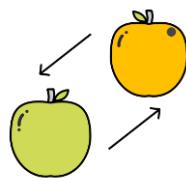


```
StringBuilder();
StringBuilder(int size);
StringBuilder(String obj);
StringBuffer(CharSequence chars);
```



If necessary, objects of the classes **StringBuffer**, **StringBuilder**, and **String** can be transformed into one another. A number of methods exist for this. You can also use constructors:

- A constructor of the class **StringBuffer/StringBuilder** can accept a **String** object as its parameter.
- Objects of the class **StringBuffer/StringBuilder** can be transformed into an object of the class **String** using the method **toString()** or a constructor of the class **String**.



As you have seen, the classes **StringBuffer** and **StringBuilder** were added to Java to address the issue of ineffective memory use.

## Methods of the Class **StringBuffer/StringBuilder**

Take a look at some methods of the class **StringBuilder**. The class **StringBuffer** has the same methods.

Method	Description
int capacity()	Returns the current capacity
void ensureCapacity(int minimumCapacity)	Ensures that the capacity is at least equal to the specified minimum
	Sets the length of the character sequence:
void setLength(int newLength)	<ul style="list-style-type: none"> <li>• If the new size is larger than the stored string, the string will be supplemented with whitespaces at the end.</li> <li>• If the new size is smaller, the string will be trimmed.</li> </ul>
StringBuilder append(...)	Appends the string representation of the argument to the sequence
StringBuilder insert(...)	Inserts the string representation of the argument into the sequence
char charAt(int index)	Returns the char value in the sequence at the specified index
StringBuilder delete(int start, int end)	Removes the characters in a substring of the sequence
StringBuilder deleteCharAt(int index)	Removes the char at a specified position in the sequence
StringBuilder replace(int start, int end, String str)	Replaces the characters in a substring of the sequence with characters in the specified String
String substring(int start)	Returns a new String that contains a subsequence of characters currently contained in the character sequence
String substring(int start, int end)	
StringBuilder reverse()	Causes the character sequence to be replaced by the inverse of the sequence

The methods **equals()** and **hashCode()** are not overridden for the classes **StringBuffer** and **StringBuilder**. Based on this, **it is impossible to compare the content of two objects**.

*Click on the drop-down list to see the example.*

*Hover your cursor over the info icon for an explanation of the code.*

```
StringBuilder sb = new StringBuilder(10);
sb.append("Hello... ");
char c = '!';
sb.append(c);
sb.insert(8, " Java");
sb.delete(5, 8);
System.out.println(sb);
```

**The output to console is:**

Hello Java!

## Working with References to the Strings **StringBuffer/StringBuilder**

When calling methods of the objects **StringBuffer/StringBuilder**, changes are made in the content of the current object, and no new object is created, as is the case when working with **String**.

Let's take a look at several examples.

*Click on the dropdown list to see the example.*

```
public static void main(String [] args) {
    StringBuilder str = new StringBuilder("Learning ");
    updateString(str);
    System.out.println(str);
}
static void updateString(StringBuilder string){
    string.append("java!");
}
```

**The output to console is:**

Learning java!

```
public static void main(String [] args) {
    String str = new String("Learning ");
    updateString(str);
    System.out.println(str);
}
static void updateString(String string){
    string += "java!";
}
```

**The output to console is:**

Learning

It is also worth noting that the **StringJoiner** class was added to Java 8. Its main purpose is to join several strings into one and set a separator, a prefix, and a suffix. The class has two constructors:

- **StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)**
- **StringJoiner(CharSequence delimiter)**

```
StringJoiner joiner = new StringJoiner(":", "<<", ">>");
String result = joiner.add("blanc").add("rouge").add("blanc").toString();
```

```
System.out.println(result);
```

**The output to console is:**

```
<<blanc:rouge:blanc>>
```

As you have seen, when calling methods of the objects **StringBuffer/StringBuilder**, changes are made in the content of the current object, and no new object is created.

## Conclusion

In this lesson, you have seen that:

- To solve the problem of ineffective memory use, the classes **StringBuffer** and **StringBuilder** were added to Java. Basically, this is an expandable string where changes can be made without compromising performance.
- When calling methods of the objects **StringBuffer/StringBuilder**, changes are made in the content of the current object, and no new object is created.
- The **StringJoiner** class was added to Java 8. Its main purpose is to join several strings into one and set a separator, a prefix, and a suffix.

## Check Your Knowledge!

1. What is the result of compiling and running the following code?

```
StringBuilder sb1 = new StringBuilder("I like Java.");//1
```

```
StringBuilder sb2 = new StringBuilder(sb1);//2
```

```
System.out.println(sb1.equals(sb2));
```

true

false correct

A compilation error in line 1

A compilation error in line 2

Answer

Correct:

The result is false since the equals method has not been overridden in the class **StringBuilder** and it compares objects only by the reference. To compare two objects based on their content, the capabilities of the class **String** are necessary.

2. What is the result of running the following code?

```
public class Test {
    public static void main(String[] args) {
        StringBuilder sb1 = new StringBuilder("Bon");
        sb1.insert(2, 'r');
        System.out.println(sb1);
    }
}
```

Born correct

Bor

Bonr

A compilation error

Answer

Correct:

The insert method will insert a character at the position indicated. The positions are counted from zero. The result:

Born

3. What is the result of compiling the following code?

```
char[] name = new char[] {'P','a','u','T'};
String boy = new String(name); //1
StringBuilder sd1 = new StringBuilder("String Builder");
String str5 = new String(sd1); //2
StringBuffer sb2 = new StringBuffer("String Buffer"); //3
String str6 = new String(sb2); //3
String empName = null; //4
```

Successful compilation correct

A compilation error in //1

A compilation error in //2

A compilation error in //3

A compilation error in //4

Answer

Correct:

All of the actions for initializing the objects are correct.

## Regular Expressions

### Introduction

In this lesson, you will explore the concept of regular expressions, you will study metacharacters that are used to build them and will find out how to use regular expressions in Java.

### Major Concepts

**Regular expressions** are a great tool to process strings. Using them, you can set a pattern that a string or substring should correspond to.

The technology of regular expressions is very often compared with the Pareto principle (20/80): by spending very little time studying the syntax of regular expressions, you can solve a huge number of problems related to the search of information and processing of strings.

A regular expression is written using alphabetic and numeric characters, also **metacharacters** are used that are characters that have a special meaning (are only used in the syntax of regular expressions).

# Components of regular expressions



Character



Metacharacter



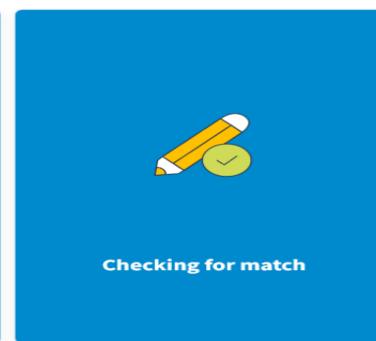
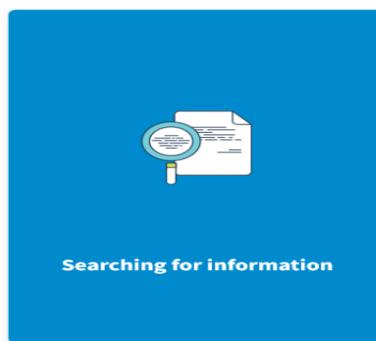
Character class



Quantifiers

One of the purposes of regular expressions is the syntax analysis of text. Other purposes:

*Click on the cards to learn more.*



The following options to search for information exist:

- Searching for a word
- Searching for words that start with certain characters
- Searching for words that end with certain characters

Checking for a match to a certain pattern. For example, validating a phone number, email address, password, etc.

Therefore, using regular expressions you can set a pattern that a string or substring should correspond to.

## Metacharacters of Regular Expressions

Pursuant to [Oracle documents](#), metacharacters of regular expressions can be split into the following groups:

*Click on each drop-down list to see the group.*

### Metacharacters to search for a match of the strings or text boundaries

- ^ — string beginning.
- \$ — string end.
- \b — word boundary.
- \B — not a word boundary.
- \A — input start.
- \G — end of the previous match.

- `\Z` — input end, except for the end terminator, if applicable.
- `\z` — input end.

#### ▼ Metacharacters to search for character classes

- `\d` — numeric character.
- `\D` — non-numeric character.
- `\s` — whitespace character.
- `\S` — non-whitespace character.
- `\w` — alphanumeric character or an underscore.
- `\W` — any character, except for an alphabetic, numeric character or the underscore character.
- `.` — (full stop) any character, except for the new string character.

#### ▼ Metacharacters to search for text delimiter characters

- `\t` — tabulation character.
- `\n` — new line character.
- `\r` — carriage return character.
- `\f` — switching to a new page.
- `\u0085` — next line unicode character.
- `\u2028` — line separator unicode character.
- `\u2029` — paragraph separator unicode character.

#### ▼ Metacharacters to group characters

- `[abc]` — any of the listed (a,b, or c).
- `[^abc]` — any, except for the listed (neither a, nor b, nor c).
- `[a-zA-Z]` — merging ranges (Roman characters from a to z without considering case).
- `[a-d[m-p]]` — combining characters (from a to d and from m to p).
- `[a-z&&[def]]` — overlapping characters (characters d,e,f).
- `[a-z&&[^bc]]` — subtracting characters (characters a, d-z).

#### ▼ Quantifiers

Note the specifics of writing and using some of the metacharacters.

*Click on the tabs to see the description.*

Escaping

Metacharacter \b

Metacharacter ?

If you need to use the designation of a metacharacter or quantifier as a regular character, then escaping is applied:

- `\<metacharacter>` (example: `\*`, `\+`, `\.`, `\?`)
- `[<metacharacter>]` (example: `[+]`, `[?]`, `[*]`, if then follows a quantifier)



The metacharacter \b is interpreted differently depending on the context where it is used:

- If it is specified in a class of characters, then it denotes the "backspace" character ( [\b] ).
- If it is specified outside a class of characters, it means the word boundary (meaning that it limits words consisting of characters [a-zA-Z0-9\_]).

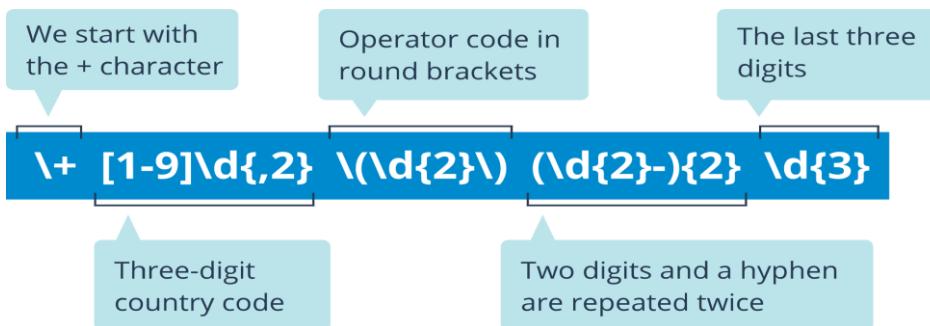


The metacharacter ? is also interpreted differently depending on where it is used:

- ? shows that the preceding characters (ranges of values) is a not a mandatory part of the expression.
- +? shows the minimum number of characters (ranges of values) that may occur once or several times.
- \*? shows the minimum number of characters (ranges of values) that may occur several times. This is the so called "lazy" quantifier.



As an example, look at the form for specifying the cell phone number:



Example of strings matching the template:

+380(99)22-44-888  
+380(67)98-54-321

Pay attention to other examples.

*Click on the tabs to see the examples.*

## Examples of regular expressions

[a-zA-Z\_\\\$][a-zA-Z0-9\_\\\$]\*

Description of an integer value:

((\\+|\\-)?[1-9][0-9]\*)|0

Description of a real value:

(\\+|\\-)?(([1-9][0-9]\*)|0)?\\.\\d+

So, you have explored the groups of metacharacters of regular expressions, as well as specifics of recording and using certain metacharacters.

## Example of a regular expression for a date

A date in the format yyyy-mm-dd located in the range of 1900-01-01 to 2099-12-31:

((19)|(20))\\d\\d-((0[1-9])|(1[012]))-((0[1-9])|([12]\\d)|([3]\\d))

Example of strings matching the template:

1900-01-01  
1986-01-07  
2099-12-31

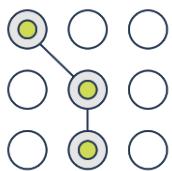
So, you have explored the groups of metacharacters of regular expressions, as well as specifics of recording and using certain metacharacters.

## Regular Expressions in Java

**java.util.regex** is a package of the standard Java library containing major classes to work with regular expressions.

You already know that regular expressions are a technology searching for matches in a string based on patterns. To set such a pattern and to start the search, the Java package **java.util.regex** has classes **Pattern** and **Matcher**.

*Click on the cards to learn more about these classes.*



The class **Pattern** is needed to create templates used to conduct a search in a string, file, or another object containing a sequence of characters.



The **Matcher** class is needed to create patterns using special syntax constructions.

Pay attention to the sequence of actions when working with regular expressions:

## The sequence of actions when working with regular expressions:

**1** Create a regular expression pattern and compile its internal representation (the static method *compile()* of the **Pattern** class).

**2** Associate the regular expression with the source text (the method *matcher()* of the **Pattern** class).

**3** Check to see if the match is successful (the method *find()* of the **Matcher** class).

**4** Request data (methods of the **Matcher** class).

**5** Get additional information about the match (methods of the **Matcher** class).

In string literals describing a regular expression pattern, you can often see "\\" (for example, for metacharacters). In Java, it has to be doubled for the compiler to interpret it correctly:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String regex1 = "\\w";
String regex3 = "When\\?";
```

You can supplement what you have learned by watching the following video lesson.

## Methods of the Pattern Class

Pay attention to the main methods of the Pattern class.

*Click on the drop-down lists to learn more about each method.*

### Compile()

To use regular expressions, it is necessary to create an object of the **Pattern** class. To do this, you need to call one of the two available static methods *compile()* located in this class:

- *compile(String regex)* accepts one parameter – a string with a regular expression. The method compiles this regular expression into a pattern.
- *compile(String regex, int flags)* accepts two parameters – a string with a regular expression and an integer representation of a set of flags (modes of comparing the pattern with text). The method compiles this regular expression into a pattern with the set flags.

Pay attention to their description and application:

```
public static Pattern compile (String literal)
public static Pattern compile (String literal, int flags)
```

The list of the possible values of the *flags* parameter is defined in the **Pattern** class and is accessible as static fields of the class. For example:

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
```

Set of available flags:

Flag	Purpose
CASE_INSENSITIVE	Searching for a match without considering the case for the ASCII characters, meaning that strings "abc", "Abc", and "ABC" will be treated as matches of the regular expression "abc".
UNICODE_CASE	Searching for a match for characters that are not part of ASCII.
UNIX_LINES	The character "\n" is treated as the end character of the line, where the match is being searched.
MULTILINE	If there are several characters "\n" in the text, where the match is being searched, then the text is considered to consist of several lines.
LITERAL	All pattern characters, including the metacharacters, are treated as regular characters.
DOTALL	If the pattern has the "." character, then any character will correspond to it, including the "\n" character (if it is absent, then the "." metacharacter will correspond to any character, excluding the "\n" character).
COMMENTS	Whitespaces and comments starting from the character "#" and until the end of the line are allowed in the pattern string (during compilation, the whitespaces and comments will be ignored).
CANON_EQ	Canonical equivalence of UNICODE characters (meaning that different coding options of one character are considered to be identical).

The **Pattern** class is a constructor of regular expressions.

The method *compile* calls the **private** constructor of the **Pattern** class to create a compiled representation. A regular expression specified as a string should first be compiled into an instance of that class. The resulting pattern can be then used to create an object **Matcher** that can compare arbitrary sequences of characters with the regular expression. Thus, a pattern instance is created as an immutable object.

When creating a pattern instance, the regular expression is checked for correct syntax. If errors are found, an exception **PatternSyntaxException** will be thrown.

#### pattern()

The method *pattern()* returns a regular expression (string), from which this pattern was compiled.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
Pattern pattern = Pattern.compile("java");
System.out.println(Pattern.pattern())
```

#### matches()

The method *matches* (*String regex, CharSequence input*) compiles the given regular expression (set using **regex**) and tries to compare the given input (parameter **input**) with it. It returns:

- **true**—if the text matches the regular expression
- **false**—if it doesn't

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
System.out.println(Pattern.matches("J.+a","Java"));
System.out.println(Pattern.matches("J.+a","Java JavaScript"));
```

#### ▼ flags()

The method **flags()** returns the values of the parameter **flags** of a regular expression that were set when creating the regular expression (the default value is 0 if no values have been set).

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
Pattern pattern = Pattern.compile("abc");
System.out.println(pattern.flags());
Pattern otherPattern = Pattern.compile("abc", Pattern.CASE_INSENSITIVE);
System.out.println(otherPattern.flags());
```

#### ▼ split()

The method **split (CharSequence text, int limit)** splits the given input sequence **text** into an array of strings **String**. The array returned using this method contains each substring of the input sequence that ends with a different subsequence corresponding to this pattern or ends with the end of the input sequence. The parameter **limit** controls the number of uses of the pattern and, thus, affects the length of the resulting array.

- If **limit>0** – the pattern will be used maximum **limit-1**times, the array length will not exceed **limit**, and the last array entry will contain all input data except for the last matched delimiter.
- If **limit<0** – the pattern will be used as many times as possible and the array can have any length.
- If **limit=0** – the pattern will be used as many times as possible, the array can have any length, and the trailing empty strings will be discarded.

```
String str1 = "One two,three!four;five six.seven";
Pattern p1 = Pattern.compile("[ ,!;.]");
String s[] = p1.split(str1);
System.out.println("Source string -> " + str1);
for (String i : s) {
    System.out.println("Lexeme: " + i);
}
```

**he output to console is:**

```
Source string -> One two,three!four;five six.seven
Lexeme: One
Lexeme: two
Lexeme: three
Lexeme: four
Lexeme: five
Lexeme: six
Lexeme: seven
```

```
String str1 = "One two,three!four;five six.seven";
Pattern p1 = Pattern.compile("[ ,!;.]");
String s[] = p1.split(str1, 2);
System.out.println("Source string -> " + str1);
for (String i : s)
    System.out.println("Lexeme: " + i);
```

## The output to console is:

Source string -> One two,three!four;five six.seven  
 Lexeme: One  
 Lexeme: two,three!four;five six.seven

Thus, you have studied the main methods of the **Pattern** class, such as: *compile()*, *pattern()*, *matches()*, *flags()*, *split()*.

## Methods of the Matcher Class

The class **Matcher** performs loads of different operations comparing the source sequence with the pattern. To do this, you need to create an object responsible for searching for matches to the pattern and further processing the search results.

To create an object of the **Matcher** class, you need to use the method *matcher()* of the **Pattern** class.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String text = "This is my second java 45 project.\n" +
    "It is wonderful to learn polysemantics and arrays.\n" +
    "The weather is cold like it should be in winter, but we are all looking forward to spring.";
```

Pattern p = Pattern.compile("\\b[\\w]{2}\\b");  
 Matcher m = p.matcher(text);

Pay attention to the main methods of the **Matcher** class.

*Click on the drop-down lists to learn more about each method.*

### ▼ find()

The method *find()* is designed to search for the next subsequence of characters in the input sequence that matches the pattern. There are two ways of how this method works:

1. The search starts at the beginning of the given text.
2. The search starts from the first character after the preceding match. This is possible only if the result of the previous invocation of this method is *true* and the matcher has not been reset.

If the search is successful, the method will return the boolean *true*.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String text = "This is my second java 45 project.\n" +
    "It is wonderful to learn polysemantics and arrays.\n" +
    "The weather is cold like it should be in winter, but we are all looking forward to spring.";
```

Pattern p = Pattern.compile("\\b[\\w]{2}\\b");  
 Matcher m = p.matcher(text);  
 while (m.find()) {  
 int start = m.start();  
 int end = m.end();  
 System.out.println("Found matches " + text.substring(start,end) + " from "+ start + " to " + (end-1) + " positions");

```
}
```

### The output to console is:

```
Found matches is from 5 to 6 positions
Found matches my from 8 to 9 positions
Found matches 45 from 23 to 24 positions
Found matches It from 35 to 36 positions
Found matches is from 38 to 39 positions
Found matches to from 51 to 52 positions
Found matches is from 98 to 99 positions
Found matches it from 111 to 112 positions
Found matches be from 121 to 122 positions
Found matches in from 124 to 125 positions
Found matches we from 139 to 140 positions
Found matches to from 166 to 167 positions
```

The method *find (int start)*, similar to the method *find()*, tries to find a subsequence in the input sequence matching the pattern, starting from the given index set by the ***start*** parameter. If the result is successful, then the boolean ***true*** is returned.

For example, *m.find(1)* – the search in the text starts from position 1, at that position 0 will not be considered.

If the parameter ***start*** equals a negative value or a value exceeding the length of the input sequence, the method will generate an exception ***java.lang.IndexOutOfBoundsException***.

#### matches()

The method *matches()* works as follows: the entire text is compared with the pattern. If it matches the pattern, the method returns the boolean ***true***.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
Pattern p = Pattern.compile("\\w*");
Matcher m = p.matcher("Thanks!");
System.out.println(m.matches());
```

#### lookingAt()

The method *lookingAt()* compares only the given text with the pattern. Unlike the method *matches()*, the entire text does not need to match the pattern. The method returns the boolean ***true***, if at least some part of the source sequence matches the pattern.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
Pattern p = Pattern.compile("\\w*");
Matcher m = p.matcher("Thanks!");
System.out.println(m.lookingAt());
```

If the match was successful, then additional information can be obtained using the methods ***start***, ***end***, and ***group***.

#### replaceFirst()

The method `replaceFirst(String replacement)` works as follows: in the input sequence, the first subsequence matching the pattern is replaced with the parameter `replacement`.

This method scans the input sequence looking for a match to the pattern. At that, the characters that are not part of the match are added directly to the resulting string. Then the match is replaced with a string received from the parameter `replacement`.

Note that the backward slash (\) and dollar sign (\$) in the replacement string (`replacement`) may cause the results to be different from the results that are processed as a string for replacement of letters. Dollar signs may be treated as references to the captured subsequences.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String text = "This is my second java 45 project.\n" +
    "It is wonderful to learn polysemantics and arrays.\n" +
    "The weather is cold like it should be in winter, but we are all looking forward to spring.";
System.out.println("Before:\n" + text);
Pattern p = Pattern.compile("\b[\w]{2}\b");
Matcher m = p.matcher(text);
text = m.replaceFirst("lab2");
System.out.println("\nAfter:\n" + text);
```

**The output to console is:**

Before:

This is my second java 45 project.

It is wonderful to learn polysemantics and arrays.

The weather is cold like it should be in winter, but we are all looking forward to spring.

After:

This lab2 my second java 45 project.

It is wonderful to learn polysemantics and arrays.

The weather is cold like it should be in winter, but we are all looking forward to spring.

#### ▼ replaceAll()

The method `String replaceAll(String replacement)` works the same way as the method `replaceFirst(String replacement)`. But unlike this method, `String replaceAll` replaces **all found matches** with characters taken from the string `replacement`.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String text = "This is my second java 45 project.\n" +
    "It is wonderful to learn polysemantics and arrays.\n" +
    "The weather is cold like it should be in winter, but we are all looking forward to spring.";
System.out.println("Before:\n" + text);
Pattern p = Pattern.compile("\b[\w]{2}\b");
Matcher m = p.matcher(text);
text = m.replaceAll("lab2");
System.out.println("\nAfter:\n" + text);
```

**The output to console is:**

Before:

This is my second java 45 project.

It is wonderful to learn polysemantics and arrays.

The weather is cold like it should be in winter, but we are all looking forward to spring.

After:

This lab2 lab2 second java lab2 project.

lab2 lab2 wonderful lab2 learn polysemantics and arrays.

The weather lab2 cold like lab2 should lab2 lab2 winter, but lab2 are all looking forward lab2 spring.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
String text = "This is my second java 45 project.\n" +
    "It is wonderful to learn polysemantics and arrays.\n" +
    "The weather is cold like it should be in winter, but we are all looking forward to spring.";
System.out.println("Before:\n" + text);
Pattern p = Pattern.compile("\b([\\w]{2})\\b");
Matcher m = p.matcher(text);
text = m.replaceAll("$1_lab2");
System.out.println("\nAfter:\n" + text);
```

**The output to console is:**

Before:

This is my second java 45 project.

It is wonderful to learn polysemantics and arrays.

The weather is cold like it should be in winter, but we are all looking forward to spring.

After:

This is\_lab2 my\_lab2 second java 45\_lab2 project.

It\_lab2 is\_lab2 wonderful to\_lab2 learn polysemantics and arrays.

The weather is\_lab2 cold like it\_lab2 should be\_lab2 in\_lab2 winter, but we\_lab2 are all looking forward to\_lab2 spring.

Thus, you have explored the main methods of the **Matcher** class: *find()*, *matches()*, *lookingAt()*, *replaceFirst()*, *replaceAll()*.

## Methods to Extract Information About Groups

For a more convenient processing of the input sequence, regular expressions use groups that help highlight parts of the found subsequence. **Groups** are a means of processing a set of characters as one.

In the patterns, they are specified with brackets "(" and ")". Each left-to-right opening bracket numbers a group. The expression ((A)(B(C))) defines four groups:

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (C)

Group numbers start with one. The zero group matches the entire found subsequence.

Below you can see methods used to extract information about groups.

**String group()**

Returns the input subsequence matched by the previous match.

**String group(int group)**

>Returns the input subsequence captured by the given group during the previous match operation.

**int groupCount()**

>Returns the number of capturing groups in this matcher's pattern.

**int start()**

Returns the start index of the previous match.

`int start(int group)`

Returns the start index of the subsequence captured by the given group during the previous match operation.

`int end()`

Returns the offset after the last character matched.

`int end(int group)`

Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

`boolean hitEnd()`

Returns true if the end of the input was hit by the search engine in the last match operation performed by this matcher.

Pay attention to the example of using groups as well as own and incomplete quantifiers.

*Click on the drop-down list to study the example.*

```
package by.epam.learn.string;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class GroupMain {
    public static void main(String[] args) {
        String base = "java";
        groupView(base, "([a-z]*)((a-z)+)");
        groupView(base, "([a-z]?)((a-z)+)");
        groupView(base, "((a-z)+)([a-z]*)");
        groupView(base, "((a-z)?)([a-z]?)");
    }
    private static void groupView(String base, String regExp) {
        Pattern pattern = Pattern.compile(regExp);
        Matcher matcher = pattern.matcher(base);
        if (matcher.matches()) {
            System.out.println("group 1: " + matcher.group(1));
            System.out.println("group 2: " + matcher.group(2));
            System.out.println("main group: " + matcher.group() + " [end]"); // eq.group(0)
        } else {
            System.out.println("nothing matches");
        }
    }
}
```

**The output to console is:**

```
group 1: jav
group 2: a
main group: java [end]
group 1: j
group 2: ava
main group: java [end]
group 1: java
group 2: main group: java [end]
nothing matches
```

- In the first case, all possible characters belong to the first group, but at the same time, there is a minimum number of characters for the second group.
- In the second case, the smallest number of characters is selected for the first group, since a weak match is used.
- In the third case, the entire string will correspond to the first group, and no characters are left for the second group, since the second group uses a weak match.
- In the fourth case, the string does not match the regular expression, since the smallest number of characters is selected for two groups.

