# Programming With Java

**Ashik Ahmed Bhuiyan, Ph.D. &**
**Md Amiruzzaman, Ph.D.**


A Member of The Pennsylvania Alliance for Design of Open Textbooks



The contents of this eTextbook were developed under a grant from the Fund for the Improvement of Postsecondary Education, (FIPSE), U.S. Department of Education. However, those contents do not necessarily represent the policy of the Department of Education, and you should not assume endorsement by the Federal Government.

# About PA-ADOPT

The Pennsylvania Alliance for Design of Open Textbooks (PA-ADOPT) is made up of four participating institutions from Pennsylvania State System of Higher Education (PASSHE) that are all regional and primarily undergraduate institutions, situated in Southeastern Pennsylvania. The PA-ADOPT project addresses gaps in the open eTextbook marketplace, improve student learning, and mitigate rising student costs. PA-ADOPT was made possible by the US Department of Education Open Textbook Pilot Program.

# About OER

Open Educational Resources (OER) are instructional, learning and research materials, digital or non, that open-source and in the public domain or that are licensed so that users have free and perpetual permission to engage in the following activities:

- Retain: the right to make, own, and control copies of the content
- Reuse: the right to use the content in a wide range of ways
- Revise: the right to adapt, adjust, modify, or alter the content itself
- Remix: the right to combine the original or revised content with other open content to create something new
- Redistribute: the right to share copies of the original content, revisions, and remixes with others.

# About the Authors

## Ashik Ahmed Bhuiyan

Ashik Ahmed Bhuiyan

Ashik Ahmed Bhuiyan, Ph.D. is an assistant professor in the Department of Computer Science at West Chester University of Pennsylvania (WCUPA). He teaches various undergraduate and graduate courses, including data structures, algorithms, introduction to programming, fundamentals of computer science, and research seminars.

He earned his Ph.D. in electrical and computer engineering from the University of Central Florida (UCF), where he was a member of the Real-Time & Intelligent Systems Lab, working under the supervision of Zhishan Guo and Abusayeed Saifullah (Wayne State University). He completed his bachelor's degree in computer science and engineering from Bangladesh University of Engineering and Technology (BUET) in 2013. His research focuses on energy efficiency in real-time embedded systems, parallel computing, and mixed-criticality scheduling. His work received the Best Student Paper Award at the 40th IEEE Real-Time Systems Symposium (RTSS 2019).

Outside of academia, he enjoys watching movies, reading books, and spending time with his wife and 7-month-old baby.

# Md Amiruzzaman

Md Amiruzzaman

Md Amiruzzaman, Ph.D., is an Assistant Professor in the Department of Computer Science at West Chester University. Before joining WCU, he worked as a software developer for almost 10 years for several companies. He has also held the position of Assistant Professor at Kent State University. He has completed a Bachelor's Degree in Computer Science from National University. Along with that, he has completed four Master's degrees with major in Computer Engineering in 2008 from Sejong University, Computer Science in 2011 from Kent State University (also, partly at Korea University), and Technology in 2015, also from Kent State University, and a Master's in Cybersecurity in 2023 from Georgia Institute of Technology. He received his Ph.D. degrees from Kent State University in 2016 (Mathematics Edu), 2019 (Evaluation and Measurement) and 2021 (Computer Science). In the past, he has worked as a Research Assistant at Sejong University and Korea University. He has also taught at National University and Korea University. His research interests include Visual Analytics of urban data, Data Mining, Machine Learning, Deep Learning, and Data Hiding.

# Table of Contents

# 1 Introduction

## 1.1 What Is a Computer?

A computer is an electronic and programmable device that allows users to store, retrieve, and process information and data (see Figure 1.1). Computers can be programmed to perform mathematical calculations or logical computations using different types of computer programming languages such as Java, C, C++, C#, and Python.

## 1.2 What Is Computer Programming?

Figure 1.1: A modern day laptop computer. Image by Lukas from Pixabay.

Computer Programming is a process of writing code that can translate into instructions for a computer system. In other words, computer programming allows programmers to write a set of instructions to perform specific tasks. There are many programming languages, and different programmers like different languages. Java is one of the commonly used programming languages and is illustrated in Example 1 below.

## 1.3 What Is Java? History of Java

Java is a popular computer programming language. It is a platform-independent language, which means that it works in almost all types of operating systems, such as Windows, Linux, OSX, and the like.

Java was first introduced in 1995 by Sun Microsystems. Since its introduction, Java has evolved. Today, many computer applications are developed using Java programming language.

## 1.4 A Sample Java Program

The following example shows a sample Java program (see Example 1). Line number 1 shows the class name “SampleClass,” then the opening curly brace (i.e., “{”) for the class. In line 2, the “main” (i.e., (String args[])) method starts and ends at line 5. Line number 4 shows a built-in function to print the message

“Hello Class!!” displayed on the screen. The “SampleClass” ends with the closing curly brace (i.e., “}”) at line 6.

### Example 1

A sample Java program to display a message on the screen. In this example, “Hello Class!!” is the message:

**Example 1 Code**

| Line | Code |
|---|---|
| 1 | `Public class SampleClass {` |
| 2 | `    public static void main(String args[]) {` |
| 3 | `        // print "Hello Class!!" on the screen when this program runs` |
| 4 | `        System.out.println("Hello Class!!”);` |
| 5 | `    }` |
| 6 | `}` |

Here is the output:

```
Hello Class!!
```

The file name for the “SampleClass” will be “SampleClass.java,” which means that the class name and the source code file are the same. Also, whatever you place within the double quotes (i.e., “”) in the System.out.println(); will be displayed in the screen as an output.

Also, note that anything you write using “//” will not be checked by the compiler. So, programmers use “//” to write programming comments. If you want to add multi-line comments in your program file or source code, then you can use “/* … */”, which is known as a block comment.

```
/*
example of
multiple line
comments
*/
```

## 1.5 What Is Source Code?

Programming statements are also known as source code. In other words, in Java, programming source code means the .java file where programmers write and save their programming statements. Compilers use source code to generate object codes or output files. The Example 1 code above is an example of a Java source code.

In Java programming, "assigned" refers to giving a value to a variable or a constant. We use "=" operator to accomplish this.

## 1.6 Variables and Constants

A variable is a data item that holds a value, and the value of a variable can change over time. However, a constant is a variable whose value cannot change once assigned.

### 1.6.1 Keywords/Reserved Words

There are some words in Java programming that are reserved and cannot be used to create variables such as the following:

```
public
class
static
void
int
double
```

### 1.6.2 VariableS

In Java, there are multiple different ways variables can be declared. Different ways of declaring variables depend on the data type or what type of data value the variable will hold. Examples include integer, floating point, string, character, and the like.

### 1.6.3 Identifier

The identifier helps to identify something uniquely. For example, within a program, variable names can be identifiers that enable us to identify uniquely the variables within the program. A class name is an identifier since it helps to identify the class uniquely. On the other hand, a variable name is given to a memory location that holds a value.

An identifier is often used to name variables, classes, packages, methods, and functions. Variables are designed to hold a value within a memory location, and that value can be changed while a program is running.

## 1.7 Rules for Variable Declaration

The following rules should be followed when declaring a variable in Java:

1. A variable declaration must begin with a *variable type*, such as *int*, *double*, *string*, and *char*.
2. Variable names must begin with a letter (i.e., either upper-case or lower-case alphabet), and not with a number or digit.
3. Reserved keywords cannot be used to name a variable.
4. Mathematical operators cannot be used in a variable name.
5. Variable names must be unique within a context.

### 1.7.1 How To Declare a Variable

In Java, declaring or creating a variable requires a programmer to know the type of variable first, then the name of the variable, and finally the initial value of the variable (see Example 2). Although the initial value is not required, it is advised. Some compilers may show a warning or error message if a variable is not initialized—this depends on the compiler settings.

The example below shows an integer type of variable: “a” is declared, and subsequently a value is assigned (Example 2). However, commenting on the initialization indicates that the initialization is not required until the variable is used. The second example in line 7 shows that both the declaration and initialization can be done at the same time. Notice that “int” is short for integer type. Variable declaration must end with a “;” as same as other statements must do, too.

```
type variablename = value ;
```

Since the variable “a” and “b” are the same type, one could declare them as,

```
int a = 12, b = 13;
```

Note that the above example did not require adding the keyword “int” two times, since both variables are the same type and one use of “int” is enough. Multiple declarations of the same type are separated by a comma in between them.

### 1.7.2 Types of Variables in Terms of the Value They Store

In Java, we use different types of variables to store different types of values:

- int - To store whole numbers, we use an integer type of variable. A shortened form for integer is “int”. Examples of whole numbers are 1, 124, and 8080. Please see the Example 2 to understand how an “int” variable is declared and used in a Java program.
- string - stores text data. In general, it can be a mix of characters and numbers or just characters or just numbers. We use double quotes to represent string variables, such as “Hello”.
- float - stores floating point numbers, with decimals, such as 199.99 or
- char - stores single characters, such as ’a’ or ’B’. Unlike String values, singles quotes are used to represent character values.
- boolean - stores true or false values. When printed as a boolean variable, then it will show either true or false in the screen. Note that a comparison operator also returns a boolean value, i.e., true or false.

**Example 2**

This example shows how to declare and use ”int” type variables:

```
public class ExampleClass {
    public static void main(String args[]) {
        // declare a "int" type of variable "a"
        int a = 5;
        // declare a "int" type of variable "b"
        int b = 6;
        // declare a "int" type of variable "sum"
        int sum = a + b;
        System.out.println("Sum of a + b = " + sum);
    }
}
```

The output of example 2:

```
Sum of a + b = 11
```

### Example 3

The following example shows how to declare and use ”string” type variables:

```
public class StringClass {
    public static void main(String args[]) {
        // declare a "String" type of variable "a"
        String a = "Hello";
        // declare a "String" type of variable "b"
        String b = "Class!!";
        // displaying the message by combining two variables
        System.out.println(a + b);
    }
}
```

The output of Example 3:

```
Hello Class!!
```

### Example 4

The following example shows how to declare and use ”int” type variables:

```
public class EaxmpleFloat {
    public static void main(String args[]) {
        // declare a "float" type of variable "a"
        float a = 5.3f;
        // declare a "int" type of variable "b"
        float b = 4.7f;
        // declare a "int" type of variable "sum"
        float sum = a + b;
        System.out.println("Sum of a + b = " + sum);
    }
}
```

The output of Example 4:

```
Sum of a + b = 10.0
```

### Example 5

The following example shows how to declare and use ”char” type variables:

```
public class ExampleCharacter{
    public static void main(String []args){
        // declare c1 as character variable and assign 'H'
        char c1 = 'H';
        // declare c2 as character variable and assign 'i'
        char c2 = 'i';
        // declare c3 as character variable and assign '!'
        char c3 = '!';
        System.out.println("Output: " + (c1) +(c2) + (c3));
    }
}
```

The output of Example 5:

```
Output: Hi!
```

### Example 6

This example shows how to declare and use "boolean" type variables:

```
public class ExampleClass {
    public static void main(String args[]) {
        // declare variable "a" as boolean
        boolean a = true;
        // declare variable "b" as boolean
        boolean b = false;
        System.out.println("The value of a is " + a + "; the value of b is
        " + b);
    }
}
```

The output of Example 6:

```
The value of a is true; the value of b is false
```

## 1.7.3 Type of Variables in Java Based on Where They Are Declared

There are several types of variables, such as

1. local variable—this type of variable is defined within a method or a function. Its scope is limited within the method or function where it is defined.

2. instance variable—this type of variable is defined within a class and not within a method of the class.

3. static variable—this type of variable is shared among instances of a class.

**Example 7**

An example of static, instance, and local variables:

```
public class SampleJavaClass{
    // static variable
    public static int a = 10;
    // instance variable
    public int b = 15;
    // a static method
    public static void display(){
        // local variable
        int a1 = 20;
        System.out.println("Local value: " + a1 + "\n");
    }
    public static void main(String []args){
        // call the display method
        display();
        // calling static variable
        System.out.println("Static variable: " + a);
        // creating an object "jc" from SampleJavaClass
        SampleJavaClass sjc = new SampleJavaClass();
        // calling the instance variable using the object "sjc"
        System.out.println("Instance variable: " + sjc.b);
    }
}
```

The output of Example 7:

```
Local value: 20

Static variable: 10
Instance variable: 15
```

The examples above shows assignment of values to variables. So, a programmer can change value of variable. For example,

```
public static void display(){
    // local variable
    int a1 = 20;
    a1 = 25;
    System.out.println("Local value: " + a1 + "\n");
}
```

So, it is allowed to update the value of variable a1 in a subsequent statement. This was possible as a1 is a variable. Reassignment would not be possible if a1 was a constant. See more on this in discussion below.

## 1.8 Constants

Constants are a type of variable whose values do not change within the program. Sometimes we declare a constant variable in a different class. In that event, the value of the constant will remain unchanged. If we try to change the value of a constant, then the compiler will show an error message.

Here is the syntax of a constant variable.

```
final type variablename = value;
```

Here is an example of a double type of constant:

```
final double PI = 3.141;
```

Note that, programmers often use all upper case letters to define a constant variable (see Example 8). This is an attempt to separate the constant variables from the rest of the variables.

### Example 8

The example below shows how to declare and use a constant variable:

```
public class CircleClass {
     public static void main(String args[]) {
     // regular double type of variable for the radius
     double radius = 4.2;
     // PI declared as a constant variable
     final double PI = 3.141;

     System.out.println("The area of the circle is: " + (PI*radius*radius));
     }
}
```

The output of Example 8:

```
The area of the circle is: 55.40724000000001
```

## 1.9 ASCII-Table

The American Standard Code for Information Interchange (ASCII) is the most widely used character encoding format in computer science. In ASCII encoding, there are total 256 (i.e., 0 to 255) alphabetic, numeric and special characters codes. Please see the ASCII character tables below.

| Dec | Hex | Oct | Character |
|---|---|---|---|
| 0 | 0x00 | 000 | NUL |
| 1 | 0x01 | 001 | SOH |
| 2 | 0x02 | 002 | STX |
| 3 | 0x03 | 003 | ETX |
| 4 | 0x04 | 004 | EOT |
| 5 | 0x05 | 005 | ENQ |
| 6 | 0x06 | 006 | ACK |
| 7 | 0x07 | 007 | BEL |
| 8 | 0x08 | 010 | BS |
| 9 | 0x09 | 011 | TAB |
| 10 | 0x0A | 012 | LF |
| 11 | 0x0B | 013 | VT |
| 12 | 0x0C | 014 | FF |
| 13 | 0x0D | 015 | CR |
| 14 | 0x0E | 016 | SO |
| 15 | 0x0F | 017 | SI |
| 16 | 0x10 | 020 | DLE |
| 17 | 0x11 | 021 | DC1 |
| 18 | 0x12 | 022 | DC2 |
| 19 | 0x13 | 023 | DC3 |
| 20 | 0x14 | 024 | DC4 |
| 21 | 0x15 | 025 | NAK |
| 22 | 0x16 | 026 | SYN |
| 23 | 0x17 | 027 | ETB |
| 24 | 0x18 | 030 | CAN |
| 25 | 0x19 | 031 | EM |
| 26 | 0x1A | 032 | SUB |
| 27 | 0x1B | 033 | ESC |
| 28 | 0x1C | 034 | FS |
| 29 | 0x1D | 035 | GS |
| 30 | 0x1E | 036 | RS |
| 31 | 0x1F | 037 | US |

| Dec | Hex | Oct | Character |
|---|---|---|---|
| 32 | 0x20 | 040 | SP |
| 33 | 0x21 | 041 | ! |
| 34 | 0x22 | 042 | ”’ |
| 35 | 0x23 | 043 | # |
| 36 | 0x24 | 044 | $ |
| 37 | 0x25 | 045 | % |
| 38 | 0x26 | 046 | & |
| 39 | 0x27 | 047 | ’ |
| 40 | 0x28 | 050 | ( |
| 41 | 0x29 | 051 | ) |
| 42 | 0x2A | 052 | * |
| 43 | 0x2B | 053 | + |
| 44 | 0x2C | 054 | , |
| 45 | 0x2D | 055 | - |
| 46 | 0x2E | 056 | . |
| 47 | 0x2F | 057 | / |
| 48 | 0x30 | 060 | 0 |
| 49 | 0x31 | 061 | 1 |
| 50 | 0x32 | 062 | 2 |
| 51 | 0x33 | 063 | 3 |
| 52 | 0x34 | 064 | 4 |
| 53 | 0x35 | 065 | 5 |
| 54 | 0x36 | 066 | 6 |
| 55 | 0x37 | 067 | 7 |
| 56 | 0x38 | 070 | 8 |
| 57 | 0x39 | 071 | 9 |
| 58 | 0x3A | 072 | : |
| 59 | 0x3B | 073 | ; |
| 60 | 0x3C | 074 | ”¡ |
| 61 | 0x3D | 075 | = |
| 62 | 0x3E | 076 | ”¿ |
| 63 | 0x3F | 077 | ? |

| Dec | Hex | Oct | Character |
|---|---|---|---|
| 64 | 0x40 | 100 | @ |
| 65 | 0x41 | 101 | A |
| 66 | 0x42 | 102 | B |
| 67 | 0x43 | 103 | C |
| 68 | 0x44 | 104 | D |
| 69 | 0x45 | 105 | E |
| 70 | 0x46 | 106 | F |
| 71 | 0x47 | 107 | G |
| 72 | 0x48 | 110 | H |
| 73 | 0x49 | 111 | I |
| 74 | 0x4A | 112 | J |
| 75 | 0x4B | 113 | K |
| 76 | 0x4C | 114 | L |
| 77 | 0x4D | 115 | M |
| 78 | 0x4E | 116 | N |
| 79 | 0x4F | 117 | O |
| 80 | 0x50 | 120 | P |
| 81 | 0x51 | 121 | Q |
| 82 | 0x52 | 122 | R |
| 83 | 0x53 | 123 | S |
| 84 | 0x54 | 124 | T |
| 85 | 0x55 | 125 | U |
| 86 | 0x56 | 126 | V |
| 87 | 0x57 | 127 | W |
| 88 | 0x58 | 130 | X |
| 89 | 0x59 | 131 | Y |
| 90 | 0x5A | 132 | Z |
| 91 | 0x5B | 133 | [ |
| 92 | 0x5C | 134 | \ |
| 93 | 0x5D | 135 | ] |
| 94 | 0x5E | 136 | ^ |
| 95 | 0x5F | 137 | _ |

| Dec | Hex | Oct | Character |
|---|---|---|---|
| 96 | 0x60 | 140 | ` |
| 97 | 0x61 | 141 | a |
| 98 | 0x62 | 142 | b |
| 99 | 0x63 | 143 | c |
| 100 | 0x64 | 144 | d |
| 101 | 0x65 | 145 | e |
| 102 | 0x66 | 146 | f |
| 103 | 0x67 | 147 | g |
| 104 | 0x68 | 150 | h |
| 105 | 0x69 | 151 | i |
| 106 | 0x6A | 152 | j |
| 107 | 0x6B | 153 | k |
| 108 | 0x6C | 154 | l |
| 109 | 0x6D | 155 | m |
| 110 | 0x6E | 156 | n |
| 111 | 0x6F | 157 | o |
| 112 | 0x70 | 160 | p |
| 113 | 0x71 | 161 | q |
| 114 | 0x72 | 162 | r |
| 115 | 0x73 | 163 | s |
| 116 | 0x74 | 164 | t |
| 117 | 0x75 | 165 | u |
| 118 | 0x76 | 166 | v |
| 119 | 0x77 | 167 | w |
| 120 | 0x78 | 170 | x |
| 121 | 0x79 | 171 | y |
| 122 | 0x7A | 172 | z |
| 123 | 0x7B | 173 | { |
| 124 | 0x7C | 174 | \| |
| 125 | 0x7D | 175 | } |
| 126 | 0x7E | 176 | ” |
| 127 | 0x7F | 177 | DEL |

### Example 9

This example shows how to declare and use ”char” type variables and ASCII values:

```
public class ExampleCharacter{
    public static void main(String []args){
        // declare c1 as character variable and assign ASCII value for 'H'
        char c1 = 72;
        // declare c2 as character variable and assign ASCII value for 'i'
        char c2 = 105;
        // declare c3 as character variable and assign ASCII value for '!'
        char c3 = 33;
        System.out.println("Output: " + (c1) +(c2) + (c3));
    }
}
```

The output of Example 9:

```
Output: Hi!
```

## 1.10 Widening and Narrowing

Often, a programmer may need to widen or narrow a value of a variable. When converting a variable from a smaller size of a primitive type to a larger size of a primitive type, we call this widening, the opposite of narrowing.

In Java, not all primitive types are the same in size and range. The sizes of the primitive types are as follows:

**Table 1: Primitive Types, Sizes, and Ranges**

| Primitive Types | Size in byte | Ranges |
|---|---|---|
| **byte** | 1 byte | -128 to 127 |
| **short** | 2 bytes | -32,768 to 32,767 |
| **int** | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| **long** | 8 bytes | 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807 |
| **float** | 4 bytes | 3.4e-038 to 3.4e+038 |
| **double** | 8 bytes | 1.7e-308 to 1.7e+038 |
| **char** | 2 bytes | u0000 (0) to uffff |
| **boolean** | 1 byte | true and false |

### 1.10.1 Widening

Variable value widening is an interesting concept in Java and is also known as upcasting. The conversion that implicitly takes place is outlined in the following situations:

1. Widening takes place when a smaller primitive type value is automatically accommodated in a larger/wider primitive data type.

2. Widening also takes place when a reference variable of a subclass is automatically accommodated in a reference variable of its superclass.

More specifically, widening occurs when a smaller primitive type value is automatically accommodated in a larger or wider primitive data type. See Example 10 for more details.

**Example 10**

This example shows how to widen a variable value:

```
public class WideningExample {
    public static void main(String[] args) {
        byte b=10;
        //byte value is widened to short
        short s= b;
        //byte value is widened to int
        int i=b;
        //byte value is widened to long
        long l=b;
        //byte value is widened to float
        float f=b;
        //byte value is widened to double
        double d=b;

        System.out.println("short value : "+ s);
        System.out.println("int value : "+ i);
        System.out.println("long value : "+ l);
        System.out.println("float value : "+ f);
        System.out.println("double value : "+ d);
    }
}
```

The output of Example 10:

```
short value : 10
int value : 10
```

```
long value : 10
float value : 10.0
double value : 10.0
```

### 1.10.2 Narrowing

Like the widening concept, narrowing in Java means downcasting or downgrading a primitive type using an explicit conversion that is performed in the following situations:

- Narrowing a wider/bigger primitive type value to a smaller primitive value.
- Narrowing a superclass reference to a subclass reference during inheritance.

More specifically, narrowing takes place when a larger primitive value is automatically accommodated in a smaller primitive data type. See Example 11 for more details.

**Example 11**

This example shows how to narrow a variable value:

```
public class NarrowingExample {
    public static void main(String[] args) {
        double d =10.5;
        //Narrowing double to byte
        byte b = (byte)d;
        //Narrowing double to short
        short s= (short)d;
        //Narrowing double to int
        int i= (int)d;
        //Narrowing double to long
        long l= (long)d;
        //Narrowing double to float
        float f= (float)d;


        System.out.println("Original double value : " +d);
        System.out.println("Narrowing double value to short : "+ s);
        System.out.println("Narrowing double value to int : "+ i);
        System.out.println("Narrowing double value to long : "+ l);
        System.out.println("Narrowing double value to float : "+ f);
        System.out.println("Narrowing double value to byte : "+ b);
    }
}
```

The output of Example 11:

```
Original double value : 10.5
Narrowing double value to short : 10
Narrowing double value to int : 10
Narrowing double value to long : 10
Narrowing double value to float : 10.5
Narrowing double value to byte : 10
```

## 1.11 Exercise

1. What is computer programming? What is source code in Java?

2. Which is a correct declaration of a variable?

```
int a = 10
a int = 10
int a = 10;
int; a = 10;
```

3. Write a Java program that has two double type variables and values that are 3.6, and 5.4. Also, display the sum of those variables.

4. How do you declare a string type of variable?

5. What are the different types of variables in Java based on their locations?

6. Provide an example of valid and invalid variable declarations.

7. In what situation should a programmer declare a constant variable?

8. Write a Java program that multiplies two double type variables' values and displays the result.

9. Write a Java program only using character variables to display "Hello Class!!". Hint: each character variable will hold only one character, e.g., char c1 = 'H';

10. Write a Java program only using character variables to display "Hello Class!!". Instead of using characters, use ASCII values. Hint: each character variable will hold only one character, e.g., char c1 = 72;

# 2 Control Statements and Loops

## 2.1 What Is a Control Statement?

A control statement deals with one or more conditions to be evaluated by the program, one of which must be true. Below, we represent a general form of a typical control statement structure used in many programming languages.