- Group 0 is always the same, since it represents the entire expression.

So, you have found out that for a more convenient processing of the input sequence regular expressions use groups, and also you have studied main methods to extract information about groups.

## Conclusion

Therefore, in this lesson you have found out:

- Which metacharacters exist and what they are used for.
- How to match and use regular expressions.
- The purpose of some methods of the classes **Pattern** and **Matcher** and how they are used.
- What methods exist to extract information about groups.

## Check Your Knowledge!

1. Which metacharacter replaces the following class of characters?

[^0-9]

\s

\i

\d

\D correct

Answer

Correct:

\D — non-numeric character

2. Which metacharacter denotes a word boundary?

^

\B

\b correct

\w

Answer

Correct:

The character \b denotes a word boundary.

3. What is the result of running the following code fragment?

```
String str = "Lena, Sveta, Lulu, Natalia, lana";
```

```
Pattern p = Pattern.compile("L.*?\b");
```

```
Matcher m = p.matcher(str);
```

```
str = m.replaceAll("XXX");
```

```
System.out.println(str);
```

Lena, Sveta, Lulu, Natalia, lana

XXX, Sveta, XXX, Natalia, XXX

XXX, Sveta, XXX, Natalia, lana correct

Runtime error

Answer

Correct:

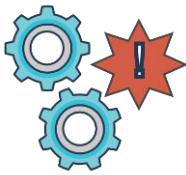
All substrings starting from the character L and containing any number of arbitrary characters were replaced with XXX.

## Exceptions and Their Types

### Introduction

In this lesson, you will start exploring exception in Java. You will study what exceptions and what kinds of exceptions exist, as well as study the types of exceptions.

### Concept of an Exceptional Event



Exceptions and errors occur during execution of a program when the occurring issue may not be resolved in the current context and the program cannot execute further. Examples of common errors include: index out of bounds of the array, calling a method on a null reference, dividing by zero, incorrect data format, etc.

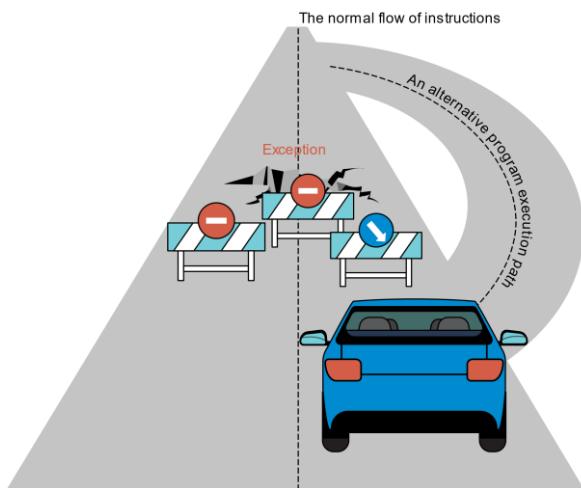
An exception should not be treated as something bad that you have to get rid of at any cost. The exception is a source of additional information about the application's execution progress. This information allows the learner to better adapt code to specific conditions of its use, as well as to detect errors at an early stage and to get protection against them in future. Otherwise, "suppressing" exceptions will hide the error, so the error will show up later, when it may be way more challenging to find its cause. As a result, it may be difficult to find the error.

In Java, each exception corresponds to a certain class, whose instance is initiated when an exceptional situation occurs. If a relevant class does not exist, it may be created by the developer.

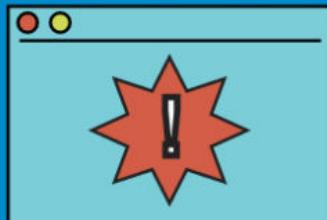
**Summary:**

**An exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's commands or instructions.

**Handling an exception** means to create an alternative path of program execution.



Let's see how a program behaves in case of an error.

**Creating an exception****Throwing an exception****Handling the exception**

In case of an error, Java creates an **exception object** that contains information about the error, its type and state of the program when the error occurred.

After the exception object has been created, it is passed to the runtime system. This is called to **throw an exception**.

The runtime system attempts to find a block of code specified as an exception handler for this particular type and scope of the exception. If the runtime system does not find an appropriate exception handler, the program terminates and prints to the console the exception description and tracing of methods that were checked in finding a handler.

Let's consider an example of an exception that occurs during a division by zero operation.

```
public static void main(String[] args) {
    int x1 = 10;
    int x2 = 0;
    int y = x1/x2;
}
```

## Output

Exception in thread "main" java.lang.ArithmaticException:  
/ by zero at com.epam.test\_excp.Main.main(Main.java:6)

There are different types of exceptions in Java. Let's study their classification.

## Kinds of Exceptions

In Java, all exceptions are described through classes that form a hierarchical structure. At the top of this hierarchy is the **Throwable** class that has two subclasses - **Error** and **Exception** - from the package **java.lang**. All other exceptions are their subclasses.

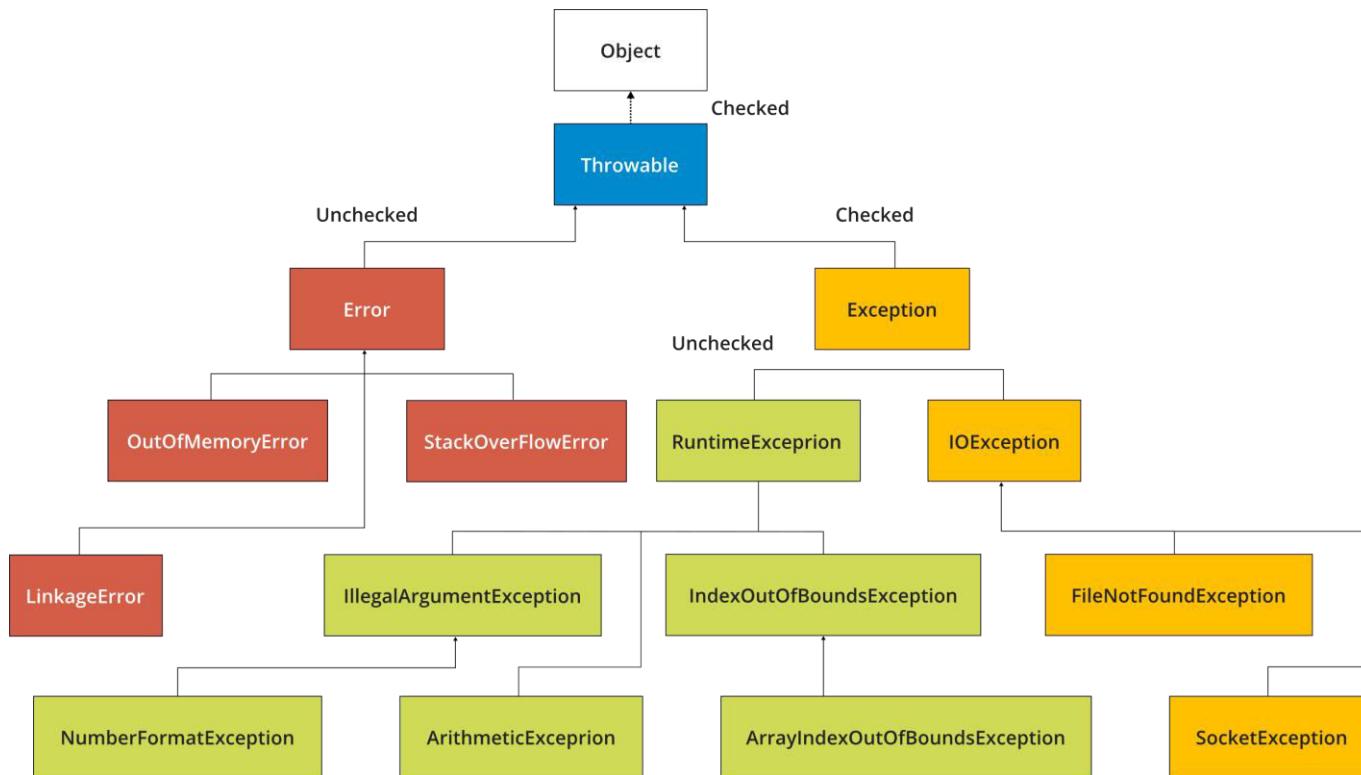
As an example, in the image you see a part of the exceptions hierarchy that shows different kinds and categories of exceptions. Depending on the kind and category of an exception, it may be catch and processed.

There are two categories of exceptions:

- Checked
- Unchecked

Unchecked exceptions can be divided into two kinds - errors and runtime exceptions.

*Click on each + sign to learn more.*



## Errors

In red, you can see a family of exceptions of the **Error** type. It combines all errors that are external to the program. This means that the program cannot anticipate or recover from such errors. This family is part of the unchecked category.

## Exceptions

You can see a family of exceptions of the **Exception** type in yellow. It combines all the errors that are legal to occur – for instance, lack or inaccessibility of some external dependency, like file or internet address. In such a case, the program should recover and continue executing. This family is part of the checked category.

## Runtime exceptions

In green, you can see a family of exceptions of the **RuntimeException** type. It combines all errors that are internal to the program—programming bugs, such as logic errors or improper use of an API. This means that the program usually cannot anticipate or recover from them. This family is part of the unchecked category.

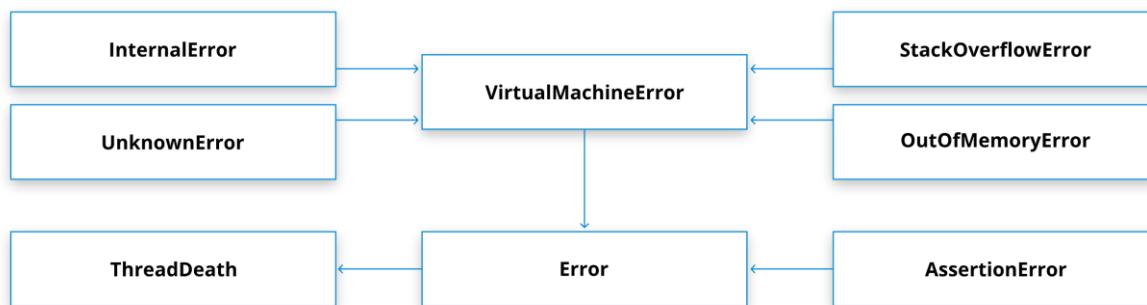
Let's explore the kinds of exceptions in detail and how they are processed.

*Click on the headings to learn more.*

## Error

Exceptions of the type **Error** occur when the program is executed and are related to errors in the external environment, where the program is executed. For example, stack overflow, disk defective or a memory error. These errors are not subject to correction by user programs. The program may try to catch them to inform the user, after which the program terminates.

In the image, you can see examples of error classes.



In the table, you can find the description of the most common exceptions of the **Error** type.

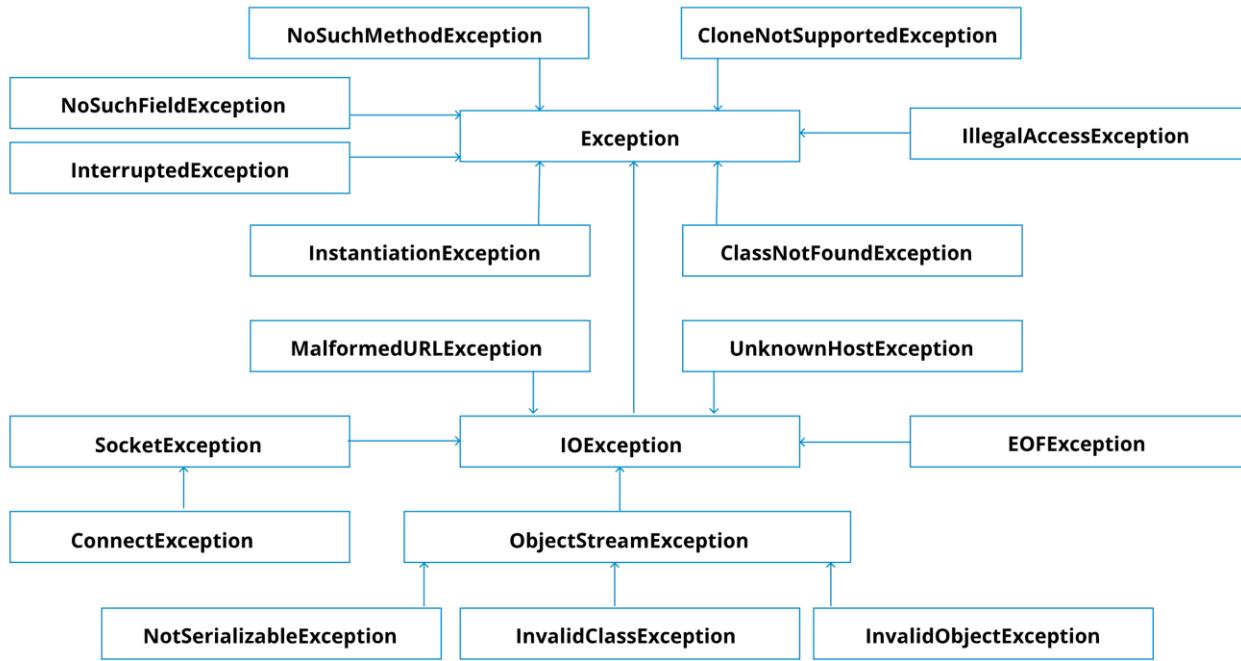
Exception	Value
AssertionError	An assertion has failed
InternalError	Internal error has occurred in the Java Virtual Machine
OutOfMemoryError	The Java Virtual Machine cannot allocate a memory for object
StackOverflowError	A stack overflow occurs because a program recurses deeply
ThreadDeath	Thrown in the victim thread when the (deprecated) <i>stop()</i> method is invoked
UnknownError	Thrown when an unknown but serious exception has occurred in the Java Virtual Machine
VirtualMachineError	The Java Virtual Machine is broken or has run out of necessary resources

## Exception

All exceptions of the **Exception** type, except the **RuntimeException** family, are part of the checked exceptions, meaning that these exceptions are expected. For example, class not found or input/output error. Therefore, the occurrence of such an exception may be detected already at the code compilation stage/ The compiler checks whether a method can throw an exception. If it can, the compiler will request to specify the placement of the code for exception handling: either in the method where it occurs or at somewhere else.

The **Exception** class has a special subclass **IOException** that describes the family of exceptions related to the input/output operations.

In the image, you can see examples of exception classes.

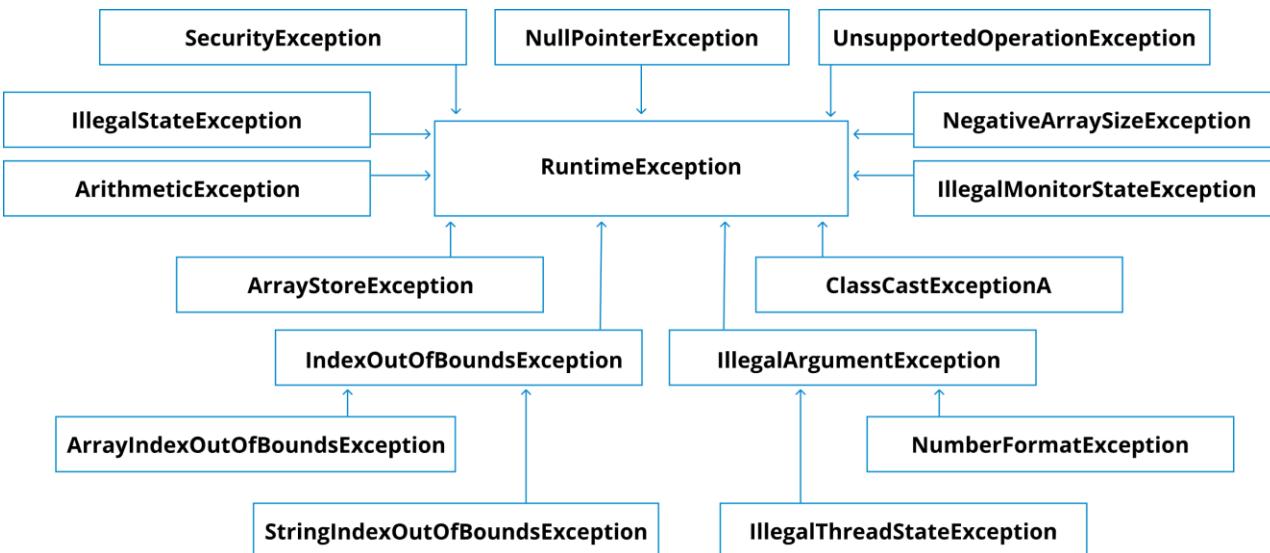


In the table, you can find the description of the most common exceptions of the **Exception** type.

Exception	Value
ClassNotFoundException	Class not found
CloneNotSupportedException	An attempt to clone an object whose class does not implement the Cloneable interface
ConnectException	Attempting to connect a socket to a remote address and port
EOFException	When an end of file or end of stream has been reached unexpectedly during input
IllegalAccessException	Access to a class denied
InstantiationException	An attempt to create an instance of an abstract class or interface
InterruptedException	Occurs when a thread needs to be interrupted
InvalidClassException	When the serialization runtime detects the problems with a class
InvalidObjectException	One or more deserialized objects failed validation tests
IOException	The general class of exceptions produced by failed or interrupted I/O operations
MalformedURLException	When the URL is malformed
NoSuchFieldException	The required field does not exist
NoSuchMethodException	The required method does not exist
NotSerializableException	The class is not serializable
ObjectStreamException	The general class of all exceptions specific to Object Stream classes
ReflectiveOperationException	The general class of exceptions defining error when using reflection
SocketException	A error creating or accessing a Socket
UnknownHostException	The IP address of a host could not be determined

## Run time exception

All exceptions of the **RuntimeException** type is part of the category of unchecked exceptions. In other words, they are unexpected, they occur during execution of a program, and the compiler has no possibility to check these situations in advance. Usually, they indicate logical program errors or an improper use of the API. For example, dividing by zero, index to an array is out of range, unacceptable typecasting.



In the table, you can find the description of exceptions of the **RuntimeException** type.

Exception	Value
ArithmeticsException	Arithmetics error like dividing by zero
ArrayIndexOutOfBoundsException	Array index out of range
ArrayStoreException	An attempt to store the wrong type of object into an array
ClassCastException	Unacceptable typecasting
IllegalArgumentException	A method has been passed an illegal argument
IllegalMonitorStateException	An unacceptable monitor operation
IllegalStateException	The Java environment or Java application is not in an appropriate state for invoke the method
IllegalThreadStateException	A thread is not in an appropriate state for the requested operation
IndexOutOfBoundsException	Some index type is out of range
NegativeArraySizeException	Trying to create an array with negative size
NullPointerException	Unacceptable use of a null reference
NumberFormatException	Attempt to convert a string to one of the numeric types, but that the string does not have the appropriate format
SecurityException	Attempt to violate security
StringIndexOutOfBoundsException	String character index is out of range
UnsupportedOperationException	An unsupported operation was encountered

## Conclusion

In this lesson, you have found out that:

- An exceptional event (exception) is an event that occurs during the execution of a program and disrupts the normal flow of the program's commands or instructions.
- Handling an exception means to create an alternative path of program execution.
- When an error occurs, the program creates an exception, throws, and handles it.
- Depending on the kind and category of an exception, it may be caught and processed. There are two categories of exceptions: checked and unchecked. Unchecked exceptions can be divided into two kinds—errors and runtime exceptions.

## Check Your Knowledge!

1. Choose the kinds and categories of exceptions that exist in Java.

Checked exception

Error

Checked error

Unchecked exception

correct

Answer

Correct:

There are two categories of exceptions in Java: checked, unchecked. Unchecked exceptions can be divided into two kinds—errors and runtime exceptions.

2. Which statements about the exceptions of the "Error" type are true?

Exceptions of the type "Error" are a subclass of **java.lang.Exception**

Exceptions of the type "Error" are a subclass of **java.lang.Error**

Exceptions of the type "Error" lead to the termination of a program

Exceptions of the type "Error" can be caught but this should not be done

correct

Answer

Correct:

Exceptions of the type "Error" are a subclass of **java.lang.Error**. They are not subject to correction by user programs.

The program may try to catch them to inform the user, after which the program terminates.

3. Choose the exception that indicates an illegal conversion of a string into a numeric format.

correct

Answer

Correct:

**NumberFormatException** – illegal conversion of a string into a numeric format.

4. Choose an exception that indicates that the required operation is incompatible with the current thread state.

correct

Answer

Correct:

**IllegalThreadStateException** – thread is not in an appropriate state for the requested operation.

## Handling Exceptions

### Introduction

In this lesson, you will find out how exceptions are handled in Java. You will learn to use blocks **try**, **catch**, and **finally** to write an exception handler.

### Handling Exceptions

You are probably wondering: Why do we need to handle exceptions? The reason is simple: if there is a chance to recover after an error occurs and to continue program execution, then handling exceptions is exactly what we need.



In practice, one of the two ways to handle exceptions is used:

- Catching and handling the exception using the **try-catch** blocks in the method
- Passing the exception to a other part of the code for handling—a **throws** component to the method declaration. It is used primarily for checked exceptions.

In this module, we will study both ways. We will start with the **try-catch** blocks.

## try-catch

Handling an exception can easily be integrated into existing code. For this, a part of code where an exception may occur is enclosed within the **try** block, after which follows the **catch** block. The **catch** block is a handling block for a specific exception specified in the block's parameters. Handling an exception using the **try-catch** blocks involves the following syntax:

```
try {
    <code controlled for occurring exceptions>
} catch (<ExceptionType> <objName>) {
    <exception handler>
}
```

The **try** block is similar to an ordinary dynamic block. The **catch** block is similar to a method description since it accepts a parameter – a reference to the exception object – and handles this object. Let's explore how these blocks work.

*Click on the tabs to learn more about the behavior of these blocks.*

**Behavior of the try block**

**Behavior of the catch block**

If an exception occurs inside the **try** block, then an exception object is generated and is then thrown from the **try** block. Then the Java runtime system searches the exception handler. First of all, the system looks the **catch** block following after **try**.

If the exception does not occur within the **try** block, then the **catch** block is not executed, and the program execution is passed to the statement following it.



The Java runtime system compares the exception object that occurred with what the **catch** block expects:

- If the exception complies with the type of the **catch** block parameter, then this block is executed, and then the statement following the **catch** block is executed.
- If the exception is not expected, then the exception is handled by the Java runtime system: it terminates the program and prints information about the exception.



Let's consider an example of handling exception when working with arrays. As you know, when an array element is accessed, Java checks the availability of this element. If the index is specified incorrectly, then an exception of the

type **ArrayIndexOutOfBoundsException** is generated classified as an unchecked exception. In this example, we enclose the work with the array within the **try** block and in the **catch** block we simply inform about the occurred situation.

*Hover the mouse cursor over the info icon to see the explanation for the code.*

```
public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5, 6};
    try {
        int sum = 0;
        for (int i = 0; i <= 6; i += 2) {
            sum += arr[i];
        }
    } catch(ArrayIndexOutOfBoundsException exp) {
        System.out.println("Program Error!");
    }
    System.out.println("Program Finish!");
}
```

## Output

Program Error!  
Program Finish!

The following capabilities exist for the implementation of handling exception:

- Nested **try-catch** blocks
- Multiple **catch** blocks
- Multi-catch block

Let's explore them in more detail.

## Nested try-catch Blocks

You can nest **try** blocks inside each other. Thus, we create a system for handling exceptions on different levels. Usually, nesting manifests itself in methods invokes, when one method within its own **try** block invokes another method that has its own **try-catch** blocks. Let's consider an example of blocks nesting.

### EXTERNAL BLOCK

```
public void method_1() {
    // ...
    try { method_2(); }
    catch(RuntimeException exp) {
        // ...
    }
}
```

### INTERNAL BLOCK

```
public void method_2() {
    try { // ...
    } catch(ArithmeticException exp) {
        // ...
    }
}
```

The handling exceptions in such block constructions happens as follows:

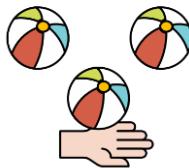
- If an exception occurs in the internal **try** block, and its **catch** block does not have a corresponding exception handler, then the program checks the **catch** blocks of the external **try** blocks for the appropriate exception handler.
- If the needed exception handler is not found in any of the **catch** blocks, the exception will be handled by the Java runtime system.
- If the exception handler is found in an external block, then after the exception is handled the program will continue to execute in the method where the exception was handled.

So, you have reviewed how nested **try-catch** blocks work. Next, we will consider the specifics of using multiple **catch** blocks.



## Multiple catch

You are not insured against a situation when several different exceptions occur within one **try** block. By creating nesting or creating many successive **try-catch** blocks, you create more overhead costs for the system to monitor the exceptions, as well as lower the readability and clarity of the code. There is another option—to assign several **catch** blocks with one **try** block. This is called *multiple catch*.



```
try {
    <code controlled for occurring exceptions>
} catch (<ExceptionType1> <objName1>) {
    <exception handler ExceptionType1>
} catch (<ExceptionType2> <objName2>) {
    <exception handler ExceptionType2>
} ...
```

When an exception occurs in such a **try** block, the handler is searched for by successively checking the described **catch** blocks. The runtime system passes the exception to the first catch block, whose argument type matches the type of the thrown exception. The system considers it a match if the thrown object can legally be assigned to an exception handler argument. Therefore, when using several **catch** blocks, the subclasses of exceptions should come before any of their superclasses.

*Click on the title to see the example.*

In the following code, the exception handler of the type **ArithmetiException** will never be reached, even if this exception does occur in the **try** block. This happens because an exception handler of the type **Exception** is described before it, which is a superclass for **ArithmetiException**. Therefore, the first handler will always be executed. It will be correct to place a handler for **ArithmetiException** first and then for **Exception**.

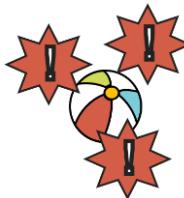
```
public class Test {
    public static void main(String[] args) {
        try {
            int b = 0;
```

```

int a = 42 / b;
} catch (Exception e) {
    System.out.println("Exception");
} catch (ArithmaticException e) {
    System.out.println("Unattainable");
}
}
}

```

## Multi-catch



When using multiple **catch** blocks, a situation can occur related to the duplicating of code the handling exception. Starting from Java 7, there is a possibility to avoid code duplication using single **catch** block, that can handle more than one type of exception. This is done by combining the types of the exceptions in the **catch** block argument using the following syntax:

```
catch (<exc. type1> | <exc. type2> | .... | <exc. typeN> <ident.>) { /* ... */ }
```

This does not only help reduce the duplication of code, but also helps fight the temptation to catch an overly broad exception. For example:

```

try { <controlled code block> }
catch (ArithmaticException | ArrayIndexOutOfBoundsException exp) {
    <common handler code>
}

```

Let's review how multi-catch is used in practice.

*Click on the title to see the example of multiple handling.*

The *a* and *b* variables are initialized by the values of arguments from the command line that are passed to the *main()* method through the array *args*. Since the arguments are of type **String**, then to initialize the *a* and *b* variables they have to be cast to the type **int**, which is done by the static method *parseInt()* of the class **Integer**. Then we divide *a* by *b*.

By analyzing the described example, we can detect three issues:

- The command line can have no arguments or only one. Then we will have a situation—the index the args array is out of range – an exception of the type **ArrayIndexOutOfBoundsException**.
- The values of the command line arguments might not match the format of integer representation. In that case, we will have an exception of converting a string into an integer value – **NumberFormatException**.
- The value of the *b* variable can be zero. In that case, we will have an exception of dividing by zero – **ArithmaticException**.

Printing the information about the exception to the console was chosen as the handler code for all these exceptions, which means that we do not have to duplicate code and can combine it into one handler.

```

int a = Integer.parseInt(args[0]);
int b = Integer.parseInt(args[1]);
System.out.println(a / b);
} catch (ArithmaticException | NumberFormatException | ArrayIndexOutOfBoundsException e) {
    System.out.println( e.getMessage() );
}
}

```



We can only combine those types of exceptions that are located in different inheritance branches.

Therefore, you have reviewed how multi-catch of exceptions is used. Now is the time to explore how the completion handler works.

## The finally Block

In addition to exception handlers in Java, there is a **finally** block to perform completing actions when the **try** block exits. It is used to clean up the resources when by a return or break of try block. For example, to close a data stream or to close a database connection. The **finally** block control the exit from the **try** block in terms of performing actions that correctly complete the work of the **try** block. It has the following syntax:

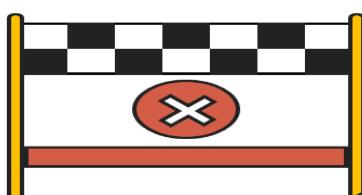
```

try { <controlled code block> }
finally { <cleanup code > }

```

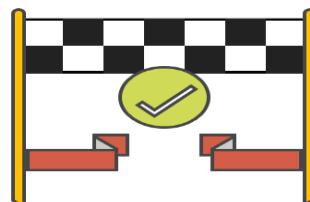
Let's explore how these blocks work.

*Click on each card to learn more.*



The **finally** blocks is always executed when exiting the **try** block:

- An exception has occurred.
- Execution of all statements without errors.
- The statements **return**, **continue**, or **break** are used.



The **finally** block is not executed when the following happens in the **try** block:

- The virtual machine exits while the try code is being executed.
- The thread executing the code of the **try** or **catch** blocks is interrupted or killed.

The **finally** block is a key tool to prevent a leakage of resources; it allows the developer to avoid having cleanup code accidentally bypassed. Let's consider an example of executing and using the **finally** block.

Click on the title to see the example of multiple handling.

In this example, we describe two methods `procA()` and `procB()` that are invoked from the method `main()`.

- The `procA()` method in the **try** block, to which the **finally** block is associated, uses the statement **return**. This means that before return from the method, the body of the **finally** block will be executed.
- The `procB()` method in the **try** block, to which the **finally** block is also associated, uses the statement **return** with a parameter. This means that before return from the method, the body of the **finally** block will be executed, which also uses the **return** statement with a another value of the parameter.

As a result, the value defined in the **finally** block will be returned from the `procB()` method.

Hover the mouse cursor over the info icon to see the comment for the code.

```
public class TestFinally {
    static void procA() {
        try {
            System.out.println("Method procA()");
            return;
        } finally {
            System.out.println("Block finally of method procA()");
        }
    }
    static int procB() {
        try {
            System.out.println("Method procB()");
            return 1;
        } finally {
            System.out.println("Block finally of method procB()");
            return 0;
        }
    }
    public static void main(String [] args) {
        procA();
        System.out.println( procB() );
    }
}
```

## Output

```
Method procA()
Block finally of method procA()
Method procB()
Block finally of method procB()
0
```

In Java, one block **try** may contain both the block **finally** and the block **catch**. Thus, the **try** block is controlled for the occurrence of an exception. This also guarantees a release of close in any exit conditions.

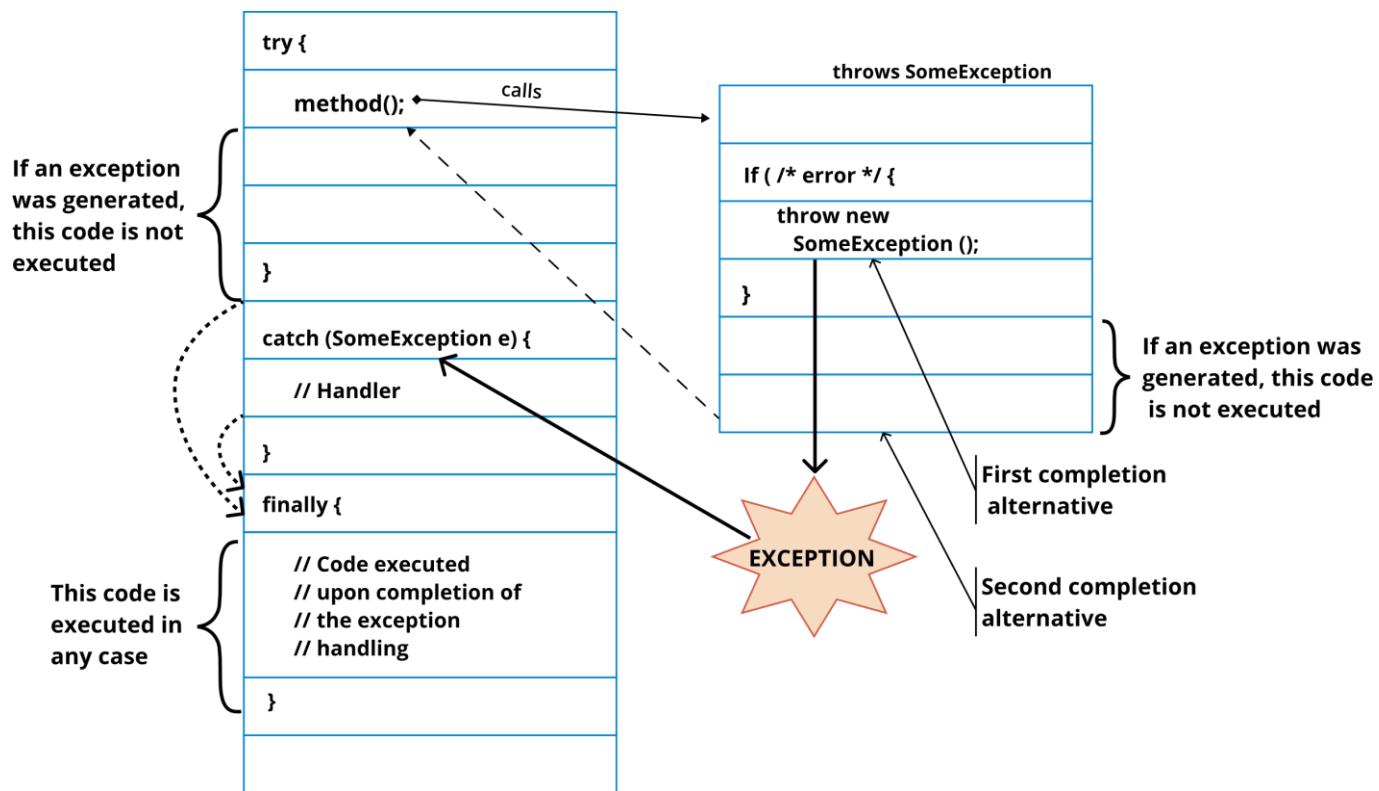


If the **try** block is linked to the **catch** and **finally** blocks, then exception handlers are described first and then the **finally** blocks specified.

Let's consider an example of code execution shown in the image below in different situations.

So, on the left we have code that has the **try** block with the **catch** and **finally** blocks. In the **try** block, we invoke some method `method()`, where an exception of the type **SomeException** can occur. Let's assume that an exception

occurs in `method()` and the method does not contain the matching handler. In this case, the method is exited early. Since it was invoked in the **try** block, the exception is **caught** by the catch block—first the exception is handled, and then the finally block is executed.



## Conclusion

In this lesson, you have studied how to:

- Handle exceptions in Java using the **try-catch** blocks. For this, a part of code where an exception may occur, is enclosed within the **try** block, after which follows the **catch** block. The **catch** block is a handling block for a specific exception specified in the block's parameters.
- Apply of the **try-finally** blocks to cleanup the code.

Now in your own programs, you can foresee exceptions and resolve them successfully.

## Check Your Knowledge!

1. What is the result of running the following code?

```

int a = 10;
String name = null;
try {
    a = name.length();
    a++;
} catch (RuntimeException e) {
    ++a;
}
System.out.println(a);
5
10
  
```

11 correct  
12  
Output error

Answer

Correct:

An exception NullPointerException will be generated in the `a = name.length();` operator and will be caught by the catch block. The application will work without errors. The increment operation will be performed only once, meaning that `a` will have a value of 11.

2. What is the result of running the following code?

```
public class Main {
    public int getInt() {
        int returnVal = 10;
        try {
            String[] students = {"Harry", "Paul"};
            System.out.println(students[5]);
        } catch (Exception e) {
            System.out.println("About to return: " + returnVal);
            return returnVal;
        } finally {
            returnVal += 10;
            System.out.println("Return value is now: " + returnVal);
        }
        return returnVal;
    }
    public static void main(String[] args) {
        Main main = new Main();
        System.out.println("In Main: " + main.getInt());
    }
}
```

About		to		return:	10
Return	value		is	now:	20
In Main: 20					
About		to		return:	10
Return	value		is	now:	20
In Main: 10 correct					
About		to		return:	10
Return	value		is	now:	10
In Main: 10					

Compilation error

Answer

Correct:

The operation `students[5]` will generate an exception—index out of range. In this block, through the `return` statement, the value 10 is set to return. After which the finally block will be executed, in which the value of the variable `returnVal` changes, but the return value does not change. Therefore, the `getInt()` method will return 10.

3. What is the result of running the following code?

```
public class TwistInTaleCatchError {
    public static void main(String[] args) {
        try {
            method();
        } catch (StackOverflowError e) {
            for (int i=0; i<2; ++i)
                System.out.println(i);
        }
    }
    public static void method() {
        method();
    }
}
```

```
    }  
}  
0  
Description of the  
java.lang.StackOverFlowError exception  
0  
1 correct  
0  
1  
2  
Description of the  
java.lang.StackOverFlowError exception  
Answer  
Correct:  
A situation of infinite recursion will occur, which will lead to a stack overload and an exception caught by the catch block. The loop in the catch block will print 0 and 1.
```

# Throwing and Passing Exceptions

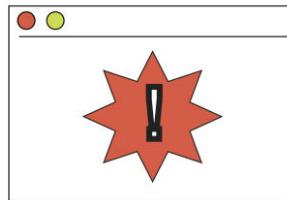
## Introduction

You already know how to handle exceptions. In this lesson, you will learn to throw and pass exceptions using the statement **throw** and the keyword **throws**.

## Throwing an Exception. Statement **throw**

In Java any code can throw an exception. This may be:

- Your code
- Code written by someone else
- Library classes code
- Java runtime environment



For example, your code may need to explicitly throw of an exception for indicate a wrong result of an operation or incorrect values of a method parameter.

Regardless of what throws the exception, Java allows the writer to do this using the statement **throw** with a parameter—an instance of the class that is a subclass of **Throwable**. In these cases, the following syntax is used:

```
throw <exception object>;
```

For example:

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
throw new ArithmeticException("test");
```

Let's consider an example of throwing an exception.

*Click on the title to see an example of throwing an exception.*

In the *main()* method, we invoke the *testG()* method enclosed within the **try-catch** blocks to control the occurrence of an exception of the **ClassCastException** type. The method *testG()* has its own **try-catch** blocks to handle the same type of exceptions **ClassCastException**. We will model a situation: we use the **throw** statement to throw an exception of the type **ClassCastException** in the **try** block of the *testG()* method. We successfully catch it in the method and throw it again in the **catch** block (why not?). The method is exited early to find a handler that is found in the *main()* method and executed. In the result we see that two handlers were executed: inside the method and outside it.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public class Test {
    static void testG() {
        try {
            throw new ClassCastException("Demo");
        }
    }
}
```

```

} catch (ClassCastException exp) {
    System.out.println("Exception in method!");
    throw exp;
}
}

public static void main(String[] args) {
    try {
        testG();
    } catch (ClassCastException e) {
        System.out.println("Exception of method!");
    }
}
}

```

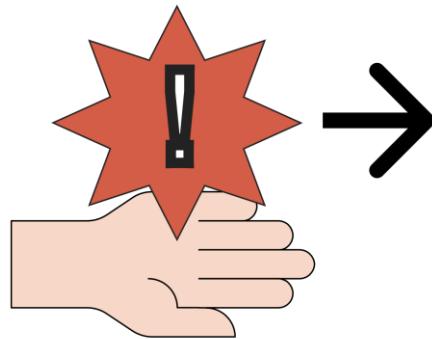
## Output

Exception in the method!

Exception of the method!

## Passing Exceptions. Keyword throws

You already know how to handle exceptions in methods where they occur. Sometimes this is justified. Nevertheless, you can let a method not to catch and handle an exception but to pass it to another code for handling. For example, a method does not have all the information about how to handle an exception and may throw over the exception to the code that invoked this method. This is called **pass an exception**.



To pass exception to the invoking method for handling, a component is added to the declaration of a method: **throws <types of exceptions>** .

[**access**] **<type of returned value>** **<method name>[(<parameters>)]** **throws <exception types>**  
 { **<method body>** }

For example:

*Hover the mouse cursor over the info icon to see the comment for the code.*

```

public double calc(int a) throws ArithmeticException {
    return 7.0 / a;
}

```

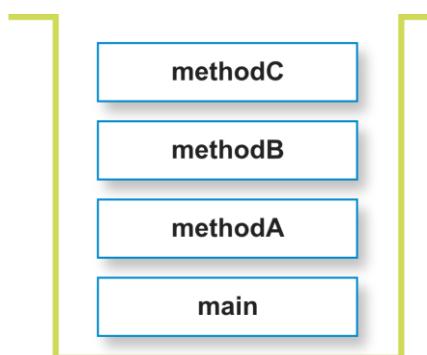
 The **throws** method component can specify several exceptions, listing them separated by commas.

Let's consider cases of passing an exception from a method with account for the exception category and kind.

*Click on the title to see an example of throwing an exception.*

## Methods call stack/searching for an exception handler

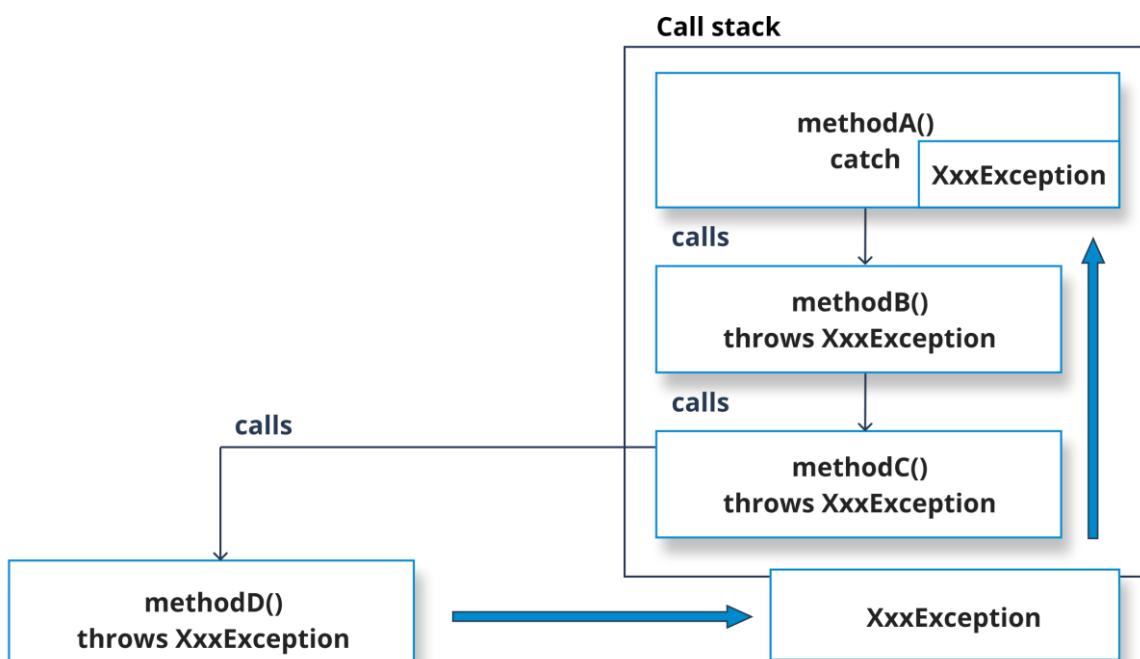
A typical application/program includes many levels of calling methods that are controlled by the **call stack**. In other words, when a method is called, the point of return is recorded in the stack (address of the statement). For example, the image on the right shows that we call method *methodA()* from the main *main()* method, and from there we call method *methodB()*, from there we call method *methodC()*, and from there we call some other method. Thus, we create a call chain of methods.



When an exception occurs inside a method, the method "throws" it into the virtual machine. The virtual machine starts searching for the exception handler:

- The search is executed back along the call stack until the handler is found.
- If the virtual machine fails to find the handler also in the *main()* method, it terminates the program.

The image below shows that method *methodA()* has a handler exception of the **XxxException** type. However, this exception occurs in method *methodD()*. This method has no handler, and therefore the virtual machine throw over it along the call stack until method *methodA()*, where the exception is handled.



## Example of rethrowing a checked exception

In this example, in the `main()` method we call the method `testExcp()`, inside of which an exception of the type **Exception** is thrown. The invocation of the method `testExcp()` itself is controlled for an exception occurrence by the blocks **try-catch**. However, the compiler throws an error in the method `testExcp()`.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public static void main(String[] args) {
    try {
        testExcp();
    } catch(Exception exp) {
        System.out.println(exp.getMessage());
    }
}
public static void testExcp() {
    throw new Exception("test");
}
```

The compilation error occurred because exceptions of the **Exception** type is checked exceptions and it is necessary to explicitly specify where to find its handler. In other words, the compiler has to be sure that the exception will be handled. For this, it is necessary to enclose the place of the expected occurrence of the exception in **try-catch** blocks, or using **throws** to specify a method further up the call stack to handle it.

Any exception that can occur in the method is part of the public interface of method programming. Those who call the method should know about the exceptions that can be thrown by the method, so that they can decide what to do with them.



If a method throws a checked exception but does not handle it, then it has to specify this using the keyword **throws**, so that the compiler can understand where the handler might be.

## Example of rethrowing an unchecked exception

In this example, in the `main()` method we call the `testExcp()` method, inside of which an exception of the type **RuntimeException** is thrown. The invocation of the method `testExcp()` itself is controlled for an exception occurrence by the blocks **try-catch**. In that case, the complier does not throw an error, because exceptions of the **RuntimeException** type are unchecked, and the decision on their handling rests with the developer. In other words, the compiler can neither inform this error, nor request to handle it.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public static void main(String[] args) {
    try {
        testExcp();
    } catch(RuntimeException e) {
        System.out.println(e.getMessage());
    }
}
public static void testExcp() {
    throw new RuntimeException("test");
```

Runtime exceptions can occur in any part of the program. Typically, there can be multiple exceptions. The necessity to add runtime exceptions to the declaration of each method will make the program difficult to understand. Therefore, the compiler does not require to specify such exceptions (although you can do this).

Thrown over exceptions through the **throws** keyword is called **exceptions passing**. There are two approaches to passing:

- **passive** – means that you just let the exception pass up the call stack until the appropriate handler without catching it along the way.
- **active** – means that you catch the exception, execute some actions, and re-throw it, or wrap it in a new exception and then throw:
  - When you need to add additional information about the exception
  - When you need to add additional information about the actions that were undertaken in relation to the data when the exception occurred.

Let's analyze the examples of the described approaches.

*Click on the tabs to see the examples.*

**Passive passing**

**Active passing**

In this example, the *doJob2()* method throws an exception of the **RuntimeException** type and does not handle it. The method *doJob()* calls the *doJob2()* method, but it also does not handle the exception but specifies that it pass it further. The *doJob()* method only executes some completing actions before passing the exception.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public void doJob() throws RuntimeException {
    try {
        doJob2();
    } finally {
        //...
    }
}
public void doJob2() {
    throw new RuntimeException();
}
```

In this example, the *doJob2()* method throws an exception of the **Exception** type and does not handle it. The *doJob()* method calls the *doJob2()* method and catches the exception to add information to it and then re-throws it further. Before the pass, some completing actions are executed in the *doJob()* method.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public void doJob() throws Exception {
    try {
        doJob2();
    } catch (Exception e) {
        throw new Exception(e.getMessage() + "more info");
    } finally { /* ... */ }
}
public void doJob2() throws Exception {
    throw new Exception();
}
```

You already know what happens when exceptions are re-thrown for handling. Now let's explore how **superclass methods are overridden**, if they have a **throws** component. An overridden method of a subclass:

- Cannot throw a wider exception than the method of its superclass.

- Can throw the same exception type that is specified for the superclass method, or an exception that is a subclass.
- May not specify the **throws** component when overriding.

Let's consider examples of different situations when we override a method that has the **throws** component in its declaration.

#### ▼ An incorrect example of method overriding

In this example, the *doJob()* method in the **Work** class throws and pass an occurred exception of the **IOException** type without handling it. The **Demo** class is a subclass of the **Work** class and when overriding the *doJob()* method it specifies that it pass an exception of the **Exception** type. In other words, a subclass method specifies that it can throw other types of exceptions than those related to the input/output.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
class Work {
    public void doJob() throws IOException {
        ...
    }
}
class Demo extends Work {
    @Override
    public void doJob() throws Exception {
        ...
    }
}
```

#### ^ A correct example of method overriding

In this example, the *doJob()* method in the **Work** class throws and pass an occurred exception of the **IOException** type without handling it. The **Demo** class is a subclass of the **Work** class and when overriding the *doJob()* method it specifies that it pass an exception of the **FileNotFoundException** type. In other words, a subclass method specifies that it can throw a more certain exception related to the input/output.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
class Work {
    public void doJob() throws IOException {
        ...
    }
}
class Demo extends Work {
    public void doJob() throws FileNotFoundException {
        ...
    }
}
```

#### ▼ Possible option for method overriding

In this example, the *doJob()* method in the **Work** class throws and pass an occurred exception of the **IOException** type without handling it. The **Demo** class is a subclass of the **Work** class and when overriding the *doJob()* method it does specify that it pass an exception. In other words, a subclass method can throw any exception and it does not specify explicitly that it will propagate it.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```

class Work {
    public void doJob() throws IOException {
        //...
    }
}
class Demo extends Work {
    public void doJob() {
        //...
    }
}

```

You have explored how exceptions are propagated and how to use the component of method declaration **throws**.

## Conclusion

In this lesson, you have found out that:

- To throw an exception, the **throw** statement with a parameter is used.
- To pass an exception to other methods for handling, a component **throws <exception types>** is added to the declaration of a method.
- Passive exception passing means that you just let the exception pass up the call stack until the appropriate handler without catching it along the way.
- Active passing means that you catch the exception, execute some actions, and re-throw it, or wrap it in a new exception and then throw it: when you need to add additional information about the exception or about the actions that were undertaken in relation to the data when the exception occurred.
- Overriding superclass methods when they have a **throws** component has its specifics.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

4/4 points

1. Which statements about overriding a method with a **throws** component in its signature are true?

The overridden subclass method can specify a wider exception than the method of its superclass

The overridden subclass method can specify only the same exception type that is specified by the method of its superclass

The overridden subclass method may omit the throws component

The overridden subclass method can specify a certain exception (subclass) than the method of its superclass

correct

Answer

Correct:

An overridden method of a subclass: cannot throw a wider exception than the method of its superclass; can throw the same exception type that is specified for the superclass method, or an exception that is a subclass; may not specify the throws component when overriding.

2. What is the result of running the following code?

```

public class OverAndOver {
    static String s = "";
    public static void main(String[] args) {
        try {
            s += "1";
            throw new Exception();
        } catch (Exception e) {
            s += "2";
        } finally {
            s += "3";
        }
    }
}

```

```

    doStuff();
    s += "4";
}
System.out.println(s);
}
static void doStuff() {
    int x = 0;
    int y = 7 / x;
}
}
12
123
1234

```

Compilation error

Only output of information about the exception correct

Answer

Correct:

The generation of the exception throw new Exception(); will be caught by the catch block. In the finally block, the method doStuff() will be called, where the exception will also be generated but it will not be caught.

3. Which options inserted in the place of //INSERT\_CODE will lead to a successful compilation of the following code?

```

public class Person { }
public class Father extends Person {
    public void dance() throws ClassCastException { }
}
public class Home {
    public static void main(String[] args) {
        Person p = new Person();
        try {
            ((Father)p).dance();
        } //INSERT CODE
    }
}
catch (NullPointerException e) {}
catch (ClassCastException e) {}
catch (Exception e) {}
catch (Throwable t) {}
catch (ClassCastException e) {}
catch (NullPointerException e) {}
catch (Exception e) {}
catch (Throwable t) {}
catch (ClassCastException e) {}
catch (Exception e) {}
catch (NullPointerException e) {}
finally { }

```

correct

Answer

Correct:

Successively placed catch blocks should contain exceptions that are not dependent hierarchically. If these are present, then subclasses should be placed above their superclasses.

4. Which declaration of the *ioRead()* method should be used instead of the comment for the code compilation and execution to be successful?