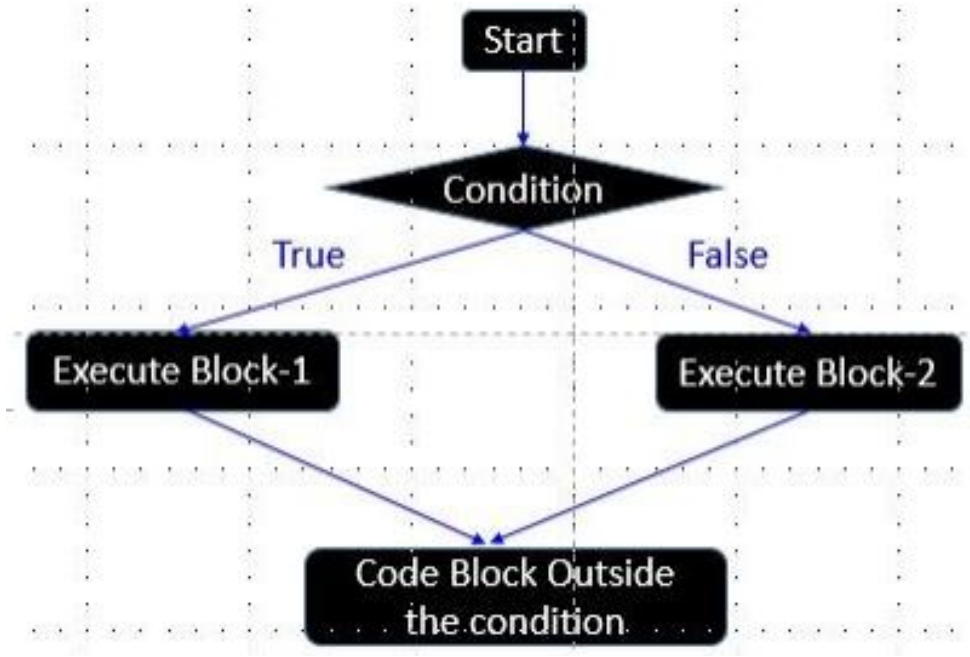


Figure 1: Basic Structure of a Branching Statement

Here, the condition will be evaluated as true or false. If the condition is true, block-1 will be executed. If the condition is false, block-2 will be executed.

**Example 1**

Now, let us describe this concept by using a real-life example. John wakes up at 7.00 AM and has his breakfast. Now, he wants to go out to buy groceries. However, he does not like rain, and he will first check if it is raining or not before heading out. If it is raining, he will buy the groceries on a different day and clean the house instead. Otherwise, he will buy groceries today. Whatever he does, i.e., clean the house or buy groceries, he will have lunch after that. We can depict this scenario as shown in Figure 1 above. Here,

- Start = Wake up and have breakfast
- Condition = Whether it will rain today
- Block-1 = Clean house.
- Block-2 = Go to the grocery store.
- Code block outside the condition = Have lunch.

In this example, we have assumed that Block-1 will execute if the condition is true. We can express the scenario in relation to Figure 2 below as follows:

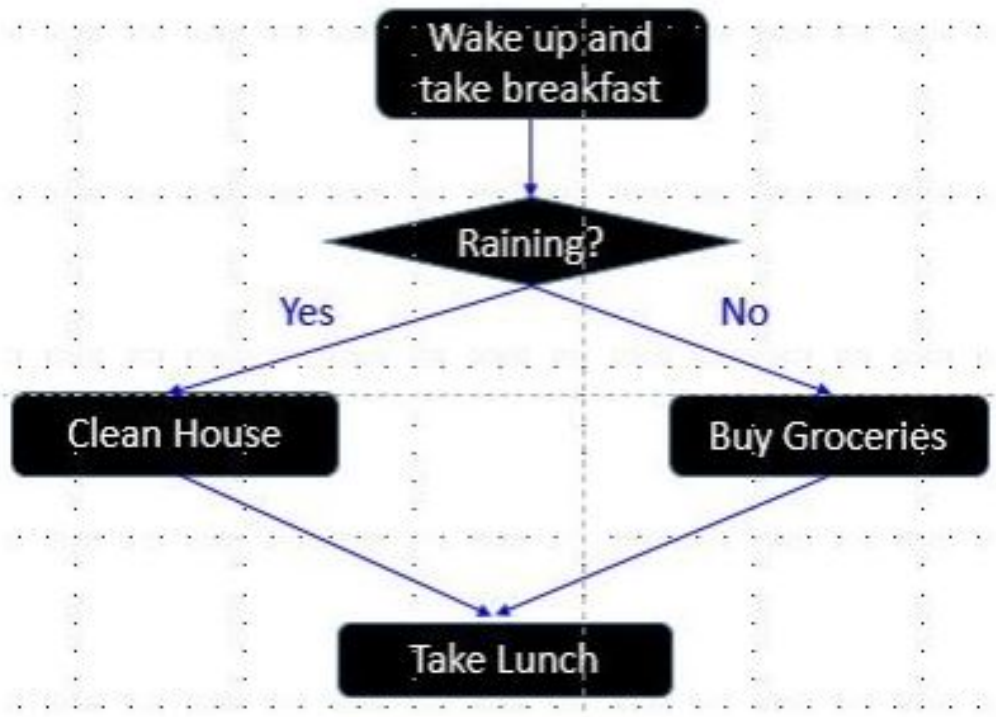


Figure 2: Example of an if-else statement within a real-life scenario

From the above discussion, it should be clear that a control statement is a construct in a computer program, where the program needs to be decided based on a condition (or a logical test). Once the decision is made, the program executes single or multiple action(s) based on that decision. The decision is thus made based on the outcome of a logical test. Here, the outcome of the condition (or the logical test) is a boolean value, i.e., TRUE or FALSE. In a decision structure, the following happens:

- A condition is evaluated. For instance, in the previous example, John decides based on the weather conditions. He considers whether it is a good day to go grocery shopping.
- An operation (or multiple operations) is performed among a set of choices. For example, John can go grocery shopping or clean the house. He will do the latter only if it rains.

Without the knowledge of the control statement, we may think that the codes (or statements) are executed sequentially, i.e., in the order they appear. However, a control statement allows us to control the program/code execution flow based on the result of the conditional statement known during run time. Now, we will describe two types of control statements supported by Java: **if-else** statements and **switch-case** statements.

## 2.2 if-Else Statement

The if statement is used to route the program execution path in two (or multiple) directions. Below, we present the primary form of the if statement:

```
if (Condition)
    Statement-1;
else
    Statement-2;
```

Here, the condition represents an expression that returns a boolean value. As described earlier, the program flow works like this: If the condition is true (recall that the condition returns a boolean value, i.e., true or false), statement-1 will execute. Otherwise, statement-2 will run.

### Example 2

Now, let us describe the example of John's work schedule by the if statement (shown below):

```
boolean isItRaining;
// Activity outside the condition
Wake up and have breakfast

if (isItRaining == TRUE) // Block-1
    Clean the house.
else // Block-2
    Go grocery shopping

// Activity outside the condition
Have lunch
```

Imagine a scenario like this. John wakes up and has his breakfast. He then goes to the window and pulls back the curtain to see if it is raining (assuming he has not checked the weather forecast yet). Initially, there was a question in his mind: Is it raining outside? However, after pulling back the curtain, he got the answer. The answer is either YES (true) or NO (false). Based on this answer, he will decide on his next task.

### Example 3

Consider an integer variable. If it is even, divide it in two. If it is odd, multiply it by 3, then add 1:

```
main() {
     int var = 47;
          // if the var is even
     if (var % 2 == 0 ){
          var = var/2;
     }

     // if the var is odd
     else {
          var = 3*var+1;
     }
}
```

This code initializes a variable var to 47 and checks whether it is even or odd using the modulus operator (%). The modulus operator returns the remainder of a division, so var % 2 == 0 checks if var is divisible by 2 (even). If it is even, var is divided by 2. Otherwise, if the var is odd, it is updated to 3 * var + 1. Since 47 is odd, the program computes var = 3 * 47 + 1 = 142 and stops after this calculation.

**Practice Problem**

- Write a Java program to find the maximum between two numbers using an if-else statement. The code should take two numbers from the user as input and find the maximum between them. Assume both these numbers are unique. Hint: To take input from the user, you can use the Scanner class in Java. For example:

```
Scanner scanner = new Scanner(System.in);
int number = scanner.nextInt(); // Reads an integer from the user
```

- Write a Java code to check if a number is positive or negative using if-else statements. The code should take two numbers from the user as input and determine if each is a positive or negative number. Assume that 0 is a positive number.

### 2.2.1 if-Else Statements With Multiple Operations

So far, we have described the basic structure, assuming that only a single operation is to be executed under the if statement. This restriction is not mandatory, and multiple operations can be performed under a single condition. If we choose to perform multiple operations, they must be enclosed in curly braces. Consider Example 4:

### Example 4

Reconsider Example 1 with some minor edits. Let's say that John wakes up at 7.00 AM and has his breakfast. If it rains, he will buy groceries on a different day and clean the house instead. In addition, he will call his mom. If it does not rain, he will buy groceries today and meet a friend (who lives on the way to the grocery store). Whatever he does—that is, clean the house and call his mom or buy groceries and meet his friend—he will have lunch after that. We can depict this scenario using the code below.

```
boolean isItRaining;
// Activity outside the condition
Wake up and have Breakfast;

if (isItRaining == TRUE) // Block-1
{
    Clean the house.
    Call Mom.
}
else // Block-2
{
    Go grocery shopping.
    Meet a friend on the way home.
}

// Activity outside the condition
Have lunch
```

Other than multiple actions, all other things are the same as depicted in Example 1. Notice the use of curly braces that enclosed the set of steps.

### Example 5

(Multiple operations inside branches) Consider a variable and print whether it is even or odd. If it is even, divide it in two. If it is odd, multiply it by 3, then add 1. In both cases, print the updated value of the variable.

```
main() {
int var = 47;

// if the var is even
if (var % 2 == 0 ){
    System.out,println("This is an even value");
    var = var/2;
}
```

```
// if the var is odd
else {
     system.out,println ("This is an odd value");
     var = 3*var+1;
}
system.out.prinln("Updated value of var is " + var);
}
```

---

Notice that we are doing multiple operations here inside the if-else block. First, we print a message (whether the number is even or odd). Second, we are performing a mathematical operation. Finally, we are printing the value of var irrespective of the conditions. Hence, we place the system.out.prinln("Updated value of var is " + var) statement outside any block.

### 2.2.2 if Without Else

After reviewing all these examples, the reader may be tempted to think that the Else clause is mandatory. This is not true. Let us consider the following Example 6 for better clarification.

**Example 6**

This time, consider a slightly different example. Let's say that John wakes up at 7.00 AM and has his breakfast. Now, he starts reading an article and plans to read it until noon, and then he will have lunch. The only thing he will do before lunch other than reading is drink water (if he is thirsty). We can depict this scenario using the code below. Here ,

- Start = Wake up and have breakfast
- Condition = Whether John is thirsty
- Block-1 = Drink water.
- Code block outside the condition = Have lunch.

In this example, we see that no else statement is associated with the if statement. John will drink water if he is thirsty (otherwise, he does not need to do anything). So, he would still have lunch at noon, whether he had drunk some water or not. We depict this scenario using the code below.

---

```
boolean isThirsty;
// Activity outside the condition
Wake up and have Breakfast;
```

```
if (isThirsty == TRUE){ // Block-1
     Drink water.
}
// Activity outside the condition
Have lunch
```

---

**Example 7**

Write a Java program that checks if an integer variable is negative or positive. If it is negative, then make it positive and print the updated value.

---

```
main() {
     int var = -47;

     // if the var is negative
     if (var < 0 ){
          var = -1 * var;
     }
     system.out.prinln("Updated value of var is " + var);
}
```

---

## 2.2.3 Relational and Logical Operators

We have gained some knowledge regarding the boolean condition and how it directs the program execution flow. We have examined only one condition inside the if statement in all these examples. These examples are often restricted, as we may need to evaluate multiple conditions simultaneously. Fortunately, we can test multiple boolean expressions by joining them with a relational and logical operator. We use a similar if-else statement to route the program execution path with two (or multiple) directions, with the only difference being that multiple boolean expressions are evaluated simultaneously. Below, we present the basic form of the of such statements:

---

```
if (Condition 1 # Condition 2 # Condition n)
     Statement-1;
else
     Statement-2;
```

---

Each condition may contain a relational operator, and # represents a logical operator; see the tables below. Each of these conditions represents an expression that returns a boolean value. Depending on the value of these expressions and the logical operators (i.e., #), the overall condition (i.e., `Condition 1 # Condition 2 # Condition n`) inside the if statement is either

true or false. If the overall condition is true, statement-1 will execute. Otherwise, statement-2 will run.

**Table 1: Relational Operator**

| Relational Operator | What it denotes |
|---|---|
| **>** | Greater than |
| **>=** | Greater than or equal to |
| **<** | Less than |
| **<=** | Less than or equal to |
| **==** | Equal to |
| **! =** | Not equal to |

A ***relational operator*** evaluates if a specific relationship between two values is true or false. For example, the less-than operator (<) determines whether one value is less than another. For reference, consider Example 7, where we check if the variable var is positive or negative (by comparing it with zero). For better understanding, the reader is encouraged to review the earlier examples. Similarly, the equality operator (==) determines if the values of two variables are equal. Now, let us talk a bit about the logical operator. A ***logical operator*** returns a boolean result evaluated based on the outcome of one, two, or more boolean expressions. Sometimes expressions that use logical operators are called "compound expressions." For example, the following if statement from Example 8 is a compound expression.

**Table 2: Logical Operator**

| Logical Operator | What it denotes | Description |
|---|---|---|
| **&&** | Logical AND | Returns true if all statements are true |
| **\|\|** | Logical OR | Returns true if at least one statement is true |
| **!** | Logical NOT | Reverse the result |

```
if (isItRaining == TRUE OR outsideTemp < 55)
    Clean the house.
```

Here, we see that two different boolean conditions are evaluated and joined via a logical operator (i.e., OR operator) to form one final expression. If one of these statements is true—that is, it is raining, or the outside temperature is below 55 degrees Fahrenheit—the overall expression will be evaluated as true. The reader may ask why the general statement becomes true when only one statement is

true. We will explain the answer via the truth table (Table 3). A truth table is used to check whether the compound expression is true or false, based on the input values. Below we provide the truth table for the logical AND (&&), logical OR (||), and the logical NOT (!) operator.

**Table 3: Truth Table**

| A | B | A&& B | A\|\|B | !A | !B |
|---|---|---|---|---|---|
| True | True | True | True | False | False |
| True | False | False | True | False | True |
| False | True | False | True | True | False |
| False | False | False | False | True | True |

Here A and B are boolean expressions, such as if it is raining outside, if the temperature is below 55F, and so forth. From the truth table, we see that one expression is true (false), logical OR (AND) returns the overall expression is true (false).

### ☛Concept Check

Consider the following code snippet:

```
if (isItSunny == TRUE AND 55 < outsideTemp < 80)
    Play Tennis.
else
    Watch a movie.
```

What you should do if:

- ✓The weather is sunny and the temperature is 91 degrees Fahrenheit.
- ✓The weather is sunny and the temperature is 71 degrees Fahrenheit.
- ✓It is raining outside.

### ☛Concept Check

Consider the following code snippet:

```
if (workFromHome == TRUE OR workFromOffice == TRUE or inBusinessTrip == TRUE)
    Get Salary.
else
    Do not get a salary.
```

What will happen if a person

✓ works from the office

✓ was on a business trip

If we express an if-else statement (with multiple boolean expressions), it will look like the following: As described earlier, the program control will evaluate all these conditions and determine a final evaluation as true or false. If the condition is true, block-1 will be executed. If else, block-2 will be executed.

**Example 8**

Now, let us describe this concept again with a real-life example. John wakes up at 7.00 AM and has his breakfast. Now, he wants to go out to buy groceries. However, he will check the weather before heading out. If it is raining or the temperature is below 55F, he will not go for groceries and will instead clean the house. Otherwise, he will buy groceries today. Whatever he does—clean the house or buy groceries—he will have lunch afterwards. We can depict this scenario by using the figure (Figure 3) presented below. Here,

- Start = Wake up and eat breakfast
- Condition 1 = Is it raining
- Condition 2 = Is the temperature below 55 degrees Fahrenheit
- Block-1 = Clean house.
- Block-2 = Go to the grocery store.
- Code block outside the condition = Have lunch.

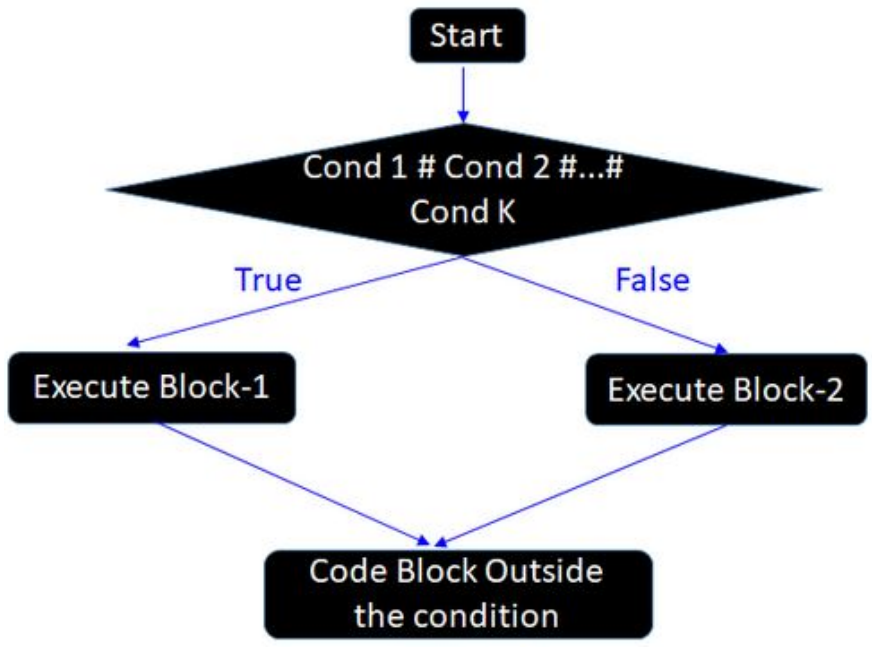


Figure 3: Example of an if-else statement with multiple boolean expressions.

We can describe the example of John's work schedule using the following code snippet.

```
boolean isItRaining, int outsideTemp ;
// Activity outside the condition
Wake up and have Breakfast

if (isItRaining == TRUE OR outsideTemp < 55) // Block-1
    Clean the house.
else // Block-2
    Go grocery shopping.

// Activity outside the condition
Have lunch
```

We already have seen how the truth table works. In this example, a logical OR statement joins two different boolean conditions. Hence, if any (or both) of these conditions are true, the whole expression becomes true. That is, if it rains outside, or it is sunny, but the temperature is below 55, or it is raining, and the temperature is below 55, John will clean the house.

**Example 9**

Consider a 24-hour time format and let an office run from 9.00 (i.e., 9.00 AM) to 16.00 (i.e., 4.00 PM) on weekdays. The following code snippet shows a Java program that takes hours and days as input (from the user) and determines whether the office is open or closed. For the sake of simplicity, we take all input as integers and represent a day with a number, i.e., Saturday = 0, Sunday = 1, . . . , Friday = 6).

We have used the onlinegdb compiler to execute this code snippet. It is a free online compiler and debugger that supports multiple programming languages, including but not limited to C, C++, Python, Java, etc. It offers an easy-to-use interface where users can write, compile, and debug code directly in their browser without installing additional software. It comes with features like code execution, syntax highlighting, and debugging tools, and benefits students and developers practicing coding online.

In Line 12, multiple boolean expressions are joined by the && operator. Each of these expressions contains a relational operator. According to the truth table, the overall expression will return true (and thus print "Office is Open") if all these expressions are true. Conversely, if one of the expressions is false, it will print "Office is closed".

```java
import java.util.Scanner;
public class Main{
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        int currentDay, timeInHour;

        System.out.println("Enter the Day:");
        currentDay = myObj.nextInt();
        System.out.println("Enter the Hour:");
        timeInHour = myObj.nextInt();

        if((timeInHour < 17) && (timeInHour > 8) && (currentDay <= 6)
        &&(currentDay > 1)){
            System.out.println("Office is Open");
        }
        else{
            System.out.println("Office is Close");
        }
    }
}
```

**Practice Problem**

Write a Java program that determines if a person is a resident of the US for tax purposes. A person is a resident for tax purposes if they meet one of the following criteria:

- the person is a citizen of the US
- the person is a green card holder in the US
- the person legally stayed in the US and paid taxes for at least five years

This program will ask the above questions and decide if this person is a resident for tax purposes. Then, it will print, “You are a resident for tax purposes.” Otherwise, it will print “You are not a resident yet.”

### 2.2.4 Nested if-Else Statement

Sometimes an if statement can be nested inside another if statement. Consider Example 9; here, we evaluate four boolean expressions to determine if the office is open. We can write this program in several different ways. For example, we need to check the office time for weekdays. In Table 4, first, we review the day and if it is a weekday, the condition in Line 14 is true. If not, we can say the office is closed, as we did on Line 22 (no need to check the time). Then, if this is

a weekday, we need to check the time (Line 15). Finally, we say that the office is open if this time also falls during office hours (Line 16). Otherwise, the office is closed (Line 19).

**Table 4: Example of a Java program with nested if-else statements**

| Line | Code |
|---|---|
| 1 | `import java.util.Scanner;` |
| 2 | `public class Main{` |
| 3 | `    public static void main(string[] args) {` |
| 4 | `        Scanner myObj = new Scanner(System.in);` |
| 5 | `        int currentDay, timeInhour;` |
| 6 | |
| 7 | `        System.out.printIn("Enter the Day: ");` |
| 8 | `        currentDay = myObj.nextInt();` |
| 9 | `        System.out.printIn("Enter the Hour: ");` |
| 10 | `        timeInhour = myObj.nextInt();` |
| 11 | |
| 12 | `        if((currentDay <= 6) && (currentDay > 1)){` |
| 13 | `            if((timeInHour < 17) && (timeInHour > 8))` |
| 14 | `                System.out.println("Office is Open");` |
| 15 | `            else` |
| 16 | `                System.out.printIn("Office if Closed");` |
| 17 | `        }` |
| 18 | `        else{` |
| 19 | `            System.out.printIn("Office is Closed");` |
| 20 | `        }` |
| 21 | `    }` |
| 22 | `}` |

### Example 10

Consider an insurance company providing each customer with various membership levels depending on the membership duration. If the customer has been a member of this company for less than two years, their level is basic. If they have been members for more than (or equal to) two years but less than five years, their status will be silver. If they have been members for more than (or equal to) five years but less than ten years, their status will be gold. At or after ten years, it will become platinum status. The following Java program will allow a user to enter the membership duration and then display the membership level. We will use a nested if-else statement to solve this problem.