```

import java.io.*;
public class Quest {

```

```
//declaring ioRead()
public static void main(String[] args) {
    try {
        ioRead();
    } catch (IOException e){}
}
private static void ioRead() throws IOException{} correct
public static void ioRead() throw IOException{}  
public static void ioRead(){ }
public static void ioRead() throws Exception{ }
```

Answer

Correct:

The signature of the method should contain an component throws IOException, otherwise there will be a compilation error.

# Custom Exceptions

## Introduction

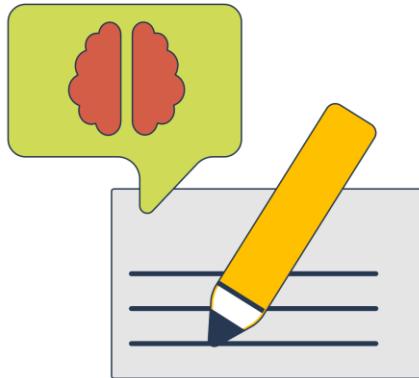
In this lesson, you will learn to create custom exceptions, as well as will get recommendations on how to handle exceptions.

## Custom Exceptions

To increase the quality and speed of code perception, a developer may create a custom exception as a subclass of the **Throwable** class or its subclasses. You can use such an exception to handle a situation that is not an exception from the point of view of the language but a logical error in the program.

According to the naming notation, a subclass of any exception class should end with the word **Exception**.

When writing custom exceptions, keep in mind the following:



- If you want to write a **checked exception**, then you need to expand the **Exception** class.
- If you want to write a **runtime exception (unchecked)**, then you need to expand the **RuntimeException** class.



Custom exceptions are used to handle logical errors of program in the same way as for system exceptions.

Let's consider an example of using a custom exception.

### Example of using a custom exception

The example describes the **Student** class that has setters to set fields. The *mark* field has limitations for the range of values; therefore, the *setMark()* method first checks the input value. If the value is out of range, an exception is thrown. This is not a common exception but a logical one. To handle it in the same way as common exceptions, a custom exception **MarkException** is described as a subclass of **Exception**. The program can resume its work after this error and the range of values is known; therefore, this exception refers to the checked category (checked exception). Since the *setMark()* method does not have information about how to handle this exception, an exception passes to the calling method. The *setMark()* method is called in the *main()* method. According to the rules for

handling checked exceptions, a call to the `setMark()` method is controlled for occurring exceptionhandled, which is handled in the `main()` method — in that case, by showing information about the occurred situation to the user.

*Hover the mouse cursor over the info icon to see the comment for the code.*

```
public class MarkException extends Exception {
    @Override
    public String getMessage() {
        return "Unacceptable value!";
    }
}
public class Student {
    private String firstName;
    private int group;
    private double mark;
    public Student(String firstName, int group) {
        this.firstName = firstName;
        this.group = group;
    }
    public void setMark(double mark) throws MarkException {
        if (mark < 0 || mark > 100) {
            throw new MarkException();
        }
        this.mark = mark;
    }
}
public class Main {
    public static void main(String[] args) {
        Student stud = new Student("Ivan", 505);
        try {
            stud.setMark(101);
        } catch (MarkException ee) {
            System.err.println( ee.getMessage() );
        }
    }
}
```

### Вывод в консоли:

Unacceptable value!

You have explored how to create custom exceptions. Now it is time to get familiar with recommendations on how to handle exceptions.

## Recommendations on How to Handle Exceptions

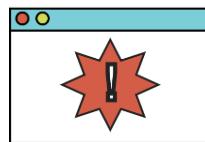
In the previous lessons, you studied the rules for handling exceptions and what they are used for. Now we propose to consider practices and recommendations on how to handle exceptions.

*Click on each heading to learn more.*

### Specify exceptions

Specify the **most specific exception classes**, even if you need to use several of them:

- This informs the code that caused exceptions how to handle them.
- This allows to change the **throw** statement when the method throws an additional exception.

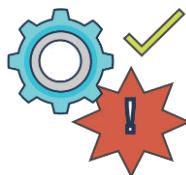


#### ▼ Use the mechanism for handling exceptions correctly

Do not use exceptions to control the flow of the application execution—this is considered an anti-pattern:

- In this case they mostly work as the **goto** statement, which makes the code very difficult to read.
- The virtual machine does not optimize the handling exception the same way as another code.

It is better to use correct conditions to break the looping statements or **if-else** for resolution, when code blocks should be executed.



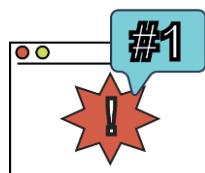
#### ▼ Log events in handlers

Log the exceptions where you handle them and not where they are thrown:

- There may be not enough information about the occurred situation to be implemented in the method of occurrence. Also, the exception may be a part of the expected behavior and be handled by the customer.
- You should avoid multiple registration of one and the same exception.



#### ▼ Do not lose the initial exception



When catching and wrapping the exception for its further passing (active propagation), you should always memorize the initial exception as a reason so that you don't lose it when tracing the call stack. For example, this applies when a decision is made to use a custom exception with error codes and a single handling location.

#### ▼ Do not generalize exceptions

Do not generalize exceptions when you catch specific exceptions by passing their supertype. This will force you to specify the reason in the handler, which will make the code bulky and difficult to read, and sometimes you might lose information.



For example, in the following code, the `doNotGeneralizeException()` method may throw an exception of the type **NumberFormatException** and **IllegalArgumentException** when executing the method `doSomething()`. The method `doNotGeneralizeException()` catches exceptions only to wrap them in the **Exception** type and throw them further for handling. In some part of the program, in a code block controlled for an occurrence of an exception of the **Exception** type, the `doNotGeneralizeException()` method is called, which means that the handler for this exception exists. However, for the handling of a concrete exception it is required to analyze what kind of exception went through the **Exception** type. This leads to a massive handling code.

```
public void doNotGeneralizeException() throws Exception {
    try {
        doSomething();
    } catch (NumberFormatException exp) {
        throw new Exception(exp);
    } catch (IllegalArgumentException exp) {
        throw new Exception(exp);
    }
}
// ...
try {
    doNotGeneralizeException();
} catch (Exception e) {
    if (e.getCause() instanceof NumberFormatException ) {
        log.error("NumberFormatException: " + e);
    } else if (e.getCause() instanceof IllegalArgumentException) {
        log.error("IllegalArgumentException: " + e);
    } else {
        log.error("Unexpected exception: " + e);
    }
}
```

#### ▼ Avoid unnecessary wraps of exceptions

Avoid exception wrapper classes that are only used to pass exceptions between the layers of the application and do not convey any additional useful information. In the most of cases, only one level of user exceptions is required.

For example, in the following code fragment a custom exception **PersistenceException** may be thrown in the `persistCustomer()` method that is passed to another method for handling. The `manageCustomer()` method calls the `persistCustomer()` method. If in this method an exception **PersistenceException** occurs, then it simply wraps the exception into another custom exception **BusinessException** that also passes it to another method for handling. The `createCustomer()` method calls the `manageCustomer()` method and, if an exception **BusinessException** occurs, then the method simply wraps it into another custom exception **ApiException** that also passes it to another method for handling. Therefore, we simply re-throw the exception between the program components changing its type that does not pass any useful information and only creates additional wrapping.

```

}
public void manageCustomer(Customer c) throws BusinessException {
    // manage a Customer
    try {
        persistCustomer(c);
    } catch (PersistenceException ex) {
        throw new BusinessException(ex, ex.getCode());
    }
}
public void createCustomer(Customer c) throws ApiException {
    // create a Customer
    try {
        manageCustomer(c);
    } catch (BusinessException ex) {
        throw new ApiException(ex, ex.getCode());
    }
}

```

## Conclusion

In this lesson, you learned to write custom (user) exceptions as a subclass of the **Throwable** class.

You also received practical recommendations to be used for handling exceptions:

- Specify exceptions
- Use the handling mechanism correctly
- Log events in handlers
- Do not lose the initial exception
- Do not generalize exceptions
- Avoid unnecessary wraps of exceptions

## Check Your Knowledge!

1. What is the result of compiling the following code?

```

public class Main {
    public static void main(String[] args) {
        try {
            System.out.println(methodA());
        }
        catch (Throwable ex) {
            System.out.println("Main Catch");
        }
    }
    static int methodA() {
        if(methodB() >0) {
            try {
                throw new UnknownError();
            }
            catch (Exception ex) {
                return 2;
            }
        }
        else {
            return 3;
        }
    }
    static int methodB()
    {
        try {
            throw new RuntimeException();
        }
        catch (Exception ex) {
            return 0;
        }
        finally {

```

```

        return 1;
    }
}
1
2
3
0

```

Main Catch correct

Answer

Correct:

From the main() method, methodA() is called, and from methodA(), methodB() is called, which returns 1 (the finally block is executed). Then an UnknownError exception is thrown in the methodA() method, which is caught in the main() method, and therefore Main Catch is printed to the console.

2. What is the result of compiling the following code?

```

class ExceptionOne extends Exception {}
class ExceptionTwo extends ExceptionOne {}
abstract class Abstract {
    abstract void method() throws ExceptionOne;
}
public class Main extends Abstract {
    static int a,b,c,d;
    @Override
    void method() throws ExceptionTwo {
        throw new ExceptionTwo();
    }
    public static void main(String[] args) {
        Main main = new Main();
        try {
            main.method();
            a++;
        }
        catch (ExceptionTwo ex) {
            b++;
        }
        catch (ExceptionOne ex) {
            c++;
        }
        finally {
            d = a + b + c;
        }
        System.out.println(a + " " + b + " " + c + " " + d);
    }
}

```

compilation error

0 1 1 2

1 0 0 1

0 0 1 1

0 1 0 1 correct

Answer

Correct:

The main() method calls method(), which throws an exception of type ExceptionTwo. This exception is caught in the catch(ExceptionTwo ex) block of the main() method, after which the finally block of the main() method is executed. The console prints: 0 1 0 1.

3. You are given the following code:

```

class A{
    public void f() throws IOException{}
}
class B extends A{}

```

How can we override the *f()* method in the **B** class without causing a compilation error?

public void f() throws Exception {}  
public void f() throws IOException {}  
public void f() throws InterruptedException, IOException {}  
public void f() throws IOException, FileNotFoundException {}  
public void f() throws FileNotFoundException {}  
public void f() throws FileNotFoundException, InternalError {}  
incorrect

Answer

Incorrect:

Обратитесь повторно к материалам данного модуля, чтобы найти правильный ответ.

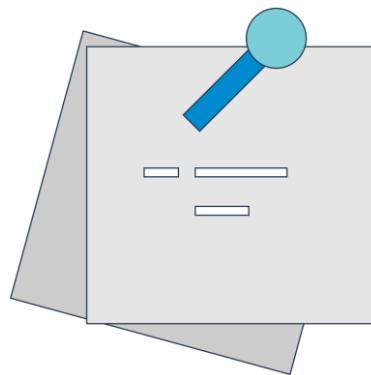
# Annotations

## Introduction

In this lesson, you will explore annotations and how to work with them.

## What Are Annotations?

Annotations are meta tags that are added to code and applied when declaring packages, classes, constructors, methods, fields, parameters, and local variables. Annotations contain additional information and link it with all the listed program elements. In fact, annotations are sort of additional modifiers that do not lead to changes in previously created code. In many situations, annotations help avoid the need to create a template code by activating utilities for its generation from the annotations in the source code. Since they can shorten code and reduce code coupling, annotations are used widely in different technologies and frameworks.



You can mark structural elements of Java code with annotations: classes, interfaces, fields, methods, and parameters. In this case, annotations are placed before the elements, usually one line above. To declare an annotation, its name should start with the @ character. Let's look at an example of using the **@Test** annotation from the **JUnit** library:

*Hover your cursor over the info icon to see an explanation of the code.*

```
@Test
void testReturnMaxForNegativeValues() {
    int[] values = new int[]{-4, -3, -2, -145};
    int result = MaxMethod.max(values);
    assertEquals(-2, result);
}
```

A declaration of any type can be annotated. You can even annotate other annotations. Annotations always precede declarations. In the following example, since the target annotation **@Target** is declared with the **ElementType.TYPE** value, it can be used to annotate a class declaration.

*Hover your cursor over the info icon to see an explanation of the code.*

```
1 @Target(ElementType.TYPE)
2 public @interface BaseAction {
3     int level();
4     String sqlRequest();
5 }
```

When using an annotation, you need to set the values of its elements if they were not set by default when declaring the annotation. Let's look at an example where the **@BaseAction** annotation accompanies a class declaration. This annotation marks the **Base** class. After the name of the annotation comes a list of initializing values for the annotation elements enclosed in round brackets. To pass a value to an annotation element, the name of this element is assigned the value. Therefore, in the given example, the line "SELECT name, phone FROM phonebook" is assigned to the *sqlRequest()* method, a member of the annotation of the **BaseAction** type. Also, in the assignment operation, there are no parentheses after the name *sqlRequest*. When a member method gets an initializing value, only the method name is used. Therefore, member methods look like fields in this context.

*Hover your cursor over the info icon to see an explanation of the code.*

```
@BaseAction(level = 2, sqlRequest = "SELECT name, phone FROM phonebook")
public class Base {
    public void doAction() {
        Class clazz = Base.class;
        BaseAction action = (BaseAction) clazz.getAnnotation(BaseAction.class);
        System.out.println(action.level());
        System.out.println(action.sqlRequest());
    }
}
```

An annotation contains only a declaration of methods; you do not need to add a body to these methods since this part is implemented by the language itself. Besides, these methods cannot contain parameters of the **throws** section and act like fields. The following types of returned values are allowed: primitives, String, Enum, Class, annotations, as well as arrays of any of these types.

All types of annotations automatically expand the **Annotation** interface from the **java.lang.annotation** package. The **Annotation** interface contains the *annotationType()* method, which returns an object of the **Class** type that represents a calling annotation.

If you need access to an annotation when executing the application, then before declaring the annotation, you should set a **RUNTIME** retention rule that allows for the maximum life of the annotation:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
```

If we mark the **@BaseAction** annotation like this, then the **doAction()** method of the **Base** class from the above example will be executed without errors.

Now that you have explored annotations and what they are used for, let's look at the different types of annotations.

## Types of Annotations

Annotations are classified into the following basic types: marker annotations, single-element annotations, and normal annotations. Let's take a detailed look at each type.

**Marker annotations**

**Single-element annotations**

**Normal annotations**

A marker annotation does not contain member methods. It is created to mark a declaration. Since the interface of the marker annotation does not have member methods, it is enough to define the presence of an annotation:

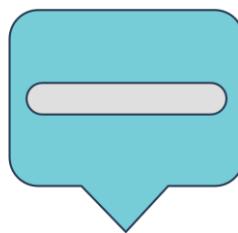


```
public @interface MarkerAnnotation {}
```

To check for the presence of an annotation, use the method *isAnnotationPresent()*.

Now that you have reviewed the most basic types of annotations, let's move on to built-in annotations.

A single-element annotation contains a single member method. For this type of annotation, you can use a short form to set the value for the member method: when creating the annotation, specify only the value of the member method. You do not need to specify the name of the member method. But to use a short form, you should use the name *value()* for the member method.



Now that you have reviewed the most basic types of annotations, let's move on to built-in annotations.

Normal annotations contain several member methods. With these, you should use complete syntax for each parameter:



```
parameterName = value
```

Now that you have reviewed the most basic types of annotations, let's move on to built-in annotations.

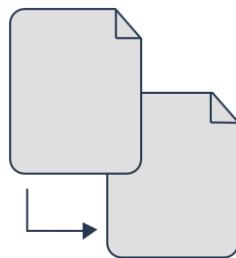
## Predefined Annotations

Several annotations defined in Java are used to create new annotations. Let's review the most common predefined annotations in Java.

*Click on the tabs to learn more about these annotations.*

## @Override

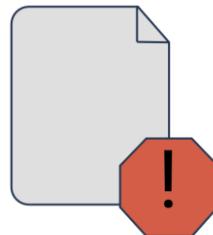
## @Deprecated



The **@Override** annotation informs the compiler that the method marked with it overrides a method of a superclass or interface. If the overridden method is not found during compilation, an error will be thrown. This way, it is convenient to control the correctness of overriding methods in the code. It is recommended always marking the methods being overridden with this annotation.

*Hover your cursor over the info icon to see an explanation of the code.*

The **@Deprecated** annotation indicates that the marked elements are not recommended for use anymore and can be deleted from now on. In the Java world, it is customary to ensure of backward compatibility. Therefore, even unsuccessful elements are not usually deleted right away in new API versions; they are marked with this annotation so that library users can prepare to delete them in the future. Imagine a café wants to change its profile from street food to vegetarian cuisine. However, to keep from disappointing their regular customers, it decided to keep old menu items for a while, marking them as **@Deprecated**.



*Hover your cursor over the info icon to see an explanation of the code.*

```
class CafeKitchen{
    @Deprecated
    Burger makeBurger(){
        ...
    }
    Falafel makeFalafel(){
        ...
    }
}
```

Java also has other predefined annotations, for example:

## @SuppressWarnings

These suppress specific warnings by the compiler in the indicated method.

## @SafeVarargs

In specific cases of combining a variable number of arguments and parameterization, this annotation suppresses the compiler's warning.

## @FunctionalInterface

This annotation marks an interface as functional. In other words, the preferred means of implementing it is to use lambda.

Now that you have reviewed the basic types of built-in annotations, it's time to find out how to create a custom annotation.

## Custom Annotations

You can also create your own annotations. Library developers often use this capability to shorten code and reduce its coupling. In Java, the **java.lang.annotation** package includes several built-in annotations that are used to create custom annotations:

### @Retention

Specifies how long the annotation should be stored

### @Target

Indicates what elements the annotation can be applied to

### @Inherited

Specifies whether the created annotation can be applied automatically to descendant classes

### @Documented

Indicates whether it is necessary to save information on how to use the specified annotation when running the application in the automatically generated documentation

The example below shows a `TestClass` annotation that can be used to mark classes (`@Target(ElementType.TYPE)`). The information on marking with this annotation will be included in the Javadoc of the marked class (`@Documented`) and will also be available when running the program (`@Retention(RetentionPolicy.RUNTIME)`). All descendants of the marked classes will also be considered marked (`@Inherited`).

```
@Target(ElementType.TYPE)
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@interface TestClass { }
```

You have now learned how to create your own annotations; let's continue by exploring how to process annotations.

## Processing Annotations

In one of the examples earlier in this lesson, you saw how to process an annotation in the `doAction()` method of the `Base` class. This implementation is rather primitive.

*Click on the title to take another look at the code.*

```
@BaseAction(level = 2, sqlRequest = "SELECT name, phone FROM phonebook")
public class Base {
    public void doAction() {
        Class clazz = Base.class;
        BaseAction action = (BaseAction) clazz.getAnnotation(BaseAction.class);
        System.out.println(action.level());
        System.out.println(action.sqlRequest());
    }
}
```

With this type of processing, every time the developer uses annotations, they will have to write code to extract the values of its elements and the corresponding reaction to these values.

The best approach would be to implement the annotation in such a way that it is enough for the developer to simply annotate the class, method, or field and pass the required value. The system's reaction to the annotation should be automatic. Below is an example of working with an annotation based on the **Reflection API**.

*Click on the title to see the example.*

The **BankingAnnotation** annotation is used to set the level of security verification when calling a method.

*Hover your cursor over the info icon to see an explanation of the code.*

```
package by.epam.learn.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface BankingAnnotation {
    SecurityLevelType securityLevel() default SecurityLevelType.MEDIUM;
}
```

In the above example, the default value of **securityLevel** (medium) was used since there was no explicit value. The possible values of the security levels can be represented as follows:

```
package by.epam.learn.annotation;
public enum SecurityLevelType {
    LOW, MEDIUM, HIGH
}
```

In this example, the methods of the system logic class perform actions with a bank account. For different reasons, specific implementations of the **AccountManager** interface may require additional actions. Let's look at an example of annotating methods.

*Click on the title to see the example.*

```
package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
public class AccountManagerImpl implements AccountManager {
    @BankingAnnotation(securityLevel = SecurityLevelType.HIGH)
    public double depositInCash(int accountId, int amount) {
        System.out.println("deposit in total: " + amount );
        return 0;
    }
    @BankingAnnotation(securityLevel = SecurityLevelType.HIGH)
    public boolean withdraw(int accountId, int amount) {
        System.out.println("amount withdrawn: " + amount);
        return true;
    }
    // run again without comment
    // @BankingAnnotation(securityLevel = SecurityLevelType.LOW)
    public boolean convert(double amount) {
        System.out.println("amount converted: " + amount);
        return true;
    }
    @BankingAnnotation // default value MEDIUM
    public boolean transfer(int accountId, double amount) {
        System.out.println("amount transferred: " + amount);
        return true;
    }
}
```

The configuration and launch of BankingMain.java will look like this:

```
package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
public class BankingMain {
    public static void main(String[] args) {
        AccountManager manager = SecurityFactory.registerSecurityObject(
            new AccountManagerImpl());
        manager.depositInCash(10128336, 6);
        manager.withdraw(64092376, 2);
        manager.convert(200);
        manager.transfer(64092376, 300);
    }
}
```

The approach of calling an intermediate method for inclusion in annotation processing is quite common and used in JPA, Hibernate, Spring, and other technologies.

A call of the *createSecurityObject()* method registering class methods to process annotations can be placed in the constructor of an intermediate abstract class implementing the **AccountManager** interface or in the static logic block of the interface itself. In this case, all methods of all implementations of the **AccountManager** interface will react to annotations.

*Click on the title to see the example.*

#### ^ An example of the logic for processing annotation values

In the example, we create a **SecurityService** class, providing the actions of the application based on the **SecurityLevelType** value passed to the annotated method.

```
package by.epam.learn.annotation;
import java.lang.reflect.Method;
import java.util.Arrays;
public class SecurityService {
    public void onInvoke(SecurityLevelType level, Method method, Object[] args){
        System.out.printf("%s() was invoked with parameters: %s ",
            method.getName(), Arrays.toString(args));
        switch (level) {
            case LOW -> System.out.println("security: " + level);
            case MEDIUM -> System.out.println("security: " + level);
            case HIGH -> System.out.println("security: " + level);
            default -> throw new EnumConstantNotPresentException(
                SecurityLevelType.class, level.toString());
        }
    }
}
```

The static method *registerSecurityObject(AccountManager target)* of the factory class creates a proxy instance based on the implementation of the **AccountManager** interface. In addition to all the features of the original, it can implicitly access the chosen business logic, depending on the value of the annotation field.

*Click on the title to see the example.*

#### ▼ An example of creating a proxy instance

```
package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
```

```

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class SecurityFactory {
    public static AccountManager registerSecurityObject(AccountManager target) {
        return (AccountManager) Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new SecurityInvocationHandler(target));
    }
    private static class SecurityInvocationHandler implements InvocationHandler {
        private Object targetObject;
        SecurityInvocationHandler(Object target) {
            this.targetObject = target;
        }
        public Object invoke(Object proxy, Method method, Object[] args)
            throws InvocationTargetException, NoSuchMethodException, IllegalAccessException {
            SecurityService service = new SecurityService();
            Method invocationMethod = targetObject.getClass().getMethod(method.getName(),
                (Class[]) method.getGenericParameterTypes());
            BankingAnnotation annotation = invocationMethod.getAnnotation(BankingAnnotation.class);
            if (annotation != null) {
                // method annotated
                service.onInvoke(annotation.securityLevel(), invocationMethod, args);
            } else {
                /* if annotation of method is required, it should
                throw an exception to the fact of its absence */
                /* throw new InvocationTargetException(null, "method " + invocationMethod
                + " should be annotated "); */
            }
            return method.invoke(targetObject, args);
        }
    }
}

```

## Output

```

depositInCash() was invoked with parameters: [10128336, 6] security: HIGH
--method execution: deposit in total: 6
withdraw() was invoked with parameters: [64092376, 2] security: HIGH
--method execution: amount withdrawn: 2
--method execution: amount converted: 200.0
transfer() was invoked with parameters: [64092376, 300.0] security: MEDIUM
--method execution: amount transferred: 300.0

```

Using annotations can be mandatory or optional:

If the *invoke()* method throws an exception about the absence of an annotation in a method, this indicates that the class methods implemented when implementing the interface **should** explicitly **use** the annotation.

The absence of a thrown exception in the block after **else** means that annotating all methods of the **AccountManager** class is **optional**. The method itself will be called, despite the absence of an annotation.

Examples of correct use of annotations cannot be short.

## Conclusion

In this lesson, you have seen that:

- Annotations contain additional information and link it with different program elements. Annotations help avoid the need to create a template code by activating utilities to generate it from annotations in the source code. They also allow for shortening code and reducing coupling.
- The predefined **@Override** annotation informs the compiler that the method is meant to override a method declared in a superclass or interface.
- The predefined **@Deprecated** annotation indicates that elements are not recommended for further use and can be deleted from now on.
- There are several predefined annotations for creating new annotations from the **lang.annotation** package: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
- It is good practice to implement an annotation in such a way that it is enough for the developer to simply annotate the class, method, or field and pass the required value. The system's reaction to the annotation should be automatic.

## Check Your Knowledge!

1. Which annotation suppresses compiler warnings in a marked method?

**@Retention**

**@Target**

**@SuppressWarnings** correct

**@Deprecated**

Answer

Correct:

The **@SupressWarnings** annotation suppresses specific warnings by the compiler in a marked method.

2. Which rule must be used to access an annotation when running an application?

**ElementType.TYPE**

**Retention**

**@Override**

**RUNTIME** correct

Answer

Correct:

If we need access to an annotation when executing an application, then before declaring the annotation, we should set a **RUNTIME** retention rule for the maximum life of the annotation.

# Generic Types

## Introduction

In this lesson, you will explore generics, including their properties and why they are used in the Java language.

## The Concept of Generics

One of the most important innovations in the fifth version of the Java language was the addition of parameterization (generics) of data and method types. This innovation adds the flexibility of defining generic classes and methods, which is a natural advantage of dynamic typing. However, we still have the compile-time type safety checks granted by static typing.

**Generics** refers to the ability of a class/interface/method to be multipurpose in defining data types they work with. Users can then specify data types when creating an object or calling a method.



## Properties of Generics

The addition of generics brought some advantages to code.

*Click on each active element to learn more.*



### Type

safety

Generics provide type safety during compilation. This is possible since errors related to the use of data types and errors that can occur during runtime are now detected at the compilation stage.

No need for type casting

Generics eliminate the need for explicit type casting between objects since the compiler has already been informed of what types of data it works with.

### Homogeneous

collections

Generics allow for the creation of homogeneous collections that are checked by the compiler. In other words, generics guarantee that all operations with the collection elements are correct since the compiler has been informed about the types of elements in the collection and can run checks.

Now let's explore what the above properties look like in practice. To start with, let's look at the **type multipurposeness before generics**: Multipurposeness is assured by the **Object** type as a supertype for any class in Java.

Below you can see an example of a description of a multipurpose class, in which the **DynamicArray** class is described and contains:

- An *elements* field as an array of values of the **Object** type
- A method for adding objects to the array *addElement()* that takes an argument of the **Object** type
- A method for receiving an object by its index *getElement()* that returns an object of the **Object** type

This means that the class works with any data type. Let's assume that you want to use this class to work with numerical data. To do this:

1. Introduce the Demo class, where you create an object of the **DynamicArray** class, and on this object, call the *addElement()* method to place numbers into the array.
2. Add a string to the array and make sure that there is no error.
3. Get the object from the array and cast it to the type **Integer** since it should contain numerical data. As a result, you get a runtime error since a string will be returned that cannot be cast to a number using explicit casting.