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        int mem_Duration;
        System.out.println("Enter your membership duration:");
        mem_Duration = myObj.nextInt();

        if(mem_Duration >= 10)
            System.out.println("Platinum Status");
        else{
            if(mem_Duration >= 5)
                System.out.println("Gold Status");
            else{
                if(mem_Duration >= 2)
                    System.out.println("Silver Status");
                else
                    System.out.println("Basic Status");
            }
        }
    }
}
```

**Practice Problem**

Assume the following grading rubrics for a test.

(A+)=(93-100), A=(88-92), (B+)=(79-82),

(C+)=(72-78), C=(66-71), D=(60-65), F=<60

Write a Java program that will allow a user to enter a test score and then display the grade for that score. Use a nested if-else statement to solve this problem.

## 2.2.5 the If-Else if-Else Statement

So far, we have examined several conditions with if-else statements including nested if-else ones or only if blocks. We can test a series of conditions with a set of nested if-else statements. However, it is often simpler to use the if-else-if statement than a nested if-else statement. The general format of an if-else if chain looks like this:

```
if( boolean Expression 1){
    Statement 1;
    Statement 2, etc;
}
```

```
// requires an IF statement above
else if( boolean Expression 2){
     Statement 1;
     Statement 2, etc;
}
else if( boolean Expression 3){
     Statement 1;
     Statement 2, etc;
}
// insert more else-if statement if necessary
else {
     Statement 1;
     Statement 2, etc;
}
```

---

The boolean expressions inside if/else if blocks must be unique. The code starts by evaluating boolean expression 1. If expression 1 is true, the program control will immediately execute all the statements inside the if block. In contrast, it will ignore the rest of the else if (and else) blocks. If expression 1 is false, the code evaluates the next else-if block (i.e., expression 2). Again, if it is true, all the statements inside the else-if block will be executed immediately, and the rest of the blocks will be ignored. This process continues until one of the expressions is true. If none of the expressions are true, by default, the ending else clause will execute. Note that the end else clause is optional, but helpful in most cases.

**Example 11**

We will solve the same problem here as shown in Example 10. Refer to the snapshot given below. In this code snippet, if the user enters something less than two years, the if block will execute, and the code will be basic status. Also, the code will skip the rest of the conditions (else-if, else statements). Suppose the user provides something greater or equal to two but less than five. In this case, the code will print ”silver status” and skip the remaining conditions. If the user provides something so that all the conditions become false, the else statement will execute by default.

---

```
import java.util.Scanner;
public class Main{
     public static void main(String[] args) {
          Scanner myObj = new Scanner(System.in);
          int mem_Duration;

          System.out.println("Enter your membership duration:”);
          mem_Duration = myObj.nextInt();
```

```
        if(mem_Duration < 2)
            System.out.println("Basic Status");
        else if((mem_Duration >= 2) && (mem_Duration < 5))
            System.out.println("Silver Status");
        else if((mem_Duration >= 5) && (mem_Duration < 10))
            System.out.println("Gold Status");
        else
            System.out.println("Platinum Status");
    }
}
```

---

## Practice Problem

- Write a code that takes the lengths of a triangle's three sides as input from the user. Assume all these values are integers. Next, you need to check if an input is valid or not (0 or negative value is invalid). Also, the sum of any two sides must be greater than the third. Now, determine whether it is an equilateral, isosceles, or scalene triangle. Note that an equilateral triangle is a triangle in which all three sides are equal. A scalene triangle is a triangle that has three unequal sides. Finally, an isosceles triangle is a triangle with (at least) two equal sides.
- Write a Java program that takes a random string that may contain letters, numbers, and alphanumeric characters. Then, write a Java program that will count the number of letters, numbers, and alphanumeric characters in the string. For example, if the user provides a string BN!X254@0%QK$#", the code will print the following:
    - THIS STRING HAS FIVE LETTERS, FOUR NUMBERS, AND FIVE ALPHANUMERIC CHARACTERS.
- Write a Java program that does the following: (i) Take electricity consumption in KWH as input (from the user) (ii) Calculate the total electricity bill according to the following given conditions:
    - For first 75 units, 14 cents/unit
    - For next 125 units, 18 cents/unit
    - For next 100 units, 23 cents/unit
    - Anything beyond these units, 30 cents/unit. An additional surcharge of 18% will be added to the bill.

**Example 12**

While writing code using the if/else statement, we need to pay attention to using the = and == operators. The former is referred to as an assignment operator and is used to assign a value to a variable. Meanwhile, the latter is a relational operator used to find the equality between operands. Let us consider an example where a program takes two integer variables (say firstVar and secVar) as input from the user. If the firstVar is equal to the secVar, it will print that "both of them are equal." Otherwise, it will print that "they are not equal." Consider the following code snippet. We present it as an exercise to test the output and justify the reason.

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        int firstVar, secVar;
        firstVar = myObj.nextInt();
        secVar = myObj.nextInt();

        if(firstVar = secVar)
            System.out.println("They are Equal");
        else
            System.out.println("They are not Equal");
    }
}
```

## 2.3 Switch-Case Statement

Like the if-else ladder, the switch-case statement executes one statement from multiple options. However, in such a statement, the value of a specific variable is tested against various values. Consider the following program that will take a month as an integer: i.e., January = 1, February = 2, …, December = 12. If the user provides 1, it will print "January." If the user provides 2, it will print "February." We write this program using an if-else statement.

```
System.out.print("Enter a month 1-12: ");
int month = kb.nextInt();
if (month == 1)
    System.out.println("January");
else if (month == 2)
    System.out.println("February");
...
else if (month == 12)
```

```
    System.out.println(“December");
```

---

Now, we will write the same program using a switch statement. While writing a switch-case statement, we need to consider some essential points. They are mentioned below:

- Keyword switch (an expression here, usually a variable with a value).
- Opening curly brace.
- Keyword case with a set of values. The expression in the parentheses (after the switch statement) is compared to these values.
- Block of codes.
- Break; (typically after each case)
- Ending curly brace

---

```
switch (month){
    case 1:
        System.out.println("January");
        break;
    case 2:
        System.out.println("February");
        break;
    ...
    case 12:
        System.out.println(“December");
        break;
}
```

---

### 2.3.1 Program Flow

The program starts by evaluating the switch expression. Then the value of the expression is compared with each of the cases. Next, the control flow enters the code block written with a specific case when it finds a case that equals the expression (written beside the switch statement). Finally, the control flow enters the code block written with a specific case when it finds a case that equals the expression (written beside the switch statement). Once this block is executed, the break statement jumps control past the rest of the cases, i.e., the rest of the cases are ignored. The switch-case statement is similar to if/else-if statements.

However, it is crucial to handle the else statement. Consider an example code where the user will enter a month. The code will print the season in that month, e.g., December-February is winter, March-May is spring, and so forth. Any input other than a month (such as Australia, Mango, Ocean, Linda, @%1yy05L4###, etc.) is invalid. We can write the following if/else-if/else block to write this program. In the following code snippet, if/else-if blocks handle all the valid scenarios. In contrast, the else block takes care of the rest; that is, if the user provides input other than the name of twelve months, the code inside the else block will execute.

```
if (month.equals("January"))
    //..... Do Something;
else if (month.equals("February"))

    //..... Do Something;
....
else
    // Invlid Month
```

In a switch-case statement, the default keyword handles these scenarios. Like the else block (if/else construct), if none of the if statements are evaluated as true, the program flow will execute the default statement. See the code below as an example. For simplicity, we denote each month numerically, i.e., `January = 1`, `February = 2`, ... `December = 12`, and so on.

```
switch (month)
{
    case 1:
        System.out.println("January");
        break;
    case 2:
        System.out.println("February");
        break;
...
    case 12:
        System.out.println("December");
        break;
    default:
        System.out.println("Invalid Month");
        break;
}
```

## Example 13

Let's say that we want to check if it is summer in your region. For example, the following code snippet will print "Summer" if it is May, June, or July. For simplicity, we denote each month numerically, i.e., January = 1, February = 2, ... December = 12, and so on.

```
Scanner myObj = new Scanner (System.in);
int month;

System.out.println ("Enter the Month: ");
month = myObj.nextInt ();

switch (month)
{
     case 5:
     case 6:
     case 7:
          System.out.println("Summer");
          break;
}
```

## Example 14

The "break" statement is essential for many reasons. See the code below. If the user enters 1 (i.e., January), it will print January, February, March, and Invalid Month.

```
Scanner myObj = new Scanner (System.in);
int month;

System.out.println ("Enter the Month: ");
month = myObj.nextInt ();

switch (month)
{
     case 1:
          System.out.println("January");
     case 2:
          System.out.println("February"); ...
     case 3:
          System.out.println("March");
     Default:
          System.out.println("Invalid Month");
}
```

### 2.3.2 Advantage and Disadvantage of Using Switch-Case (Compared to If-Else Statement)

A switch-case statement is more aesthetic when a logical OR joins several conditions. Compare the following two code snippets.

```
switch (number)
{
     case 1:
     case 4:
     case 7:
          case 10:
     case 15:
     case 21:
               System.out.println("Do Something");
}
```

Below is how the same code will look if we write them using an if-else construct.

```
if (number == 1 || number == 4 || number == 7 || number == 10 ||
number == 15 || number == 21){
     System.out.println("Do Something");
}
```

However, a switch-case statement has limitations. It cannot express relational operators like <, >, ≤, ≥, =, etc. Also, the strange syntax makes the code error-prone—i.e., missing the break statement may lead to unexpected outcomes.

## 2.4 Loops

Did you ever get punished for forgetting to do your homework in middle school? For example, being asked to write "I will complete my homework on time" 100 times? Boring and tedious, isn't it? Instead of your using pen and paper, if the teacher asks you to write this sentence using Java programming, then the statement below will do this job:

```
System.out.print("I will complete my homework on time");
```

But how can we print the same statement 100 times or more? Obviously, using tons of print statements is not useful and will not be feasible in many cases. Yet, there are situations when we need to execute a statement or set of statements several times. In general, running these statements is very complicated and time-consuming. Hence, the Java programming language provides a feature to

handle such complex execution statements as loops. Loops allow a set of statements, instructions, or code blocks to execute repeatedly until a particular condition is met.

### 2.4.1 Java Loops and Syntax

In Java, three types of loops are mainly used—the while loop, the for loop, and the do-while loop. Although the syntax slightly varies, all of them have the same purpose. They execute a set of statements repeatedly until a specified condition becomes false. Typically, a variable in the loop governs the loop execution. Generally, we can describe a loop using the following four elements:

- **Initialize a control variable**. This is the first part, which is performed before entering the loop. Here, we initialize the control variable with an initial value; this initialization is executed only once at the beginning.
- **boolean condition (or conditions)**. The boolean condition (typically involves the control variable) decides whether the control will enter the loop body or not. If the condition is true, the loop body will execute. Meanwhile, if it is false, the loop will terminate, which is also known as the exit condition.
- **Update the control variable (mentioned in the first step)**. This expression, often an increment or decrement statement, updates the control variables. Typically, the updated expression executes after the loop body is executed.
- **Loop body**. The loop body contains a statement or set of statements. These statements repeatedly execute if the boolean expression is true.

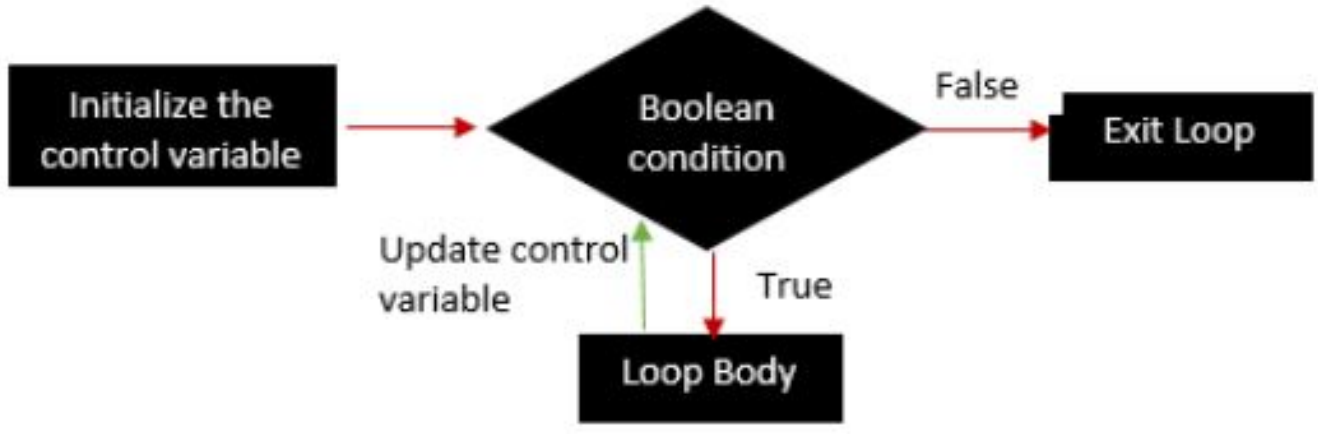


Figure 4: Structure of a loop in Java.

In a nutshell, the loop works as follows. First, we initialize a control variable. Second, we use this variable in a boolean expression. If the boolean expression is true, the control enters the loop body. Next, it executes the statement(s)

written inside the loop body. Third, we update the control variable and return to the second step. If the expression is still valid, it enters the loop body and then goes to the third step again. These steps continue until the boolean expression becomes false. We can express a loop by the following diagram:

### 2.4.2 the While Loop

The general syntax of a while loop looks is depicted below:

```
while(boolean expression){
    Loop-body
}
```

Details on how the loop works have already been mentioned above. However, this general syntax does not depict the whole scenario:—that is, it does not show the role of the control variable. We present the following examples to demonstrate how the while loop works.

**Example 15**

In this example, we write a Java program that prints all numbers from 1 till 500.

```
int count = 1; // control variable initialization
while(count <= 500) // boolean expression
{// loop body starts
    System.out.println(count);
    count = count + 1; // update control variable
}
```

In this example, we initialize the control variable (i.e., count) to 1 and provide the boolean expression in the while loop. Suppose the test expression or condition is true. In that case, the program control enters the loop body, where the control variable is also updated. Eventually, the boolean expression will become false (when the count variable becomes 501). Then the loop will break, and the program control passes next to the loop’s body.

**☛Concept Check**

What will happen if we do not update the control variable, i.e., skip the COUNT++ statement?

### Example 16

Now, we will write a Java program that prints all the even numbers from 0 till 500.

```
int count = 0; // control variable initialization
while(count <= 500) // boolean expression
{// loop body starts
     System.out.println(count);
     count = count + 2; // update control variable
}
```

This example is almost the same as Example 15, other than the change we make in the control variable initialization and update. We want to print all the even numbers from 0-500. Hence, we initialize the variable to 0 and increase it by two at each iteration until the boolean expression becomes false.

### Example 17

In this example, we write a Java program that prints the sum of all numbers from 1 till 500.

```
int count = 1, sum = 0;
while(count <= 500){
     sum = sum + count;
     count = count + 1; // update control variable
}
System.out.println(sum);
```

In this example, we declare an extra variable (sum) and initialize it to zero. Then, we use the count variable to iterate through all numbers from 1-500 and add them to the sum. When the loops break, i.e., the boolean expression becomes false, the code control goes outside the loop body and prints the value of the variable sum.

### ☛Concept Check

What will happen if:

1. We replace count <= 500 by count < 500?
2. We do not initialize the variable sum?
3. We initialize the variable sum with a different value other than ZERO?

Scope of a Variable in a Loop. Let us declare a variable inside the loop. In this case, the scope of this variable is within the loop body. Hence, if the loop is terminated, we cannot access the variable outside the loop body.

**Example 18**

Consider the following code snippet. The System.out.println() statement outside the loop body is invalid in this code. As discussed above, the scope of a variable is only inside the loop body, and a variable is not accessible outside its scope. Hence, accessing the variable outside its scope is causing an error.

```
public class Main{
    public static void main(String[] args) {
        int count = 1;
        while(count <= 5){
            int var = count * 10;
            count++;
        }
        System.out.println("Value of var: " + var);
    }
}
```

When compiled, this code generates the following error message.

```
Main.java:17: error: cannot find symbol
System.out.println("Value of var: " + var);
symbol: variable var
location: class Main
1 error
```

## 2.4.3 the For Loop

The general syntax of a for loop looks like the statement below:

```
for(initialization;boolean expression;update expression)
{
    Loop-body
}
```

Unlike the while loop, the for loop allows for initialization and updating of the control variable inside the parentheses. Let us explain by using the same example that we used in the while loop.

### Example 19

We repeat Example 15 and print all numbers from 1 to 500 using a for loop.

```
// control variable initialization, boolean expression, and variable update
for(int count = 1; count <= 500; count++){
    // loop body starts
    System.out.println(count);
}
```

In this example, we both initialize and update the control variable (i.e., count) inside the parenthesis. The initialization is executed only once at the beginning. Then, the boolean expression is evaluated. If it is true, the program control enters the loop body. Unlike the while loop, the program control goes back to the parentheses, where the control variable is updated. The loop will break when the boolean expression becomes false.

### Example 20

Write a Java program that prints the sum of all numbers from 1 until 500, using for loop.

```
int count, sum = 0;
for(count = 1; count <= 500; count++){
    sum = sum + count;
}
System.out.println(sum);
```

### ☛Concept Check

Consider the following code snippet:

```
for(int count = 1; count <= 15; count++){
    System.out.println("Value of count inside loop: " + count);
}
System.out.println("Value of count outside the loop: " + count);
```

What will happen if we execute this code? Can you justify the output?

**Multiple Initialization and Update Statements.** A for loop can contain multiple initialization and update statements, separated by commas. For example, consider the following code snippet:

```
int i, sum;
for( i = 1, sum = 0 ; i <= 10 ; sum +=i, ++i )
    System.out.println(i);
```

In this code snippet, we write two initialization expressions (i.e., i = 1 and sum = 0) and two update expressions (i.e., sum += i and ++i). As usual, these statements execute sequentially.

**Empty Loop**. In Java programming, we also can write an empty loop that contains no statement in the loop body. Consider the following example of an empty loop:

```
int counter;
// The following loop has nothing in the loop-body
for (counter = 2000; counter >=0; counter—)
```

An empty loop is often used in applications waiting for something or needing some delay.

### 2.4.4 the Do-While Loop

The general syntax of a do-while loop is depicted below:

```
do{
    Loop-body;
} while(boolean expression);
```

It is very similar to the syntax of a while loop, other than the following differences:

- Involves two keywords
- ; after the boolean expression

Like before, the loop body is repeated until the boolean expression evaluates as false. However, contrary to the while loop and the for loop, the do-while loop will execute at least once. In a do-while loop, first, the loop body is executed. Then, it evaluates the boolean expression. Based on these characteristics, the while loop and for loop are known as the pretest loops, and the do-while loop is a posttest loop.

**Example 21**

Write a Java program using a do-while loop that prints all numbers from 1 until 500.

```
int count = 1; // control variable initialization
do{// loop body starts
    system.out.println(count);
    count++; // update control variable
} while(count <= 500); // boolean expression
```

Unlike the while loop, the loop body will execute at least once. Then, it will continuously evaluate the boolean expression, and the loop will break when the expression becomes false (when the count variable becomes 501).

**☛Concept Check**

Can you write a Java program where a pretest (while, for) and posttest (do-while) loop does not generate the same result?

## 2.4.5 Nested Loops

A loop that is inside another loop is called a nested loop. For example, consider the following code snippet. Here, we present the skeletons of three nested loops: i.e., a while loop nested in a while loop, a for loop nested in a for loop, and a for loop nested in a while loop.

```
// Nested Loop-1
while(boolean expression){
    //body of the outer loop
    while(boolean expression){
        //body of the inner loop
    }
}
// Nested Loop-2
for(initialization;boolean expression;update statement){
    //body of the outer loop
    for(initialization;boolean expression;update statement){
        //body of the inner loop
    }
}
// Nested Loop-3
for(initialization;boolean expression;update statement){
    //body of the outer loop
    while(boolean expression){
```

```
        //body of the inner loop
    }
}
```

---

A nested loop may seem tricky at the beginning. However, while writing a nested loop, keep in mind:

- For each iteration of an outer loop, the inner loop goes through all of its iterations.
- The inner loops must complete their iterations before the outer loop.
- Suppose we multiply the number of iterations of all the loops. In that case, we will find the total number of iterations of a nested loop.
- More than two loops can be nested.

The following code snippet prints the value of variable j (from one to three) for each value of variable i (from one to two).

---

```
for (int i = 1; i <= 2; i++){
    for (int j = 1; j <= 3; j++){
        System.out.println(i + " " + j);
    }
}
/* Output: 1 1
           1 2
           1 3
           2 1
           2 2
           2 3
*/
```

---

### ☛Concept Check

- Consider the code snippet provided above. Can you generate the same output, but using a nested while loop?
- Consider the following code snippet. What will be the output if we set the value of numRows to 5?

---

```
System.out.print("Enter number of rows: ");
int numRows = kb.nextInt();

for (int rowNum = 1; rowNum <= numRows; rowNum++){
```

```
    for (int colNum = 1; colNum <= rowNum; colNum++)
        System.out.print("*");
    System.out.println();
}
```

---

## 2.5 Exercise

1. Let, x = 10, y = 8, z = 18. Determine the values (TRUE or FALSE) of the following expressions:

```
A. (x >= y) || (y <= 10)
B. (y <= z) || !(x == 10)
C. (x >= z) && (y != 10) && !(x == y)
D. !((x >= y) || (y <= 10)) && (y != 10)
```

2. Write a Java program that takes the length and width of a rectangle from the user and checks if it is square or not.

3. Write a Java program that takes a character as input and check whether it is a lowercase (a to z) or uppercase (A to Z) letter or a non-letter (e.g., 8, #, *, etc.)

4. Write a Java program that takes the salary of 3 people as input from the user. Now, determine the highest, lowest, and average salary. If all of them have the same income, print “all of them have the same salary.”

5. Write a Java program that takes an input of an integer number and then checks whether this number is divisible by 4, 6, and 10, or some of them or none of them. If multiple of these numbers, divide your number; you must mention all of them.

6. Professor Simeone adopts a policy in his CSC 141 course. He will not allow a student to sit for the final exam if his/her attendance is below 80%. Write a Java program that takes the number of classes held and the number of classes attended by a particular student. Then, print the percentage of classes attended by that student. Also, print if the student is allowed to sit for the final exam or not.

7. In algebra, a quadratic equation is expressed in the following form:

$$ax^2 + bx + c = 0$$

Here, a, b, and c are the coefficients of the equation. Write a Java program that takes the value of a, b, and c as input from the user and finds all roots of a quadratic equation.

8. The dates for each season in the northern hemisphere are as follows:

```
A. Spring: March 20 - June 20
B. Summer: June 21 - September 21
C. Autumn: September 22 - December 20
D. Winter: December 21 - March 19
```

   Write a Java program that takes a date as input and outputs the date's season in the northern hemisphere. The input is a string representing the month and an int representing the day. First, you must check if the string and int are valid (an actual month and day). For example, `July 23` is valid, while `March 53` or `Covid 19` is invalid.

9. Write a Java program that takes electricity consumption in KWH as input (from the user), and calculate the total electricity bill according to the given condition:

```
A. 0 - 50 units: 10 cents/unit
B. 51 - 150 units: 16 cents/unit
C. 151 - 250 units: 22 cents/unit
D. 251 unit or higher: 30 cents/unit
```

   If the total consumption is less than 251 KWH, an additional surcharge of $7 is added to the bill. Otherwise, the surcharge amount will be $12.

10. Write a Java program to create a simple calculator. This calculator will perform the add, subtract, multiply, divide, modulus, and exponent operations. The program should take two numbers and an operator as input from the user. Then, it will operate according to the operator entered. The input must be provided in the following format and any other format should be considered as invalid.

```
A. number 1 <operator> number 2
```

11. Write an infinite loop using the while loop, the for loop, and the do-while loop.

12. Write a Java program that takes a string as input and counts the frequency of each character in this string.

13. Write a Java program that takes an integer as input from the user. Now it should calculate the sum and product of all input digits. For example, if the user inputs 5143, the sum of digits is $(5+1+4+3=13)$ and the product of digits is $(5\times1\times4\times3=60)$.

14. Write a Java program that takes an integer as input and reverses the input digits. Let the input is 51430, then the output will be 3415.

15. A palindrome is a word, number, phrase, or other sequence of characters that reads the same backward as forward, such as madam, racecar, 1110111, etc. Write a Java program that takes a sequence of characters as input and determines if these characters form a palindrome.

16. In mathematics, the greatest common divisor (GCD) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For two integers x and y, the greatest common divisor of x and y is denoted GCD (x , y). For example, the GCD of 8 and 12 is 4, that is, GCD (8 , 12) = 4 [Source: Wikipedia]. Write a Java program that takes two non-zero integers as input and calculate their GCD.

17. The Least Common Multiple (LCM) of two or more numbers is the smallest number that is divisible by all these numbers. For example, assume two integers, “a” and “b.” Their LCM is denoted as LCM (a,b). If a = 12 and b = 8, LCM (12,8) = 24. Write a Java program that takes non-zero integers as input and calculate their LCM.

18. Repeat the same problem as mentioned above in No. 17. This time, take three non-zero integers as input and calculate their LCM.

19. In the Fibonacci series, each number (other than the first two) is the sum of the previous two numbers. Here, the first two numbers are 0 and 1. So, the series looks like the one below:

```
A.0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

    Write a Java program that takes an integer n as input and calculates the nth Fibonacci number.

# 3 String

## 3.1 What Is String?

Simply put, a string is a sequence of characters. In Java programming the String class represents the character strings, and all string literals are implemented as an instance of the String class.

## 3.2 How to Declare String

In Java, string variables hold the value of strings. Please note that the “String” datatype starts with the capital letter “S”, and not small letter “s”. Example 1 shows how to declare a string variable.

```
String s1 = "Hello Class!!”;
```

**Example 1**

The example shows how to declare a string variable and assign value to it.

```
public class StringDeclaration{
    public static void main(String args[]){
        // declare s1 string variable and assign value "Hello Class!!”
        String s1 = "Hello Class!!”;
        // print the value of string variable s1
        System.out.println(s1);
    }
}
```

The output of Example 1:

```
Hello Class!!
```

## 3.3 String Input

The Scanner class is used in Java to take user input(s). The Scanner class is part of java.util package. See in example 2, a Scanner object input is declared, and to access the Scanner class the java.util.Scanner package is imported to access the Scanner class. Notice that the Scanner object input.nextLine() allows user input, which is assigned to a String variable inputString and later displayed using the System.out.println(inputString).

**Example 2**

The example shows how to declare a string variable and assign value to it.

```
import java.util.Scanner;
public class KeyboardInput {
    public static void main (String[] args) {
        // create a scanner object
        Scanner input = new Scanner(System.in);
        // use the scanner object to read line from keyborad
        String inputString = input.nextLine();
        // print the input string
        System.out.println(inputString);
    }
}
```

The output of example 2:

```
Hello Class!!
```

# 3.4 Helpful String Methods

- `length()`
- `toUpperCase()`
- `toLowerCase()`
- `charAt(int index)`
- `substring(int start, int end)`
- `indexOf(String str)`

## 3.4.1 String Length() Method

This method returns the length of a string as an int value. The method counts the number of unicode characters in a string and returns the count value.

**Example 3**

The example shows how to use length() method to find the length of a string variable.

```
public class LengthExample {
    public static void main(String args[]) {
        // declare a string variable s1 and assign "Hello Class!!"
        String s1 = "Hello Class!!";
        // print the length of s1 variable
        System.out.println(s1.length());
    }
}
```

The output of example 3:

```
13
```

When we create a string variable by using the String constructor. As shown in the example above (see example 3).

### 3.4.2 String toUpperCase() Method

The toUpperCase() method helps to convert a string variable's values to upper case letters.

**Example 4**

The example shows how to use toUpperCase() method to convert all characters of a string variable to uppercase letters.

```
public class ToUpperCaseExample {
    public static void main(String args[]) {
        // declare a string variable and initialize with "Hello"
        String s1 = new String("Hello");
        // print the variable s1
        System.out.println(s1);
        // print the variable after converting to all upper case letters
        System.out.println(s1.toUpperCase());
    }
}
```

The output of example 4:

```
Hello
HELLO
```

### 3.4.3 String toLowerCase() Method

The toLowerCase() method helps to convert a string variable's values to lowercase letters.

**Example 5**

The example shows how to use the toLowerCase() method to convert all characters of a string variable to lowercase letters.

```
public class ToLowerCaseExample {
    public static void main(String args[]) {
        // declare a string variable and initialize with "Hello"
        String s1 = new String("Hello");
        // print the variable s1
        System.out.println(s1);
        // print the variable after converting to all upper case letters
        System.out.println(s1.toLowerCase());
    }
}
```

The output of example 5:

```
Hello
hello
```

### 3.4.4 String charAt() Method

The charAt() method returns a character from a string variable using the position or the index of the character of the string variable.

Syntax for charAt() method

```
public char charAt(int index)
```

**Example 6**

The example shows how to use charAt() method to get a character from a string variable using the index of the character.

```
public class CharAtExample{
    public static void main(String args[]){
        // declare a string variable and initialize with "Hello"
        String s1="Hello";
        // declare a character variable and initialize that with
        // 5th character of the s1 string
```

```
        char c1=s1.charAt(4);
        System.out.println(c1);

        // declare another character variable and initialize that with
        // 1st character of the s1 string
        char c2=s1.charAt(0);
        System.out.println(c2);
    }
}
```

The output of example 6:

```
o
H
```

## 3.4.5 String Substring() Method

The substring() method returns a string from a string variable using the beginning index or beginning and ending index of the string variable.

Syntax for substring() method

```
public String substring(int start)
```

and

```
public String substring(int start, int end)
```

### Example 7

The example shows how to use substring() method to get a string from a string variable using the index of the string.

```
public class SubstringExample{
    public static void main(String args[]){
        // declare a string variable and initialize with "Hello class!!”
        String s1="Hello class!!”;
        //substring(beginningIndex)
        System.out.println(s1.substring(6));
        // substring(beginningIndex, endIndex)
        System.out.println(s1.substring(0,5));
    }
}
```

The output of example 7:

```
class
Hello
```

### 3.4.6 String indexOf() Method

The indexOf() method returns the position as an integer value of the first occurrence of specified character(s) in a string.

Syntax for indexOf() method

```
public int indexOf(String str)
```

**Example 8**

The example shows how to use indexOf() method to get the index of the substring within a string variable.

```
public class IndexOfExample{
    public static void main(String args[]){
        // declare a string variable and initialize with "Hello class!!,
        hello class!!”
        String s1="Hello class!!, hello class!!”;
        // find the index of "class"
        System.out.println(s1.indexOf("class"));
        // find the index of "Hello"
        System.out.println(s1.indexOf("Hello"));
    }
}
```

The output of example 8:

```
6
0
```

## 3.5 Chaining Method Call

In Java, method chaining is the chain of methods being called at the same time. In other words, a single object is used to call multiple methods in a single statement.

### Example 9

The example shows how to call multiple methods at the same time in Java. This example utilizes concept of class, constructor, setter, and method which is discussed in detail in chapter 4 and 6.

```
public class ChainingMethodCallExample {
    // private member variables
    private String name;
    private int age;
    // set name method
    public ChainingMethodCallExample setName(String name) {
        this.name = name;
        return this;
    }
    // set age method
    public ChainingMethodCallExample setAge(int age) {
        this.age = age;
        return this;
    }
    public void getChainingMethodCallExampleDetails() {
        // print information
        System.out.println("Person name is " + name + " and " + age + " years
        old.");
    }

    public static void main(String[] args) {
        // create an object
        ChainingMethodCallExample person= new ChainingMethodCallExample();
        // call multiple methods
        person.setName("John").setAge(22).getChainingMethodCallExampleDetails(
        );
    }
}
```

The output of example 9:

```
Person name is John and 22 years old.
```

# 3.6 String Operations

## 3.6.1 Concat

In Java, the concat() method helps to concatenate strings. In the example below, "s1" and "s2" is concatenated using the concat() method, and the resulting string put back on "s1" (see Example 10).

### Example 10

The example shows how to use concat() method to join two string variables.

```
public class StringConcat{
    public static void main(String args[]){
        // declare s1 string variable and assign value "Hello"
        String s1 = "Hello";
        // declare s1 string variable and assign value " Class!"
        String s2 = " Class!";
        // concat s1 and s2 and assign the value to s1
        s1 = s1.concat(s2);
        // print the value of string variable s1
        System.out.println(s1);
    }
}
```

The output of example 10:

```
Hello Class!!
```

## 3.6.2 Compare String Variable

Often a string may be compared with another. Java allows programmers to write code to compare strings. There are different ways to compare strings in Java, such as

- Using `equals()` method
- Using `compareTo()` method

**String equals() method** In Java, uppercase letters and lowercase letters are not the same. So, if a string variable value is ”S” and another is ”s”, then comparing them will show that they are not the same. This means variables ”s1” and ”s2” are not same or not equal (see Example 11),

```
String s1 = "S"
```

and

```
String s2 = "s"
```

### Example 11

The example shows how to use ”equals” method to compare string variables.

```
public class TestStringComparison{
    public static void main(String args[]){
        // declare s1 string variable and assign value "S"
        String s1 = "S";
        // declare s2 string variable and assign value "s"
        String s2 = "s";
        // compare s1 and s2
        System.out.println(s1.equals(s2));//false
    }
}
```

The output of example 11:

```
false
```

### Example 12

The example shows how to use ”equals” method to compare string variables.

```
public class TestStringComparison{
    public static void main(String args[]){
        // declare s1 string variable and assign value "John"
        String s1 = "John";
        // declare s2 string variable and assign value "John"
        String s2 = "John";
        // declare s3 string variable and assign value "John"
        String s3 = new String("John");
        // declare s4 string variable and assign value "Mike"
        String s4="Mike";
        System.out.println(s1.equals(s2));//true
        System.out.println(s1.equals(s3));//true
        System.out.println(s1.equals(s4));//false
    }
}
```

The output of example 12:

```
true
true
false
```

Note that if a string variable is ”John” and another is ”john”, then equals() method will return false since ”John” and ”john” contain characters in different cases and therefore are different variables. However, if you want to ignore the case, then you can use equalsIgnoreCase().

### Example 13

The example shows how to use ”equalsIgnoreCase” method to compare string variables.

```
public class TestStringComparison{
    public static void main(String args[]){
        // declare s1 string variable and assign value "John"
        String s1 = "John";
        // declare s2 string variable and assign value "John"
        String s2 = "John";
        // declare s3 string variable and assign value "John"
        String s3 = "john";
        System.out.println(s1.equals(s2));//true
        System.out.println(s1.equals(s3));//false
        System.out.println(s1.equalsIgnoreCase(s3));//true
    }
}
```

The output of example 13:

```
true
false
true
```

**String compareTo() method**. The ”compareTo()” methods can help to determine if two strings are equal. If strings are equal, then the method will return 0, otherwise a nonzero value. Please note that the compareTo() method will return a nonzero value, i.e., either a positive or a negative value depending on which string is larger than the other (see Example 14).

### Example 14

The example shows how to use ”compareTo” method to compare string variables.

```
public class TestStringCompareTo{
    public static void main(String args[]){
        // declare s1 string variable and assign value "John"
        String s1 = "John";
        // declare s2 string variable and assign value "John"
        String s2 = "John";

        // declare s3 string variable and assign value "Mike"
        String s3="Mike";
        System.out.println(s1.compareTo(s2));//0
        System.out.println(s1.compareTo(s3));//-3
```

```
        System.out.println(s3.compareTo(s1));//3
    }
}
```

---

The output of example 14:

```
0
-3
3
```

Similar to equalsIgnoreCase(), if you want to ignore case and compare two strings, then compareToIgnoreCase() method should be used.

**Example 15**

The example shows how to use ”compareToIgnoreCase” method to compare string variables.

---

```
public class TestStringCompareTo{
    public static void main(String args[]){
        // declare s1 string variable and assign value "John"
        String s1 = "John";
        // declare s2 string variable and assign value "john"
        String s2 = "john";
        System.out.println(s1.compareTo(s2));//-32
        System.out.println(s1.compareToIgnoreCase(s2));//0
    }
}
```

---

The output of example 15:

```
-32
0
```

## 3.7 Exercise

1. What is a string in Java? Please provide an example.
2. What is the difference between a character and a string variable in Java? How do you declare a character variable and a string variable?
3. Write a program to take two string inputs from keyboard and contact them; display the result.
4. Write a program to find the length of the following string. Make sure to ignore the white space when computing the length of the string.

```
Hello Class
Hello, My name is John!
```

5. What is the difference between equals() method and equalsIgnoreCase() method? Explain your answer using examples.

6. What is the difference between compareTo() method and compareToIgnoreCase() method? Explain your answer using examples.

7. Write a program to take user input as string and display the input values in uppercase letters.

8. Write a program to take user input as string and display the input values in lowercase letters.

9. Write a program that takes two string inputs from the keyboard and displays TRUE if the second string value exists in the first input.

10. Write a program that takes two string inputs from the keyboard and displays an index if the second string value exists in the first input.

11. Write a program that will take a string as inputs and print the most frequently appearing character (in this string) and its frequency. For example, if you enter ”AliBaba”, the output is ”A:3”.

12. Write a program that will take a string as input and switch the cases of the letters (lowercase to uppercase and vice versa) without using built-in methods. For example, if the user provides hgFh@gbBBT5e54& as input, the method will convert this string to HGfH@GBbbt5E54&.

13. Write a program that will take a string as input. Now, check if the string is valid as a password. A password is valid if it contains

    A. at least nine characters,

    B. has at least one lowercase and one uppercase letter,

    C. at least one number, and

    D. one alphanumeric character.

    You are not allowed to use any built-in methods provided in the String class.

14. Write a Java program that only takes a string formed only by letters (any non-letter input is invalid). The length of the string must be between 7-16 characters. Now, do the following:

    A. Replace all the vowels (in this string) with the immediate next consonants from English alphabet list.

    B. Replace all the consonants (in this string) with the immediate next vowels from English alphabet list. If the consonants appears after ”U” alphabetically, replace it with ”A” or ”a”. Consider the following input-output for understanding:

        Input: *”Peninsula”*, Output: *”Ufojouvob”*.

        Input: *”Delaware”*, Output: *”Efobabuf”*.

        You are not allowed to use any built-in methods provided in the STRING class.

15. Write a program that will take a date as input in MM/DD/YYYY style. Now display it this way: DD First 3 letters of Month, YYYY. For example, if the user provides 08/24/2002, the program will print “24 Aug, 2002.” If the user provides 18/24/2002 or 07/42/2001, the program will print ”Invalid Input”. You are not allowed to use any built-in methods provided in the String class.

# 4 Methods in Java

A method is a system or a way of doing something. However, in Java, a method is a collection of given instructions or statements that are grouped and perform a specific task. In other words, a method in Java or Java is an assortment of explanations that play out a particular assignment and return the outcome to the guest.

## 4.1 Why Write a Method

Assume four friends, John, Chris, Aby, and Mary, are hired as Java developers in four different companies. Soon, they will move to four different states, respectively, Michigan, Colorado, Pennsylvania, and Utah. All these states have flat state income tax rates (Michigan 4.25%, Colorado 4.55%, Pennsylvania 3.07%, and Utah 4.95%). We write the following program to calculate their state income tax.

```
public static void main(){
    String f1 = "John", f2 = "Chris", f3 = "Aby", f4 = "Mary";
    int salary_f1 = 89600, salary_f2 = 106300, salary_f3 = 92000, salary_f4 =
    86200;
    double rate_MI = 4.25, rate_CO = 4.55, rate_PA = 3.07, rate_UT = 4.95;
    double tax_f1, tax_f2, tax_f3, tax_f4;

    tax_f1 = salary_f1 * rate_MI/100.0;
    tax_f2 = salary_f2 * rate_CO/100.0;
    tax_f3 = salary_f3 * rate_PA/100.0;
    tax_f4 = salary_f4 * rate_UT/100.0;

    System.out.println(f1 + "pays $" + tax_f1 + "per year.");
    System.out.println(f2 + "pays $" + tax_f2 + "per year.");
    System.out.println(f3 + "paya $" + tax_f3 + "per year.");
    System.out.println(f4 + "pays $" + tax_f4 + "per year.");
}
```

This code is pretty simple. However, if we want to calculate the tax for one more friend, we must repeat similar statements. In addition, if someone moves to a different state, all these calculations must be rechecked and probably adjusted. We can rewrite this code with a method, and it will look like the example below:

```
public static void main(){
     String f1 = "John", f2 = "Chris", f3 = "Aby", f4 = "Mary";
     int salary_f1 = 89600, salary_f2 = 106300, salary_f3 = 92000, salary_f4 =
     86200;
     double rate_MI = 4.25, rate_CO = 4.55, rate_PA = 3.07, rate_UT = 4.95;

     calc_Tax(f1,salary_f1,rate_MI);
     calc_Tax(f2,salary_f2,rate_CO);
     calc_Tax(f3,salary_f3,rate_PA);
     calc_Tax(f4,salary_f4,rate_UT);
}
static void calc_Tax(String name, int salary, double Rate){
     double tax = salary * Rate/100.0;
     System.out.println(name + "pays $" + tax + "per year.");
}
```

Here, we write a method named `calc_Tax(param list)`, which takes multiple parameters—i.e., the name of the friend, his/her salary, and the flat tax rate of the state where he/she will be moving. For now, imagine it like a magic box.[1] First, we call the method `calc Tax(param list)`, which prints the relevant information. This code snippet is easily manageable for a few reasons: For instance, if one of the friends moves to a different state, there is no need to define a new formula. Instead, call the method with a modified tax rate value, and it will adjust accordingly. Let us examine another example that motivates the use of methods.

### Example 1

Write a Java program that prints the sum of all numbers from 1 to 100, 1 to 1000, and 100 to 5000.

```
public static void main(){
     int sum_1 = 0, sum_2 = 0, sum_3 = 0;
     for (int i = 1; i <= 100; i++){
          sum_1 = sum_1 + i;
     }
     for (int i = 1; i <= 1000; i++){
          sum_2 = sum_2 + i;
     }
     for (int i = 100; i <= 5000; i++){
          sum_3 = sum_3 + i;
     }
```

[1] Alternatively, think about a fruit juicer. We put in water and different types of fruits (consider them as input parameters), and it produces the juice (i.e., output).

```
    System.out.println("Sum of 1 to 100 is: " + sum_1);
    System.out.println("Sum of 1 to 1000 is: " + sum_2);
    System.out.println("Sum of 100 to 5000 is: " + sum_3);
}
```

---

Notice that all these for loops are doing almost the same job. They add all the numbers that fall within a range. We can replace this code and write it using methods, as shown below:

---

```
public static void main(){
    calc_Sum(1,100);
    calc_Sum(1,1000);
    calc_Sum(100,5000);
}
static void calc_Sum(int low, int high){
    int sum = 0;
    for (int i = low; i <= high; i++){
        sum = sum + i;
    }
    System.out.println("Sum of " + low + " to " + high + " is: " + sum);
}
```

---

Needless to say, the code with the method is more manageable and organized.

Based on these discussions, we conclude that a method is beneficial in several ways, such as the following:

1. Code reuse-ability; define once, and use it several times.
2. Structured and organized code.
3. Easy to debug and modify the code.

We will discuss more details on methods in the following sections.

## 4.2 Java Methods

Based on the earlier discussion, we might understand that a method is a set of statements (or blocks of code) that performs a specific task. These code blocks are often a reusable portion of a program, also known as a procedure or subroutine. In Java, a method always belongs to a class and has the following properties:

- It can take in single/multiple or no inputs (arguments).

- Depending on the return type, it can return an answer. A void method returns nothing.

### 4.2.1 Defining a Method

Now, let us observe a bit more about methods. Typically, a method in Java often looks like the example below:

```
// Method signature
modifier(s) returnType methodName(parameter list){
    // method body
}
```

Now, let us discuss the pieces.

- **The modifier**: This is an optional label that identifies specific properties of the method.
- **Return Type**: A method could be a void method (we have covered some examples on this type) or a value-returning method (we will provide examples on this type). The return type specifies if a method will return a value or not.
- **Method Name**: This is the descriptive name of the method. Generally, we follow the same rules while naming a method as we did for the variables. For example, say we write a method that calculates the average of some variables. We can call it calculateAvg() or avgCalculation() or something similar, so someone can easily guess what the method does.
- **Input Parameter**: We can pass data to a method as input by giving a parameter (or arguments) to a method. A method may accept zero, one, or multiple parameters separated by a comma. A parameter (or parameter list) is provided inside the () provided after the method name. While providing the input parameter, we must indicate the type and the number of parameters.
- **Method Body**: A statement or set of statements that describe what the method will do.

Based on the discussion presented above, we show the skeleton of a method below:

```
// Method signature
public static int returnMax(int var1, int var2){
    // Method body: return the greater value
    if(var1 >= var2)
        return var1;
    else
        return var1;
}
```

**☛Concept Check**

Write a method skeleton that will:

1. take three integer numbers as input, calculate their average, and print the average value. This method will not return anything.

2. take your first name as input and count the number of characters. This method will return the character count.

### 4.2.2 Calling a Method

When a program starts, the main method automatically begins execution. However, unlike the main method, other methods are executed only when they are called.[2] When a method is called, the program control jumps to that method, performs whatever is mentioned in the method body, and returns to the position from where it was called. Reconsider Example 1, where we wrote a method that calculates the sum of all numbers that fall within a certain range. In this example, multiple statements call the calc Sum() method from the main() method, like below:

```
calc_Sum(1,100);
```

We call a method by its name followed by the parentheses (and input parameters, if any). Note that a method call is a complete statement. Hence, putting a semicolon at the end of a method call is mandatory. Notice how the program works in this case. The program starts execution from the main() method. When we call the calc_Sum(parameters) method, the JVM branches to that calc Sum. Then, it executes the method body, i.e., calculating the sum of all numbers within a specific range. Once the calc_Sum(parameters) method completes execution, the JVM jumps back to the main() and resumes execution from where it was called.

[2] Recall the example of a fruit juicer. It is plugged in, and all the ingredients are there, but it cannot produce fruit juice until you switch it ON.

**📢 Tip**

Method modifiers and the return type are not mentioned while calling the method.

**Practice Problem**

1. Consider the problem shown in Chapter 2, Example 10. Solve this problem using a method that will take the membership duration as an input parameter and print the membership status.

2. Implement the following grading rubrics using a method.

    A. A+ = (93 - 100), A = (88 - 92), B+ = (83 - 87), B = (79-82), C+ = (72 - 78), C = (66 - 71), D = (60 - 65), F = < 60

The method will take a test score as input (provided by the user) and then display the grade for that score.

**☛Concept Check**

- What will happen if we call a method inside a loop?
- Is it possible to call a method from another method (other than the main method)?

### 4.2.3 Passing Arguments to a Method

When calling a method, we can pass values (also known as arguments or parameters) to it. Reconsider the method call from Example 1. In this example, multiple statements call the `calc_Sum()` method from the main() method, like below:

```
calc_Sum(1,100);
```

This statement calls the `calc_Sum(1,100)` method and passes two integer values as arguments. Now, let us examine the method definition, which is presented below:

```
static void calc_Sum(int low, int high){
    int sum = 0;
    for (int i = low; i <= high; i++){
        sum = sum + i;
    }
```

```
    System.out.println("Sum of " + low + " to " + high + " is: " +sum);
}
```

---

Declaration of the integer variable appears inside the parentheses, i.e., (int low, int high). This declaration enables the method to accept the values as an argument. The statement `calc_Sum(1,100)` executes the method. The argument (i.e., the integer values) inside the parentheses is copied into the method's parameters—i.e., low becomes 1, and high becomes 100. Figure 1 illustrates this concept: It is also possible to pass a variable as an input argument. Consider the following code snippet:

---

```
public static void main(String[] args) {
    int myAge = 31, myWeight = 147;
    printMyInfo(myAge,myWeight);
}
static void printMyInfo(int age, int weight){
    System.out.println("My weight is " + weight + " lbs and I am " +age +
    "years old");
}
```

---

The output will look like the example below:

```
My weight is 147 lbs and I am 31 years old.
```

### Example 2

In the code snippet shown above, we declared two integer variables and initialized their values. When we call the method `printMyInfo(myAge, myWeight)`, the value of the myAge and myWeight variables are copied to the input parameters (i.e., age and weight).

### 📢 Tip

When passing parameters to a method, the number of parameters and their data type must match.

### ☛Concept Check

Consider the following code snippet that calls and defines a method. This code snippet contains errors. Modify and correct all the mistakes.

---

```
public class Main {
    static void myMethod(String fname, int age, String city) {
    System.out.println(name + " is " + age + " years old");
```

```
    System.out.println(name + " lives in " + city);
}
public static void main(String[] args) {
    String Name = "Fiona";
    int age = 5;
    myMethod(age, name);
}
```

---

### 4.2.4 Different Types of Methods

So far, we have discussed the advantages of using a method, its components, writing a method, and calling it. Now, we will discuss different types of methods.

Void Methods

A void method performs a task and terminates without returning anything to the caller method. We have already discussed the process of creating a method (refer to Subsection 4.2). We need to provide the method definition, consisting of a header and a body. In the method header, we offer much important information about the method. We presented the skeleton of a method in Subsection 4.2. Now, we will change the skeleton slightly to represent a void method.

---

```
// Method signature
public static void methodName(parameter list){
    // method body
}
```

---

**Example 3**

Let us present an example where we write a program using void methods with and without input arguments (or parameters). In this program, a user will print a generic text in the first line, e.g., *"Hi There"*, *"Good afternoon"*, or something similar. Then, the user will declare his (or her) name and hometown in the following line. Of course, we can write the same code without using any methods. However, the purpose of this example is to explain writing a void method.

In this example, we write two void methods **welcomeMsg()** and **presenter(String name, String city)**. The former does not take any input arguments and prints plain text. The latter takes two input arguments, i.e., the user's name and hometown, and prints these pieces of information. Notice that we did not add any return statement at the end of these methods.

```
public class Main{
    static void welcomeMsg(){
        System.out.println("Hi There! Good Morning");
    }
    static void presenter(String name, String city){
        System.out.println("I am " + name + " from " + city + "welcome you");
    }

    public static void main(String[] Args){
        String name = "Ashik Bhuiyan", city = "West Chester";
        welcomeMsg();
        presenter(name, city);
    }
}
```

The code presented above generates the following output:

```
Hi there! Good Morning
I am Ashik Bhuiyan from West Chester. Welcome you all.
```

### Example 4

In the S.I. unit system, the units to measure distance and temperature are kilometer (km) and degrees Celsius, respectively. However, in the U.S., they use miles and degrees Fahrenheit as units of measurement. Therefore, we will write a Java program that takes the distance and temperature in miles and degrees Fahrenheit and converts it to the S.I. units. We present the complete code below:

```
import java.util.Scanner;
public class Main {
    static void milesToKM(double miles) {
        double km = 1.61 * miles;
        System.out.println(miles + " miles is equivalent to " + km +
        "Kilometers");
    }
    static void fahrenheitToCelsius(double fahrenheit) {
        double celsius = (fahrenheit - 32) / 1.8;
        System.out.println(fahrenheit + " F is equivalent to " +celsius +
        "C");
    }
    public static void main(String[] args) {
        double distanceInMiles, temperatureInF;
        Scanner myObj = new Scanner(System.in);

        System.out.println("Enter the distance (in Miles): ");
```

```
        distanceInMiles = myObj.nextDouble();

        System.out.println("Enter the temperature (in F): ");
        temperatureInF = myObj.nextDouble();

        //call the method
        milesToKM(distanceInMiles);
        fahrenheitToCelsius(temperatureInF);
    } // end of main method
}
```

---

In this example, we wrote two void methods, `milesToKM(parameter)` and `fahrenheitToCelsius(parameter)`. Just as in Example 3, none of these methods return anything.

### Value-Returning Methods

We already see that we can pass data or information into a method by using parameter variables. Unlike a void method, it is possible to return a value to the statement from where a method is called. When a method returns a value to the caller, it is known as a value-returning method. Similar to the void methods, we need to provide the method definition, consisting of a header and a body. We provide the skeleton below that represents a value-returning method. We consider that the method will return an integer value in this skeleton.

---

```
    // Method signature
    public static int methodName(parameter list){
        return an int value;
    }
```

---

### Example 5

Let us consider that we want to calculate the sum of all numbers from 1 to 1000. We have already solved such a problem in Example 1. We will rewrite this example with the help of a value-returning method.

---

```
public static void main(){
    int sum = calc_Sum(1,1000);
    System.out.println("Sum of numbers from 1 to 1000 is: " + sum);
}
static int calc_Sum(int low, int high){
    int temp = 0;
    for (int i = low; i <= high; i++){
        temp = temp + i;
```

```
    }
    return temp;
}
```

Let us examine the following statement:

```
    int sum = calc_Sum(1,1000);
```

After this statement, the calc_sum(1,1000) method is invoked. Then it calculates the sum of all numbers from 1 to 1000 and stores it in a variable temp. When the calculation is over (i.e., the method is finished) it returns the temp variable to the caller method. Finally, the integer variable (i.e., `sum`) captures the value (i.e., `temp`) returned by the method.

### Example 6

We will repeat the problem shown in Example 4. The only difference is that, instead of void methods, we will solve this problem using value-returning methods. This code will generate the same output as shown in Example 4.