*Click on the dropdown element to study the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class DynamicArray {
    private Object[] elements;

    public void addElement(Object element) {
        // ...
    }

    public Object getElement(int index) {
        // ...
    }
    // ...
}

public class Demo {
    public static void main(String[] args) {
        DynamicArray dynArray = new DynamicArray();
        dynArray.addElement(Integer.valueOf(10));
        Integer first = (Integer) dynArray.getElement(0);
        // ...
        dynArray.addElement("Java");
        Integer second = (Integer) dynArray.getElement(1);
    }
}
```

There is no insurance against the errors shown in the example. They can only occur during runtime since the compiler cannot perform such a check.

Next, let's take a look at the changes in the type multipurposeness when using generics. We'll start with the syntax.

*Click on the tabs to learn more.*

**Defining a generic class/interface**

**Instantiating objects of a generic type**

When a class or interface needs to be defined as generic, the following syntax is used:

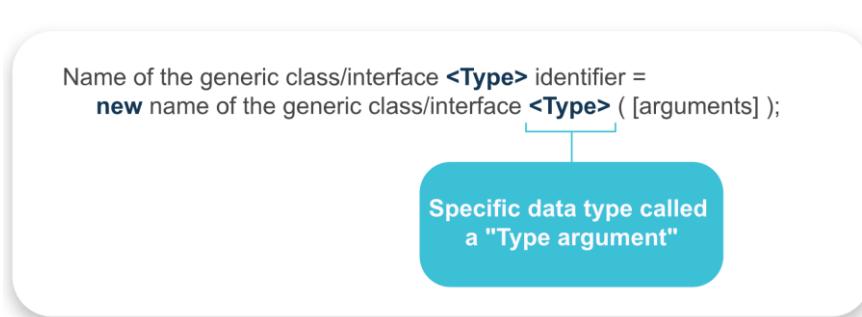
[access] class/interface class/interface name <T1, T2, ..., Tn> { }

"Type Parameter" component

The "Type parameter" component is always specified in angle brackets (<>) and always follows the name of the class or interface. It is also called "type variable" and represents a formal designation of the data type that will be used in this class or

interface. In other words, by defining the type variables through T1, T2, etc., you will be able to use them anywhere inside the class or interface.

If an object needs to be created based on a generic type, we use the following syntax:



The type argument is specified both when describing the reference and calling a constructor so that the compiler can check that the reference and object types match.

The type argument can be any reference type (class or interface).



A primitive type cannot be used as a type argument.

Note that there are **conventions regarding how to name a type parameter**. The Java language gives recommendations for naming the type parameter:



Use a one-character designation in uppercase. This is in sharp contrast to how class and interface types are designated, which makes it is easy to distinguish them.

The most common names of type parameters are:

- E - Element (for elements of collections)
- T - Type (for any type)
- K - Key (for a key)
- V - Value (for a value)
- N - Number (for numerical data)
- S, U, V, etc. - for the 2nd, 3rd, and 4th types of parameters

By parsing the syntax, let's change the example of declaring a multipurpose **DynamicArray** class from the previous clause by making it generic:

*Click on the dropdown element to study the example.*

1. In the **Demo1** class, create the reference *dynArray* and an object based on the generic class with an argument of the **Integer** type. This will tell the compiler that only objects of the **Integer** type can be placed inside the *dynaArray* object.
2. Add objects of the **Integer** class and the **String** class to the *dynArray* object. As a result, the compiler will report an error when trying to add a string.

3. Get an object from `dynArray`: Unlike in the previous example, here you do not need to perform explicit casting of the returned object to the **Integer** type since the compiler knows that only objects of the **Integer** type are stored in `dynArray`.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class DynamicArray<T> {
    private T[] elements;

    public void addElement(T element) {
        // ...
    }

    public T getElement(int index) {
        // ...
    }
    // ...
}

public class Demo1 {
    public static void main(String[] args) {
        DynamicArray<Integer> dynArray = new DynamicArray<Integer>();
        dynArray.addElement( Integer.valueOf(10) );
        Integer xx = dynArray.getElement(0);
        dynArray.addElement("Java");
        Integer yy = dynArray.getElement(1);
    }
}
```

Now that you have explored the properties of generics and analyzed some examples of multipurpose types before and on generics, it's time to move on to the diamond operator.

## The Diamond Operator



In Java version 7 and later, you can replace the type arguments required to invoke a generic type constructor with an empty set of type arguments (<>). The compiler can determine the type arguments on its own from the context. This pair of angle brackets <> is unofficially called [a diamond operator](#).

This makes code more readable. For example, we could create an instance of the generic class **DynamicArray<Integer>** as follows:

```
DynamicArray <Integer> mb1 = newDynamicArray <>()
```

For more about the diamond operator, see the [official documentation](#).

## Multiple Type Parameters

A generic class or interface can have several type parameters since different types of fields can be used in classes and interfaces.

Let's analyze the use of multiple type parameters on the **KeyValue** interface and the **KeyValueImpl** class. In this example, the **KeyValue** interface is parameterized by two type parameters that relate to the Key and Value types. A

class implementing a generic interface is also generic by the same number of type parameters. In our example, this is the **KeyValueImpl** class.

```
public interface KeyValue <K, V> {
    public K getKey();
    public V getValue();
}
public class KeyValueImpl<K, V> implements KeyValue <K, V> {
    private K key;
    private V value;
    public KeyValueImpl (K key, V value) {
        this.key = key;
        this.value = value;
    }
    @Override
    public K getKey() { return key; }
    @Override
    public V getValue() { return value; }
}
```

What will happen if we don't specify the type argument when using a generic class or interface? Let's keep going to find out.

## The Raw Type

Using the name of a generic class or interface without specifying the type argument is called a **raw type**. This means that the compiler cannot perform all the necessary checks to guarantee the safe use of a data type.

For instance, creating an object based on the generic type **DynamicArray** without specifying a type argument will allow objects of any type to be placed inside the *dynArray* object.

```
DynamicArray dynArray = new DynamicArray();
dynArray.addElement( Integer.valueOf(10) );
dynArray.addElement( Double.valueOf(10.5) );
```

It is important to understand that:

- A described non-generic type of a class or interface is a not a raw type.
- Raw types are used in inherited code since many library classes did not have a generic nature before Java 5.

## Conclusion

In this lesson, you have studied the concept of parameterization and the properties of generics. You have seen that:

- Generics refers to the generic nature of a class/interface/method when defining the data types they work with. Generics allow users to use specific data types when invoking a method or creating an object.
- Generics make code more stable since they allow the compiler to detect most errors.
- An empty set of type arguments (<>) may replace the type arguments required to invoke a constructor of a multipurpose type. This pair of brackets is called a diamond operator.
- A raw type is the name of a multipurpose class or interface without any type arguments.

## Check Your Knowledge!

1. Which of the following are considered advantages of generics?

Assurance of type safety

The need for explicit casting of types between objects

The ability to create homogeneous collections checked by the compiler

The compiler is informed about the types of data it works with.

correct

Answer

Correct:

The addition of generics assured type safety since the compiler knows the types of data it works with. As a result, most errors can be eliminated at the compilation stage. Moreover, since the compiler is informed, there is no need for explicit type casting between objects.

2. What is the result of the following code?

```
class X {  
    public <X> X(X x) { }  
}
```

A compilation error

Successful compilation without warnings

Successful compilation but with warnings correct

Answer

Correct:

The compilation is successful, but there is a warning that the class and the parameterized type have the same name. This use is difficult to read and makes it difficult to use the class.

# Generic Methods

## Introduction

In this lesson, you will see that methods can be declared as generic and explore the syntax used to declare parameterized methods.

## Generic Methods

Not just classes and interfaces can be generic in the Java language. It is also possible to declare a generic method. This is similar to declaring a generic type, but the scope of the type parameter is limited by the method where it is declared.

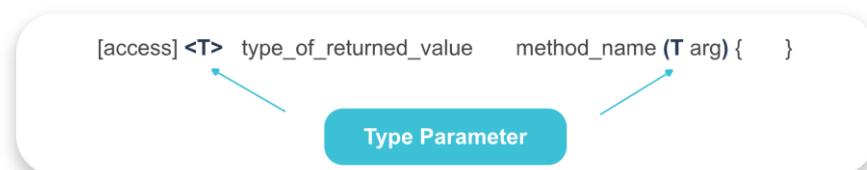
Usually, a **parameterized/generic method** defines a basic set of operations that will be applied to different data types received by the method as a parameter. The following methods can be described as generic:

- Class methods (static)
- Instance methods (not static)
- Constructors

Next, you will explore the syntax for using generic methods.

## Syntax

When declaring a generic method, the location of the type parameter is always specified after the access modifier and before the type of the returned value.



Below is an example of declaring a usual class **GenericMethod** with a generic static method *asByte()*. This method can accept parameters of any type; however, this method's task is to transform an object to a value of the primitive type **byte**. You can't do this with all data types—only with types extending **Number**. Therefore, the *asByte()* method, using the **instanceof** statement, first checks whether the received object falls within the **Number** type, and only after this is the *byteValue()* method used for the transformation. Otherwise, a zero value is returned.

*Hover your cursor over the info icon to see an explanation of the code.*

```
class GenericMethod {
    public static <T> byte asByte(T num) {
        if (num instanceof Number) {
            return ((Number) num).byteValue();
        }
        return 0;
    }
}
```

Generic methods can be present both in ordinary classes and in parameterized classes. A method's type parameter might not be related to the type parameter of its generic class at all.

Let's see how to call a generic method based on its type.

*Hover your cursor over the underlined elements to learn more.*

### A generic method of an instance

```
reference_to_instance.<Type>method_name( arguments_list );
```

### A generic method of a class (static)

```
class_name.<Type>method_name( arguments_list );
```

For example:

```
class Box<T> {  
    private T value;  
  
    public Box(T value) {  
        this.value = value;  
    }  
}
```

Study the following example of applying generic methods when using the `asByte()` method of the **GenericMethod** class.

*Click on the dropdown element to study the example.*

In this example, we create the **Main** class, where we call the static generic method `asByte()` with arguments of the **Integer**, **Float**, and **Character** types.

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(GenericMethod.asByte(Integer.valueOf(7)));  
        System.out.println(GenericMethod.asByte(Float.valueOf(7.0F)));  
        System.out.println(GenericMethod.asByte(Character.valueOf('7')));  
    }  
}
```

### Code output

```
7  
7  
0
```



You can call generic methods without specifying the type argument; the compiler determines the type on its own from the context.

### Conclusion

In this lesson, you have seen that:

- A generic method determines a basic set of operations with different data types that are received by the method as a parameter.
- Class methods (static), instance methods (not static), and constructors can be declared as generic.
- 

## Check Your Knowledge!

1. Which of the options below reflects the correct location of the type parameter?

After the access modifier and the type of the returned value

Before the type of the returned value and after the access modifier correct

Before the access modifier and the type of the returned value

Answer

Correct:

The location of the type parameter is always specified before the type of the returned value and after the access modifier.

2. Which classes can have generic methods?

Only ordinary classes

Only parameterized classes

Both ordinary and parameterized classes correct

Answer

Correct:

Both ordinary and parameterized classes can have generic methods.

# Restrictions and Erasure. Generics and Inheritance

## Introduction

In this lesson, you will explore the possibilities for restricting type parameters that give a stricter definition of the data types being used. You will become familiar with the concept of type erasure and the process of erasure. You will also discover the limitations of using generics and whether inheritance works with generic types.

## Types of Restrictions

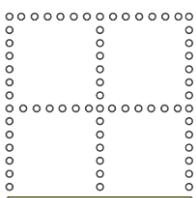
Sometimes we want to declare a generic type or method that is parameterized not with all the possible types but only with a subset of them. In other words, not all data types can be used with a generic class or method. For example, a generic class is declared to work with numerical data types, and using types that do not support operations with numerical data will lead to an error.

To avoid such errors and additional checks for the type parameter, you can set bounds. There are two types of bounds: upper and lower.

*Click on the cards to see the description of bounds.*

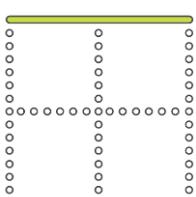
A lower bound is described as

<T **super** Type> and means that T can only be of the **Type** type or its supertype.



An upper bound is described as

<T **extends** Type> and means that T can only be of the **Type** type or its subtype.



Below is an example of how to use bounds for the type parameter.

*Click on the dropdown element to study the example.*

In this example, we declare a generic class **Div** with two fields of the T type and the *perform()* method that performs integer division. The T parameter is upper-bounded at the **Number** class, meaning that it can only use descendant classes of **Number** as the type argument. Therefore, when using the *perform()* method, we do not need to check

whether the references *x* and *y* belong to the **Number** type or cast them before calling the *doubleValue()* method. The compiler will do this on its own.

*Hover your cursor over the info icon to see an explanation of the code.*

```
class Div < T extends Number > {
    private T x;
    private T y;
    Div(T a, T b) {
        x = a;
        y = b;
    }
    public int perform() {
        return (int)(x.doubleValue() / y.doubleValue());
    }
}

public class Main {
    public static void main(String [] args) {
        double aD = Math.random() * 100;
        double bD = Math.random() * 100;
        short aS = (short)(Math.random() * 100);
        short bS = (short)(Math.random() * 100);
        Div<Double> divD = new Div <(aD, bD);
        System.out.println(aD + " : " + bD + " = " + divD.perform());
        Div<Short> divS = new Div <(aS, bS);
        System.out.println(aS + " : " + bS + " = " + divS.perform());
        Div<Character> obj3 = new Div <'9', '2'>;
    }
}
```

### Code output (when the line with the error is deleted)

61.95198795487858 : 12.204340305758842 = 5

26 : 14 = 1

To learn more about bounded type parameters, see the official Oracle documentation: [Bounded Type Parameters](#) and [Generic Methods and Bounded Type Parameters](#). Next, let's move on to type erasure.

## Type Erasure

The virtual machine does not know anything about generic types or methods since the Java compiler uses type erasure. In other words, it performs the following actions:

- It replaces all references to the type parameters and their bounds or **Object** (if the type parameter is not bounded). The resulting byte code contains only ordinary classes, interfaces, and methods.
- It includes typecasting if it is necessary to preserve type safety.
- It creates bridge methods to preserve polymorphism in descendants of generic types.

Below is an example of the code erasure mechanism involving the static generic method *count()*, which counts the number of elements of the *array* array equal to *value*. The type parameter has no bounds; therefore, when erasing the method's code, wherever we have the **T** parameter type, the compiler will include the **Object** class.

*Click on the tabs to see the examples.*

The method's original code

The method's code after erasure

```
public static <T> int count(T[] array, T value) {
    int num = 0;
    for (T element : array) {
```

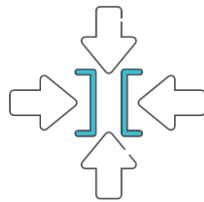
```

if (element.equals(value)) {
    num++;
}
return num;
}

```

In short, type bounds are applied with compilation only. Type erasure means that the compiled bytecode contains no information about generics whatsoever. So, there are no generics checks at runtime.

## Using Limitations



The introduction of generics allowed developers to be stricter with data type safety. However, it also limited the use of the type parameter. Note the following limitations:

*Click on the dropdown elements to learn more about the limitations.*

### ▼ Cannot instantiate generic types with primitive types

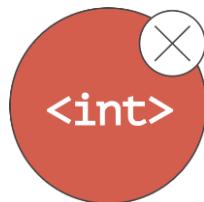
Below is an example of correct type arguments—Integer and Double.

```
KeyValue<Integer, Double> pair = new KeyValueImpl<>(10, 5.5);
```

An example of incorrect type arguments—**int** and **double**—which will result in a compilation error:

```
KeyValue<int, double> pair = new KeyValueImpl<>(10, 5.5);
```

Generics only use object data types!



### ▼ Cannot create instances of type parameters

In this example of the `testInstance()` method, an object of the T type is created. However, it is unknown what will be included instead of this type as a result of compilation.

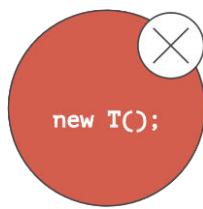
*Hover your cursor over the info icon to see an explanation of the code.*

```

public static <T> T testInstance() {
    T elem = new T();
    return elem;
}

```

When performing code erasure, the compiler replaces **T** with the **Object** type or a bound that can be of an interface type. Note that you cannot create instances of an interface.



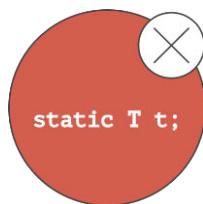
#### ✗ Cannot declare static fields whose types are type parameters

In this example of the **MobileDevice** class, a static field of the type parameter is declared. But this field is common for all class instances. This means that when we create objects with different type arguments, we need to override the type of static field for each of them. And this is impossible.

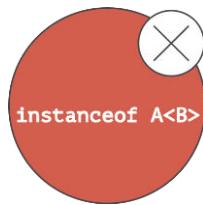
*Hover your cursor over the info icon to see an explanation of the code.*

```
public class MobileDevice <T> {
    private static T os;
    // ...
}
```

Since a static field is used conjointly by all class objects that can be parameterized with different types, there will be uncertainty about the type of the static field.



#### ✗ Cannot use instanceof with the type argument



In this example of the *test()* method, the **DynamicArray<E>** object passed via parameters is checked for whether it belongs to the **DynamicArray<Integer>** type, which is not known during runtime.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public static <E> void test(DynamicArray<E> array) {
    if (array instanceof DynamicArray<Integer>) {
        // ...
    }
}
```

Due to type erasure, during runtime, we don't know which type argument was used to create a generic class object.

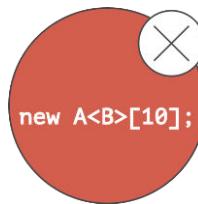
### ✗ Cannot create arrays of parameterized types

In this example, memory is allocated for an array of objects of the generic **KeyValueImpl** class with the arguments **Integer** and **Double**; however, it is not known whether all objects will be of the specified types.

*Hover your cursor over the info icon to see an explanation of the code.*

```
KeyValue<Integer, Double>[] pairs = new KeyValueImpl <Integer, Double> [5];
```

Since the compiler performs code erasure, during runtime, it will be impossible to check whether all objects included in the array were parameterized with the specified types.



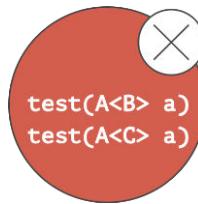
### ✗ Cannot overload methods with type parameters

In this example of the **Example** class, we declare two overloaded methods *test()*, with one parameter of the generic **KeyValue** interface and different type arguments that cannot be differentiated during runtime.

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Example {  
    public void test(KeyValue <Integer, String> a) {}  
    public void test(KeyValue <String, Double> a) {}  
}
```

After type erasure, the parameter types of the *test()* methods will match.



In this example, we declare the generic exception class **MathException** as a descendant of the **Exception** class. Note that exceptions do not support generics.

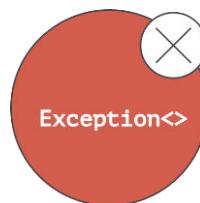
*Hover your cursor over the info icon to see an explanation of the code.*

```
public class MathException<T> extends Exception {}
```

A generic class cannot extend a **Throwable** class directly or through its descendants since exception classes are not generic.

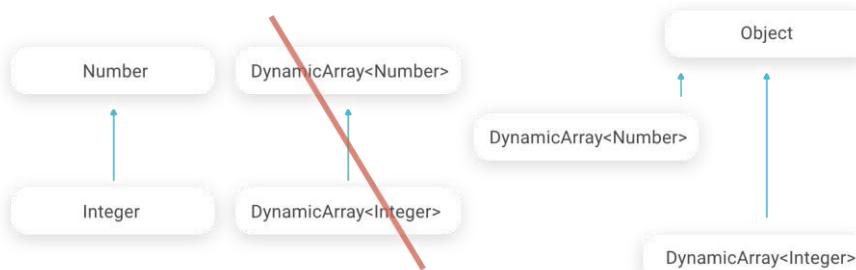
As you already know, you can assign an object of one type to a reference to an object of another type if they are compatible. For example, you can assign an **Integer** value for a **Number** variable since **Number** is one of the supertypes for **Integer**.

But does this work for generic types?



## Inheritance in Generics

If we declare a method that accepts a **DynamicArray<Number>** type as its parameter, can the **DynamicArray<Integer>** or **DynamicArray<Double>** types be passed to it? The answer is no since **DynamicArray<Integer>** and **DynamicArray<Double>** are not subtypes of **DynamicArray<Number>**, even if **Integer** and **Double** are subtypes of **Number**.



*Hover your cursor over the info icon to see an explanation of the code.*

```
DynamicArray<Double> da1 = new DynamicArray<>();
DynamicArray<Integer> da2 = new DynamicArray<>();
da1 = da2;
```

For more about inheritance and generics, see [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- The upper and lower bounds of type parameters help avoid additional checks and errors when using different data types.
- Due to type erasure, the resulting byte code contains only ordinary classes, interfaces, and methods.
- The introduction of generics led to limitations in how type parameters can be used. For example, we can no longer create an object with primitive type arguments, instantiate a type parameter, or declare a static field for type parameters.

## Check Your Knowledge!

1. You are given a generic X class whose type parameter has an upper bound in the form of the DI interface. Which declaration will allow instances of this X class to be created with type arguments that are descendants of the DI interface?

```
interface BI { }
interface DI extends BI { }
class X <T super DI> { }
class X <T implements DI> { }
```

class X <T extends DI> { }  
 correct  
 class X <T : DI> { }  
 2. What can substitute type parameters when performing type erasure?

Bounds of the type parameters

The Object class

The T type parameter

correct

Answer

Correct:

During erasure, the compiler replaces all references to the type parameters with their bounds or Object (if the type parameter is not bounded).

3. Which expressions will make the following code compilable, if it took the place of the "//CREATE\_INSTANCE" placeholder?

```
class Base<T> { }  

class Derived<T> { }  

class Test {  

  public static void main(String[] args) {  

    //CREATE_INSTANCE  

  }  

}
```

Base<Number> b = **new** Base<Number>()  
 Base<Number> b = **new** Derived<Number>()  
 Base<Number> b = **new** Derived<Integer>()  
 Derived<Number> b = **new** Derived<Integer>()  
 Base<Integer> b = **new** Derived<Integer>()  
 Derived<Integer> b = **new** Derived<Integer>()

correct

Answer

Correct:

Placing Base<Number> b = new Base<Number>() or Derived<Integer> b = new Derived<Integer>() instead of //CREATE\_INSTANCE will make the code compilable.

# The Wildcard

## Introduction

In this lesson, you will study the concept of the wildcard and see how it is used in different situations.

## The Concept of the Wildcard

A **wildcard** refers to an unknown/any type when declaring a reference. In other words, it is a way of expressing restrictions applied to a type using the terms of an unknown type. A question mark ("?") is used to denote a wildcard. The wildcard can be useful in different situations, for example:

- For a method that can be implemented with the help of functional capabilities given by the **Object** class.
- When code uses generic methods that do not depend on the type parameter.

Now let's explore the specifics of applying the wildcard.



## Using the Wildcard

If we aim to create a method that accepts any parameterized version of the generic **DynamicArray** type, we could try **DynamicArray<Object>** as the method's parameter type. However, this method would accept only **DynamicArray<Object>** instances. It would not be able to accept **DynamicArray<Integer>**, **DynamicArray<String>**, **DynamicArray<Double>**, etc. This happens because they are not subtypes of **DynamicArray<Object>**. To use a method parameter of a parameterized type, we use **DynamicArray<?>**.

In the example below, the declaration of the **Div** class is supplemented with the instance method *equalsDiv()*, which compares the results of integer division of two objects of the generic class **Div**.

*Click on the dropdown element to study the examples.*

### ▼ Incorrect method description

It is logical to use the **Div<T>** type as the method's parameter type. Then we need to declare the **Demo** class, where we will create two objects, **d\_1** and **d\_2**, with the **Div<Integer>** and **Div<Double>** types, respectively. After that, we will invoke the **equalsDiv()** method on the **d\_1** object by passing **d\_2** as a parameter. As a result, we will get a compilation error.

```
class Div<T extends Number> {
    private T x;
```

```

private T y;

Div(T a, T b) {
    x = a;
    y = b;
}

public int perform() {
    return (int) (x.doubleValue() / y.doubleValue());
}

public boolean equalsDiv(Div<T> data) {
    return (this.perform() == data.perform());
}

class Demo {
    public static void main(String[] args) {
        Div<Integer> d_1 = new Div<>(10, 20);
        Div<Double> d_2 = new Div<>(5.5, 1.1);
        boolean res = d_1.equalsDiv(d_2);
    }
}

```

#### ▼ Correct method description

You can use a bounded wildcard to loosen the restrictions on the variable and outline the group of types that can be used:

- **<? extends SuperClass>** – **upper bound**: means that any subtype of the **SuperClass** or **SuperClass** itself fits the type parameter.
- **<? super SubClass>** – **lower bound**: means that any supertype of the **SuperClass** or **SuperClass** itself fits the type parameter.

For example, **DynamicArray <Number>** is more limiting than **DynamicArray <? extends Number>**, since the former matches only datasets of the **Number** type, while the latter matches datasets of the **Number** type or any of its subclasses (**Integer**, **Double**, **Float**, etc.)

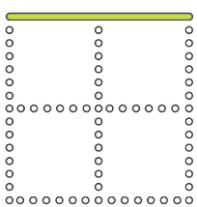
Below are some examples of using upper-bounded and lower-bounded wildcards on the previously declared generic **KeyValue** interface and generic **KeyValueImpl** class.

*Click on the tabs to study the examples.*

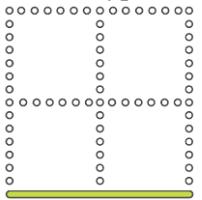
#### Upper bound

#### Lower bound

In this example, we declared a reference to the **KeyValue** interface with wildcards: The first one has an upper bound at **Number**, and the second one is unlimited. This means that when creating an object of the **KeyValueImpl** class as the first argument, we can use only descendant types of **Number** or this type itself. And for the second type argument, we can use any type of data.



Here we declared a reference to the **KeyValue** interface with wildcards: one with a lower bound at **Integer** and the other one without limitations. This means that when creating an object of the **KeyValueImpl** class, we can only use **Integer** supertypes and this type itself as the first argument and any type of data as the second argument.

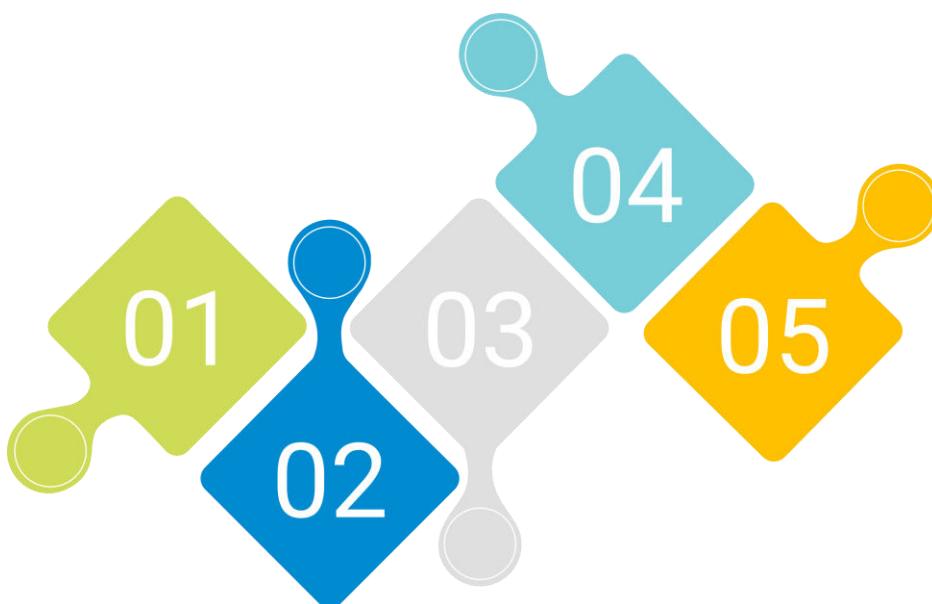


*Hover your cursor over the info icon to see an explanation of the code.*

```
KeyValue <? super Integer, ? > kv = new KeyValueImpl(12, 'k');
kv = new KeyValueImpl (99, "bbb");
kv = new KeyValueImpl (9.99, "bbb");
kv = new KeyValueImpl(int[], Long)(new int[] {1,2}, 100L);
```

Next, let's look at an example of using a bounded wildcard and an unbounded wildcard to work with multidimensional spaces.

*Click on the + signs to see the description of each stage.*



First, we need to declare the following classes:

- **Coord2** – a point in a two-dimensional space.
- **Coord3** – a point in a three-dimensional space and a descendant of the **Coord2** class.
- **Coord4** – a point in a four-dimensional space and a descendant of the **Coord3** class.
- **CoordM** – a space declared as an array of points (The parameter type of this class is upper-bounded at the **Coord2** type, i.e., the array type is limited by the **Coord2** type and its descendants.).

We need to create a **Main** class, where a *showXY()* method will be implemented to return any space in a plane: The method should accept spaces with any number of dimensions, so the type of parameter should be **CoordM<?>**.

We need to implement the *showXYZ()* method to show multidimensional spaces: The method should accept spaces with at least three coordinates, so the type of parameter should be upper-bounded at **Coord3** in order not to pass a two-dimensional space by mistake.

We need to implement the `showAll()` method to show a space with all the known dimensions: The method should accept spaces with at least four coordinates, so the type of parameter should be upper-bounded at **Coord4**.

We need to create the two-dimensional space `ss` on the basis of an array of points of the **Coord2** type and try to print it by invoking the `showXY()` and `showXYZ()` methods. In the latter case, we will have a compilation error since the method cannot accept two-dimensional spaces. We need to create the four-dimensional space `cc` based on an array of points of the **Coord4** type and try to print it by invoking the `showXY()`, `showXYZ()`, and `showAll()` methods. All methods are executed without errors.

Take a look at the following example.

*Click on the dropdown element to study the examples.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
public class Coord2 {
    int x, y;
    Coord2(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class Coord3 extends Coord2 {
    int z;
    Coord3(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
}
public class Coord4 extends Coord3 {
    int t;
    Coord4(int x, int y, int z, int t) {
        super(x, y, z);
        this.t = t;
    }
}
public class CoordM <T extends Coord2> {
    T space[];
    CoordM(T[] array) {
        space = array;
    }
}
public class Main {
    public static void showXY(CoordM<?> obj) {
        System.out.println("Array coordinate (x,y): ");
        for (int i = 0; i < obj.space.length; i++) {
            System.out.print("(" + obj.space[i].x + ", " + obj.space[i].y + ")");
        }
        System.out.println();
    }
    public static void showXYZ( CoordM<? extends Coord3> obj ) {
        System.out.println("Array coordinate (x,y,z): ");
        for (int i = 0; i < obj.space.length; i++) {
            System.out.print("(" + obj.space[i].x + ", "
                + obj.space[i].y + ", " + obj.space[i].z + ")");
        }
        System.out.println();
    }
    public static void showAll( CoordM<? extends Coord4> obj ) {
        System.out.println("Array coordinate (x,y,z,t): ");
        for (int i = 0; i < obj.space.length; i++) {
            System.out.print("(" + obj.space[i].x + ", " + obj.space[i].y
                + ", " + obj.space[i].z + ", " + obj.space[i].t + ")");
        }
    }
}
```

```

        System.out.println();
    }
    public static void main(String[] args) {
        Coord2 sp_1[] = {new Coord2(0,0), new Coord2(1,1),
                         new Coord2(5,5), new Coord2(-1,-1)};
        CoordM<Coord2> ss = new CoordM<>(sp_1);
        System.out.println("Object ss");
        showXY(ss);
        // showXYZ(ss);
        Coord4 sp_2[] = {new Coord4(1,2,3,4), new Coord4(0,0,2,6),
                         new Coord4(2,4,2,4)};
        CoordM<Coord4> cc = new CoordM<>(sp_2);
        System.out.println("\nObject cc");
        showXY(cc);
        showXYZ(cc);
        showAll(cc);
    }
}

```

## Output

Object ss  
 Array coordinate (x,y):  
 (0,0); (1,1); (5,5); (-1,-1);  
 Object cc  
 Array coordinate(x,y):  
 (1,2); (0,0); (2,4);  
 Array coordinate(x,y,z):  
 (1,2,3); (0,0,2); (2,4,2);  
 Array coordinate(x,y,z,t):  
 (1,2,3,4); (0,0,2,6); (2,4,2,4);

For more about when to use upper- and lower-bounded wildcards, see the [Oracle documentation](#).

## Conclusion

In this lesson, you have seen that:

- A wildcard refers to an unknown type when declaring a reference.
- A wildcard can loosen limitations on type parameters.

## Check Your Knowledge!

1. Which statement about the following code is true?

```

interface BI { }
interface DI extends BI { }
interface DDI extends DI { }
public class C <T> { }
public class WildCard {
    static void foo(C<? super DI> arg) {     }
    public static void main(String []args) {
        foo( new C<BI>() ); //ONE
        foo( new C<DI>() ); //TWO
        foo( new C<DDI>() ); //THREE
        foo( new C() ); //FOUR
    }
}

```

-  There will be a compilation error in line //ONE.  
 There will be a compilation error in line //TWO.  
 There will be a compilation error in line //THREE. correct

There will be a compilation error in line //FOUR.

Answer

Correct:

The parameter in the foo method allows only objects of the supertypes of the DI interface to be accepted.

2. Which expressions for creating objects to replace //Stmt#1 will make the following code compilable?

```
class Base<T> { }
class Derived<T> { }
class Test {
    public static void main(String []args) {
        // Stmt #1
    }
}
Base<? extends Number> b = new Base<Number>()
Base<? extends Number> b = new Derived<Number>()
Base<? extends Number> b = new Derived<Integer>()
Derived<? extends Number> b = new Derived<Integer>()
Base<?> b = new Derived<Integer>()
Derived<?> b = new Derived<Integer>()
```

correct

Answer

Correct:

The expressions `Base<? extends Number> b = new Base<Number>()`, `Derived<? extends Number> b = new Derived<Integer>()`, and `Derived<?> b = new Derived<Integer>()` will make the code compilable.

# Enums

## Introduction

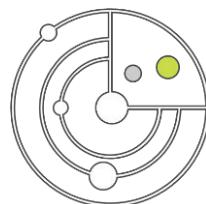
In this lesson, you will find out what enums are and how they are described in Java. You will also find out how to use the **Enum** class and become familiar with its methods.

## The Concept of Enums

An **enum** is a special data type consisting of a set of predefined constants called elements of the enum type. A variable of the enum type must be equal to one of these values (or null). In other words, an enum can be understood as a class in which all of its instances are defined from the start at the compilation stage, and in which other instances cannot be created.

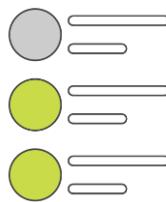
Enums are usually used if you need to predefine a fixed set of values:

*Click on the cards to learn more.*



Examples of natural types of enums:

- Planets in the Solar System
- Parts of the world
- Months
- Days of the week



Sets of data for which all the possible values are known during compilation.

For example:

- Menu items
- Command-line flags

Enums have the following **syntax**:

```
<access> enum <identifier> {<constant name 1>, <constant name 2>, ... , <constant name N>}
```

It is important to note that if an enum is not declared inside a class, it is usually declared with the **public** modifier. If an **enum** is declared inside some class, it can be declared with the **private** modifier or any other as well.

Let's take a look at some examples of how to declare and use enums.

*Click on the dropdown lists to see the examples.*

▼ Declaring an enum type

*Hover your cursor over the info icon to see an explanation of the code.*

```
public enum Season { WINTER, SPRING, SUMMER, FALL }
```

▼ Using an enum

*Hover your cursor over the info icon to see an explanation of the code.*

```
Season season = Season.SPRING;
```

By declaring an enum, you are implicitly describing a class:

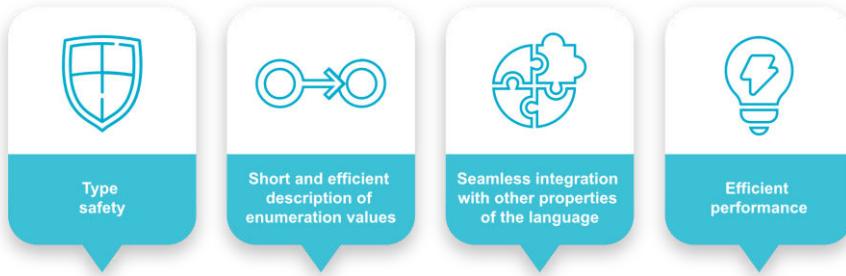
*Hover your cursor over the info icon to see an explanation of the code.*

```
final class Season {
    public static final int WINTER = 0;
    public static final int SPRING = 1;
    public static final int SUMMER = 2;
```

```
public static final int FALL = 3;
```

}

Thanks to their properties, enums offers a range of advantages.



## Type safety

There is no need to check whether a value is part of a range; the compiler knows about and checks the set of possible values.

## Compactness and efficiency

The value of an enum is represented by a semantic (logical) concept.

## Seamless integration

Since an enum is an object, all the rules that apply to objects also apply to enums.

## Performance

The type is only verified once, when first invoked.

An enum can be defined in a separate file but can also be part of any class. However, the description of an enum does not have to be located in a class. In other words, we can say that an additional class is defined, but only the keyword "class" is replaced with "enum".

## The Enum Class and Its Methods

All enum types are inherited from the abstract class **java.lang.Enum**:

- The compiler does not allow enums to be inherited explicitly from **java.lang.Enum**.
- This can be checked through the reflection mechanism.

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
System.out.println(Season.class.getSuperclass());
```

It is important to note that the abstract class **java.lang.Enum** was first introduced in the fifth Java version.

The abstract class **java.lang.Enum** contains a number of methods, and most of them are described with the **final** modifier. Consequently, when describing an enum, these methods can be used but not overridden. The only method that can be overridden is *toString()*.

Returns the name of the current constant exactly as declared in its enum declaration  
**ordinal()**  
 Returns the index (position) of the current constant  
**equals()**  
 Returns true if two values of enum of the same type are equal  
**hashCode()**  
 Returns a hash code (identifier) of the current constant  
**toString()**  
 Returns the description of the current constant as a string  
**compareTo()**  
 Compares two enumerations of the same type in order  
**values()**  
 Returns an array of all enum values in the order they are declared  
**valueOf()**  
 Returns the enum constant that corresponds to the specified string

Let's explore some of these methods in more detail.

## The Methods **name()**, **ordinal()**, **values()**, **valueOf()**, and **compareTo()**



Each enum value has a name and index. They can be retrieved using the **name()** and **ordinal()** methods. For example:

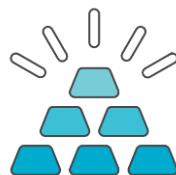
*Hover your cursor over the info icon to see an explanation of the code.*

```
System.out.println(Season.WINTER.name());
System.out.println(Season.WINTER.ordinal());
```

It is also worth noting three methods that are important for enums — **values()**, **valueOf()**, and **compareTo()**.

*Click on the dropdown lists to learn more about the methods.*

### ▼ **values() and valueOf()**



The **values()** method returns an array containing all of the values of the enum in the order they are declared.

Implementation of the **values()** method in the **java.lang.Enum** class:

```
public static E[] values();
```

There is no complete implementation of this method since it is implicit and is added at the compilation stage. The enum values are returned by the method in the order they are declared. For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
Season[] seasons = Season.values();
System.out.println(Arrays.toString(seasons));
```

The `valueOf()` method returns an enum value according to its string representation.

Implementing the `valueOf()` method in the **java.lang.Enum** class:

```
public static E valueOf(String name);
```

As with `values()`, this method is also static; therefore, there is no complete implementation of this method in the **Enum** class. The `valueOf()` method simply returns an enum value according to its string representation. For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
System.out.println(Season.valueOf("WINTER").ordinal());
```

If the value being searched for is not found in the enum, the exception **IllegalArgumentException** will be thrown. For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
Season.valueOf("Summer");
```

#### ▼ compareTo()



It is important to note that **java.lang.Enum** implements the **Comparable** interface. The main goal is to be able to compare enum values with each other, for example, when sorting. The comparison is done by the *index* of the enum.

Note the order of the values in the **Season** enum:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

It is important to analyze and understand the results of comparing the enum values. To do this, we have to use the `compareTo()` method, which returns:

- A negative integer if the constant value comes before the parameter passed to the method (also a constant value)
- 0 if the constant value is equal to the parameter passed to the method (i.e., this == other)
- In all other cases, a positive integer if the constant value comes after the parameter passed to the method

*Hover your cursor over the info icon to see an explanation of the code.*

```
System.out.println(Season.SPRING.compareTo(Season.WINTER));
System.out.println(Season.SPRING.compareTo(Season.SPRING));
System.out.println(Season.SPRING.compareTo(Season.SUMMER));
System.out.println(Season.WINTER.compareTo(Season.SUMMER));
```



Constant values of the enum can only be compared within the same enum type.

Thus, you have explored the methods *name()*, *original()*, *values()*, *valueOf()*, and *compareTo()*. Now let's move to the *equals* method.

## The Method *equals()*

The *equals()* method is responsible for comparing enum values and does so based on references. References are used when comparing because the enum values themselves are unique constants: Only one instance of the WINTER season is possible, and only one SPRING season, one SUMMER season, and one FALL season. This means that there will always be only one reference to an instance. Therefore, it is possible to compare using the `==` operator.



Implementing the *equals()* method in the **java.lang.Enum** class:

```
public final boolean equals(Object other) {
    return this == other;
}
```

Take a look at the following examples.

*Click on the tabs to study the examples.*

**Example 1**

**Example 2**

*Hover your cursor over the info icon to see an explanation of the code.*

```
boolean isEqualToItself = Season.WINTER.equals(Season.WINTER);
boolean isEqualToDifferentSeason = Season.WINTER.equals(Season.SUMMER);
System.out.println(isEqualToItself);
System.out.println(isEqualToDifferentSeason);
```

Therefore, you have found out that the *equals()* method is responsible for comparing enum values and does so based on references. Let's get into other methods.

*Hover your mouse cursor over the info icon to see an explanation of the code.*

```
Season season = Season.WINTER;
```

```
System.out.println(season == Season.WINTER);
System.out.println(season == Season.SUMMER);
```

Therefore, you have found out that the *equals()* method is responsible for comparing enum values and does so based on references. Let's get into other methods.

## The Methods **hashCode()**, **toString()**, and **clone()**

Let's analyze how the *hashCode()*, *toString()*, and *clone()* methods are overridden in the **java.lang.Enum** class.

**hashCode()**

**toString()**

**clone()**

The *hashCode()* method uses a standard implementation from the **java.lang.Object** class. Here is an implementation of the *hashCode()* method in the **java.lang.Enum** class:

```
public final int hashCode() {
    return super.hashCode();
}
```

For example:

*Hover your cursor over the info icon to see an explanation of the code.*

```
int hashOfWinter = Season.WINTER.hashCode();
int hashOfSummer = Season.SUMMER.hashCode();
System.out.println(hashOfWinter);
System.out.println(hashOfSummer);
```

Invoking the *toString()* method gives us the name of an enum element. Therefore, the constant value **WINTER** is returned by the *toString()* and *name()* methods.

Here is an implementation of the *toString()* method in the **java.lang.Enum** class:

```
public String toString() {
    return name();
}
```

*Hover your cursor over the info icon to see an explanation of the code.*

```
String winter = Season.WINTER.toString();
System.out.println(winter);
```

Invoking the *toString()* method gives us the name of an enum element. Therefore, the constant value **WINTER** is returned by the *toString()* and *name()* methods.

Here is an implementation of the *toString()* method in the **java.lang.Enum** class:

```
public String toString() {
    return name();
}
```

*Hover your cursor over the info icon to see an explanation of the code.*

```
String winter = Season.WINTER.toString();
System.out.println(winter);
```

Thus, you have studied how the **hashCode()**, **toString()**, and **clone()** methods are overridden in the **java.lang.Enum** class. Let's turn to custom fields and methods.

## Custom Fields and Methods

A description of an enum type can be supplemented with constructors, variables, and methods. Methods are always placed after the list of constants, which should end with a semicolon.

Fields and methods in enums can be declared with any access. It is important to note that enum constructors should have either a **private** or a **package-private** visibility scope. An attempt to specify any other access modifier will lead to a compilation error.

 An enum constructor cannot be invoked directly!

A constructor is invoked automatically with arguments that are defined as enumeration fields after the constant values.

*Click on the dropdown list to see the example.*

*Hover your cursor over the info icon to see an explanation of the code.*

```
enum DocumentStatus {
    NEW(31),
    DRAFT(23),
    PUBLISHED(52),
    ARCHIVED(77);

private int statusCode;

private DocumentStatus(int statusCode) {
    this.statusCode = statusCode;
}

public int getStatusCode() {
    return statusCode;
}

public static void main(String[] arg) {
    for (DocumentStatus docSt : DocumentStatus.values()) {
        System.out.println("Name = " + docSt.name() +
            ", statusCode is = " + docSt.getStatusCode());
    }
}
```

It is also necessary to mention that the **toString()** method can be overridden.

*Click on the dropdown lists to see the examples.*

```
enum Season {
    WINTER, SPRING, SUMMER, FALL;
    public String toString(){
        return "Season: " + this.name();
    }
}
```

```
public static void main(String[] args) {
    System.out.println(Season.SPRING);
    System.out.println(Season.SUMMER);
    System.out.println(Season.WINTER);
    System.out.println(Season.FALL);
}
```

**Output to console:**

Season: SPRING  
 Season: SUMMER  
 Season: WINTER  
 Season: FALL

For each constant, the *toString()* method can be overridden separately to represent the description of its value:

```
enum Season {
    WINTER {
        public String toString() { return "Winter - cold season"; }
    },
    SPRING {
        public String toString() { return "Spring - cold-warm season"; }
    },
    SUMMER {
        public String toString() { return "Summer - hot season"; }
    },
    FALL {
        public String toString() { return "Fall - cool season"; }
    }
}

public static void main(String[] arg) {
    System.out.println(Season.SPRING);
    System.out.println(Season.SUMMER);
    System.out.println(Season.WINTER);
    System.out.println(Season.FALL);
}
```

**Output to console:**

Spring - cold-warm season  
 Summer - hot season  
 Winter - cold season  
 Fall - cool season

Next, let's will explore how to describe an enum.

## The Specifics of Describing an enum Type

There are several important things to keep in mind:

- Multiple constructors can be defined in an enum, and the necessary one is chosen using the parameters.
- An enum constructor should not contain the keyword **super**; otherwise, there will be a compilation error.
- It is impossible to call static enum fields (if they are not **public static final**) from constructors, initialization blocks, or initialization expressions.
- An enumeration can contain abstract methods. In this case, each constant must implement all abstract methods using anonymous classes.

**Look at the following examples.**

**Click on the dropdown lists to see the examples.**

```

public enum Operation {
    PLUS {
        double eval(double x, double y) {
            return x + y;
        }
    },
    MINUS {
        double eval(double x, double y) {
            return x - y;
        }
    },
    TIMES {
        double eval(double x, double y) {
            return x * y;
        }
    },
    DIVIDED_BY {
        double eval(double x, double y) {
            return x / y;
        }
    };
}

abstract double eval(double x, double y);

public static void main(String args[]) {
    double x = 2.0;
    double y = 4.0;
    for (Operation op : Operation.values())
        System.out.println(x + " " + op + " " + y
            + " = " + op.eval(x, y));
}
}

```

### Output to console:

2.0 PLUS 4.0 = 6.0  
 2.0 MINUS 4.0 = -2.0  
 2.0 TIMES 4.0 = 8.0  
 2.0 DIVIDED\_BY 4.0 = 0.5

```

public enum BinaryOperation {
    PLUS ("+") {
        public int calculate(int a, int b) {
            return a + b;
        }
    },
    MINUS ("-") {
        public int calculate(int a, int b) {
            return a - b;
        }
    },
    DIVISION ("/") {
        public int calculate(int a, int b){
            return a / b;
        }
    },
    MULT ("*") {
        public int calculate(int a, int b){
            return a * b;
        }
    };
}

```

```

private String operation;

public abstract int calculate(int a, int b);

BinaryOperation(String operation) {

```

```

        this.operation = operation;
    }

public static void main(String args[]) {
    int x = 10;
    int y = 2;
    for (BinaryOperation op : BinaryOperation.values())
        System.out.println(x + " " + op.operation + " " + y
            + " = " + op.calculate(x, y));
}
}

```

**Output to console:**

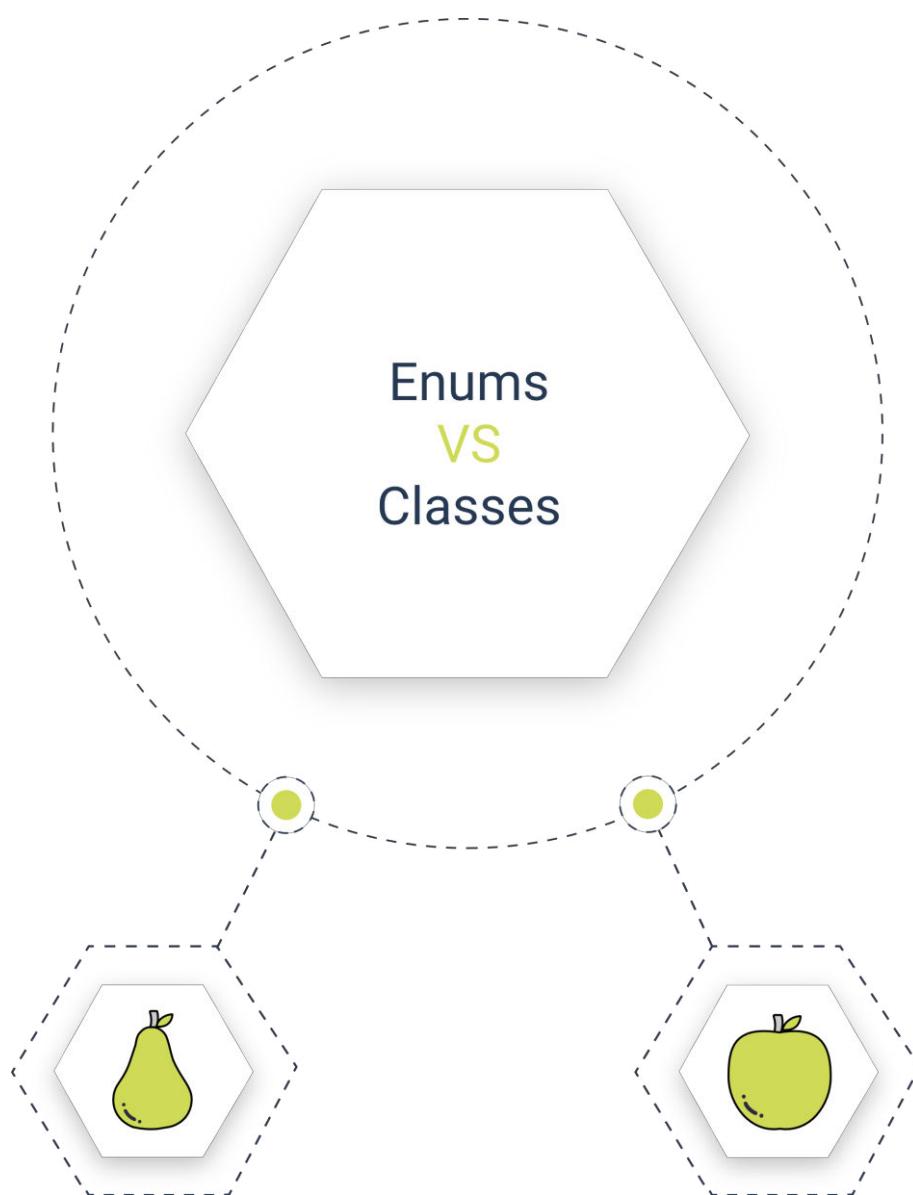
10 + 2 = 12  
 10 - 2 = 8  
 10 / 2 = 5  
 10 \* 2 = 20

Thus, you have found out that you need to keep in mind certain specifics when describing an enum type. Now, let's go through the differences between enums and ordinary classes.

## The Difference Between Enums and Ordinary Classes

It is important to see the difference between enums and ordinary classes.

*Click on the + signs to see the explanation.*



Enums, just like classes, can have fields (attributes) and methods. The difference is that enum constants are public, static, and final, meaning they are immutable, or cannot be redefined.

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

To summarize:

- By default, an enum type is an inheritor of the abstract **java.lang.Enum** class and therefore cannot extend other classes.
- By default, an enum implements the **java.lang.Comparable** interface and thus can be ordered.
- By default, enum values are variables of the **public static final** type.
- Some methods are defined as final, i.e., they cannot be overridden: (*name()*, *ordinal()*, *equals()*, *hashCode()*, *compareTo()*, and *clone()*).
- The *toString()* method returns the constant name and can be overridden.
- An enum type can have its own fields and methods, including abstract ones, but each constant must implement all abstract methods using anonymous classes.
- An enum cannot be declared as a generic type.

## Conclusion

In this lesson, you:

- Discovered how to describe and apply an enum type
- Explored the methods of the **Enum** class: *name()*, *ordinal()*, *equals()*, *hashCode()*, *toString()*, and *clone()*
- Found out how to describe custom fields and methods in enums

## Check Your Knowledge!

1. Which of the following statements about enums are true?

An enum cannot have **public** methods or fields.

An enum can have **public** methods or fields.

An enum can declare a constructor with **private** access.

An enum cannot declare a constructor with **private** access.

correct

Answer

Correct:

An enum can have public methods or fields and can declare a constructor with private access.

2. Which of the following statements is true based on the following enum description?

```
public enum PrinterType {
    private int pagePrintCapacity;
    DOTMATRIX(5),
    INKJET(10),
    LASER(50);
    private PrinterType(int pagePrintCapacity) {
        this.pagePrintCapacity = pagePrintCapacity;
    }
    public int getPrintPageCapacity() {
        return pagePrintCapacity;
    }
}
```

The enum description is correct and will be compiled without any warnings or errors.

The enum description is incorrect and will lead to a compilation error. correct

The enum description will lead to a runtime error.

Answer

Correct:

The enum description is incorrect and will lead to a compilation error since constants must be declared first, and then fields and methods.

3. What is the result of running the following code?

```
enum Numbers {ONE, TWO, THREE, FOUR, FIVE}
public class Quest9 {
    public static void main(String[] args) {
        Numbers n1 = Numbers.ONE;
        Numbers n2 = Numbers.ONE; //Line1
        if (n1 == n2) {
            System.out.print ("true");
        } else {
            System.out.print ("false");
        }
        System.out.println( Numbers.FIVE.ordinal() ); //Line2
    }
}
```

false4

true4 correct

false5

true5

Compilation error in //Line 1

page

A compilation error in //Line 2

Answer

Correct:

Objects of enum elements are created in a single copy. Therefore, comparing them here gives a true value. The number 4 indicates the index of the enum element when declaring it, and the numbering starts from 0.

## What is a Wrapper Class?

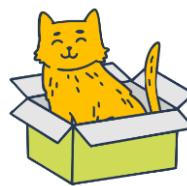
### Introduction

In this lesson, you will find out what wrapper classes are, why they were introduced to Java, and what they are used for. You will also explore the hierarchy of wrapper classes and their properties.