```
import java.util.Scanner;
public class Main {
    static double milesToKM(double miles) {
        double km = 1.61 * miles;
        return km;
    }
    static double fahrenheitToCelsius(double fahrenheit) {
        double celsius = (fahrenheit - 32) / 1.8;
        return celsius;
    }
    public static void main(String[] args) {
        double km, celsius, distanceInMiles, temperatureInF;
        Scanner myObj = new Scanner(System.in);

        System.out.println("Enter the distance (in Miles): ");
        distanceInMiles = myObj.nextDouble();
        System.out.println("Enter the temperature (in F): ");
        temperatureInF = myObj.nextDouble();

        //call the method
        km = milesToKM(distanceInMiles);
        celsius = fahrenheitToCelsius(temperatureInF);
        System.out.println(distanceInMiles + " miles is equivalent to" + km +
        " Kilometers”);
```

```
        System.out.println(temperatureInF + " F is equivalent to " +celsius +
        " C");
    } // end of main method
}
```

---

In this example, we convert the void methods, i.e., `milesToKM(parameter)` and `fahrenheitToCelsius(parameter)`, to a value-returning method.

**☛Concept Check**

Answer the following questions:

- Consider Example 6 and the following statement:

  ```
  km = milesToKM(distanceInMiles);
  ```

  What will happen if we do not use any variable in the LHS?

- We have used the System.out.println() method several times. Why do we not need to define this method? Also, is it a value-returning or void method?

- What will happen if we add a return statement inside the main() method? Why?

## 4.2.5 Scope of Variables

A variable declared inside a method is local to that method. A local variable is inaccessible to the code outside the method. Hence, different methods may have local variables with the same name. We demonstrate this concept using the code below.

**Example 7**

In the code snippet shown below, we provide a comparison between two major cities in the US. We write two methods, each having three variables with the same name. We initialize one of them (*city*) via a parameter passed to the method, and we initialize the other two (*population and area*) by providing hard-coded values. The same naming of these variables does not create any issues. This is because these variables are written in different methods, and the program can see only one of them at a specific time. The population and area of

`Chicago_details()` do not influence the population and area of `Atlanta_details()`.[3]

```
public static void main(String[] args){
     String city_s = "Chicago", city_a = "Atlanta";
     Chicago_details(city_s);
     Atlanta_details(city_a);
}
public static void Chicago_details(String name){
     String city = name;
     double population = 2.7, area = 234.5;
     System.out.println(name + " has a area of " + area + " square miles and
     with a population of " + population + " million");
     }
     public static void Atlanta_details(String name){
          String city = name;
     double population = 0.5, area = 136.3;
     System.out.println(name + " has a area of " + area + " square miles and
     with a population of " + population + " million");
}
```

The output will be:

```
Chicago has an area of 234.5 square miles and a population
of 2.7 million
Atlanta has an area of 136.3 square miles and a population
of 0.5 million
```

### Example 8

We will examine another example. We declare a variable (var), assign a value to it, and pass the variable to a method, changeVar(var). Inside the changeVar(var) method, we update the value of var. However, the value of var (before and after calling the method) inside the main() remains the same.

```
public static void main(String[] args){
     int var = 100;
     System.out.println("var inside main, before calling changeVar(): " + var);
     changeVar(var);
     System.out.println("var inside main, after calling changeVar(): " + var);
}
public static void changeVar(int var){
```

[3] We can visualize this concept differently. For example, say you write a question on a paper, create three different copies, and give it to three other students. They receive the same question but will each write a different answer, and the explanation written by one will not influence the others.

```
    var = var * 2;
    System.out.println("var inside changeVar(): "+ var);
}
```

---

The output will be:

```
var inside main, before calling changeVar(): 100
var inside changeVar(): 200
var inside main, after calling changeVar(): 100
```

**☛Concept Check**

Consider the following code snippet (slightly modified from Example 7).

---

```
public static void main(String[] args){
    String city_s = "Chicago", city_a = "Atlanta";
    Chicago_details(city_s);
    Atlanta_details(city_a);
    System.out.println("Population Density of Chicago is: " +(population_Chi/
    area_Chi));
    System.out.println("Population Density of Atlanta is: " +(population_Atl/
    area_Atl));
}
public static void Chicago_details(String name){
    double population_Chi = 2.7, area_Chi = 234.5;
    System.out.println(name + " has a area of " + area_Chi + " square miles
    and with a population of " + population_Chi + "million");
}
public static void Atlanta_details(String name){
    double population_Atl = 0.5, area_Atl = 136.3;
    System.out.println(name + " has a area of " + area_Atl + " square miles
    and with a population of " + population_Atl + "million");
}
```

---

- Will it work? Why or why not?
- If it does not work, make the necessary changes inside the main() method (do not write any additional methods) so that it works.

### 4.2.6 Common Mistakes

While writing or invoking a method, we need to be careful. Below we mention some common mistakes. We also provide a code snippet that demonstrates most of these points.

1. Do not put a semicolon at the end of a method header.

2. Do not use method modifiers and return types in method calls.
3. Write the empty parentheses while calling a method even if it does not accept arguments.
4. Mention the data type of each input parameter in a method header.
5. Always pass the input parameter(s) to a method that requires it (or them).
6. Use a return statement if it is not a void method.
7. Attempting to access a variable declared inside a method (from code outside) will cause an error.
8. A variable receives a method's return value; it needs to be compatible with a method's return value.
9. Passing an argument of a data type that is unmatched by the data type of the parameter variable. Java automatically performs a widening conversion, but it does not do it the other way around. For more about widening conversion, refer to Section 1.10 for details.

---

```
// Method signature
public static void main(String[] args){ // no ;
     String s1 = "Good Morning”;
     String s2 = "John Doe";

     // No method modifiers and return types
     demoMethod(); // empty parentheses needed here
     //parameter passed to the method that needs it
     demoMethodTwo(s1);
     String msg = demoMethodThree(s2);
     System.out.println(msg);

     // Uncomment the lines below and it will cause an error.

     // The variable temp is local to demoMethodThree()
     // System.out.println(temp);

     // weight is not compatible with return value of demoMethodThree()
     // int weight = demoMethodThree(s2);
}
public static void demoMethod(){ // no ;
     System.out.println("Hello World");
}
```

```
// Need to mention the data type of input param
public static void demoMethodTwo(String str){ // no ;
    System.out.println(str);
}
// Return statement needed for a value-returning method
public static String demoMethodThree(String name){ // no ;
    String s1 = "I am ", s2 = " Welcome you all";
    String temp = s1.concat(name);
    temp = temp.concat(s2);
    return temp;
}
/* Output: Hello World
Good Morning
I am John Doe Welcome you all */
```

---

## 4.3 Exercise

1. Write a Java program that takes an integer number (say num) as input (from the user), which is between 100 to 9999999. Now, write a function void revNum(int num), which reverses this number and prints the difference between the original number and the reversed number. While printing the difference, ignore whether the difference is positive or negative. For example, you should publish as XYZ only if the difference is -XYZ.

2. Consider Problem 16 and Problem 17 from Chapter 2. Now write the following two methods:

    A. *int GCD(int num1, int num2)*; it will return the GCD of num1 and num2.

    B. *int LCM(int num1, int num2)*; it will return the LCM of num1 and num2.

3. A prime number n is a natural number with the following properties:

    A. n is greater than 1, and

    B. n is not a product of any natural numbers other than itself.

    For example, 11 is prime because while factorizing 11, its only factors are itself and 1, i.e., 1 × 11 or 11 × 1. Now, write a method void *primeCheck(int n)* that checks if n is a prime number and prints the decision.

4. Write a Java method void `findPrimes(int lowLimit, int hiLimit)`, which will print all the prime numbers between lowLimit and hiLimit (including themselves). Here, *2 ≤ lowLimit < hiLimit ≤ 999999*.

5. A bank provides 3.5% compound interest on savings account if it fulfills the following two criteria:

    A. The deposit must be 10000USD or higher.

    B. The client cannot withdraw money before two years.

    Write a function datatype calculateInterest(parameters) that takes the deposit amount, how long the money will be deposited, and calculates (and returns) the compound interest.

6. Write a method that will take a string as input and switch the cases of the letters (lowercase to uppercase and vice versa) without using built-in methods. For example, if the user provides `hgFh@gbBBT5e54&` as input, the method will convert this string to `HGfH@GBbbt5E54&`.

7. Consider Problem 19 from Chapter 2. Solve this problem by writing a method `int fib(int num)`. Here, num is an integer value provided by the user, and the function will return the $num^{th}$ Fibonacci number.

8. Write a Java program that takes a string formed only by letters (any non-letter input is invalid). The length of the string must be something between 7-16. Now, write a method:

    A. `void countVowel(String str)`, which will return the total number of vowels in this string.

    B. *`void countConsonants(String str)`*, which will return the total number of consonants in this string.

9. Consider the problem given above. Now, write a method:

    A. `void replaceVowel(String str)`, which will replace all the vowels (in this string) with the immediate next consonants from the English alphabet list.

    B. `void removeConsonants(String str)`, which will replace all the consonants (in this string) with the immediate next vowels from the English alphabet list. If the consonants is something after ”U”, replace

it with ”A” or ”a”. Consider the following input-output for understanding:

- **Input**: ”Peninsula”, Output: ”Ufojouvob”.
- **Input**: ”Delaware”, Output: ”Efobabuf ”.

10. Write a method *void passwordChecker(String pwd)* to check if a password is valid. The password is valid if it

    A. is at least nine characters long,

    B. has at least one lowercase and one uppercase letter,

    C. at least one number, and

    D. one alphanumeric character.

11. Write a program that will take a string as input. Now, write two methods, `findSpace(String input)` and `removeSpace(String input)`. The former method will print the location of white spaces in the string, and the latter will print the modified string after removing all the white spaces.

    For example, if the user inputs ”Have a nice day!” It will invoke the findSpace(String input) method and print: ”Whitespaces found at positions 4, 6, and 11.” Then, it will invoke the removeSpace(String input) method and print: ”Haveaniceday!”

    If the user inputs ”Superman-12455” The findSpace(String input) method prints: ”Whitespaces not found in this string.”, and the removeSpace(String input) method prints: ”Superman-12455”

12. Consider Problem 8 from Chapter 2. Solve this problem by writing a method `String session(String month, int day)`. Here, month and day are the month and day provided by the user. This function will return the season (as string).

13. In this problem, you will create a basic calculator to perform the following binary operations ={+, −, ∗, /, %}. Then, write a program that will do the following:

A. The user will provide the operator. Any input other than the operators listed above is invalid. If an invalid input is provided, terminate the program and print a message saying that the input is not valid.

B. Take two numbers as inputs.

C. Write a method int operatorName (datatype input1, datatype input2), which will operate on the parameter variables (i.e., input1 and input2). For example, `int minus(input1, input2)` will return $(input1 - input2)$, while `int multiply(input1, input2)` will return $(input1 \times input2)$.

D. Show the output.

# 5 Arrays

In this chapter, we introduce a data structure called an array—a collections of related data items of the same type. An array is a static data structure—their length remains the same length once they are created. First, we will discuss declaring, creating, and initializing an array. Then, we will discuss how to manipulate an array and observe some standard array algorithms. We will also examine some common mistakes while using an array. We will present several examples to demonstrate these topics.

## 5.1 Introduction to Arrays

The primitive variables (int, float, doubles) we have encountered in the previous chapters are designed to hold one value at a time. On the other hand, an array in Java is a collection of values, which must be the same type. You declare a reference variable and use the `new` keyword to create an array instance in memory. We use the following syntax to declare an array.

```
dataType[] arrayName;
```

For example, consider the following statement that declares an array reference variable. In this statement, numbers is an array reference variable. It can refer to an array of `int` values.

```
int[] numbers;
```

It is important to note that declaring an array reference variable does not create an array. The next step is to use the new keyword to create the array and assign its memory address to the numbers variable. Here's a complete example:

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

This statement creates an array of integer values with a length of 10, using the new keyword. The numbers variable then contains the memory address of the newly created array. By following this approach, you can create and work with arrays in Java, storing and manipulating groups of values efficiently.

Now, let us answer an important question, *why use an array?* Arrays provide a convenient and efficient way to store and access multiple values of the same type. The following examples highlight the motivation behind using arrays.

### Example 1

Suppose we want to write a program that reads the temperatures of a city for each day of the month and performs some calculations, such as finding the maximum, minimum, and average temperature. Without using an array, we may want to declare separate variables for each day of the month, making the code cumbersome and challenging to manage:

```
// Assume all the temperature are recorded as an integer number
int temperatureDay1;
int temperatureDay2;
int temperatureDay3;
// ... and so on
```

Performing calculations on these individual variables becomes tedious and often impractical. As the number of days increases, the number of variables grows, making the code repetitive and error-prone; e.g., we need 365 or 366 variables to read the temperatures for each day of the year. However, utilizing an array makes the task much simpler and more manageable. We can declare a single array variable to hold the temperatures for all the days:

```
int [] temperature = new int[30]; // Assume it is a 30 day month
```

This approach reduces code repetition, enabling us to perform operations on the entire array or specific subsets. For instance, using array operations, finding the maximum temperature, or calculating the average temperature becomes much more straightforward.

### Example 2

Let us consider another example demonstrating the need for a different data type array, such as storing a collection of strings. Suppose we write a program that manages a student roster for a class size of fifty. Each student has a name, and we need to store these names. Without using an array, we face similar challenges as in the previous example. For example,

```
String student1;
String student2;
String student3;
// ... declare 50 variables for 50 students
```

Like the last example, when the number of students increases, this approach quickly becomes cumbersome and impractical. An array can efficiently store and manage the names of multiple students:

```
String[] studentRecord = new String[50];
```

In this case, we create an array called studentRecord that can store the names of up to fifty students.

The above examples should motivate the use of an array. In both examples, one piece of information still needs to be included, i.e., *how to access a particular day of the month or a particular student from all the students*. We will discuss this topic in the next section.

## 5.2 Array Indexing and Array Length

In this section, we will discuss how to initialize array with values. We will also explain how arrays are indexed and how to access individual elements in an array using their index. Finally, we explain how to perform input/output operation in an array.

### 5.2.1 Initialize and Access the Array Elements

While we give a single name to an array, we can access and utilize its elements as individual variables. We assign each element a unique number called an index. The first element has an index of 0; the second is an index of 1, and so on. For example, the temperature array in Example 1 contains thirty elements, and the index ranges from 0 to 29. We can easily access and modify individual elements of the array using indices, starting from 0 for the first element (as shown below).

```
temperatures[0] = 25; // 1st Day temperature
temperatures[1] = 28; // 2nd Day temperature
// and so on…
temperatures[29] = 28; // Last Day temperature
```

In the same way, in Example 2, we can assign individual names using the indices:

```
studentRecord[0] = "Alice";
studentRecord[1] = "Bob";
// ... and so on
```

### 📢 Tip

In Java, the index always starts at zero. The index of the last element in an array is one less than the total number of elements. If an array has N elements, the index for the last element is (N-1). See Example 3 for details.

### Example 3

```
public static void main(String[] args) {
    // Declare and initialize an array
    int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Get the length of the array
    int length = numbers.length;

    System.out.println("Array length: " + length); // Print the length

    // Demonstrate the difference between array length and last index
    int lastIndex = length - 1;
    System.out.println("Last index is " + lastIndex + ", and the value at last
    index: " + numbers[lastIndex]);
}
```

In this example, we declare and initialize an array called numbers. We then obtain the array's length using the .length property and store it in an integer variable *length*. The length or size of the array represents the number of elements it can hold. Here, we also show the difference between the array length and the last index. As mentioned, we can obtain the last item by subtracting one from the length. The output of this code snippet is shown below:

```
Array length: 10
Last index is 9, and the value at last index: 10
```

### ☛Concept Check

What will happen if we do not subtract one from length, i.e., assign lastIndex as lastIndex = length?

Now, let us describe a bit more about the array memory management. We declare an integer array as follow:

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

Here, memory space holds 10 int numbers, each with a different address. Suppose, at the time, that their memory addresses start from 1000. The

memories unite each other and are separated by the same distance. If 4 bytes represent the int storage, henceforth, the other memories will have 1004, 1008, and 1012, and so on.

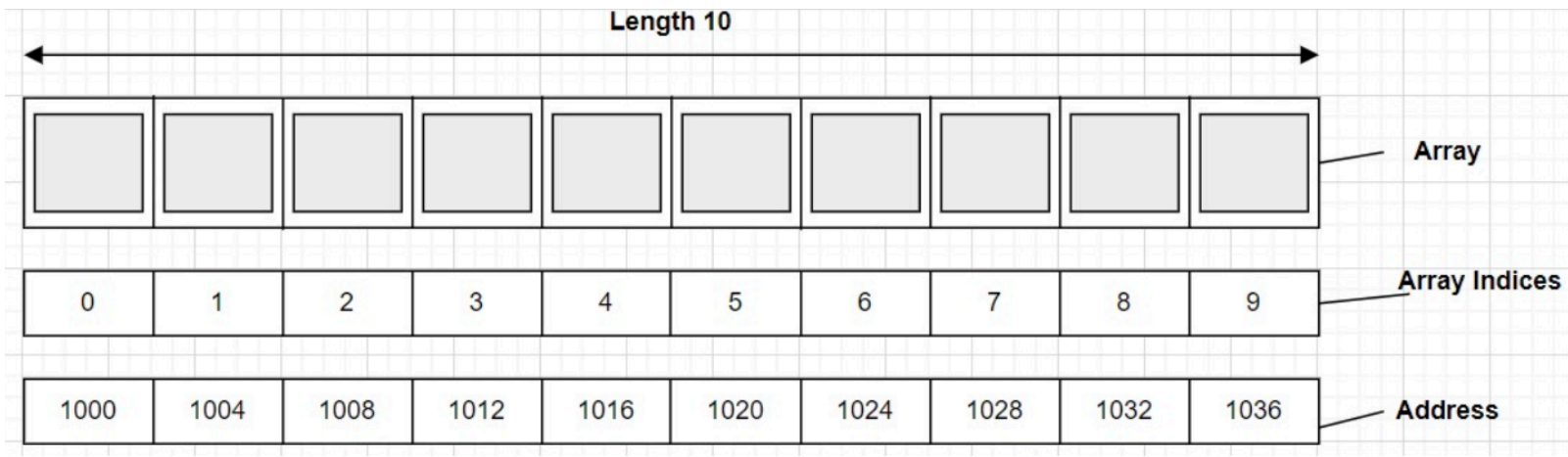


Figure 1: Array Memory Management. ©2025 Bindu sai Jammula. Used with permission.

## 5.2.2 Input and Output the Array Content

As with a regular variable, we can read values from the keyboard and store them in an array element. We will provide a simple example demonstrating how to input and output the contents of an array in Java. We will use the temperature array from Example 1 again.

**Example 4**

We will modify Example 1 so that a user can store the temperature of the last seven days in an array. Then, we will print the temperature on each day. We write the following code snippet for this purpose.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // Create a Scanner object to read user input
        Scanner scanner = new Scanner(System.in);

        // Declare and initialize a double array
        double[] temperature = new double[7];

        // Input values into the array
        System.out.println("Enter the temperature of last 7 days: ");
        for (int i = 0; i < temperature.length; i++) {
            temperature[i] = scanner.nextDouble();
        }

        // Output the contents of the array
        for (int i = 0; i < temperature.length; i++) {
```

```
            System.out.println("Day " + (i + 1) + ": " + temperature[i]);
        }

        // Close the scanner
        scanner.close();
    }
}
```

---

The output of the code is shown below:

```
Enter the temperature of last 7 days:
29.8 28.8 31.2 27.7 27.4 28.3 26.1
Day 1: 29.8
Day 2: 28.8
Day 3: 31.2
Day 4: 27.7
Day 5: 27.4
Day 6: 28.3
Day 7: 26.1
```

In this example, we utilize the Scanner class. Inside the `main()` method, we create a scanner object named scanner to read user input. Next, we declare and initialize an array called temperature with a size of seven. This array will store the temperature for a specific week (i.e., decimal numbers entered by the user). We prompt the user to enter seven decimal numbers using the scanner.nextDouble() method. We use a for loop to iterate through the elements of the temperature array and assign the user's input to each element. When a user provides these values, we display the array's contents using another for loop. We iterate through the elements of the array and print each value along with its corresponding index.

## 5.3 Array Manipulation

In Section 5.2, we have seen how to assign values to array elements, retrieve values from an array, and perform input/output operations. In this section, we will provide some elaborated examples that perform calculations over the array elements, and we will also see how to copy an array.

### 5.3.1 More Examples of Array Operations

Now, we will write a Java program that calculates and displays statistics for a week's temperature values, including the average, maximum, and minimum temperatures.

**Example 5**

We use the code from Example 4 to take the user input, i.e., the last seven days' temperature. Then we use this information to calculate the maximum, minimum, and average temperature. Let us discuss how we calculated the average and the maximum temperature and leave it as an exercise to understand how the minimum temperature is calculated.

**How the average is calculated:**

- We declare a variable sum and initialize it to 0.
- We run a for loop to iterate over each element in the temperature array, adding each element to the sum variable.
- When the loop terminates, we divide the sum variable by the length of the `temperatures` array.
- A new variable, averageTemperature, stores the average temperature and is printed at the end.

**How the max temperature is calculated:**

- We declare and initialize a variable maxTemperature with the first element in the temperatures array.
- We use a for loop starting from the second element to iterate over each element and check if the current element is greater than the maxTemperature. If Yes, it updates the value of maxTemperature to the current element.
- When the loop terminates, the maxTemperature variable will hold the maximum temperature.

---

```
public static void main(String args[]) {
        // Create a Scanner object to read user input
        Scanner scanner = new Scanner(System.in);

        // Declare and initialize a double array
        double[] temperatures = new double[7];

        // Input values into the array
        System.out.println("Enter the temperature of last 7 days: ");
        for (int i = 0; i < temperatures.length; i++) {
            temperatures[i] = scanner.nextDouble();
```

```
}

// Calculate the average temperature
double sum = 0;
for (int i = 0; i < temperatures.length; i++) {
    sum += temperatures[i];
}
double averageTemperature = sum / temperatures.length;

// Find the maximum temperature
double maxTemperature = temperatures[0];
for (int i = 1; i < temperatures.length; i++) {
    if (temperatures[i] > maxTemperature) {
        maxTemperature = temperatures[i];
    }
}

// Find the minimum temperature
double minTemperature = temperatures[0];
for (int i = 1; i < temperatures.length; i++) {
    if (temperatures[i] < minTemperature) {
        minTemperature = temperatures[i];
    }
}
```

```
        // Print the results
        System.out.println("Weekly Temperature Statistics”);
        System.out.println("Average Temperature: " + averageTemperature);
        System.out.println("Maximum Temperature: " + maxTemperature);
        System.out.println("Minimum Temperature: " + minTemperature);
    }
```

---

The output of this code is shown below.

```
Enter the temperature of last 7 days:
27.3 22.9 26.7 22.8 24.7 27.1 25.8
Weekly Temperature Statistics
Average Temperature: 25.32857142857143
Maximum Temperature: 27.3
Minimum Temperature: 22.8
```

We provide below another example that showcases a different type of operation, where we calculate the total and average age of the students and determine the count of students above the average age. Like before, it demonstrates how arrays can be used to store and process different types of data, and perform calculations based on the array elements.

**Example 6**

In this example, the program asks the user to enter the ages of 7 students. We store the ages in an integer array called ages. The program then calculates the total age and the average age (divide the total age by the number of students). Finally, it counts the number of students above and below the average age. We print the results using `System.out.println()` statements. We also calculate the standard deviation using the formula `sqrt((sumOfSquaredDifferences / ages.length))`, where `sumOfSquaredDifferences` is the sum of the squared differences between each age and the average age, and ages.length is the array's length.

---

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        int totalAge = 0, aboveAverageCount = 0, belowAverageCount = 0;
        // Create a Scanner object to read user input
        Scanner scanner = new Scanner(System.in);

        // Declare and initialize an array to store student ages
        int[] ages = new int[7];

        // Input values into the array
        System.out.println("Enter the ages of 7 students:");
        for (int i = 0; i < ages.length; i++) {
            ages[i] = scanner.nextInt();
        }

        // Calculate the total age
        for (int i = 0; i < ages.length; i++) {
            totalAge += ages[i];
        }

        // Calculate the average age
        double averageAge = (double) totalAge / ages.length;

        // Count the number of students above the average age
        for (int i = 0; i < ages.length; i++) {
            if (ages[i] > averageAge) {
                aboveAverageCount++;
            }
        }

        // Count the number of students below the average age
        for (int i = 0; i < ages.length; i++) {
```

```
            if (ages[i] < averageAge) {
                belowAverageCount++;
            }
        }

        // Calculate the standard deviation
        double sumOfSquaredDifferences = 0;
        for (int i = 0; i < ages.length; i++) {
            double difference = ages[i] - averageAge;
            sumOfSquaredDifferences += difference * difference;
        }
        double standardDeviation = Math.sqrt(sumOfSquaredDifferences /
        ages.length);

        // Print the results
        System.out.println("Age Statistics");
        System.out.println("Average Age: " + averageAge);
        System.out.println("Students above Average Age: "+ aboveAverageCount);
        System.out.println("Students Below Average Age: "+ belowAverageCount);
        System.out.println("Standard Deviation of Ages: "+ standardDeviation);
    }
}
```

---

The output of the code is shown below:

```
Enter the ages of 7 students:
21 20 19 20 21 19 20
Age Statistics
Average Age: 20.0
Students above Average Age: 2
Students Below Average Age: 2
Standard Deviation of Ages: 0.7559289460184544
```

### 5.3.2 Copying an Array

In Java, if we need to create an exact copy of an array, we need to copy each element of the source array to the corresponding index in the destination array. Such a process ensures that both arrays have separate memory locations and that modifying one array does not affect the other. Java offers various methods for array copying, such as utilizing loops or assigning elements manually. The following code snippets demonstrate how to copy an array.

---

```
public class Main {
    public static void main(String[] args) {
        int[] sourceArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] destinationArray = new int[sourceArray.length];
```

```
        // Copy the elements from sourceArray to destinationArray
        for (int i = 0; i < sourceArray.length; i++) {
            destinationArray[i] = sourceArray[i];
        }

        // Print the contents of both arrays
        System.out.println("Source Array: ");
        for (int element : sourceArray) {
            System.out.print(element + " ");
        }

        System.out.println("\nDestination Array: ");
        for (int element : destinationArray) {
            System.out.print(element + " ");
        }
    }
}
```

---

The output of this code is shown below:

```
Source Array:
1 2 3 4 5 6 7 8 9 10
Destination Array:
1 2 3 4 5 6 7 8 9 10
```

One tempting approach could be simply writing "sourceArray = destinationArray". If we do this, we assign a reference to the array, and both `sourceArray` and `destinationArray` refer to the exact memory location. Hence, if we modify the content of one array, it would also be reflected in another array. We justify this concern with code as shown below:

---

```
public class Main {
    public static void main(String[] args) {
        int[] sourceArray = {1, 2, 3, 4, 5, 6, 7};
        int[] destinationArray = new int[sourceArray.length];

        // Copy the elements from sourceArray to destinationArray
        for (int i = 0; i < sourceArray.length; i++) {
            destinationArray[i] = sourceArray[i];
        }

        // Print the contents of both arrays
        System.out.print("Content of Source Array before modification: ");
        for (int element : sourceArray) {
            System.out.print(element + " ");
```

```
        }

        System.out.print("\n Content of Destination Array before modification:
"");
        for (int element : destinationArray) {
            System.out.print(element + " ");
        }

        // Doesn't copy elements of sourceArray[] to destinationArray[],
        // only makes destinationArray refer to same location
        destinationArray = sourceArray;

        // Change to destinationArray[] will also reflect in sourceArray[]
        destinationArray[0] = 100;


        // Print the contents of both arrays
        System.out.print("\n Content of Source Array after modification: ");
        for (int element : sourceArray) {
            System.out.print(element + " ");
        }

         System.out.print("\n Content of Destination Array after
         modification: ");
                for (int element : destinationArray) {
                    System.out.print(element + " ");
                }
            }
        }
```

---

Output of the code is shown below:

```
Content of Source Array before modification: 1 2 3 4 5 6 7
Content of Destination Array before modification: 1 2 3 4 5 6 7
Content of Source Array after modification: 100 2 3 4 5 6 7
Content of Destination Array after modification: 100 2 3 4 5 6 7
```

## 5.4 Array Algorithms

In this subsection, we will observe two different examples involving an array. We will see the following examples:

- Sorting an integer array.
- Removing duplicates from an array.

### 5.4.1 Sorting an Integer Array

This section will consider a random unsorted array and sort it in ascending order. We provide the Java code (below) that starts with an array of integers. We then call the bubbleSort() method, which implements the bubble sort algorithm. The algorithm compares adjacent elements in the array and swaps them if they are in the wrong order. We repeat this process until the array is fully sorted. The printArray() method displays the array's contents before and after sorting.

**Example 7**

---

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {19, 5, 20, 9, 3, 29, 8, 21, 13};

        System.out.println("Original Array: ");
        printArray(numbers);

        // Perform bubble sort
        bubbleSort(numbers);

        System.out.println("Sorted Array: ");
        printArray(numbers);
    }

    public static void bubbleSort(int[] arr) {
        int arrLength = arr.length;
        for (int i = 0; i < arrLength - 1; i++) {
            for (int j = 0; j < arrLength - i - 1; j++) {
                // Compare adjacent elements
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
```

```
}
```

---

The output of the code is shown below:

```
Original Array:
19 5 20 9 3 29 8 21 13
Sorted Array:
3 5 8 9 13 19 20 21 29
```

### ☛Concept Check

1. Modify Example 7 such that the unsorted array is displayed in descending order (after sorting).

2. Consider **insertion sort**, which is another sorting algorithm. Modify the code shown in Example 7 and implement the sorting using the insertion sort.

## 5.4.2 Removing Duplicate Items From an Array

This section presents the code demonstrating a Java program that removes duplicates from an array without using the built-in Arrays.sort() method. Here's a step-by-step explanation of how the code works:

1. The removeDuplicates method takes an integer array “arr” as input and returns an integer array “uniqueArr" containing the unique elements.

2. The method first determines the array length (i.e., n). Suppose the input array is empty or has only one element (i.e., n= 0 or 1). In that case, the method returns the input array with no duplicates to remove.

3. Otherwise, the method creates a new integer array, uniqueArr, to store the unique elements and an integer variable, “uniqueCount,” to track the number of unique elements found so far. Initially, “uniqueCount” is set to 0.

4. The program then traverses arr, and for each element (index i), it checks if that element is already present in the uniqueArr. To check for the presence of an element, the program uses another loop (with index j) to iterate over the elements in uniqueArr. Suppose the element at index-i of “arr” matches any element in “uniqueArr". In that case, the inner loop is terminated using the break statement (Refer to Section 2.3 for details regarding break statement), indicating that the element is a duplicate.

5. The element is unique if the element is not present in uniqueArr (i.e., the inner loop completes without finding a match). It is added to the uniqueArr at index uniqueCount. After adding the element, the “uniqueCount” is incremented.

6. The process repeats for all elements in arr. At the end of the loop, uniqueArr contains only the unique elements from arr.

7. The method then resizes the uniqueArr using “Arrays.copyOf" to remove unused elements (set to 0). The new size of uniqueArr is equal to uniqueCount. Finally, the method returns the “uniqueArr" containing the unique elements.

**Example 8**

```
public class Main {
    public static int[] removeDuplicates(int[] arr) {
        int n = arr.length;

        // Check if the array is empty or has only one element
        if (n == 0 || n == 1) {
            return arr;
        }

        // Create a new array to store unique elements
        int[] uniqueArr = new int[n];
        int uniqueCount = 0;

        // Traverse the original array
        for (int i = 0; i < n; i++) {
            int j;
            for (j = 0; j < uniqueCount; j++) {
                // If the element is already present in the unique array
                // break the inner loop
                if (arr[i] == uniqueArr[j]) {
                    break;
                }
            }
            // If the element is not present in the unique array, add it
            if (j == uniqueCount) {
                uniqueArr[uniqueCount] = arr[i];
                uniqueCount++;
            }
        }

        // Resize the unique array to remove the extra unused elements
```

```
        uniqueArr = Arrays.copyOf(uniqueArr, uniqueCount);
        return uniqueArr;
    }
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 20, 40, 10, 50};

        System.out.println("Array before removing duplicates: " +
        Arrays.toString(arr));

        int[] uniqueArr = removeDuplicates(arr);

        System.out.println("Array after removing duplicates: " +
        Arrays.toString(uniqueArr));
    }
}
```

---

The output of the code is shown below:

```
Array before removing duplicates: [10, 20, 30, 20, 40, 10, 50]
Array after removing duplicates: [10, 20, 30, 40, 50]
```

### ☛Concept Check

In Example 8, when there is a duplicate, the program keeps the first occurrence of that item. Modify the code to keep the last occurrence of any duplicate item. Consider the same array from Example 8. The updated output will be:

```
Array before removing duplicates: [10, 20, 30, 20, 40, 10, 50]
Array after removing duplicates: [30, 20, 40, 10, 50]
```

## 5.5 Multidimensional Arrays

A multidimensional array enables storing data in multiple dimensions. The elements in a multidimensional array are arranged in rows and columns, forming a table-like structure. The most common type of multidimensional array is a 2D array or matrix. However, it is also possible to have arrays with three or more dimensions. In this section, we will focus on a 2D array.

### 5.5.1 Declaring, Initializing, and Accessing Elements in a 2D Array

In Java, we can declare a 2D array using the following syntax:

---

```
data_type[][] array_name = new data_type[rows][columns];
```

---

Here, “data_type” specifies the type of elements the array will hold, “array_name” is the array's name, “rows/columns” represents the number of rows/columns in the 2D array. For example, to declare a 2D integer array named “matrix” with 4 rows and 3 columns, you would use the following code:

```
int[][] matrix = new int[4][3];
```

The code snippet creates a 2D array with four rows and three columns, where all elements are initialized to their default values for the int data type (which is 0 for numeric types). Once we declare the 2D array, we can initialize it with specific values using the following syntax:

```
data_type[][] array_name = { {value1, value2, value3, ...}, {value4, value5, value6, ...}, ... };
```

Here, each set of curly braces “{}” represents a row in the 2D array, and the elements inside the braces represent the values for each column in that row. The number of elements in each row must be the same as the number of columns specified during the declaration. For example, if you want to initialize a 2D integer array named “matrix” with the following values:

```
1  2  3
5  6  7
9 10 11
```

You can do it as follows:

```
int[][] matrix = { {1, 2, 3}, {5, 6, 7}, {9, 10, 11} };
```

In this example, the 2D array “matrix” has three rows, each containing four elements corresponding to the values in the matrix.

### Using 2D Arrays

Once the 2D array is declared and initialized, we can access its elements using two indices, one for the row and another for the column. For example, to access the element at the 2nd row and 3rd column of the “matrix" array, we would write:

```
int element = matrix[1][2]; // The indices are 0-based, so the 2nd row is at index 1, and the 3rd column is at index 2
```

2D arrays provide a powerful way to organize and manipulate data in a structured manner. They are helpful when working with tabular data or representing matrices in mathematics and graphics, such as matrix operations, image processing, and game development. Now, we present a small example where we declare, initialize, and access elements of a 2D array representing a simple 3x3 matrix.

**Example 9**

```
public class Main {
    public static void main(String[] args) {
        // Declare a 2D integer array with 3 rows and 3 columns
        int[][] matrix = new int[3][3];

        // Initialize the 2D array with some values
        matrix[0][0] = 1;
        matrix[0][1] = 2;
        matrix[0][2] = 3;
        matrix[1][0] = 4;
        matrix[1][1] = 5;
        matrix[1][2] = 6;
        matrix[2][0] = 7;
        matrix[2][1] = 8;
        matrix[2][2] = 9;

        // Display the elements of the 2D array
        System.out.println("Elements of the 2D array:");
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();

    }
}
```

The output of the code is shown below:

```
Elements of the 2D array:
1 2 3
4 5 6
7 8 9
```

In this example, we declare a 2D integer array (i.e., matrix) with three rows and columns and initialize the array with some values. Finally, we use nested loops to access and display the elements of the 2D array in row-major order. Recall that

the indices in a 2D array are zero-based, so the first row starts at index 0, the second is at index 1, and so on. Similarly, the first column starts at index 0, the second at index 1, etc.

### 5.5.2 Matrix Multiplication Using a 2D Array

This section will provide a comparatively complicated example that uses a 2D array. Matrix multiplication is a fundamental operation in linear algebra, where two matrices are multiplied to produce a new matrix. Note that *the number of columns in the first matrix must equal the number of rows in the second matrix* for multiplication to be possible. Let's take an example of matrix multiplication using a 2D array:

**Example 10**

---

```
public class Main {
    public static void main(String[] args) {
        // Define two matrices for multiplication
        int[][] matrix1 = { {1, 2}, {3, 4} };
        int[][] matrix2 = { {5, 6}, {7, 8} };

        // Display the content of matrix1
        System.out.println("Content of matrix1:");
        for (int i = 0; i < matrix1.length; i++) {
            for (int j = 0; j < matrix1[i].length; j++) {
                System.out.print(matrix1[i][j] + " ");
            }
            System.out.println();
        }

        // Display the content of matrix2
        System.out.println("Content of matrix2:");
        for (int i = 0; i < matrix2.length; i++) {
            for (int j = 0; j < matrix2[i].length; j++) {
                System.out.print(matrix2[i][j] + " ");
            }
            System.out.println();
        }

        // Get the dimensions of the matrices
        int rows1 = matrix1.length;             // Number of rows in matrix1
        int columns1 = matrix1[0].length;       // Number of columns in matrix1
        int columns2 = matrix2[0].length;       // Number of columns in matrix2

        // Create a result matrix to store the product of matrix1 and matrix2
        int[][] result = new int[rows1][columns2];
```

```
        // Perform matrix multiplication
        // To calculate the element at position (i, j) in the result matrix,
        // we need to take the dot product of the i-th row of matrix1
        //and j-th column of matrix2.
        // The dot product is obtained by multiplying corresponding
        //elements of the row and column and summing them up.
        for (int i = 0; i < rows1; i++) {
            for (int j = 0; j < columns2; j++) {
                for (int k = 0; k < columns1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }

        // Display the result matrix
        System.out.println("Result of matrix multiplication:");
        for (int i = 0; i < rows1; i++) {
            for (int j = 0; j < columns2; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

---

The output of the code is shown below:

```
Content of matrix1:
1 2
3 4
Content of matrix2:
5 6
7 8
Result of matrix multiplication:
19 22
43 50
```

In this example, we create two matrices, i.e., `matrix1` and `matrix2`, and calculate their product, using a third matrix result. The resulting matrix stores the product of `matrix1` and `matrix2`, which is displayed as the output.

## 5.6 Array Pitfalls and Best Practices

Arrays are a fundamental data structure used in programming to store a collection of elements of the same data type. While arrays are powerful tools,

they come with pitfalls and common mistakes, especially at the beginning. Being mindful of these pitfalls will help prevent errors and ensure the correct and efficient use of arrays in programming.

### 5.6.1 Array Index Out of Bounds

One of the frequent mistakes when accessing array items is accessing elements using invalid indices, which leads to an “ArrayIndexOutOfBoundsException". Array indices in Java start from 0 and go up to "array.length - 1". Attempting to access an index beyond these bounds will result in a runtime error. To avoid this, always ensure that the index used to access elements lies within the valid range (see Example 3). See the following code snippet for better understanding.

**Example 11**

```
int[] numbers = {1, 2, 3, 4, 5};
int index = 5;
// Invalid index, should be between 0 and 4 (inclusive)
// This will throw ArrayIndexOutOfBoundsException
int value = numbers[index];
```

### 5.6.2 Uninitialized Array Elements

During an array creation, its elements are initialized with default values based on their data types. For example, integers are initialized to 0, booleans to false, and object references to null. However, it is essential to remember that arrays are not automatically populated with meaningful values. Initialize array elements properly to avoid unexpected behavior and incorrect results. For example:

**Example 12**

```
public class Main {
    public static void main(String[] args) {
        // Declare an array to store 5 integers
        int[] numbers = new int[5];

        // Attempt to access uninitialized array elements
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}
```

In the example mentioned above, we create an integer array named numbers with a size of 5. Since arrays are objects in Java, they are initialized with default values for their respective data types (which is 0 for integers). We do not explicitly assign any values to the array elements, which remain in their default state (0). When we attempt to access and print the elements using a loop, the output will be as follows:

```
Element at index 0: 0
Element at index 1: 0
Element at index 2: 0
Element at index 3: 0
Element at index 4: 0
```

When one is creating arrays, it is good practice to initialize them with appropriate default values. Doing so ensures that the array elements contain meaningful data from the start.

### 5.6.3 Incorrect Array Size

Misjudging the required size of an array can lead to inefficiencies or data truncation. If the array size is too small to hold all the necessary elements, some data might be lost or overwritten. Conversely, allocating an unnecessarily large array consumes extra memory (this may not lead to an error, but it could be a better practice to avoid doing so).

**Example 13**

---

```java
public class Main {
    public static void main(String[] args) {
        // Incorrect array size
        int[] scores = new int[10];

        // Assume that we receive the following scores for 7 students
        scores[0] = 78;
        scores[1] = 90;
        scores[2] = 85;
        scores[3] = 92;
        scores[4] = 68;
        scores[5] = 75;
        scores[6] = 88;

        // Display the scores of all students
        System.out.println("Scores of Students:");
        for (int i = 0; i < scores.length; i++) {
            System.out.println("Student " + (i + 1) + ": " + scores[i]);
```

```
            }
        }
}
```

Output of this code is shown below:

```
Scores of Students:
Student 1: 78
Student 2: 90
Student 3: 85
Student 4: 92
Student 5: 68
Student 6: 75
Student 7: 88
Student 8: 0
Student 9: 0
Student 10: 0
```

## 5.6.4 Mixing Array Types

Arrays are static data structures with a fixed data type. Mixing different data types within a single array is not allowed in most programming languages and will result in a compilation error. See the following code snippet for better understanding.

### Example 14

```
public class Main {
    public static void main(String[] args) {
        // Mixing array elements of different data types (int, string, double)
        int[] mixedArray = {1.5, 2, "Hello", 3.14, 4};

        // Attempting to sum elements with mixed data types
        for (int item : mixedArray) {
            System.out.println("Elements: " + item);
        }
    }
}
```

Output of this code is shown below:

```
Main.java:5: error: incompatible types: possible lossy conversion from double
to int
        int[] mixedArray = {1.5, 2, "Hello", 3.14, 4};
                                    ^
Main.java:5: error: incompatible types: String cannot be converted to int
```

```
        int[] mixedArray = {1.5, 2, "Hello", 3.14, 4};
                                             ^
Main.java:5: error: incompatible types: possible lossy conversion from double
to int
        int[] mixedArray = {1.5, 2, "Hello", 3.14, 4};
                                                   ^
3 errors
```

## 5.6.5 Caution During Array Traversal

When iterating over the elements of an array, using an enhanced for loop (also known as a "for each" loop) is preferred instead of the traditional for loop. The enhanced for loop simplifies array traversal, leading to cleaner and more readable code. See the following example:

### Example 15

```
public class Main
{
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        // Using traditional for loop
        System.out.print("Using traditional for loop: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i]);
        }
        System.out.println("");
        // Using enhanced for loop
        System.out.print("Using enhanced for loop: ");
        for (int num : numbers) {
            System.out.print(num);
        }

    }
}
```

The output of the code is shown below:

```
Using traditional for loop: 12345
Using enhanced for loop: 12345
```

## 5.6.6 Pass Arrays as Parameters to Methods

Arrays allow us to store and manipulate collections of elements of the same data type, making them robust data structures. Many programming scenarios expect the passing of arrays as arguments to methods for various data processing tasks. Writing efficient and maintainable code requires a crucial understanding of how to pass arrays as parameters and how they are modified inside methods.

When a method receives an array as a parameter in Java, it uses a ***pass-by-reference*** mechanism, i.e., it gets a reference to the *original array*, *not a copy of it*. Consequently, any changes made to the array elements within the method are reflected in the original array outside the method. This behavior can be advantageous when modifying array elements and retaining the changes after the method call. See the following example for better understanding.

**Example 16**

```
public class Main {
    // Method to double the elements of an array
    static void doubleElements(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            arr[i] *= 2;
        }
        System.out.println("Modified array inside the method: " +
        Arrays.toString(arr));
    }

    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5, 6};
        System.out.println("Original array before calling the method: " +
        Arrays.toString(numbers));

        // Call the method to double the elements
        doubleElements(numbers);

        System.out.println("Modified array outside the method: " +
        Arrays.toString(numbers));
    }
}
```

The output of the code is shown below:

```
Original array before calling the method: [1, 2, 3, 4, 5, 6]
Modified array inside the method: [2, 4, 6, 8, 10, 12]
Modified array outside the method: [2, 4, 6, 8, 10, 12]
```

In this example, we have a method, doubleElements, that takes an integer array as a parameter and doubles the value of each element. When we call this method with the numbers array, the elements in the original numbers array are modified directly. After the method call, the changes made to the array elements persist in the original array.

Note that the ability to modify array elements directly in a method is powerful but may introduce potential side effects. Modifying arrays inside methods may lead to unintended consequences, especially in large and complex programs. Therefore, using arrays as method parameters makes it crucial to be mindful of side effects.

**☛Concept Check**

Consider the following scenario. We will pass the original array to a method and make some changes in the passed array. Now, propose an approach such that the initial array remains unchanged.

## 5.7 Exercise

1. Write a Java program that takes an integer number (say num) between 20 to 100 as input (from the user). Now, declare an integer/double array with total "`num`" items. Finally, reverse the array in place (without using extra space).

2. Write a Java program to find the second smallest element in an array. For example, if the initial array contains [23, 43, 18, 98, 11, 30, 29, 63, 18, 84, 11, 110, 37], the updated array will be [23, 43, 98, 11, 30, 29, 63, 84, 11, 110, 37].

3. Write a Java program to merge two sorted arrays into a new sorted array. For example, if array A1 contains [23, 43, 58, 98, 111, 130] and array A2 contains [29, 63, 68, 84, 91, 110, 137], the merged array will contain [23, 29, 43, 58, 63, 68, 84, 91, 98, 110, 111, 130, 137].

4. Consider a sorted integer array, i.e., `sortedArray`, containing integers in ascending order. Your task is to find the frequency of each element in the array and return a new array, `frequencyArray`, containing the corresponding frequencies. For example, if the sortedArray contains [1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 7, 7, 7], then the expected output for the frequency of elements should be frequencyArray = [1, 3, 4, 2, 2, 2, 1, 3].

5. Write a function that takes a character array as input and returns the count of vowels and consonants in the array. For example:
   **Input**: ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
   **Output**: Vowels: 3, Consonants: 7

6. Write a function that takes a character array as input and checks if it forms a palindrome. For example:
   **Input**: ['r', 'a', 'c', 'e', 'c', 'a ', 'r']
   **Output**: It is a palindrome.
   **Input**: ['r', 'a', 'c', 'e']
   **Output**: It is not a palindrome.

7. Write a Java program that checks if two given character arrays are anagrams of each other. For example:
   **Input**: ['l', 'i', 's', 't', 'e', 'n'], ['s', 'i', 'l', 'e', 'n', ’t']
   **Output**: true
   **Input**: ['h', 'e', 'l', 'l', 'o'], ['w', 'o', 'r', 'l', `d']
   **Output**: false

8. Write a Java program that finds the common characters in two given character arrays. You can show multiple occurrences of the same character only once. For example:
   **Input**: ['D', 'E', 'L', 'A', 'W', 'A', 'R', 'E'], ['F', 'L', 'O', 'R', 'I', 'D', `A'].
   **Output**: [`D', `L’, `A’, `R'] or [`L', `R’, `D’, `A']
   **Input**: ['h', 'e', 'l', 'l', 'o'], ['l', 'o', 'l', 'l', `y’, `p’, `o’, 'p']
   **Output**: [`l', `o’]

9. Write a program that takes a 2D array as input and calculates the transposition (rows turn to columns and vice versa) of a given 2D matrix. See the sample input/output for clarity.

   **Input**: A = [3 2
   1 2
   3 4]

   **Output**: AT = [3 1 3
   2 2 4]

10. Write a program that takes a 2D array as input and calculates the sum of its diagonal. Assume the number of rows equals the number of columns.

**Input**: A = [ 3 2 1 4
1 2 8 7
3 4 9 2
7 4 5 1]

**Output**: 15

# 6 Introduction to Classes and Objects

Class is a fundamental concept of Java, serving as the foundation for object-oriented programming (OOP). It defines the structure and behavior of objects, encapsulating concepts within its blueprint. Classes in Java are robust, establishing new data types that can be instantiated as objects. Therefore, a class acts as a template for objects, with the terms "object" and "instance" often used interchangeably in Java programming. This chapter will explore the essential elements of classes, including methods, constructors, and the "this" keyword.

## 6.1 Introduction to Class

In Java programming, it is crucial to understand the concept of classes. Class and object concepts are closely entangled, with classes serving as blueprints or templates for creating objects. For example, consider a class named "Dog." This class defines the characteristics and behaviors that all dogs should have. An individual dog (say its name is *Mojo*) would be an instance of the "Dog" class. Here, we can assume *Mojo* is an object of the Dog class. The class specifies the attributes (e.g., breed and age) and behaviors (e.g., barking and fetching) for all dog objects, including *Mojo*. Thus, classes provide a structure for organizing and defining the properties and actions of objects in Java programs. Let us delve deeper into the concept of classes in Java to gain a better understanding.

### 6.1.1 General Form of a Class

When crafting a class, you articulate its precise structure and characteristics. You outline the data it encompasses and the code that manipulates that data. While basic classes might consist solely of code or data, most real-world classes encompass both aspects. Utilizing the class keyword, you formally declare a class. Classes often evolve into more intricate constructs. Below, we present a simplified format for defining a class:

**Example 1**

```
class classname {
    type instanceVariable1;
    type instanceVariable2;
    // List all variables
    type methodName1(parameter-list) {
        // body of method
    }
```