### The Concept of Wrapper Classes

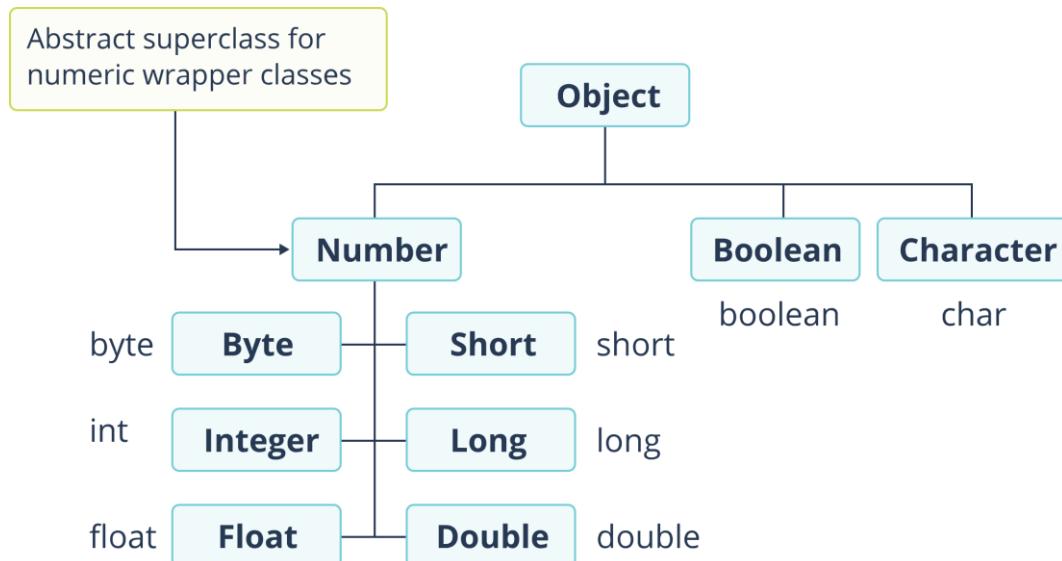
Earlier in this course, you learned about primitive data types and studied the operations that can be performed with this data. You already know that Java is an object-oriented programming language—everything is represented as objects. In this respect, the existence of primitive types violates the object-oriented nature of Java. Therefore, to represent data of primitive types as objects, wrapper classes were introduced in the **java.lang** package beginning with Java version 1.5.

The main reason wrapper classes were created was to gain all the advantages of the Java language when working with object data types—for example, to pass data of different primitive types in one array of the **Object[]** type.



Primitive types exist to enhance performance because objects are "expensive"—it takes much longer to execute operations with reference types.

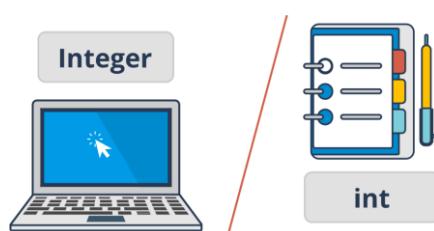
Each primitive type has its own wrapper class:



For example, the **Integer** class and the primitive type **int** can be compared to a laptop and a notepad. A computer is much larger than a notepad, so chances are you won't carry around a heavy laptop all day just to take a few notes here and there.

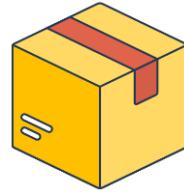
Wrapper classes have a number of important properties, which are described below:

*Scroll down to see the properties.*



Wrapper classes have a number of important properties, which are described below:

*Scroll down to see the properties.*



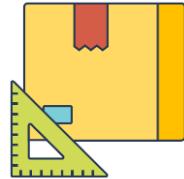
Wrapper classes override all methods of the **Object** class for their purposes—for example: `toString()`, `equals()`, and `hashCode()`.

### Implementing the Comparable interface



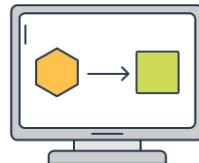
All wrapper classes for primitive types implement the **Comparable** interface (the `compareTo()` method), meaning that their objects can be compared with each other.

### Range of values



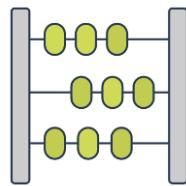
All numeric wrapper classes contain the constants `MIN_VALUE` and `MAX_VALUE`, which assure the upper and lower bounds of data types.

### Type casting methods



All numeric wrapper classes contain methods for casting an object to a primitive value—for example: `intValue()`, `longValue()`, and `floatValue()`.

### Numerical systems



All numeric wrapper classes contain methods used to represent numbers in different numeral systems.

To learn more about wrapper classes, see [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- A wrapper class is a class that encapsulates a value of a primitive type and provides basic operations for working with an object of that class.
- Wrapper classes were introduced to the **java.lang** package to represent primitive data types as objects.

You also explored the various properties of wrapper classes.

## Check Your Knowledge!

1. The wrapper classes for primitive data types include:

Byte

byte

Double

Long

Int

correct

Answer

Correct:

The wrapper classes for primitive data types are Byte, Short, Integer, Long, Float, and Double.

2. Which of the following are properties of wrapper classes?

They perform autoboxing when a method takes a primitive value as an argument but was expecting an object.

They include immutable variables such as MIN\_VALUE and MAX\_VALUE, which provide upper and lower bounds for the data type.

They include methods for converting numeric values to and from strings to represent data in different number systems (decimals, octals, hexadecimals, binaries).

correct

Answer

Correct:

All the properties listed are arguments for using an object of the corresponding wrapper class.

To learn more about wrapper classes, see [Oracle's official documentation](#).

## Conclusion

In this lesson, you have seen that:

- A wrapper class is a class that encapsulates a value of a primitive type and provides basic operations for working with an object of that class.
- Wrapper classes were introduced to the **java.lang** package to represent primitive data types as objects.

36 You also explored the various properties of wrapper classes.

## Creating Objects of Wrapper Classes

## Introduction

This lesson will present an overview of the different approaches to creating objects of wrapper classes.

## Creating Objects of Wrapper Classes

There are different approaches to creating a wrapper class or to wrapping the value of a primitive type into an object. These include using:

- A constructor with the relevant primitive type
- A constructor with a string representing the value of the primitive type
- The static method `valueOf()` with the primitive type
- The static method `valueOf()` with a string representing the value of the primitive type

Let's take a closer look at these approaches.

## Using a Constructor with the Primitive Type

The easiest way to create an object of a wrapper class is to use a constructor of the wrapper class with its primitive type.

```
Integer intObject1 = new Integer(100);
Double doubleObject1 = new Double(0.01);
```

## Using a Constructor with a String Representing the Value of the Primitive Type

To create an object of a wrapper class, you can use a constructor that accepts text as a parameter containing a representation of the value of the primitive type.

```
Integer intObject2 = new Integer("100");
Double doubleObject2 = new Double("0.01");
```

The **Character** class is an exception since it has no such constructor.

## Using the Static Method `valueOf()` with the Primitive Type

For all wrapper classes, the static method `valueOf()` can accept a value of the primitive type that corresponds to the wrapper class.

```
Integer intObject3 = Integer.valueOf(11);
Double doubleObject3 = Double.valueOf(11.1);
```

## Using the Static Method `valueOf()` with a String Representing the Value of the Primitive Type

For all wrapper classes, the static method `valueOf()` can accept text containing a representation of the value of the primitive type.

```
Integer intObject4 = Integer.valueOf("11");
Double doubleObject4 = Double.valueOf("11.1");
```

The **Character** class is an exception since it has no such method.

Now that you have explored the different ways of creating wrapper classes, it's time to explore how to create objects of the **Boolean** class.

## Creating Objects of the Boolean Class

When creating an object using a constructor or the static method `valueOf()` with a string, we need to keep a few things in mind:

- The string representing the **true** value can be specified using any case.
- All strings with a value other than **true** will lead to the creation of an object with the value **false**.

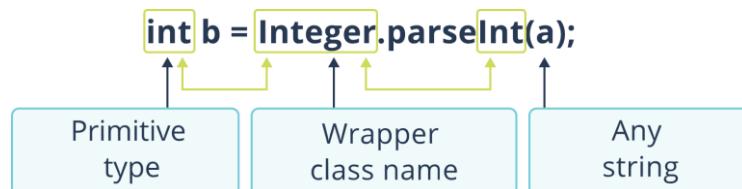
```
Boolean b = new Boolean("true");
Boolean b1 = new Boolean("false");
Boolean b2 = Boolean.valueOf("no true");
Boolean b3 = Boolean.TRUE;
Boolean b4 = new Boolean("True");
System.out.println(b + ", " + b1 + ", " + b2 + ", " + b3 + ", " + b4);
```

### Output

true, false, false, true, true

## The Static Method `parseType(String)`

You can get a primitive value for numeric wrapper classes from a string of characters using the static method `parseType(String)`. For example, for the **Integer** class:



Let's consider an example of the **Test** class, where we will parse text and will get a value of a primitive data type:

```
public class Test {
    public static void main(String[] args) {
        String str = "11";
        int number = Integer.parseInt(str);
        System.out.println(number);
    }
}
```

### Output

11

You have looked at various ways of creating wrapper classes. Explore the coding story below to consolidate and systematize your knowledge.

## Conclusion

In this lesson, you studied different approaches to creating wrapper class objects by using:

- A constructor with the relevant primitive type or a string representing the value of a primitive type.
- The static method `valueOf()` with a primitive type or a string representing the value of a primitive type.

You also studied the specifics of creating objects of the **Boolean** class. Finally, you explored how to get a primitive value for numeric wrapper classes from a string of characters using the static method `parseType(String value)`.

### Check Your Knowledge!

Q1. Which of the following lines are correct for the **Integer** type and executed without errors?

```
System.out.println(Integer.parseInt("-123"));
System.out.println(Integer.parseInt("123"));
System.out.println(Integer.parseInt("+123"));
System.out.println(Integer.parseInt("123_45"));
System.out.println(Integer.parseInt("12ABCD"));
correct
```

Answer

Correct:

In Java, to correctly convert a number represented as a string into a number itself, the string should look like a number to a normal person.

2. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
        Character c1 = new Character(9); //line1
        Character c2 = new Character("a"); //line2
        System.out.print(c1);
        System.out.print(c2);
    }
}
```

9a

A compilation error in //line1

A compilation error in //line2

A compilation error in //line1 and //line2 correct

Answer

Correct:

The Character class does not describe constructors that take numeric values or values of the String type.

3. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
        Integer i1 = new Integer(76); //line1
        Integer i2 = new Integer("24"); //line2
        System.out.print(i1 + i2);
    }
}
```

100 correct

A compilation error in //line1

A compilation error in //line2

7624

Answer

Correct:

You can pass a string or a number into constructors of the Integer class.

4. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
        Integer i2 = Integer.valueOf("0010", 2);
        System.out.print(i2);
    }
}
```

102

2 correct

12

A compilation error

Answer

Correct:

The string contains a binary representation of the number 2. The valueOf method as a second parameter contains the basis for the numeral system. The conversion result is 2.

5. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
```

```

Float f1 = new Float(3.1f);      //line1
Float f2 = new Float("3.1f");    //line2
Float f3 = Float.valueOf("3.1f"); //line3
System.out.print(f1 + "" + f2 + f3);
}
}

```

3.13.13.1 correct

3.16.2

A compilation error in //line1

A compilation error in //line2

A compilation error in //line3

Answer

Correct:

All three variables will be initialized correctly with the value 3.1. When printing to console, the values will be concatenated into one string.

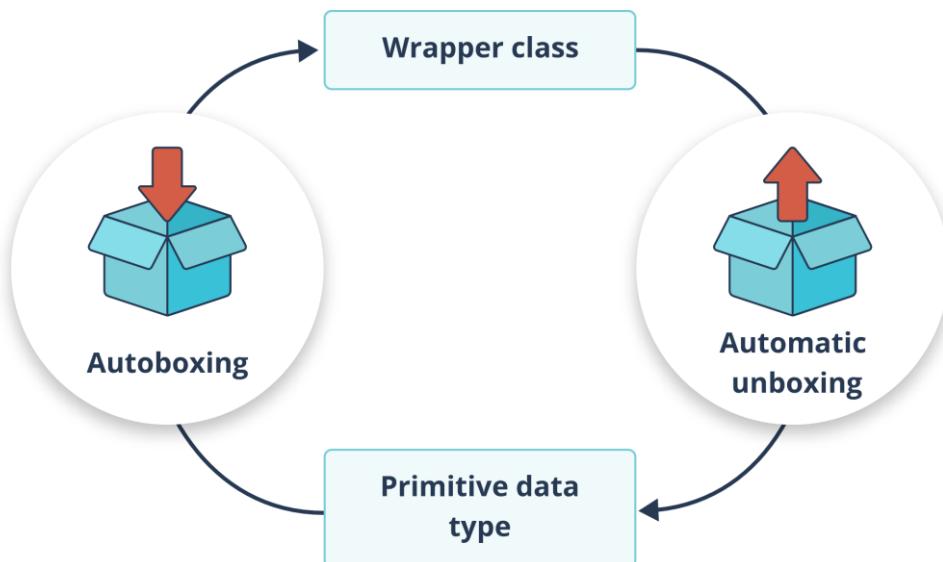
## The Mechanisms of Autoboxing and Unboxing

### Introduction

In this lesson, you will continue exploring wrapper classes. You will become familiar with the mechanism of autoboxing and unboxing, the specifics of caching integer type objects, and the concept of immutability of wrapper classes.

**Autoboxing/unboxing** refers to an automatic transformation (casting) performed by Java between primitive types and their corresponding wrapper class objects. These mechanisms were introduced beginning with Java 5 version to avoid the bulky source code associated with explicit wrapping (packing) of a primitive value into an object and extracting a primitive value from the object.

Click on the + signs in the image to learn more.



## Autoboxing

In the process of autoboxing, the static method `valueOf()` or a constructor is called implicitly. This depends on the value.

## Unboxing

During unboxing, the instance method `typeValue()` is called implicitly, where `type` is the respective primitive data type—for example, `longValue()`—which converts an object of the `Long` type into a value of the primitive `long` type.

The Java compiler applies autoboxing when the primitive value is:

- Passed as an argument of a method that expects an object of the corresponding wrapper class
- Assigned to a variable of the corresponding wrapper class

For example:

similarly

<pre>Integer intObject = 200; int a = intObject; int b = intObject - 3; Integer[] intArray = {1, 2, 3};</pre>	<pre>Integer intObject = Integer.valueOf(200); int a = intObject.intValue(); int b = intObject.intValue() - 3; Integer[] intArray = {Integer.valueOf(1),                     Integer.valueOf(2), Integer.valueOf(3)};</pre>
---	---

For more about autoboxing, see [Oracle's official documentation](#).

For quicker access to the most common objects of wrapper classes, the Java virtual machine caches them. Such objects include **integer type objects** ranging from -128 to 127.



Beginning with Java 6, one can change the upper range boundary.

Let's explore how wrapper class objects are cached in more detail.

*Click on the active elements to learn more.*



## Way of creation

- The cache is initialized the first time it is used. The first time a value from the above range is used, it is boxed into an object by invoking the `valueOf()`.
- Every time an object is initialized with a value from the range specified, no new object is created, but a reference is returned to the existing one.

## Storing in memory

- Memory is allocated for the object in a pool of integer objects. This is a special area of the "heap" called a cache-memory.

## Used types

- Caching is used for the primitive types **byte**, **short**, **int**, **char**, and **boolean**.

Take a look at a few examples of creating objects of the **Integer** type with values within the range of -128–127 and outside of this range.

*Click on the tabs to explore and compare the examples.*

**Example 1—A value within the range of -128–127**

**Example 2—A value outside of the range of -128–127**

Let's create

two objects of the **Integer** type and initialize them with a value of 10. Then we will compare the references to these objects and compare the objects themselves using the `equals()` method. As a result, we will see that both the objects and the references are equal. This means that one object exists with two references to it.

```
Integer ten = 10;
Integer copyOfTen = 10;
System.out.println( ten == copyOfTen );
System.out.println( ten.equals(copyOfTen) );
```

## Output

true  
true

Let's create two objects of the **Integer** type and initialize them with a value of 1000. Then we will compare the references to these objects and compare the objects themselves using the *equals()* method. As a result, we will see that the objects are equal, while the references are different. This means the objects are stored in the common "heap" memory and the **new** operator and a constructor were used to store them.

```
Integer thousand = 1000;
Integer copyOfThousand = 1000;
System.out.println(thousand == copyOfThousand);
System.out.println(thousand.equals(copyOfThousand));
```

## Output

false  
true

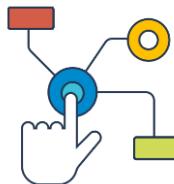
Now that you have explored how objects of wrapper classes are cached and seen what a pool of integer objects is, it's time to move on to the concept of immutability of wrapper class objects.

## Immutability of Wrapper Class Objects

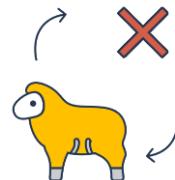
All wrapper class objects are **immutable**. This means that after a wrapper class object is created, its state cannot be changed. Immutable objects simplify a program immensely. Why?

*Click on the cards to see each reason.*

- They are easy to build, test, and use.
- They are checked only once when they are created.



- No copy constructor is required.
- No cloning implementation is required.

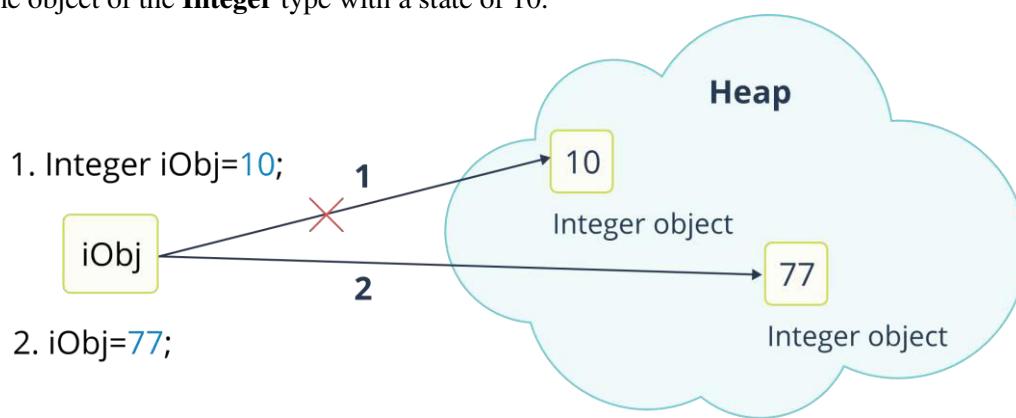


- They are automatically thread-safe and have no synchronization issues.
- They are convenient as keys in the **Map** and **Set** collections.



Let's look at an example that illustrates the immutability of wrapper class objects. We will declare an *iObj* variable of the **Integer** type and initialize it with a value of 10. This means that a storage place will be allocated in the memory of an object of

the **Integer** type with a state of 10. The *iObj* variable will be assigned the memory address of this object. Then we will assign a different value to this variable—for example, 77. This means that a new object of the **Integer** type will be created in the memory with a state of 77. The address for where the object is stored in the memory will be assigned to the *iObj* variable. We will lose access to the object of the **Integer** type with a state of 10.



So far, you have seen that wrapper classes are immutable and how this simplifies a program. Next, you'll learn which wrapper classes are used for high-precision arithmetic.

## Conclusion

In this lesson, you have seen that:

- Autoboxing/unboxing is an automatic transformation performed by Java between primitive types and their corresponding wrapper class objects. This is done to avoid the bulky source code associated with the explicit wrapping of a primitive value into an object and extracting a primitive value from the object.
- To quickly access common wrapper class objects (objects of integer types ranging from -128 to 127), the Java virtual machine caches them.
- All wrapper classes objects are immutable.

## Check Your Knowledge!

1. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
        Integer y = 567;
        Integer x = y;
        System.out.print((x == y) + " ");
        x++;
        y++;
        System.out.print(x == y);
    }
}
```

true true  
true false correct  
false false  
false true  
A compilation error

Answer

Correct:

The first two references being compared refer to the same object. The second comparison compares different objects since the `++` operation creates a new object.

2. What is the result of running the following code?

```
public class Main {
    public static void main(String[] args) {
        Integer i1 = 1234;
        Integer i2 = 1234;
        if (i1 != i2) System.out.println("different objects");
        if (i1.equals(i2)) System.out.println("meaningfully equal");
    }
}
```

```

    }
}
different objects
meaningfully equal
correct
different objects
meaningfully equal
nothing will be printed

```

Answer

Correct:

The two references are not equal. The equals() method compares objects in terms of their content, and the content of the objects is equivalent.

3. What is the result of running the following code?

```

public class Main {
    static Integer x;
    public static void main(String[] args) {
        doStuff(x);
    }
    static void doStuff(int z) {
        int z2 = 5;
        System.out.println(z2 + z);
    }
}

```

5

0

A compilation error

A runtime error correct

Answer

Correct:

"java.lang.Integer.intValue()" cannot be invoked because "x" is null.

4. What is the result of running the following code?

```

class UseBoxing {
    public static void main(String[] args) {
        UseBoxing u = new UseBoxing();
        u.go(5);
    }
    boolean go(Integer i) {
        Boolean ifSo = true;
        Short s = 300;
        if (ifSo) {
            System.out.println(++s);
        }
        return !ifSo;
    }
}

```

301 correct

300

A compilation error

A runtime error

Answer

Correct:

The number 300 will be extracted from the wrapper and increased by the prefix increment operation by 1.

5. What is the result of running the following code?

```

public class Main {
    public static void main(String[] args) {
        Integer i = new Integer(5);
        Integer b = i;
        i = 5;
        System.out.println(i == b);
    }
}

```

true  
false correct

A compilation error

A runtime error

Answer

Correct:

The operation `i=5;` initializes the reference with a new object; therefore, the operation of reference comparison will give a result of false.

## Wrapper Classes for High-precision Arithmetic

### Introduction

Now let's see which wrapper classes are used for high-precision arithmetic.

### Wrapper Classes for High-Precision Arithmetic

There are two classes for working with high-precision arithmetic—**`java.math.BigInteger`** and **`java.math.BigDecimal`**. These classes support integers and fixed-point numbers of arbitrary length.

Performing calculations for the **`double`** and **`float`** types raises an issue with precision. The concept of "computer zero" exists for these types, which means that when calculating after the fifteenth digit for **`double`** and after the sixth digit for **`float`**, "extra" significant figures appear. To eliminate such side effects, we use the **`BigDecimal`** class:

```
float res = 0.4f - 0.3f;
BigDecimal big1 = new BigDecimal("0.4");
BigDecimal big2 = new BigDecimal("0.3");
BigDecimal bigRes = big1.subtract(big2, MathContext.DECIMAL32);
System.out.println(res);
System.out.println(bigRes);
```

**Output**

```
0.099999994
0.1
```

Constants of the **MathContext** class determine the number of decimal places when rounding. For addition, multiplication, and division, we use the *add()*, *multiply()*, and *divide()* methods, which are represented in several overloaded versions. The issue of "computer zero" also comes up in comparison operations—specifically, when executing the following operator:

```
boolean res1 = 1.00000001f == 1.00000002f;
```

as is the case here

```
boolean res2 = 1 == 1f / 3 * 3;
```

**Output**

```
true
```

**Conclusion**

As you have seen, we should use the **java.math.BigInteger** and **java.math.BigDecimal** wrapper classes when doing high-precision arithmetic.

**Check Your Knowledge!**

When calculating for **double** types, after which digit do "extra" significant figures appear?

- 6
- 7
- 14
- 15 correct
- 16

Answer

Correct:

Performing calculations for the double and float types raises an issue with precision. The concept of "computer zero" exists for these types, which means that when calculating after the fifteenth digit for double and after the sixth digit for float, "extra" significant figures appear.

**Class Optional****Introduction**

In this lesson, you will find out how to create **Optional** type objects.

**Class Optional**

The **java.util.Optional** class is a container for any object type. This class is used to identify the problems of returning a **null** value using the method to perform search, perform processing, or generate any object in a situation when this action gave no acceptable result.

When a developer uses a method returning some reference to a value, this means that the developer is planning to use the returned object. If the method returns **null**, then using it will certainly throw a **NullPointerException**. The developer will have to add a special check for **null** to the code to protect it from the exception. The **Optional** class does not solve the problem of returning **null**, but using this class as a returned value clearly identifies this problem.

An object of this type can be created by static factory methods:

#### **Optional.of()**

Encapsulates a non-null value: if the value is **null**, then the following exception will be thrown: **NullPointerException**.

#### **Optional.ofNullable()**

Encapsulates a non-null value; otherwise, there will be an empty object.

#### **Optional.empty()**

Encapsulates an empty object.

Let's compare the specifics of working with the returned value with and without the class **Optional**.

## Conclusion

In this lesson, you have found out that:

- The **Optional** class is used to identify the problems of returning a **null** value using the method to perform search, processing, or generate any object in a situation when this action gave no acceptable result.
- An object of the **Optional** type can be created using a static factory method: **Optional.of()**, **Optional.ofNullable()**, or **Optional.empty()**.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. Select the correct description of the method **Optional.ofNullable()**.

Encapsulates an empty object.

Encapsulates a non-null value; otherwise, there will be an empty object. correct

Encapsulates a non-null value: if the value is **null**, then the following exception will be thrown: **NullPointerException**.

Answer

Correct:

The method **Optional.ofNullable()** encapsulates a non-null value; otherwise, there will be an empty object.

2. Which of the below methods encapsulates an empty object?

**Optional.ofNullable()**

**Optional.of()**

**Optional.empty()**

correct

Answer

Correct:

**Optional.empty()** encapsulates an empty object.

## Conclusion

In this lesson, you have found out that:

- The **Optional** class is used to identify the problems of returning a **null** value using the method to perform search, processing, or generate any object in a situation when this action gave no acceptable result.
- An object of the **Optional** type can be created using a static factory method: **Optional.of()**, **Optional.ofNullable()**, or **Optional.empty()**.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. Select the correct description of the method **Optional.ofNullable()**.

Encapsulates an empty object.

Encapsulates a non-null value; otherwise, there will be an empty object. correct

Encapsulates a non-null value: if the value is **null**, then the following exception will be thrown: **NullPointerException**.

Answer

Correct:

The method **Optional.ofNullable()** encapsulates a non-null value; otherwise, there will be an empty object.

2. Which of the below methods encapsulates an empty object?

**Optional.ofNullable()**

**Optional.of()**

**Optional.empty()**

correct

Answer

Correct:

**Optional.empty()** encapsulates an empty object.

You can find additional information in the [official documentation](#).

## Conclusion

In this lesson, you have explored the following methods of the **Optional** class:

- Methods of verifying and returning a result **get()**, **isPresent()**, **ifPresent()**
- Methods of returning a conditional result **orElse()**, **orElseGet()**, **orElseThrow()**

## Check Your Knowledge!

1. Which methods are used for verifying and returning a result?

**orElseThrow()** **get()**

**isPresent()**

**ifPresent()**

**orElseGet()**

correct

Answer

Correct:

The following methods are used for verifying and returning a result: **get()**, **isPresent()**, **ifPresent()**.

2. Is it true that the **orElseThrow()** method throws an exception of the type **NoSuchElementException** or the value produced by the transferred object of the type **Supplier**, if the **Optional** object is empty?

True correct

False

Answer

Correct:

The **orElseThrow()** method really throws an exception of the type **NoSuchElementException** or the value produced by the transferred object of the type **Supplier**, if the **Optional** object is empty

# Comments and Documenting

## Introduction

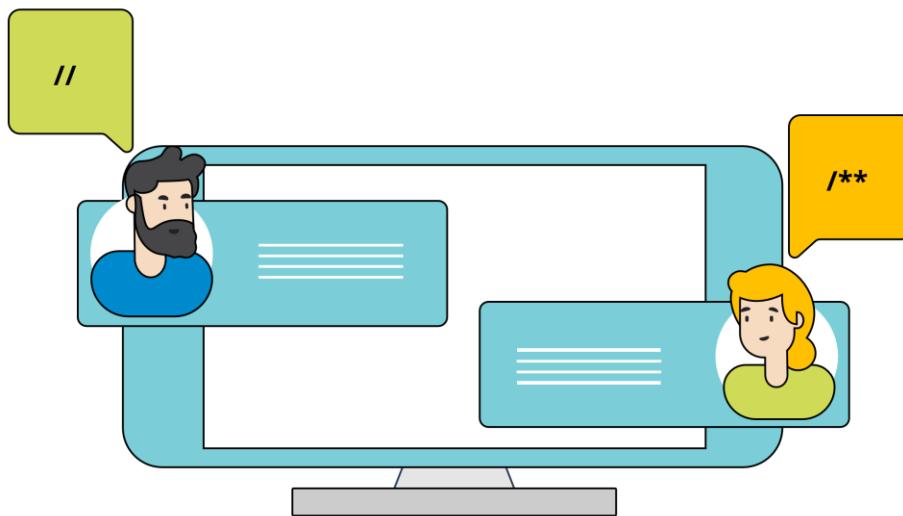
In this lesson, you will explore comments and their types in Java and will learn to work with them. You will also become familiar with the "Javadoc" tool and its functionality.

## Comments in Java

Comments are an important part of working with code, but how do they work and what are they used for?

**Comments** are a tool for the developer that allows to provide additional information about code and inform the compiler about the need to ignore a certain code line/block during compilation.

Comments are ignored by the compiler since they have a certain meaning for the developer and not for the end user. Therefore, the size of compiled classes is reduced.

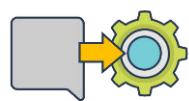


According to official documentation, Java programs can have two types of comments.

*Click on the cards to see the description of the types of comments.*



Documentation comments are used to describe code specification (its interface) that does not depend on its implementation.



Implementation comments are used to describe certain code lines/blocks.

Implementation comments can fall within one of two types – single-line and multi-line.

*Click on the tabs to learn about the types of comments.*

**Single-line**

**Multi-line**

**Use options**

Single-line comments start with // and continue to the end of the line.

```
// This is a comment
System.out.println("Hello World");

System.out.println("Hello World"); // This is a comment
```

Multi-line or block – start with /\* and end with \*/. The text inside /\* and \*/ will be ignored during compilation.

```
/* The code below will print the words
Hello World to the screen */
System.out.println("Hello World");
```

It is necessary to note that the comment type does not limit its use, and for the specified implementation comments the following variants of use is possible:

```
/* This is a comment */
System.out.println("Hello World");

System.out.println("Hello World"); /* This is a comment */

// The code below will print the words
// Hello World to the screen
System.out.println("Hello World");
```

The frequency of implementation comments sometimes reflects poor code quality. If you need to add a comment, try to rework your code to make it easier to understand.

When developing software, the largest problem with documenting code is maintaining that documentation. If documentation and code are separated, then each time when changing program code, you will have difficulties related to the necessity to make changes in the relevant parts of the documentation. The JDK contains a useful tool called **Javadoc**, generating documentation in the HTML format based on the source files.



Basically, interactive documentation on the official website is the result of applying the Javadoc utility to the source code of the standard Java library.

By adding comments starting with a sequence of characters /\*\* to your source code, it is easy to create professionally looking documentation. This is a very convenient approach, as it allows to store both: program code and documentation for this code at the same time. If you place documentation in a separate file, with time it will no longer correspond to program code. At the same time, since comments are an integral part of the source file, it is easy to update documentation by relaunching the **Javadoc** utility.

The Javadoc utility extracts information about the following elements:

*Scroll through the items to find out more.*



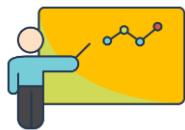
packages



Classes and interfaces declared as **public**



Methods declared as **public** or **protected**



Fields declared as **public** or **protected**

When developing a program code, you can (and even should) comment each of the listed elements. It is necessary to remember that comments:

- Are placed immediately before the element they refer to
- Start with the separation character `/**` and end with these characters `*/`
- May contain any text (description) and descriptors (tags), if they have the form `/** ... */`

It is also important to note that the line following after the documentation comment opening (`/**`) usually contains the description text. After the description come the tags, each time on a new line. Tags start with the `@` character, for example, `@author` or `@param`.

*Click on the tabs to learn about the types of tags.*

**Autonomous tags**

**Nested tags**

The Javadoc tags starting with the `@` character are called **autonomous** and should be placed at the beginning of the comment line. At that, the leading `*` character is ignored.



Tags starting with a curved bracket, for example, { @code }, are called **nested** and can be used inside a description.



The first sentence of the comments text should represent a short description. The Javadoc utility will automatically generate pages consisting of short descriptions.

In the text itself, it is possible to use elements of the HTML language, for example:

```
<em>...</em>
To highlight text in italics.
<code>...</code>
To set a monospaced font.
<strong>...</strong>
To highlight text in bold.
<img...>
To insert images.
```

Below you can see examples of implementation of comments.

```
/**
 * You can <em>even</em> insert a list:
 * <ol>
 * <li> Item one
 * <li> Item two
 * <li> Item three
 * </ol>
 */
 /**
 * <pre>
 * System.out.println(newDate());
 * </pre>
 */
```

If comments contain links to other files, for example, to images, diagrams, pictures, or screen shots of the user interface components, it is necessary to place these files in the catalog under the name of doc-files. The Javadoc utility will copy this catalog, its sub-catalogs, and files contained therein from the source catalog to the catalog for documentation. For example:

```

```

Below you can find examples of applying comments to different elements.

*Click on the drop-down element to learn about the different types of comments.*

#### ▼ Comments to a class

Comments to a class should be placed after the **import** directives, immediately before the class definition.

```
* A class representing a window on the screen.
* For example:
* <pre>
*   Window win = new Window(parent);
*   win.show();
* </pre>
*/
class Window extends BaseWindow {
    ...
}
```

Although most IDEs asterisks at the beginning of a line, there is no need to start each line of the documentation with an asterisk.

For example, the following comment is quite specific:

```
/*
A class representing a window on the screen.
For example:
<pre>
    Window win = new Window(parent);
    win.show();
</pre>
*/
class Window extends BaseWindow {
    ...
}
```

In the comments of documentation of a class, interface or package, the following tags are often used:

- **@author** author's name or information about the author. This tag creates the "author" section. The comments can have several such tags – per one for each author.
- **@version** text. This tag creates the "version" section. In this case, "text" means any description of the current version.

#### ▼ Comments to fields

We must document only public and protected fields (they are usually static constants).

```
/** 
 * The X-coordinate of the component.
 */
public int x = 1263732;
```

#### ▼ Comments to methods

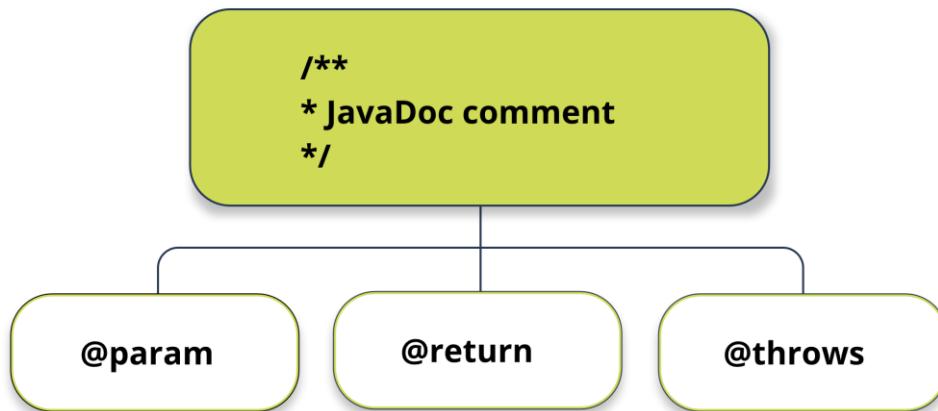
Comments should come immediately before the method they describe.

```
/** 
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 */
public char charAt(int index) {
    ...
}
```

Apart from general-purpose tags, it is also possible to use special tags – **@param**, **@return**, **@throws**.

*Click on each active element to learn more.*

## JavaDoc Tool



### @param

`@param variable description`

This tag adds the "Parameters" section to the method description and this section can be "stretched" over several lines. Apart from that, we can use elements of the HTML language. All @param tags related to one method should be grouped together.

\* @param index the index of the desired character.

### @param

`@param variable description`

This tag adds the "Parameters" section to the method description and this section can be "stretched" over several lines. Apart from that, we can use elements of the HTML language. All @param tags related to one method should be grouped together.

\* @param index the index of the desired character.

### @throws

`@throws class_description` or may be called `@exception class_description`

This tag means that the method can generate an exception.

\* `@throws StringIndexOutOfBoundsException`  
 \* if the index is not in the range `<code>0</code>`  
 \* to `<code>length() - 1</code>`.

Now consider how the tags can be used in practice.

*Click on the drop-down element to study the example.*

```

*
* @param index the index of the desired character.
* @return the desired character.
* @throws StringIndexOutOfBoundsException
*   if the index is not in the range <code>0</code>
*   to <code>length()-1</code>.
* @see java.lang.Character#charValue()
*/
public char charAt(int index) {
    ...
}

```

The following tags can be used, when writing any general comments intended to create documentation:

*Click on the tabs to learn about the types of tags.*

**@since text**

**@deprecated text**

**@see link**

This tag creates the "since" section ("since..."). The word "text" means the description of the version when this particular property was first implemented.

@since version 1.7.1



This tag adds a message that a class, method, or variable are not recommended for use. It is assumed that some expression follows after the tag @deprecated.

@deprecated Use setVisible(true) instead



Using the @see and @link tags, we can use hyperlinks to the relevant external documents or to a position in the structure of that same document.



The tag @see ... adds a link to the section "See also". It can be used in comments to both classes and methods. Here the word "link" means one of the following structures:

```
<a href="...">label  
3  
"text"
```

It is worth noting that the first option is more common. Here we need to specify the names of the class, method, or variable, and the Javadoc utility will include the relevant hyperlink in the documentation. For example, the below command creates a link to the method `raiseSalary(double)` of the class **com.epam.learn.Employee**:

```
@see com.epam.learn.Employee#raiseSalary(double)
```



Note that here we use the # character to separate the class name from the method or variable name, and not the period. The javac compiler correctly processes periods received by it as separators between the names of packages, sub-packages, classes, internal classes, methods, and variables. However, the Javadoc utility is not so intelligent, thus there is a need for special syntax.

If the tag @see is followed by the < character, then the developer should specify the hyperlink. You can refer to any URL-address. For example:

```
@see <a href="https://training.epam.com/">EPAM Learning
```

In each of these cases, you can specify an optional label that acts as an anchor for the link. If no label is specified, then the code name or URL will serve as the link's anchor. If the tag @see is followed by the " character, then the text will be displayed in the section "see also". For example:

```
@see "Java Fundamentals, chapter 2"
```

It is possible to use several @see tags for one and the same element, but they should be grouped together.

You can place hyperlinks to other classes or methods in any comment. To do this, you need to add a special tag of the form: {@link package.class#element label} to the required position. The description of the @link tag follows the same rules as that of @see tag.

## Package comments

Now that you have studied general comments, we will proceed to package, module, and overview comments.

Comments to classes, methods, and variables are usually placed directly in source files. However, to generate comments to a package in each catalog containing the package, a separate file has to be added. Two options exist.

*Click on the cards to see the description of the types of comments.*



## PACKAGE.HTML

Apply an HTML-file called package.html. The entire text contained between tags <body> ..... </body> is extracted by the Javadoc utility.



## PACKAGE-INFO.JAVA

Prepare a Java-file called package-info.java. This file should contain the initial Javadoc comment separated with /\*\* ... \*/ , after which follows the package sentence. It should not contain any code or comments.

Consider the examples of package-level documentation comments (package-info.java) and (package.html) given below.

*Click on the drop-down element to study the examples.*

### Example package-info.java

```
 /**
 * This is the core package for the application
 * @since 1.0
 */
package com.epam.learn;
```

### Example package.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Core Package</TITLE>
  </HEAD>
  <BODY>
    This is the core package of the application
    @since 1.0
  </BODY>
</HTML>
```

To generate comments to a module, information has to be added to the file module-info.java.

It is worth noting that all source files can be accompanied with overview comments. They are specified in the file called overview.html. It is located in the parent catalog containing all source files. The entire text located between tags <body> ... </body> is extracted by the Javadoc utility. These comments are displayed on the screen when the user chooses Overview in the navigation menu.

*Click on the drop-down element to study the example.*

### Example

```
<!DOCTYPE HTML>
<HTML>
<HEAD>
    <TITLE>API Overview</TITLE>
</HEAD>
<BODY>
    Short overview of the API.
</BODY>
</HTML>
```

Above we have explored the most common tags, but java has many more of them

*Click on the drop-down element to see all the possible tags and the options of their application.*

### ▼ Tags in javadoc

#### Tag

@author *name-text*

#### Description

Adds an Author entry with the specified name text to the generated documents when the -author option is used. A documentation comment can contain multiple @author tags.

{@code *text* }

Displays text in code font without interpreting the text as HTML markup or nested Javadoc tags. This enables to use regular angle brackets (< and >) instead of the HTML entities (&lt; and &gt;) in documentation comments.

Equivalent to <code>{@literal text}</code>.

@deprecated *deprecated-text*

Adds a comment indicating that this API should no longer be used (even though it may continue to work). The first sentence of deprecated text should tell the user when the API was deprecated and what to use as a replacement. Subsequent sentences can also explain why it was deprecated. It is recommended to include the @see or {@link} tags to inform the developer about the available alternative options.

{@docRoot}

Represents the relative path to the generated document's (destination) root directory from any generated page.

@exception *class-name description*

The @exception tag adds a Throws subheading to the generated documentation, with the class-name and description text. The class-name is the name of the exception that might be thrown by the method.

@hidden

Hides a program element from generating API documentation for this element.

{@index *word description*  
{@index "phrase" description}}

Declares that a word or phrase, along with an optional short description, should appear in index files generated by a standard doclet.

{@inheritDoc}

{@link package.class#member label }

{@linkplain package.class#member label }

{@literal text }

**@param parameter-name description** Adds a parameter with the specified parameter-name followed by the specified description to the Parameters section.

**@provides service-type description**

**@return description**

**@see "string"**  
**@see <a href="url">label</a>**  
**@see package.class#member label**

**@serial**  
**@serial**  
**@serial exclude**

**@serialData data-description**

**@serialField field-name field-type description**

**@since since-text**

**{@summary text }**

**{@systemProperty property-name }**  
**@throws class-name description**

**@uses service-type description**

**{@value}**

Inherits (copies) documentation from the nearest inheritable class or implementable interface into the current documentation comment at this tag's location.

Inserts an inline link with a visible text label that points to the documentation for the specified package, class, or member name of a referenced class.

Behaves the same as the {@link} tag, except the link label is displayed in plain text rather than code font. Useful when the label is plain text.

Displays text without interpreting the text as HTML markup or nested Javadoc tags. This enables you to use angle brackets (< and >) instead of the HTML entities (&lt; and &gt;) in documentation comments.

Adds a parameter with the specified parameter-name followed by the specified description to the Parameters section.

It can only appear in comments to module documentation and serves to document the implementation of the service provided by the module.

Adds a Returns section with the description text. This text should describe the return type and permissible range of values.

Adds a See Also heading with a link or text entry that points to a reference. A documentation comment can contain any number of @see tags, which are all grouped under the same heading. The @see tag has three variations. The form is the most common.

Used in the documentation comment for a default serializable field. An optional field-description should explain the meaning of the field and list the acceptable values. When needed, the description can span multiple lines. If a serializable field was added to a class after the class was made serializable, then a statement should be added to its main description to identify at which version it was added. The include and exclude arguments identify whether a class or package should be included or excluded from the serialized form page.

Uses the data description value to document the types and order of data in the serialized form. This data includes the optional data written by the writeObject method and all data (including base classes) written by the Externalizable.writeExternal method.

For a class implementing Serializable, the tag documents an ObjectStreamField component of the serialPersistentFields member of a Serializable class. Use one @serialField tag for each ObjectStreamField component.

Adds a Since heading with the specified since-text value to the generated documentation. This tag means that this change or feature has existed since the software release specified by the since-text value.

Allows the developer to explicitly show which part of the Javadoc comment will appear in the "summary", instead of relying on the standard Javadoc processing.

Defines property-name as the name of a system property.

Has the same purpose as the tag @exception.

It can appear only in the comments to module documentation and serves to document the possibility of using the service by the module.

Displays constant values. When the {@value} tag is used without an

{ @value package.class#field }

argument in the documentation comment of a static field, it displays the value of that constant. When used with the argument package.class#field in any documentation comment, the { @value } tag displays the value of the specified constant. The argument package.class#field takes a form similar to that of the @see tag argument, except that the member must be a static field.

@version *version-text*

Adds a Version subheading with the specified version-text value to the generated documents when the -version option of the Javadoc tool is used.

#### ▼ Options of using tags for different elements

Tag	Overview	Module	Package	Class	Constructor	Method	Field
@author	✓	✓	✓	✓			
{ @code }	✓	✓	✓	✓	✓	✓	✓
@deprecated		✓		✓	✓	✓	✓
{ @docRoot }	✓	✓	✓	✓	✓	✓	✓
@exception					✓	✓	
@hidden				✓		✓	✓
{ @index }	✓	✓	✓	✓	✓	✓	✓
{ @inheritDoc }				✓		✓	
{ @link }	✓	✓	✓	✓	✓	✓	✓
{ @linkplain }	✓	✓	✓	✓	✓	✓	✓
{ @literal }	✓	✓	✓	✓	✓	✓	✓
@param				✓	✓	✓	
@provides		✓					
@return						✓	
@see	✓	✓	✓	✓	✓	✓	✓
@serial			✓	✓			✓
@serialData						*	
@serialField						*	
@since	✓	✓	✓	✓	✓	✓	✓
{ @summary }	✓	✓	✓	✓	✓	✓	✓
{ @systemProperty }	✓	✓	✓	✓	✓	✓	✓
@throws					✓	✓	
@uses		✓					
{ @value }	✓	✓	✓	✓	✓	✓	✓
@version	✓	✓	✓	✓			

\* the tag @serialData can be used only in documentation comments for the methods readObject, writeObject, readExternal, writeExternal, readResolve and writeReplace

Page 361 \* the tag @serialField is used in documentation comments only for the field serialPersistentFields

Tags are included in the documentation comments in the following order (from left to right):

Note that the tags **@param** and **@return** in some cases are mandatory:

- The tag **@param** is "mandatory" (by convention) for each parameter even if the description is obvious.
- The tag **@return** is required for any method that returns anything other than void, even when it is redundant based on the method description.

## Conclusion

In this lesson, you have found out that:

- Comments allow to provide additional information about code and are ignored by the compiler.
- Java programs have two types of comments: implementation and documentation.
- The Javadoc tool extracts information about packages, classes, and interfaces declared as **public**; methods declared as **public / protected**; fields declared as **public / protected**.
- Tags are used when writing any comments intended for creating documentation.
- Tags are included in the documentation comments in the following order: **@author**, **@version**, **@param**, **@return**, **@exception**, **@see**, **@since**, **@serial**, **@deprecated**. At that, **@param** and **@return** are mandatory.

## Check Your Knowledge!

Below are several test questions to determine how well you understand the material from the lesson. If you make mistakes, you can refer back to the course materials to make sure you don't miss anything. Good luck!

*Read the questions carefully and select the appropriate answer(s). Then click Submit.*

2/2 points

1. In which options the characters of implementation comments are placed correctly?

// This is a multi-line comment //  
/\* This is a multi-line comment \*/  
// This is a single-line comment  
/\* This is a single-line comment

correct

Answer

Correct:

Single-line comments start with // and continue to the end of the line. Multi-line comments start with /\* and end with \*/.

2. Which tag can be used only in methods descriptions (except constructors)?

@param  
@hidden  
@since  
@return correct

Answer

Correct:

The tag **@return** is used in documentation comments to describe the method's result value.

Note that the tags **@param** and **@return** in some cases are mandatory:

- The tag **@param** is "mandatory" (by convention) for each parameter even if the description is obvious.
- The tag **@return** is required for any method that returns anything other than void, even when it is redundant based on the method description.

## Conclusion

In this lesson, you have found out that:

- Comments allow to provide additional information about code and are ignored by the compiler.
- Java programs have two types of comments: implementation and documentation.
- The Javadoc tool extracts information about packages, classes, and interfaces declared as **public**; methods declared as **public / protected**; fields declared as **public / protected**.
- Tags are used when writing any comments intended for creating documentation.
- Tags are included in the documentation comments in the following order: @author, @version, @param, @return, @exception, @see, @since, @serial, @deprecated. At that, @param and @return are mandatory.

## Check Your Knowledge!

1. In which options the characters of implementation comments are placed correctly?

```
// This is a multi-line comment //
/* This is a multi-line comment */
// This is a single-line comment
/* This is a single-line comment
```

correct

Answer

Correct:

Single-line comments start with // and continue to the end of the line. Multi-line comments start with /\* and end with \*/.

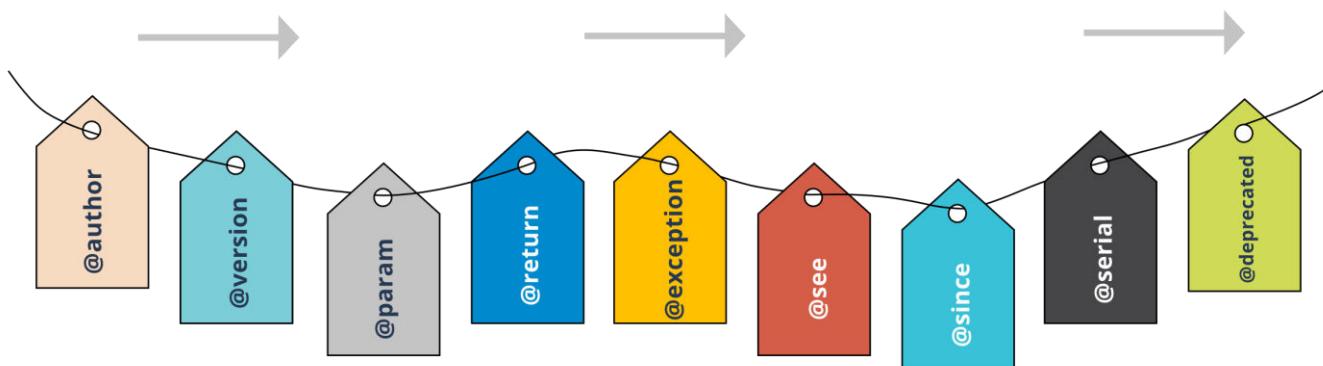
2. Which tag can be used only in methods descriptions (except constructors)?

```
@param
@hidden
@since
@return correct
```

Answer

Correct:

The tag @return is used in documentation comments to describe the method's result value.



Note that the tags **@param** and **@return** in some cases are mandatory:

- The tag **@param** is "mandatory" (by convention) for each parameter even if the description is obvious.
- The tag **@return** is required for any method that returns anything other than void, even when it is redundant based on the method description.

## Conclusion

In this lesson, you have found out that:

- Comments allow to provide additional information about code and are ignored by the compiler.
- Java programs have two types of comments: implementation and documentation.
- The Javadoc tool extracts information about packages, classes, and interfaces declared as **public**; methods declared as **public / protected**; fields declared as **public / protected**.
- Tags are used when writing any comments intended for creating documentation.
- Tags are included in the documentation comments in the following order: @author, @version, @param, @return, @exception, @see, @since, @serial, @deprecated. At that, @param and @return are mandatory.

## Check Your Knowledge!

1. In which options the characters of implementation comments are placed correctly?

```
// This is a multi-line comment //
/* This is a multi-line comment */
// This is a single-line comment
/* This is a single-line comment
```

correct

Answer

Correct:

Single-line comments start with // and continue to the end of the line. Multi-line comments start with /\* and end with \*/.

2. Which tag can be used only in methods descriptions (except constructors)?

```
@param
@hidden
@since
@return correct
```

Answer

Correct:

The tag @return is used in documentation comments to describe the method's result value.

## Generating Documentation

## Introduction

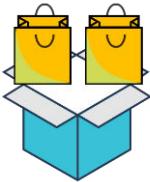
In this lesson, you will continue to explore the topic of code documenting and will study the process of comments extraction.

## Extracting Comments

To extract documentation comments from source files, you need to use the Javadoc utility that is part of the standard JDK. Let's assume that docDirectory is the name of the catalog that should store HTML-files.

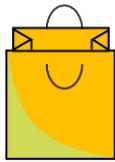
Carefully study the actions below to learn how to extract code documentation comments from source files. Also, note that step 2 can vary depending on the number of packages.

*Click on the arrow to see the steps.*



## Step 1

Go to the catalog that contains source files to be documented. If nested packages are subject to be documented, go to the catalog containing these packages. For example, for the packages com.epam.learn, you need to go to the catalog containing the sub-catalog com (this catalog should store the file overview.html.)



## Step 2 (one package)

To document one package, execute the following command:

```
javadoc -d docDirectory package_name
```



## Step 2 (several packages)

To document several packages, execute the following command:

```
javadoc -d docDirectory package_name_1 package_name_2 ...
```



## Step 2 (by default)

If files are located in a package set by default, then instead of executing the above commands we need the following command:

```
javadoc -d docDirectory *.java
```

If we omit the option -d docDirectory, then the HTML-files will be extracted to the current catalog, which will cause confusion. Therefore, this is not recommended.

When invoking the Javadoc utility, you can specify different options. For example, to include the tags @author and @version in the documentation, you can use options -author and -version (by default they are omitted).

If you need to do a fine-tuning, for example, to create documentation in a format different from HTML, then you can develop a custom doclet allowing to generate documentation in any form.



Oracle has developed a tool to check comments to a document called Oracle Doc Check Doclet or DocCheck. If you launch it on source code, it will generate a report with a description of style and tag errors in the comments and will recommend the necessary changes.

You can get additional information on the options of the Javadoc utility from the [official documentation](#).

## Conclusion

In this lesson, you have found out how to generate documentation from the existing javadocs.

### Check Your Knowledge!

1. Which command should you execute if files are located in a package set by default?

javadoc -d docDirectory \*.java correct  
javadoc -d docDirectory package\_name  
javadoc \*.java  
all commands can be used

Answer

Correct:

The command javadoc -d docDirectory \*.java is used when the files are located in a package set by default.

2. Is it possible to omit the option -d docDirectory when extracting documentation comments?

Yes, it is recommended to do this.

Yes, but this is not recommended. correct

No

Answer

Correct:

If we omit the option -d docDirectory, then the HTML-files will be extracted to the current catalog, which will cause confusion. Therefore, this is not recommended.