```
    type methodName2(parameter-list) {
        // body of method
    }
    // List all methods
}
```

---

Explanation:

- **class classname**: This line declares the beginning of a class definition with the keyword "class" followed by the class name.
- **type instanceVariable1; type instanceVariable2;**: These lines declare the class's instance variables (also known as member variables or properties). These variables represent the data that each class object will hold.
- **type methodName1(parameter-list) { // body of method }**: These lines define methods (also known as member functions) of the class. Methods are functions associated with the class that perform specific actions using the class's data. The method's return type, name, and parameter list are specified, followed by the method's body enclosed in curly braces.
- **// List all variables**: This comment reminds the programmer to list all variables (i.e., instance variables) of the class.
- **// List all methods**: This comment reminds the programmer to list all class methods.

Here's an example of a class named "Dog" based on the provided template:

**Example 2**

---

```
// Define a class named Dog
class Dog {
    // Declare instance variables
    String breed;
    int age;
    String color;

    // Define a method to make the dog bark
    void bark() {
        System.out.println("Woof! Woof!");
    }
```

```
    // Define a method to make the dog wag its tail
    void wagTail() {
        System.out.println("The dog wags its tail happily.");
    }

    // Define a method to make the dog play
    void play() {
        System.out.println("The dog plays with joy.");
    }
}
```

---

## 6.2 Objects

In Java, you can touch or interact with a computer program using an object. An object is created based on a blueprint called a class, which defines what the object can do and what information it holds. Think of it this way: Imagine you have a blueprint for building a toy robot. The blueprint tells you what parts the robot has (e.g., arms, legs, and a head) and what it can do (like walk, talk, and light up). When you build the robot following that blueprint, you create an object based on that class. So, in Java, an object is like a real-life robot you build using a blueprint. It has its unique characteristics (state) and can perform specific actions (behavior) defined by the class blueprint. Let us describe an object using the "Dog" class as a reference:

### Example 3

---

```
// Define a class named Dog
class Dog {
    // Declare member variables
    private String breed;  // Represents the breed of the dog
    private int age;       // Represents the age of the dog
    private String color;  // Represents the color of the dog

    // Constructor to initialize member variables
    public Dog(String breed, int age, String color) {
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // Method to display details of the dog
    public void displayDetails() {
        System.out.println("Breed: " + breed);
        System.out.println("Age: " + age + " years");
        System.out.println("Color: " + color);
```

```
    }
}

// Main class to create and utilize Dog objects
public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog("Labrador", 3, "Golden");

        // Access object's methods to display details
        myDog.displayDetails();
    }
}
```

---

In this example:

- We have defined a class named "Dog" with private member variables representing a dog's breed, age, and color.
- The "Dog" class has a constructor that initializes these member variables when a new "Dog" object is created. We do this by passing the required values as arguments to the constructor. *This* keyword is used within the constructor to distinguish between the class-level member variables and the constructor's parameters, as both have the same names. This keyword explicitly refers to the current object's variables, ensuring that the values provided in the parameters are correctly assigned to the object's breed, age, and color fields.
- The "displayDetails()" method within the "Dog" class displays the details of the "Dog" object.
- In the "Main" class, we create an object in the "Dog" class named "myDog" using the "new" keyword and passing specific values for breed, age, and color to the constructor.
- We then invoke the "displaydetails()" method on the "myDog" object to display its details. This method internally accesses the object's member variables to print its breed, age, and color.

In summary, an object encapsulates data (member variables) and behavior (methods) defined within its class, allowing us to create multiple class instances with different states and functionalities. Each object operates independently and maintains its state, facilitating modular and reusable code design. In the

upcoming sections, we will discuss the private member variables (see Section 6.3) and constructors (see Section 6.4).

## 6.3 Class Members and Scope

In OOP, classes serve as blueprints for creating objects. Class members, including variables and methods, define the structure and behavior of objects instantiated from the class. Understanding the scope and accessibility of class members is crucial for designing well-organized and maintainable code.

### 6.3.1 Member Variables (Properties)

Member variables define objects' characteristics and attributes in OOP. Understanding member variables and their usage is essential for effective class design and object manipulation. Member variables, properties, or attributes are data fields associated with each class instance. They define the state of objects and hold different values for each object (refer to Section 6.2). Member variables are declared within the class body and can have various data types, such as integers, strings, booleans, or custom objects.

In the context of the "Dog" class example provided earlier, member variables could include properties like breed, age, and color, which represent characteristics specific to each dog object. These variables define attributes that distinguish one dog from another. Let's illustrate this with an updated "Dog" class example:

**Example 4**

```
// Define a class named Dog
class Dog {
    // Declare member variables
    String breed;  // Represents the breed of the dog
    int age;       // Represents the age of the dog
    String color;  // Represents the color of the dog

    // Define methods...
}
```

In this example, "breed," "age," and "color" are member variables of the "Dog" class. Consider three dogs (Mojo, Vojo, and Dingo). Each of them is an instance of the "Dog" class. Each dog object (Mojo, Vojo, and Dingo) will have values for these variables, allowing us to differentiate between them (and other dogs) based on their breed, age, and color.

### 6.3.2 Member Functions (Methods)

Member functions, aka methods, encapsulate the behavior or actions that objects can perform. In object-oriented programming, methods enable objects to exhibit specific behaviors or functionalities, allowing interaction with the object's data and other objects in the system. Methods promote code reusability, modularity, and maintainability by encapsulating related functionality within the class. Methods are declared within the class and can manipulate the object's state by accessing and modifying member variables. Let's expand on the "Dog" class example provided earlier to include methods:

**Example 5**

```
// Define a class named Dog
class Dog {
    // Declare member variables
    String breed;  // Represents the breed of the dog
    int age;       // Represents the age of the dog
    String color;  // Represents the color of the dog

    // Define a method to make the dog bark
    void bark() {
        System.out.println("Woof! Woof!");
    }

    // Define a method to make the dog wag its tail
    void wagTail() {
        System.out.println("The dog wags its tail happily.");
    }

    // Define a method to make the dog play
    void play() {
        System.out.println("The dog plays with joy.");
    }
    // Define additional methods...
}
```

In this example, we've added three methods: “bark(),” “wagTail()” and “play().” The “bark()” method simulates the action of a dog barking, the “wagTail()” method simulates the action of a dog wagging its tail, and the “play()” method simulates a situation when a dog is playing. These methods encapsulate behaviors associated with dogs and allow objects of the "Dog" class to exhibit these behaviors.

### 6.3.3 Access Modifiers

Access modifiers in Java control the visibility and accessibility of class members within the program. They determine which parts of the program can access and modify class members. Java provides four access modifiers:

- **Public**: Allows access to the member from any other class.
- **Private**: Restricts access to the member within the same class.
- **Protected**: Allows access to the member within the same package or subclasses.
- **Default (Package-private)**: Allows access to the member within the same package.

Access modifiers play a crucial role in encapsulating class members and controlling their exposure to other parts of the program. They help hide a class's internal implementation details from external entities. We will focus on the **public** and **private** access modifiers for now, as they are the most commonly used and clearly contrast unrestricted access and strict encapsulation. The protected and default (package-private) modifiers are less common and we will leave them for later discussion. Let's illustrate access modifiers using the "Dog" class example provided earlier:

**Example 6**

```
class Dog {
    // Declare private member variables
    private String breed;  // Represents the breed of the dog
    private int age;       // Represents the age of the dog
    private String color;  // Represents the color of the dog

    // Define public methods to access and modify the private variables
    public String getBreed() {
        return breed;
    }
    public void setBreed(String newBreed) {
        breed = newBreed;
    }
    // Define additional methods...
}
```

In this updated example, we've modified the member variables of the "Dog" class to be private. This means these variables are accessible only within the same class and cannot be accessed or modified directly from outside the class. To provide controlled access to these private variables, we've added public accessor and mutator methods: `getBreed()` and `setBreed()`. These methods allow other parts of the program to read and modify the "breed" variable indirectly while still maintaining encapsulation and preventing direct access to it.

Now, let us turn to an example demonstrating how access modifiers prevent unauthorized access by class members. Violating these rules can lead to compile-time errors and compromise the integrity of the class's implementation.

**Example 7**

---

```
// Define a class named Dog
class Dog {
    // Declare private member variables
    private String breed;  // Represents the breed of the dog
    private int age;       // Represents the age of the dog
    private String color;  // Represents the color of the dog

    // Define public methods to access and modify the private variables
    public String getBreed() {
        return breed;
    }

    public void setBreed(String newBreed) {
        breed = newBreed;
    }

    // Define additional methods...
}

// Another class attempting to access private member variables
// of the Dog class
public class Main {
    public static void main(String[] args) {
        // Create an instance of the Dog class
        Dog myDog = new Dog();

        // Attempt to access and modify private member variables directly
        // Error: breed has private access in Dog class
        myDog.breed = "Labrador";
        // Error: breed has private access in Dog class
        System.out.println("My dog's breed is: " + myDog.breed);
    }
```

```
}
```

In this example, we have a class named "Dog" with private member variables (breed, age, and color). The "Main" class, which is outside the "Dog" class, attempts to access and modify the private member variable "breed" directly using the dot operator. Such an attempt violates the rules set by access modifiers, as private member variables are only accessible within the class where they are declared. Therefore, attempting to access or modify the "breed" variable outside the "Dog" class results in compilation errors.

### 6.3.4 Scope of Class Members

The scope of a class member defines its accessibility within the program. Class members with different access modifiers have different scopes, affecting their accessibility from other classes and methods. Understanding the scope of class members is vital for writing modular and maintainable code, helping to prevent unintended access and modification of data. Let's illustrate the scope of class members using the "Dog" class example provided earlier:

**Example 8**

```
// Define a class named Dog
class Dog {
    // Declare public member variable
    public String breed;  // Represents the breed of the dog

    // Define public method to access the member variable
    public void displayBreed() {
        System.out.println("Breed of the dog: " + breed);
    }
}
```

In this example, the member variable breed and the method `displayBreed()` are declared as public within the "Dog" class. As a result:

- The public member variable breed encapsulates the entire program, making it accessible and modifiable from any class or method within the program.
- The scope of the public method "displayBreed()" also extends throughout the entire program. It can be invoked from any class or method within the program.

Let us expand the "Dog" class example to illustrate clearly the class members' scope.

**Example 9**

```
// Define a class named Dog
class Dog {
    // Declare private member variables
    private String breed;  // Represents the breed of the dog
    private int age;       // Represents the age of the dog
    private String color;  // Represents the color of the dog

    // Define public constructor to initialize member variables
    public Dog(String breed, int age, String color) {
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // Define public method to display the details of the dog
    public void displayDetails() {
        System.out.println("Breed: " + breed);
        System.out.println("Age: " + age + " years");
        System.out.println("Color: " + color);
    }
}
```

Now, let's create another class called "Main" to demonstrate the scope of class members. In this "Main" class, we create an instance of the "Dog" class named "myDog" and initialize it with specific breed, age, and color values. We demonstrate the scope of the private member variables by attempting to access the breed variable directly from outside the "Dog" class. However, this results in a compilation error because we attempted to access private members directly from outside the class. Instead, we access the public method "displayDetails()" of the "Dog" class, which internally accesses the private member variables to display the dog's details.

**Example 10**

```
// Another class attempting to access the Dog class members
public class Main {
    public static void main(String[] args) {
        // Create an instance of the Dog class
        Dog myDog = new Dog("Labrador", 3, "Golden");
```

```
        // Attempt to access private member variables
        //directly (violating encapsulation)
        // Compilation error: breed has private access in Dog class
        // System.out.println("My dog's breed is: " + myDog.breed);

        // Access public method to display the details of the dog
        myDog.displayDetails();
    }
}
```

---

## 6.4 Constructors

In this section, we dive into constructors using the example of a "Dog" class. Constructors are like instructions that tell a program how to create a new object: They set up everything needed for the object to "exist" in the program. We start by explaining what constructors are and why they're essential. Then, we will show how you can have different types of constructors (default and ones where you give specific details), which help create dogs with different details like name, age, or breed right from the start.

We also discuss having multiple constructors in the same class, which provide different options when creating a new object. Being able to pick the constructor that fits your object's needs makes working with the program easier.

### 6.4.1 Introduction to Constructors

In Java, constructors play a critical role in creating objects. A constructor is a special type of method with the same name as the class that is used to initialize new objects. By assigning initial values to an object's attributes, constructors ensure that an object starts its life in a consistent state.

**Example 11**

Let us now discuss constructors through the "Dog" class. To prepare a Dog object for use immediately after its creation, we can define a constructor that sets up its initial state. For example, when we create a Dog object, we should specify its breed, color, and age immediately. Here's how we might add a constructor to our "Dog" class:

---

```
class Dog {
    String breed;
    String color;
    int age;
```

```
    // Constructor for Dog class
    public Dog(String breed, String color, int age) {
        this.breed = breed;
        this.color = color;
        this.age = age;
    }

    // Methods for Dog class
    void bark() {
        System.out.println(breed + " barking...");
    }

    void run() {
        System.out.println(breed + " is running...");
    }

    void play() {
        System.out.println(breed + " is playing...");
        }
}
```

When we want to create a new Dog object, we call the constructor and pass in the specific values for the breed, color, and age of the dog:

```
Dog myDog = new Dog("Poodle", "White", 2);
```

In the example above, the Dog constructor takes three parameters (breed, color, and age) and assigns them to the instance variables of the new "Dog" object using the “this" keyword. The "this" keyword differentiates between instance variables and parameters that share the same name.

This above statement creates a new Dog object named myDog, a 3-year-old white poodle. The constructor ensures that myDog has all its essential attributes set from the moment it's created, immediately making the object ready for use.

### 6.4.2 Default and Parameterized Constructors

In Java, there are two types of constructors: *default constructors* and *parameterized constructors*. Each type serves a specific purpose in initializing objects. They provide flexibility in initializing objects, allowing for default initialization and customization based on specific requirements. They ensure that objects are correctly initialized and ready for use, contributing to the reliability and maintainability of Java code.

**Default Constructors**: A default constructor doesn't take any parameters. Java automatically provides one if no other constructors are explicitly defined in the class. The default constructor initializes the object with default values or performs default initialization tasks. Let us illustrate using default constructors with the "Dog" class:

### Example 12

```
class Dog {
    String breed;
    String color;
    int age;

    // Default constructor for Dog class
    public Dog() {
        breed = "Unknown";
        color = "Unknown";
        age = 0;
    }

    // Methods for Dog class
    void bark() {
        System.out.println(breed + " barking...");
    }

    void run() {
        System.out.println(breed + " is running...");
    }

void play() {
        System.out.println(breed + " is playing...");
    }
}
```

We've added a default constructor to the "Dog" class in this example. This constructor initializes a Dog object with default values: "Unknown" for the breed and color and 0 for the age.

**Parameterized Constructors**: A parameterized constructor takes parameters to initialize the object with specific values. It allows for customization of object initialization by accepting arguments during object creation. Refer to the example in Section 6.4.1. that utilizes a parameterized constructor.

**Using the Constructors**: Now, we describe how to create Dog objects using both types of constructors:

```
// Creating a Dog object using the default constructor
Dog unknownDog = new Dog();
System.out.println("Unknown Dog:");
unknownDog.bark();
unknownDog.run();
unknownDog.play();

// Creating a Dog object using the parameterized constructor
Dog poodle = new Dog("poodle", "White", 3);
System.out.println("\nPoodle:");
poodle.bark();
poodle.run();
poodle.play();
```

In this code snippet, we create two Dog objects: one using the default constructor (which initializes an "Unknown" dog) and another using the parameterized constructor (which initializes a "poodle" dog with a specified breed, color, and age).

**Combine Both These Constructors**: Here we provide a complete example with the "Dog" and "Main" classes, where we create an object and invoke these methods.

## Example 13

```
// Define the Dog class with constructors and methods
class Dog {
    String breed;
    String color;
    int age;

    // Default constructor for Dog class
    public Dog() {
        breed = "Unknown";
        color = "Unknown";
        age = 0;
    }

    // Parameterized constructor for Dog class
    public Dog(String breed, String color, int age) {
        this.breed = breed;
        this.color = color;
        this.age = age;
    }

    // Methods for Dog class
```

```
    void bark() {
        System.out.println(breed + " barking...");
    }

    void run() {
        System.out.println(breed + " is running...");
    }

    void play() {
        System.out.println(breed + " is playing...");
    }
}

// Main class to create Dog objects and invoke methods
public class Main {
    public static void main(String[] args) {
        // Creating a Dog object using the default constructor
        Dog unknownDog = new Dog();
        System.out.println("Unknown Dog:");
        unknownDog.bark();
        unknownDog.run();
        unknownDog.play();

        // Creating a Dog object using the parameterized constructor
        Dog poodle = new Dog("poodle", "white", 3);
        System.out.println("\n poodle:");
        poodle.bark();
        poodle.run();
        poodle.play();
    }
}
```

---

Here is the output:

```
Unknown Dog:
Unknown barking...
Unknown is running...
Unknown is playing...

Poodle:
Poodle barking...
Poodle is running...
Poodle is playing...
```

### 6.4.3 Constructor Overloading

Constructor overloading in Java is a technique where a class includes more than one constructor with different parameter lists. It allows objects in that class to be initialized in different ways. Overloaded constructors can take different numbers of parameters or parameters of different types, providing flexibility and improving code readability by allowing various ways of object instantiation based on the user context.

**Example 14**

In this example, we will instantiate three objects in the “Dog" class, each using a different constructor to demonstrate the versatility provided by constructor overloading.

---

```
public class Dog {
    String breed;
    String color;
    int age;

    // Default constructor
    public Dog() {
        this.breed = "Unknown";
        this.color = "Unknown";
        this.age = 0;
    }

    // Parameterized constructor 1
    public Dog(String breed, String color) {
        this.breed = breed;
        this.color = color;
    }

    // Parameterized constructor 2 (Overloaded)
    public Dog(String breed, String color, int age) {
        this.breed = breed;
        this.color = color;
        this.age = age;
    }

    void bark() {
        System.out.println(breed + " barking...");
    }

    void run() {
        System.out.println(breed + " is running...");
```

```
    }

    void play() {
        System.out.println(breed + " is playing...");
    }

    @Override
    public String toString() {
        return "Breed: " + breed + ", Color: " + color + ", Age: " + age;
    }

    public static void main(String[] args) {
        // Using default constructor
        Dog unknownDog = new Dog();

        // Using parameterized constructor 1
        Dog goldenRetriever = new Dog("Golden Retriever", "Golden");

        // Using parameterized constructor 2 (Overloaded)
        Dog poodle = new Dog("Poodle", "White", 5);

        System.out.println(unknownDog);
        unknownDog.bark();

        System.out.println(goldenRetriever);
        goldenRetriever.play();

        System.out.println(poodle);
        poodle.run();
    }
}
```

---

In this example, we use all three constructors, producing these results:

- We create an "unknownDog" object using the default constructor, with its breed and color set to "Unknown", and age set to 0.
- We create a goldenRetriever object using the first parameterized constructor, setting its breed to "Golden Retriever" and color to “Golden," leaving the age uninitialized.
- We create a poodle object using the second overloaded parameterized constructor, which sets the breed to "Poodle," color to "White," and age to 5.

Here is the code output:

```
Breed: Unknown, Color: Unknown, Age: 0
Unknown barking...
Breed: Golden Retriever, Color: Golden, Age: 0
Golden Retriever is playing...
Breed: Poodle, Color: White, Age: 5
Poodle is running...
```

## 6.5 Passing Objects as Arguments

Understanding how to pass objects as arguments to methods is crucial for creating flexible and efficient programs in Java. This concept enables methods to use or modify objects created elsewhere in your code. To illustrate these concepts, we will continue using our "Dog" class.

### 6.5.1 Passing Objects by Value vs. Reference

Java operates on the principle of passing arguments by value: When you pass an object to a method, you pass the object's reference, not the object itself. Although this might sound like passing by reference, remember that what is being passed to the method is a *copy* of the object's reference.

The significance of this distinction becomes apparent when you modify the object inside the method. Since the method references the same object, changes in the method affect the original object. However, when you attempt to reassign the object to a new object within the method, the original reference (outside the method) remains unchanged. This behavior showcases how Java treats object passing as passing the value of the reference.

### 6.5.2 Examples and Use Cases

This example, using our “Dog" class, clarifies how Java handles objects passed to methods. It demonstrates the possibility of modifying the original object and the limitations of reassigning the object within the method.

**Example 15**

---

```
class Dog {
    String name;
    int age;

    Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class TestDog {

    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", 5);
        System.out.println("Before calling changeDog:");
        myDog.displayInfo();

        changeDog(myDog);

        System.out.println("After calling changeDog:");
        myDog.displayInfo();
    }

    public static void changeDog(Dog dog) {
        // changes the name of myDog because 'dog' is a reference to myDog
        dog.name = "Max";
        // This changes the age of myDog for the same reason
        dog.age = 6;

        // This will not affect myDog outside this method
        dog = new Dog("Rocky", 2);
    }
}
```

---

In this example, we have a "Dog" class and a "TestDog" class that contains the main method and a changeDog method. The changeDog method illustrates the effect of modifying an object passed as an argument.

- When we call the changeDog method, it receives a copy of the reference to myDog, allowing it to modify myDog's name and age.
- The attempt to reassign the dog to a new Dog object inside changeDog does not change the original myDog object in the main method—the method receives a copy of the object reference, not a direct link to the original object.

The output will be as follows:

```
Before calling changeDog:
Name: Buddy, Age: 5
After calling changeDog:
```

```
Name: Max, Age: 6
```

## 6.6 Conclusion

This chapter comprehensively explored classes and objects in Java. We began by explaining the fundamental concepts of classes, emphasizing their role as blueprints for creating objects. Through discussions on constructors, we learned how to initialize objects, understand their different types, and leverage constructor overloading for enhanced flexibility.

This chapter also deeply dove into passing objects as arguments in Java methods, clarifying the differences between passing by value and reference. Using the "Dog" class as an example, we illustrated how Java handles object references when they're passed to methods, empowering you to understand how objects are manipulated and reassigned. From basic class creation to exploring constructor overloading and object passing, this chapter provided foundational insights into Java's object-oriented programming principles.

## 6.7 Exercises

1. Define a "Car" class with properties for make, model, and year.
    1.1. Include methods to display car information and update the year.
    1.2. Test the class by creating instances and calling its methods. Sample input/output:

    ```
    Car myCar = new Car("Toyota", "Camry", 2015);
    myCar.displayInfo(); // Output: Car: Toyota Camry, Year: 2015
    myCar.updateYear(2018);
    myCar.displayInfo(); // Output: Car: Toyota Camry, Year: 2018
    ```

2. Expand the "Dog" class to include a method that calculates and displays the dog's age in dog years.
    2.1. Test this method with different ages of dogs.
    2.2. Sample input/output:

    ```
    Dog myDog = new Dog("Buddy", 5);
    myDog.displayDogYears(); // Output: Dog Buddy's age in dog years: 35
    ```

3. Implement a "Circle" class with properties for radius and methods to calculate and display the area and circumference.

3.1. Create instances of the class and test its methods.

3.2. Sample input/output:

```
Circle myCircle = new Circle(5);
myCircle.calculateArea(); // Output: Area of the circle: 78.54
myCircle.calculateCircumference(); // Output: Circumference of the
circle: 31.42
```

4. Develop a "Student" class with properties like `name`, `age`, and `grade`.

4.1. Include methods to update the grade and display student information.

4.2. Test the class by creating multiple student instances and modifying their grades.

4.3. Sample input/output:

```
Student student1 = new Student("Alice", 18, "A");
student1.displayInfo(); // Output: Name: Alice, Age: 18, Grade: A
student1.updateGrade("B");
student1.displayInfo(); // Output: Name: Alice, Age: 18, Grade: B
```

5. Expand the "Student" class to include a method that checks if the student is eligible for graduation (grade is above a certain threshold). Test this method with different student grades.

5.1. Sample input/output:

```
Student student2 = new Student("Bob", 20, "C");
student2.isEligibleForGraduation(); // Output: Bob is not eligible for
graduation
```

6. Enhance the "Dog" class to include methods to check if the dog is a puppy (age less than 1 year) or a senior dog (age more than 10 years).

6.1. Test these methods with different ages of dogs.

6.2. Sample input/output:

```
Dog myDog = new Dog("Max", 0.5);
myDog.isPuppy(); // Output: Max is a puppy
```

7. Define a "BankAccount" class with properties like accountNumber, balance, and accountHolder.

7.1. Include methods to deposit, withdraw, and display account information.

7.2. Test the class with various transactions.

7.3. Sample input/output:

```
BankAccount myAccount = new BankAccount("123456789", 1000, "John
Doe");
myAccount.deposit(500);
myAccount.withdraw(200);
myAccount.displayInfo(); // Output: Account Number: 123456789,
Balance: 1300, Account Holder: John Doe
```

8. Implement a "Triangle" class with properties for all three sides.

8.1. Develop methods to calculate and display the area and perimeter of the triangle.

8.2. Test the class by creating instances and calling its methods.

8.3. Sample input/output:

```
Triangle myTriangle = new Triangle(3, 4, 5);
myTriangle.calculateArea(); // Output: Area of the triangle: 6
myTriangle.calculatePerimeter(); // Output: Perimeter of the triangle:
12
```

9. Create a "LibraryBook‟ class with properties like "title", "author", and "checkedOut".

9.1. Include methods to check out and return books.

9.2. Test the class with different book instances and transactions.

9.3. Sample input/output:

```
LibraryBook book1 = new LibraryBook("Java Programming", "John Smith");
book1.checkOut();
book1.displayInfo(); // Output: Book: Java Programming, Author: John
Smith, Checked Out: true
book1.returnBook();
book1.displayInfo(); // Output: Book: Java Programming, Author: John
Smith, Checked Out: false
```

10. Consider the need for a balancing warning as an assigned problem in a "BankAccount" class. Implement additional functionality to check if the account balance falls below a certain threshold (say $1,000), triggering an alert. Test the class with various transactions and alert scenarios.

## 10.1. Sample input/output:

```
BankAccount myAccount = new BankAccount("123456789", 1000, "John
Doe");
myAccount.deposit(500);
myAccount.withdraw(200);
myAccount.displayInfo(); // Output: Account Number: 123456789,
Balance: 1300, Account Holder: John Doe
myAccount.checkBalanceAlert(); // Output: Your account is doing well!
Account balance is above $1000.

myAccount.withdraw(600);
myAccount.displayInfo(); // Output: Account Number: 123456789,
Balance: 700, Account Holder: John Doe
myAccount.checkBalanceAlert(); // Output: Alert! Account balance is
below $1000.
```

# 7 File Handling

## 7.1 Introduction To File Handling

A file is a named location that can be used to store data and information. For example, a data file may contain names of different people. Combining the name with address can be turned into information and store in a file as well.

- A directory contains various file collections and sub-directories. Sub-directories are directories placed inside another directory, but do not need to be filled with files or sub-directories; they can exist and be empty as well.

- The “java.io.file” library can help to create a file object. After importing the library, the following statement can be used to create a file.

```
// An example of file object creation
File file = new File(String filepath);
```

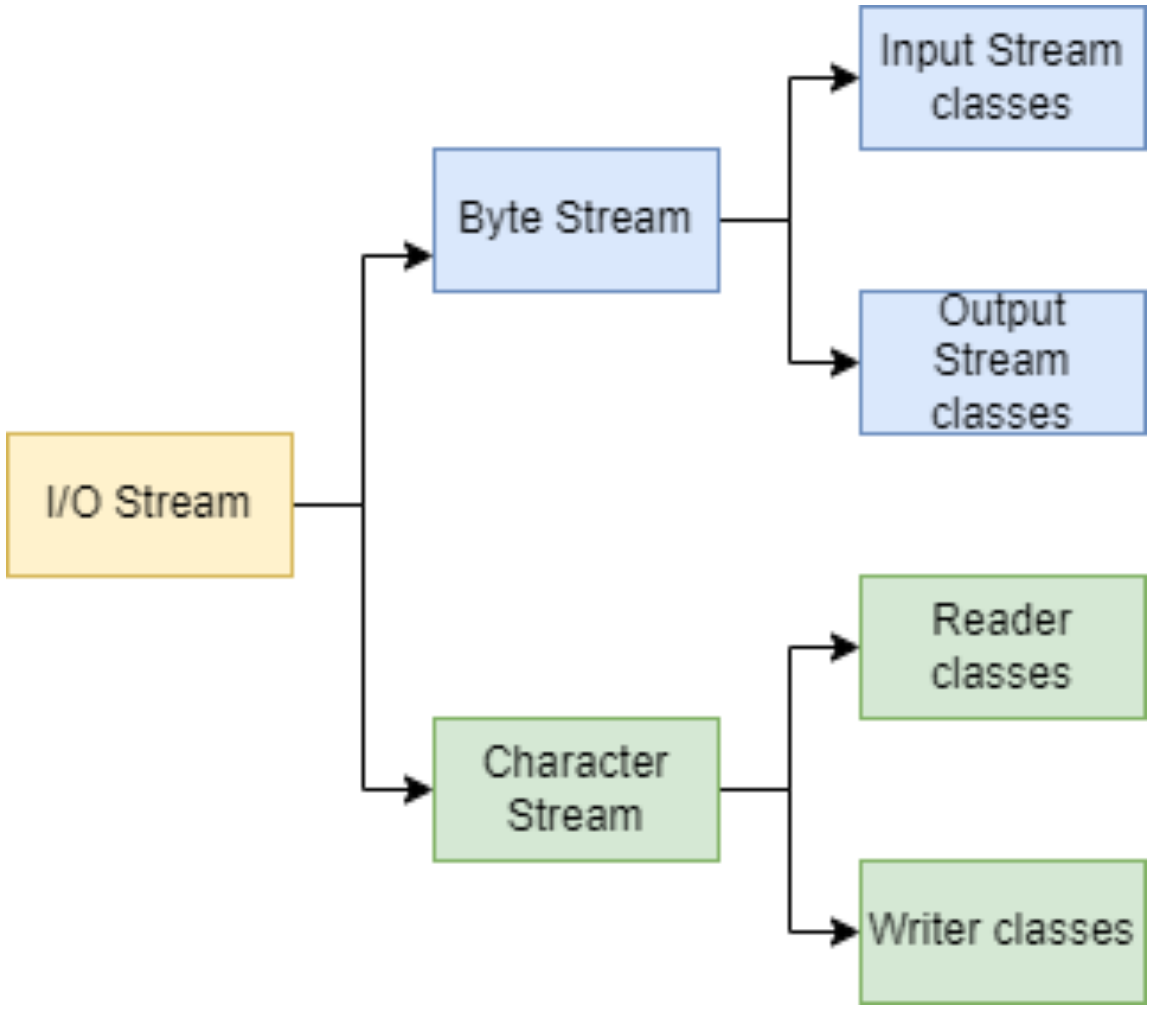


Figure 1: Classification of I/O stream in Java

### 7.1.1 Significance of File Handling

Java is a popular programming language. Java allows file handling using its libraries (e.g., java.io)—which means reading and writing from files. User inputs can be passed to a java program using file handling. Similarly, a program's output can be saved to a file using file handling.

### 7.1.2 Operations on Files

There are two main operations in file handling:

- Reading a file
- Writing a file (creating a file, updating a file, deleting a file)

To read a file, a file must exist otherwise the program trying to read from a file will encounter a file not found exception. However, when writing into a file that does not exist, the program will create a new file with the given name. Similarly, the program should check if the file exists before trying to update (i.e., adding new content to an existing file) or delete the file.

## 7.2 File Classes in Java

In Java, the java.io package/library allows a programmer to create a "File" object from the "File" class. The "File" object can then be used to obtain important information about a file. Creating a "File" object requires a file path (i.e., at least the file name) as a parameter. See the example below.

**Example 1**

```
// Import the File class
import java.io.File;
// Create a file object and pass the filename
File myFile = new File("my_test_file.txt");
```

The "File" class provides some built-in helpful methods as listed below:

**Table 1: Different methods, descriptions and their return type from File class**

| Method Name | Method Description | Return Type |
|---|---|---|
| canRead() | Use this to test whether the file is readable or not | boolean |
| canWrite() | Use this to test whether the file is writable or not | boolean |

| Method Name | Method Description | Return Type |
|---|---|---|
| createNewFile() | Use this to create an empty file | boolean |
| delete() | Use this to delete a file | boolean |
| exists() | Use this to test whether the file exists | boolean |
| getName() | Use this to get the name of the file | String |
| getAbsolutePath() | Use this to get absolute pathname of the file | String |
| length() | Use this to get the size of the file in bytes | Long |
| list() | Use this to get an array of the files in the directory | String[] |
| mkdir() | Use this to create a directory | boolean |
| canExecute() | Checks if the file is executable by the user | boolean |

A programmer can use any of the above methods; however, the example below shows the use of two built-in methods from the File class.

In Java, a “try-catch” block is used to handle exceptions by placing code that might throw an exception in the “try” block, and defining how to handle specific exceptions in the corresponding "catch” block. See the example below for more details.

## Example 2

```
import java.io.File;

public class FileClassExample {

    public static void main(String[] args) {
        File myFile = new File("my_test_file.txt");
        try {
            // true if the file is executable
            boolean fileExecutable = myFile.canExecute();
            // prints
            System.out.println(" is executable: "+ fileExecutable);

            // find the absolute path
            String filePath = myFile.getAbsolutePath();
            // prints absolute path
            System.out.print(filePath);
            }
        } catch (Exception e) {
            // if any I/O error occurs
```

```
            e.printStackTrace();
        }
    }
}
```

---

The example above shows a file name is passed to the “File” constructor when creating an object of File class. Then the object is used to an executable variable with the help of a built-in function “canExecute()”. Later the “File” object is used to get absolute path of the file using the “getAbsolutePath()” function.

### Example 3

---

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class Main
{
    public static void main(String[] args) throws IOException
    {
        String content = null;
        File file = new File("my_test_file.txt");
        FileReader reader = null;
        try {
            reader = new FileReader(file);
            char[] chars = new char[(int) file.length()];
            reader.read(chars);
            content = new String(chars);
            System.out.println(content);
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(reader != null){
                reader.close();
            }
        }
    }
}
```

---

The above code block shows how a File object can be used to read a file. Notice that after creating a File object, it is passed to a FileReader object as a parameter. Also, notice the reader close(); this is important to free up resources being used by the file. Of course, the close() needs to be called after the tasks are performed by the code.

## 7.3 File Navigation and Manipulation

The “listFiles()” method returns an array of pathnames or file location for files and directories in the directory denoted by this abstract pathname or file location. The example below shows how a File object can be used to print the file directory information.

**Example 4**

```
import java.io.File;
public class Main {
    public static void main(String[] args) {
        File f = null;
        File[] paths;
        try {
            // create new file in the location /
            f = new File("/");
            // array of files and directory in the location /
            paths = f.listFiles();
            // for each file in the path array
            for(File path:paths) {
                // prints filename and directory name
                System.out.println(path.getName());
            }
        } catch(Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

The “isDirectory()" method returns true or false. It returns true if and only if the file denoted by this abstract pathname or file location is a directory. Otherwise, the method returns false. The example below shows how a File object can be used to check if a file path is a directory.

**Example 5**

```
import java.io.File;
public class Main {
    public static void main(String[] args) {
        File f = null;
        File f1 = null;
        try {
            // create new files
```

```
            f = new File("test.txt");
            // create new file in the system
            f.createNewFile();
            // create new file object from the absolute path
            f1 = f.getAbsoluteFile();
            // prints the file path if is directory
            System.out.print("Is directory: "+ f1.isDirectory());
        } catch(Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

---

## Example 6

the “isFile()” method returns true if and only if the file denoted by this abstract pathname is a file; otherwise, the method returns false.

---

```
import java.io.File;
public class Main {
    public static void main(String[] args) {
        File f = null;
        File f1 = null;
        try {
            // create new files
            f = new File("test.txt");
            // create new file in the system
            f.createNewFile();
            // create new file object from the absolute path
            f1 = f.getAbsoluteFile();
            // prints the file path if is file
            System.out.print("Is file: "+ f1.isFile());
        } catch(Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

---

### Example 7

Demonstrate how to create, delete, and rename files and directories.

```
import java.io.File;
public class Main {
    public static void main(String[] args) {
        File f = null;
        File f1 = null;
        try {
            // create new files
            f = new File("test.txt");
            f.createNewFile();

            //rename file
            f = new File("test.txt");
            f1 = new File("test1.txt");
            boolean successr = f.renameTo(f1);
            System.out.println("Rename successful: "+ successr);

            //delete file
            f1 = new File("test1.txt");
            boolean successd = f1.delete();
            System.out.println("Delete successful: "+ successd);
        } catch(Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

## 7.4 Reading and Writing Text Files

### 7.4.1 FileReader

The FileReader class from java.io package can be used to read a stream of characters from the files. It is an easy way to read input from file. Below is an example of FileReader reading file contents from a given file:

### Example 8

```
import java.io.FileReader;
public class FileReaderExample {
    public static void main(String args[])throws Exception{
        FileReader fileReader = new FileReader("my_test_file.txt");
        int i;
```

```
        while((i = fileReader.read()) != -1){
            System.out.print((char)i);
        }
        fileReader.close();
    }
}
```

The example code above first checks if the file has something to read. If the file is empty or reaches to the end-of-file, then it will return -1, and the program will stop. Otherwise, the program will read each position of characters and convert and print them.

### 7.4.2 BufferedReader

The BufferedReader class from the java.io package can be used to read input from files. It reads a character-input stream using the buffering concept. Thus, it is efficient in reading characters, arrays, and lines from file input.

**Example 9**

An example of BufferedReader

```
// create a FileReader object
FileReader file = new FileReader(String file);

// create a BufferedReader and pass the FileReader object to it
BufferedReader buffer = new BufferedReader(file);
```

First, we create a FileReader object and then pass the FileReader object to the BufferedReader object. Please note that the file object is taking a String input as the file name. See the example below for more details.

**Example 10**

```
import java.io.FileReader;
import java.io.BufferedReader;

class BufferedReaderExample{
    public static void main(String[] args) {
    // create an array of character to store the stream-input
    char[] charArray = new char[100];

try {
    // create a FileReader
    FileReader file = new FileReader("my_test_file.txt");
```

```
    // create a BufferedReader
    BufferedReader buffer = new BufferedReader(file);

    // reads character-input
    buffer.read(charArray );
    System.out.println("Input from the file: ");
    System.out.println(charArray);

    // close the reader as same as file close
    buffer.close();
    }

    catch(Exception e) {
        e.getStackTrace();
        }
    }
}
```

---

## 7.4.3 FileWriter

The FileWriter class from java.io can be used to write in a file. If the file does not exist, the FileWriter will create the file and write to it.

**Example 11**

---

```
import java.io.FileWriter;

public class FileWriterExample{

    public static void main(String args[]) {

        String data = "This is the data in the output file";

        try {
            // create a FileWriter object
            FileWriter output = new FileWriter("my_output_file.txt");

            // write the string from data variable to the file
            output.write(data);

            // closes the writer
            output.close();
        }

        catch (Exception e) {
            e.getStackTrace();
```

```
            }
        }
}
```

The above code first creates an object of FileWriter and passes a filename to it. The FileWriter object is then used to write in the file using the write method. Finally, the FileWriter object is used to close the file.

### 7.4.4 BufferedWriter

One of the drawbacks of FileWriter is that it writes directly to a file, so it is only good for situations in which a small amount of writing is needed. Additionally, it is limited to a set number of characters or string length. When it comes to writing a large text, then BufferedWriter can be good choice. It is almost similar to FileWriter. However, it uses an internal buffer to write data into files. So, if more frequent writing is needed, then BufferedWriter is a good choice.

**Example 12**

```
import java.io.FileWriter;
import java.io.BufferedWriter;

public class BufferedWriterExample{

    public static void main(String args[]) {

        String data = "This is the data in the output file";

        try {

        // create a FileWriter object
        FileWriter file = new FileWriter("my_output_file.txt");

        // create a BufferedWriter and pass the FileWriter
        BufferedWriter output = new BufferedWriter(file);

        // write the string from data variable to the file
        output.write(data);

        // closes the BufferedWriter object
        output.close();
        }

    catch (Exception e) {
        e.getStackTrace();
        }
```

```
    }
}
```

## 7.5 Exception Handling

In programming, an exception is a situation that is leads to an unexpected behavior. This can occur in many different ways, such as dividing by zero or attempting to open a file that does not exist. These exceptions, if not handled properly, can cause serious issues. Data loss, incorrect calculations, or even a complete system crash serve as examples. The *IOException* is a checked exception in Java that occurs when an attempt to open a file denoted by a specified pathname fails. See the example below:

### Example 13

```
import java.io.*;
import java.util.Scanner;

class Main {
    public static void main(String[] args) throws IOException {
        // Open the file.
        File file = new File("input.txt");
        Scanner inputFile = new Scanner(file);
        // Read and display the file's contents.
        while (inputFile.hasNext())
        {
            System.out.println(inputFile.nextLine());
        }
        // Close the file.
        inputFile.close();

    }
}
```

The above example code will show FileNotFoundException. This exception is a part of IOException. This may occur if the file name is wrong or the file is not available in the directory where the code is looking for it. This can be managed by using “try ... catch” code block (see the example below).

### Example 14

```
import java.io.*;
import java.util.Scanner;

class Main {
```

```
    public static void main(String[] args){
        // Open the file.
        try{
            File file = new File("input.txt");
            Scanner inputFile = new Scanner(file);
            // Read and display the file's contents.
            while (inputFile.hasNext()){
                System.out.println(inputFile.nextLine());
            }
            // Close the file.
            inputFile.close();

        catch(FileNotFoundException e)
        {
            System.out.println("There was some problem opening the file");
        }
    }
}
```

---

As mentioned earlier, there could be multiple reasons for having run-time errors in file handling. Run-time errors are not exactly programming errors. It is important to handle run-time errors so that a program can continue to function without disrupting its natural flow. In file handling, there could be several reasons for having run-time errors. Trying to read a non-existent file and trying to write to a file that the program or user does not have access to it offer two examples. This could happen as a program is moved from one system to another. To avoid this type of unwanted situation, a programmer must use proper exception handling for file handling. Simple “try ... catch” block can help to ease the program flow.

## 7.6 Best Practices and Error Handling

Following steps should be taken while handling files in Java:

- Use `try…catch` blocks to catch expectations.
- Use specific exceptions in `catch` block.
- Avoid empty `catch` blocks.
- Avoid overusing checked exceptions.

## 7.7 Practical Examples and Exercises

Handling CSV files is a major advantage in Java programming. CSV stands for comma-separated values, which means all the data stored in a CSV file is usually separated by commas. For example, John, Doe, 2, 998888, USA. The file extension for a CSV file is .csv.

**Example 15**

```
// import libraries
import java.io.*;
import java.util.Scanner;
public class Csv_file_handline {
    public static void main(String[] args) throws Exception {
        // passing the file name
        Scanner sc = new Scanner(new File("test.csv"));
        // specifiying the delimiter
        // sets the delimiter pattern
        sc.useDelimiter(",");
        // returns a boolean value
        while (sc.hasNext())
        {
            // find and returns
            // the next complete token from this scanner
            System.out.print(sc.next());
        }
        // closes the scanner
        sc.close();
    }
}
```

The above example shows step-by-step how to read a CSV file using Java.

### 7.7.1 Reading and Writing Data Line by Line, Character by Character, or in Bulk.

The example below shows performance comparison of different reading options: line by line, character by character, or in bulk. The comparison is based on the time taken by each option.

**Example 16**

```
import java.io.file.Files;
import java.io.*;
import java.util.stream.Stream;
```

```
public class FileHandlingExamples {

    // read in bluk
    public static void ReadFile_Files_ReadAllBytes(String fileName)
            throws IOException {
        //String fileName = "c:\\temp\\sample-10KB.txt";
        File file = new File(fileName);

        byte[] fileBytes = Files.readAllBytes(file.toPath());
        char singleChar;
        for (byte b : fileBytes) {
            singleChar = (char) b;
            //System.out.print(singleChar);
        }
    }
    // read line by line
    public static void ReadFile_Files_Lines(String fileName)
            throws IOException {
        //String fileName = "c:\\temp\\sample-10KB.txt";
        File file = new File(fileName);

        try (Stream linesStream = Files.lines(file.toPath())) {
            linesStream.forEach(line -> {
                //System.out.println(line);
            });
        }
}

// read character by character
public static void ReadFile_BufferedReader_Char(String fileName)
        throws IOException {

    try (FileInputStream fileInput = new FileInputStream(fileName)) {
        int r;
        while ((r = fileInput.read()) != -1) {
            char c = (char) r;
            // do something with the character c
            //System.out.println(c);
        }
    }
}

// Line
public static void main(String[] args) throws
        FileNotFoundException, IOException {
    String filename = "file100KB.txt";
```

```
        long start1 = System.nanoTime();
        // reads all at once
        ReadFile_Files_ReadAllBytes(filename);
        long time1 = System.nanoTime() - start1;
        System.out.printf("Read bulk took %.3f seconds\n", time1 / 1e9);
        long start2 = System.nanoTime();
        // line by line
        ReadFile_Files_Lines(filename);
        long time2 = System.nanoTime() - start2;
        System.out.printf("Read line by line took %.3f seconds\n", time2 /
        1e9);
        long start3 = System.nanoTime();
        // read a character at a time
        ReadFile_BufferedReader_Char(filename);
        long time3 = System.nanoTime() - start3;
        System.out.printf("Read character by character took %.3f seconds\n",
        time3 / 1e9);

    }

}
```

---

The above example shows that reading in bulk is faster than reading line by line and reading character by character. Also, reading line by line is faster than reading character by character. Which means reading character by character is the slowest than the rest, and reading in bulk is fastest. The output of the above program may look like the following:

```
Read bulk took 0.008 seconds
Read line by line took 0.039 seconds
Read character by character took 0.243 seconds
```

## 7.8 Exercise

1. Write a Java program to create a new file.

2. Write a Java program to read the content of a text file and display the information on the screen.

3. Write a Java program to read the content of a text file (e.g., first.txt) and write the information on another text file (e.g., second.txt).

4. Write a Java program to read the content of a Comma-Separated-Values (CSV) file (e.g., first.csv) and write the information on another CSV file (e.g., second.csv).

5. Write a Java program to read a file line by line.

6. Write a Java program to read the first 3 lines of a file.

7. Write a Java program to find the line that matches a given word.

8. Write a Java program that allows you to write a new line to an existing file. The existing file may or may not have content, and the new write should not remove previous content. Hints: append.

9. Write a Java program to read content from a file using BufferedReader.

10. Write a Java program to read a file line by line and store it into a variable.

# References

## Chapters 1-5

A modern-day laptop computer–. https://pixabay.com/photos/business-technology-notebook-laptop-2717063/. Accessed: 7-22-2022.

Practice questions on decide if or else. https://www.codesdope.com/practice/java-decide-if-or-else/.

If-else-programming exercises and solutions in c. https://codeforwin.org/2015/05/if-else-programming-practice.html.

Tony Gaddis, Soumen Mukherjee, and Arup Kumar Bhattacherjee. Starting out with Java: From control structures through objects. Pearson Education International, 2010.

C++ if-else statements example. https://appdividend.com/2019/09/05/cif-else-statements-example/.

Java loops – a complete guide for beginners https://techvidvan.com/tutorials/java-loops/.

Logical operators in java. https://www.dummies.com/article/technology/programmingweb-design/java/logical-operators-in-java-172160/.

Online java compiler. https://www.onlinegdb.com/online-java-compiler.

Java methods. https://www.cs.fsu.edu/myers/cgs3416/notes/methods.html.

Java string: Exercises, practice, solution, August 01 2022. https://www.w3resource.com/java-exercises/string/index.php.

## Chapter 6

AppDividend. C++ if-else statements example. https://appdividend.com/2019/09/05/cif-else-statements-example/

Codeforwin (2015). If else programming exercises and solutions in C. https://codeforwin.org/2015/05/if-else-programming-practice.html.

CodesDope. Practice questions on decide if or else. https://www.codesdope.com/practice/java-decide-if-or-else/.

Florida State University. Java methods. https://www.cs.fsu.edu/myers/cgs3416/notes/methods.html.

Gaddis, T., Mukherjee, S., & Bhattacherjee, A. K. (2010). Starting Out with Java: From Control Structures through Objects. Pearson Education International.

GDB Online (2016). OnlineGDB. https://www.onlinegdb.com/online_java_compiler.

Goumbik (2017, September 4). Business Technology Notebook Laptop [Photo]. Pixabay. https://pixabay.com/photos/business-technology-notebook-laptop-2717063/.

Herbert, S. (2014). Java: The Complete Reference (9th ed.). McGraw-Hill Education. ISBN: 978-0-07-180856-9.

Lowe, D. (2016, March 26). Logical operators in Java. Dummies. https://www.dummies.com/article/technology/programmingweb-design/java/logical-operators-in-java-172160/.

Simplilearn (2023, February 22). What are Java classes and objects and how do you implement them?. https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects.

TechVidvan. Java loops – a complete guide for beginners!. https://techvidvan.com/tutorials/java-loops/.

W3resource (2022, August 1). Java string: Exercises, practice, solution. https://www.w3resource.com/java-exercises/string/index.php.

## Chapter 7

A modern day laptop computer–. https://pixabay.com/photos/business-technology-notebook-laptop-2717063/. Accessed: 7-22-2022.

Practice questions on decide if or else. https://www.codesdope.com/practice/java-decide-if-or-else/.

If else programming exercises and solutions in c. https://codeforwin.org/2015/05/if-else-programming-practice.html.

Tony Gaddis, Soumen Mukherjee, and Arup Kumar Bhattacherjee. Starting out with Java: From control structures through objects. Pearson Education International, 2010.

Herbert Schildt. Java: The complete reference (ISBN: 978-0-07-180856-9)

C++ if-else statements example. https://appdividend.com/2019/09/05/cif-else-statements-example/.

Java loops: A complete guide for beginners https://techvidvan.com/tutorials/java-loops/.

Logical operators in Java. https://www.dummies.com/article/technology/programmingweb-design/java/logical-operators-in-java-172160/.

Online java compiler. https://www.onlinegdb.com/online-java-compiler.

Java methods. https://www.cs.fsu.edu/myers/cgs3416/notes/methods.html.

Java string: Exercises, practice, solution, August 01, 2022. https://www.w3resource.com/java-exercises/string/index.php.

What are Java classes and objects and how do you implement them? https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects.