Object Oriented Programming

Practical Activities

Author: Assia Brighen

Institute: University of Jijel

Date: 2026

Specialty & Level: Computer Science L2

A journey of a thousand miles begins with a single step. 🚶 – *Lao Tzu*

CONTENTS

Contents i

List of Tables iii

List of Figures iv

Listings v

Preamble 1

Chapter 1 Netbeans environment 3

1.1 Set up the work environment 3
1.2 Setting Up the Project 4
1.3 Creating/Adding a Java Source File 6
1.4 Adding Code to the Java Source File 7
1.5 Compilation and Running the Application 8

Chapter 2 Introduction to Java 10

2.1 What is Java ? 11
2.2 How does Java work ? 11
2.3 Building your First Java Program 11
2.4 Input/Output & Control Statements 12
2.5 Array & ArrayList 15
2.6 Characters & String 18
2.7 Static Functions 20
2.8 Quick Check 21
2.9 Activities 24
2.10 Supplementary Activities 31

Chapter 3 Basics of POO 34
3.1 Classes and Objects 35
3.2 Attributes and Methods 35
3.3 Constructors 38
3.4 Static Attributes and Methods 39
3.5 Visibility and Accessor Methods 40
3.6 Quick Check 42
3.7 Activities 46

Chapter 4 Inheritance 59
4.1 What is Inheritance? 60
4.2 Overriding Methods 61
4.3 Accessing an Overridden Method 62
4.4 Constructors 63
4.5 Quick Check 65
4.6 Activities 69

Chapter 5 Polymorphism 75
5.1 What is Polymorphism 76
5.2 Types of Polymorphism 76
5.3 Benefit of Polymorphism 77
5.4 Polymorphism by Inheritance 77
5.5 Keyword *instanceof* 78
5.6 Quick Check 80
5.7 Activities 84

Chapter 6 Abstract Class and Interface 87
6.1 Abstract Classes 88
6.2 Inheritance of Abstract Classes 90
6.3 Interfaces 92
6.4 Quick Check 95
6.5 Activities 97

Bibliography 102

LIST OF TABLES

2.1 Primitive data types of Java 12
2.2 Some of the most commonly used methods of the class ArrayList 18
2.3 Some of the most commonly used methods of the class String 20

4.1 Allowed Access Levels for an Overriding Method 62

LIST OF FIGURES

1.1 First step for creating new Java project in Netbeans 5
1.2 Creating HelloWorldApp project in Netbeans 5
1.3 Step 1 of adding new Java class in Netbeans 6
1.4 Step 2 of adding new Java class in Netbeans 6
1.5 HelloWorldApp in Netbeans 7
1.6 Adding Code to the Java Source File 7
1.7 Completing the code using Ctrl+Space 8
1.8 Example of potential errors alert. 8
1.9 Running application in Netbeans 9

3.1 Point2D class. 49
3.2 Pet class 53
3.3 Vehicle class 54

4.1 Class Point3D inherits from class Point2D 70
4.2 Pets classes 71
4.3 Pet Class and its subclasses 72
4.4 Subclasses of vhicle Class 73

LISTINGS

2.1 Automatic casting in Java 13
2.2 Manual casting in Java 13
2.3 Reading data from keyboard 14
2.4 Control statements in Java 14
2.5 Declaration of simple arrays in Java 16
2.6 Declaration of multi-dimensional arrays in Java 16
3.1 Student class in Java 35
3.2 Creation of an object of the Student class in Java 35
3.3 Example of passing parameters in Java 37
3.4 Student class in Java with two constructors 38
3.5 Student class in Java with a static attribute 39
3.6 Creation of two objects of Student class in Java 39
3.7 Student class with private attributes 40
3.8 Retrieve and update value of age field outside of class Student 41
3.9 Temperature class 57
3.10 Tmois class 57
3.11 TestTemperature class 58
4.1 Pet class 60
4.2 Cat class derived from Pet class 61
4.3 Dog class inherited from Pet class 62
4.4 Constructor calling without usage of super keyword. 63
4.5 Constructor calling with usage of super keyword. 64
5.1 Example of using instancof operator 79
6.1 Example of an abstract class 88
6.2 An abstract class and Subclasses that extend it 90

PREAMBLE

This polycope is designed to serve as a practical guide for students studying Object-Oriented Programming (OOP) using Java. It provides a structured set of practical and laboratory activities intended to reinforce the theoretical material covered in class.

The primary objective of this lab handout is to help students develop a solid understanding of the fundamental principles of object-oriented programming, including classes and objects, encapsulation, inheritance, polymorphism, abstraction, and interfaces, through practical application. Each practical activity is carefully designed to encourage logical thinking, problem-solving skills, and good programming practices.

The exercises in this lab handout progress from basic concepts to more advanced topics, allowing students to gradually build confidence in Java programming. Realistic examples and guided tasks are used to illustrate how object-oriented concepts are applied in real-world software development.

This practical manual is intended to be used during laboratory sessions under the supervision of an instructor, as well as for independent study. By completing the activities contained in this lab handout, students will acquire the necessary skills to design, implement, and test robust, modular, and reusable Java applications, as well as, they will be able to:

① Understand and apply the fundamental concepts of object-oriented programming using Java.

② Design and implement classes and objects following good programming practices.

③ Use encapsulation to protect data through access modifiers and getter/setter methods.

④ Apply inheritance to promote code reuse and establish hierarchical relationships between classes.

⑤ Implement polymorphism through method overriding and dynamic method dispatch.

6. Work with abstract classes and interfaces to achieve abstraction and flexibility in program design.
7. Distinguish between compile-time and runtime polymorphism with practical examples.
8. Handle type casting and the instanceof operator safely in polymorphic contexts.
9. Develop Java programs using arrays of objects and collections.
10. Practice modular programming by organizing code into packages.
11. Write, compile, execute, and debug Java programs using standard development tools.
12. Analyze program behavior by predicting outputs and identifying compile-time and runtime errors.
13. Apply object-oriented concepts to solve real-world programming problems.
14. Develop clean, readable, and maintainable Java code following Java coding standards.

CHAPTER 1

NETBEANS ENVIRONMENT

Contents

1.1 Set up the work environment 3

1.2 Setting Up the Project 4

1.3 Creating/Adding a Java Source File 6

1.4 Adding Code to the Java Source File 7

1.5 Compilation and Running the Application 8

Key Objectives

This chapter provides an explanation of how to get started with the NetBeans IDE. You will learn how to create a new project, write and organize your Java files, and use the main features of the editor. The goal is to help you become familiar with the NetBeans environment so you can efficiently develop, run, and debug your Java programs. Note that, the screenshots were taken with Apache NetBeans IDE 18 on Windows 10.

1.1 Set up the work environment

Before beginning Java programming, you need to install the necessary tools:

① Download the JDK from the official Oracle website.

② Install the Java Development Kit (JDK): This includes the Java Runtime Environment (JRE), the Java compiler (javac), and other essential tools for developing and running Java applications.

③ Install the JDK by following the installation guide based on your operating system (Windows, macOS, or Linux).

④ Choose and Install an Integrated Development Environment (IDE): While you can use a simple text editor, an IDE like IntelliJ, Eclipse, or NetBeans provides features such as syntax highlighting, code completion, debugging tools, and project management, which significantly enhance productivity. As part of our practical work, we will be using Netbeans and we will present a brief overview of this environment.

⑤ To install NetBeans, download the installer from the official Apache NetBeans website, and choose the installer for your operating system (the .exe file for Windows).

⑥ Locate the downloaded file in your downloads folder and double-click it to start. The installer will guide you through the process.

⑦ Once complete, click "Finish" to exit the installer. You can then launch NetBeans from your desktop shortcut or application menu.

1.2 Setting Up the Project

In Netbeans, you are always working within a project. A "project" serves as the fundamental organizational unit for developing applications. It encapsulates all the necessary files, configurations, and metadata related to a specific development effort. Follow the following steps to set up a new Java project using Netbeans:

① In the IDE, choose File > New Project or click the "New Project" button in the toolbar. In the New Project wizard, select Java Application, as shown in the Figure 1.1. Then click Next.

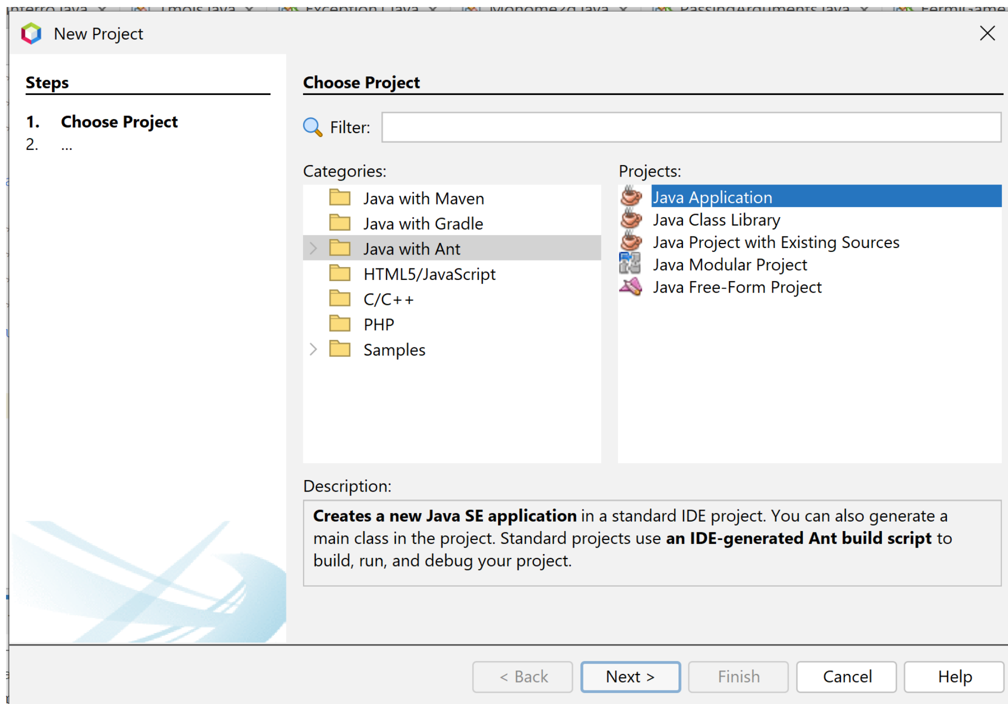


Figure 1.1: First step for creating new Java project in Netbeans

② In the Name and Location page of the wizard, type HelloWorldApp in the Project Name field, and select the option "*Create Main Class*" that allows to add the `public static void main(String[] args)` method (as shown in the Figure 1.2):

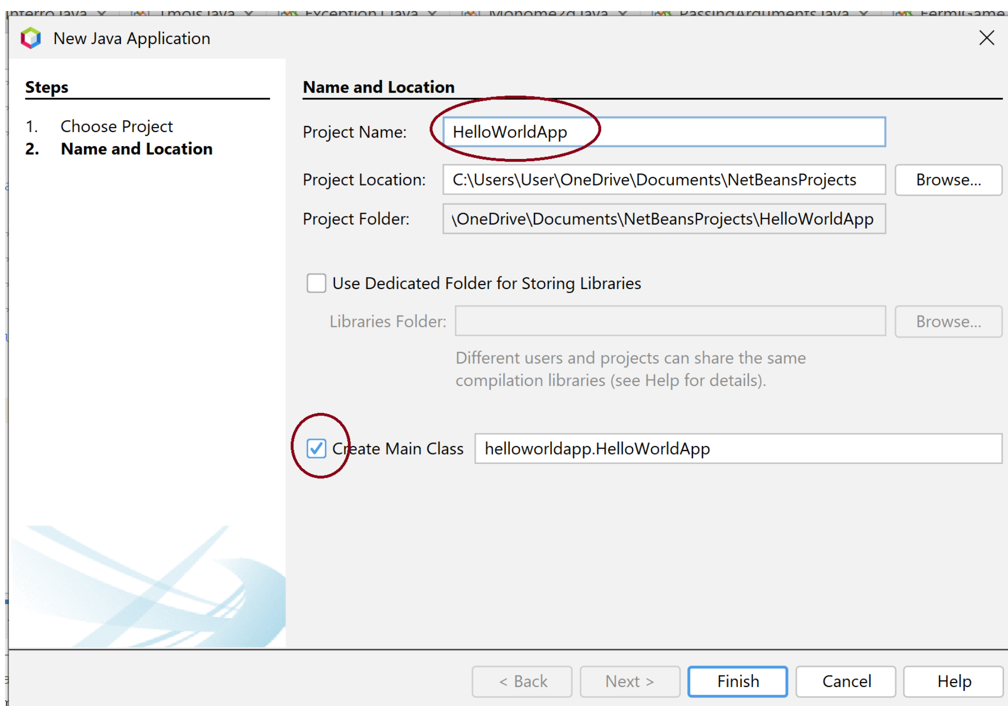


Figure 1.2: Creating HelloWorldApp project in Netbeans

1.3 Creating/Adding a Java Source File

① Right-click the package name and choose New > Java Class, (as shown in the Figure 1.3):

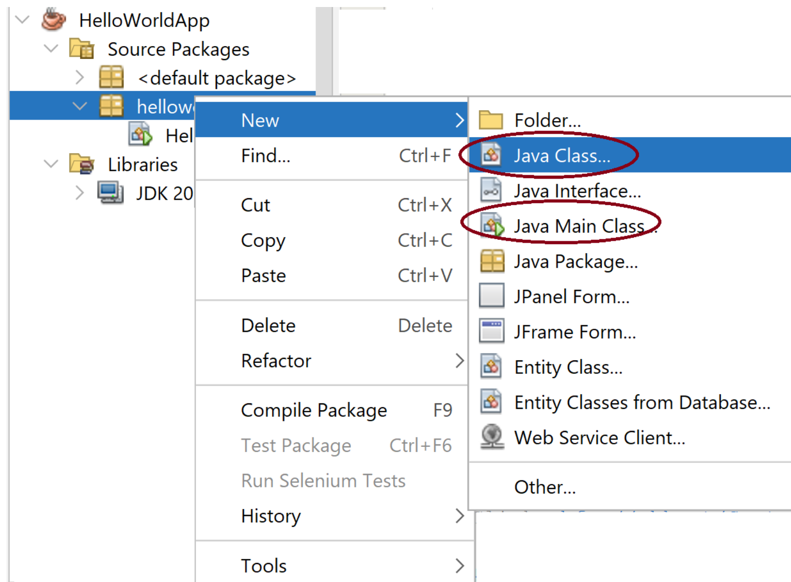


Figure 1.3: Step 1 of adding new Java class in Netbeans

② In the New Java Class wizard, type the name of the new class in the Class Name field, (as shown in the Figure 1.4):

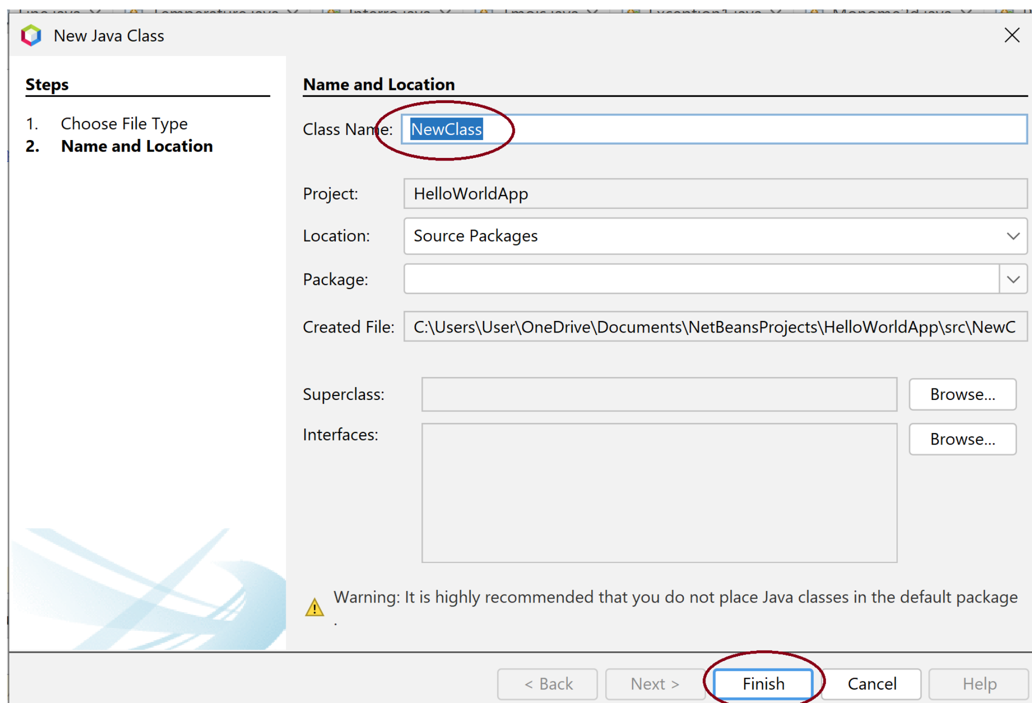


Figure 1.4: Step 2 of adding new Java class in Netbeans

③ Click Finish. The Java source file is created and opened.

④ You should see the following components, (as shown in the Figure 1.5):

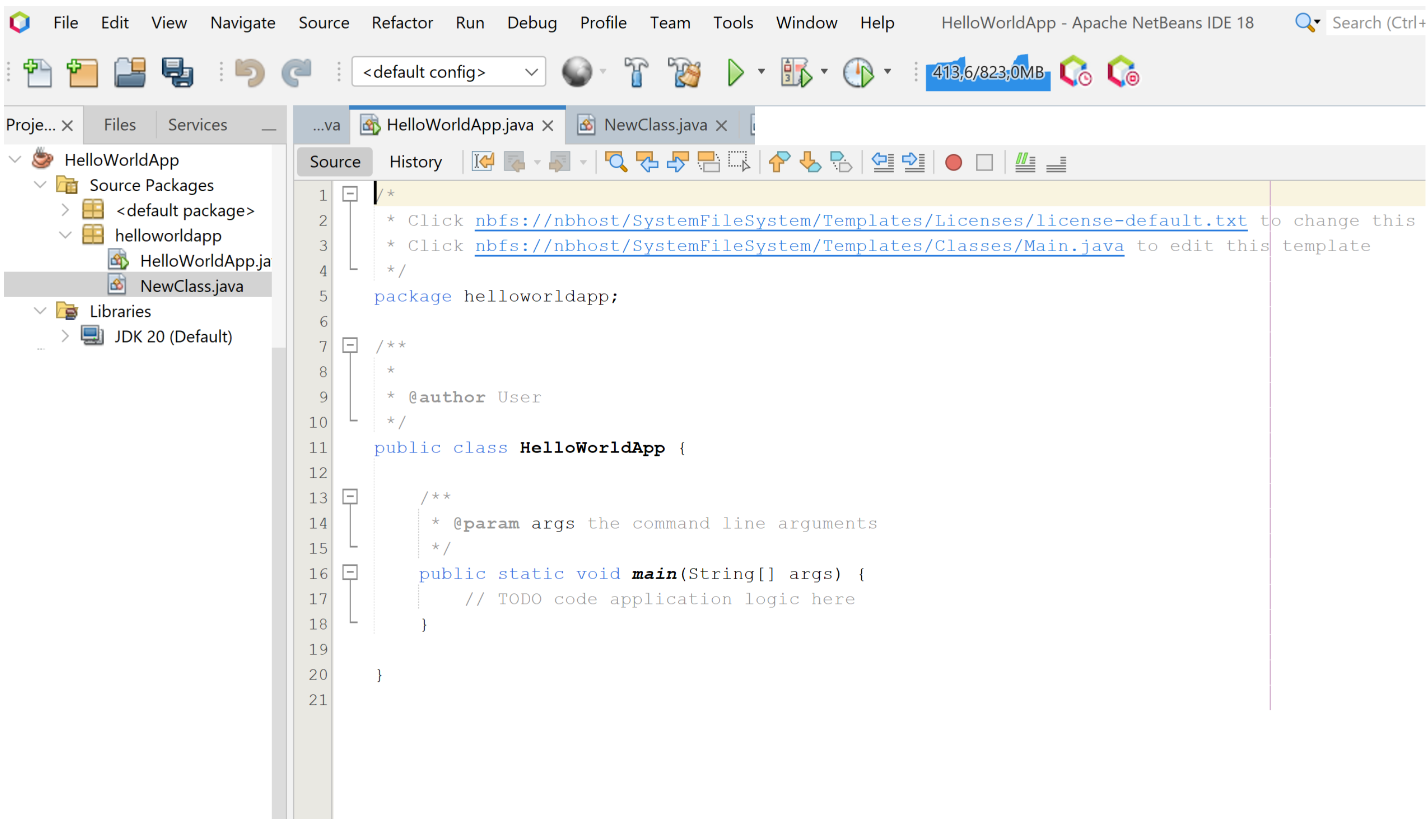


Figure 1.5: HelloWorldApp in Netbeans

1.4 Adding Code to the Java Source File

Let's add some basic content to produce a 'hello world' message.

① Within the public static void main statement, type *sout* and press Tab (sout+Tab). You should now see a `System.out.println("")` statement.

② Within the quotation marks, type *hello world*. You should now see the following (Figure 1.6):

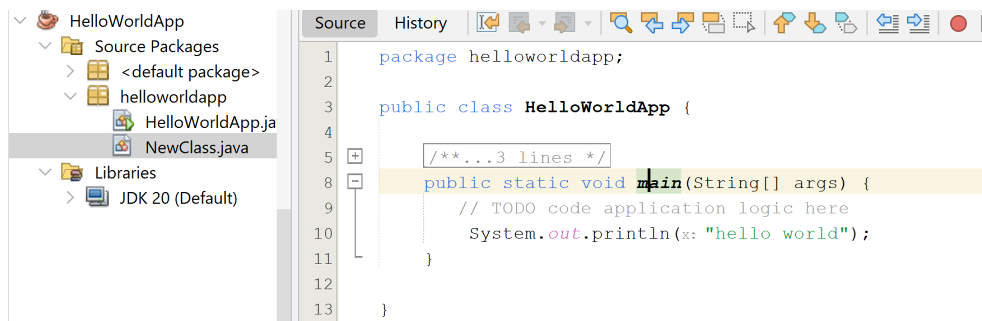


Figure 1.6: Adding Code to the Java Source File

R When you press **Ctrl+Space**, the editor shows you multiple ways of completing the code at the cursor along with their corresponding Javadoc documentation. For example, if you start typing `System.out.` and then stop, the editor will display all available methods along with their corresponding Javadoc documentation as shown in Figure 1.7.

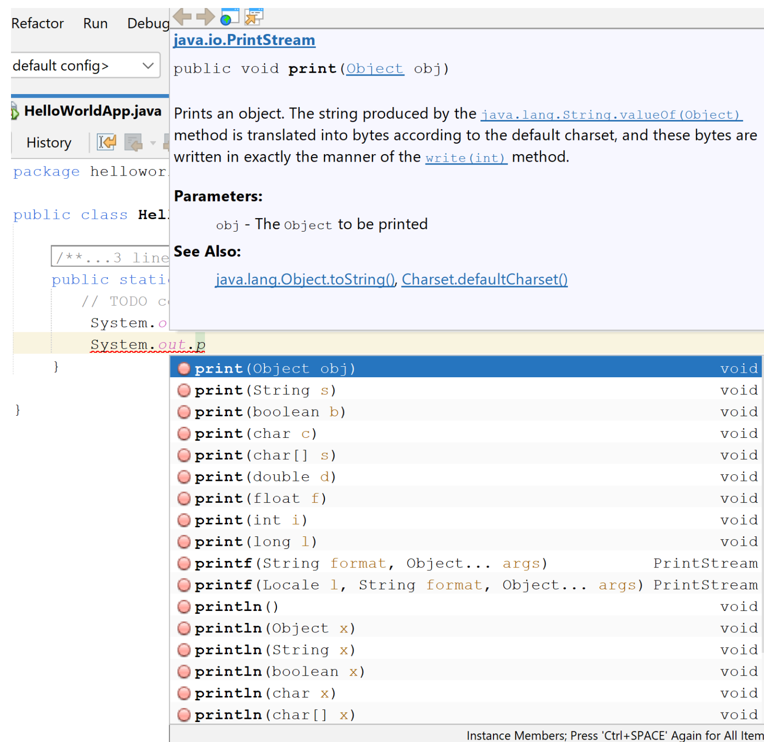


Figure 1.7: Completing the code using Ctrl+Space

1.5 Compilation and Running the Application

The editor does not always wait for compilation to alert you about potential errors. As you type, it continuously checks the Java syntax, and if it detects a problem, a small red exclamation mark appears next to the corresponding line. By positioning the mouse cursor over it you will get an explanation message indicating the nature of the error (as shown in the Figure 1.8).

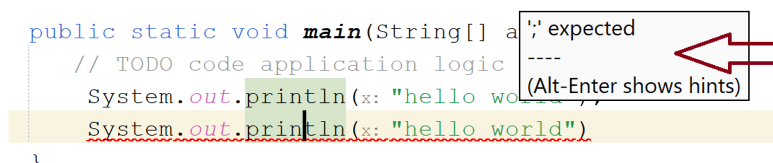


Figure 1.8: Example of potential errors alert.

For running the application:

① Make sure to save the Java source file

② Right-click the project and choose Run or choose Run Project under the Run menu. As you can start compiling and running your project via Menu Run > Project (F6) or by clicking the button run (▶) on the toolbar.

③ In the Output window (at the bottom of the NetBeans window), you should see the below (Figure 1.9):

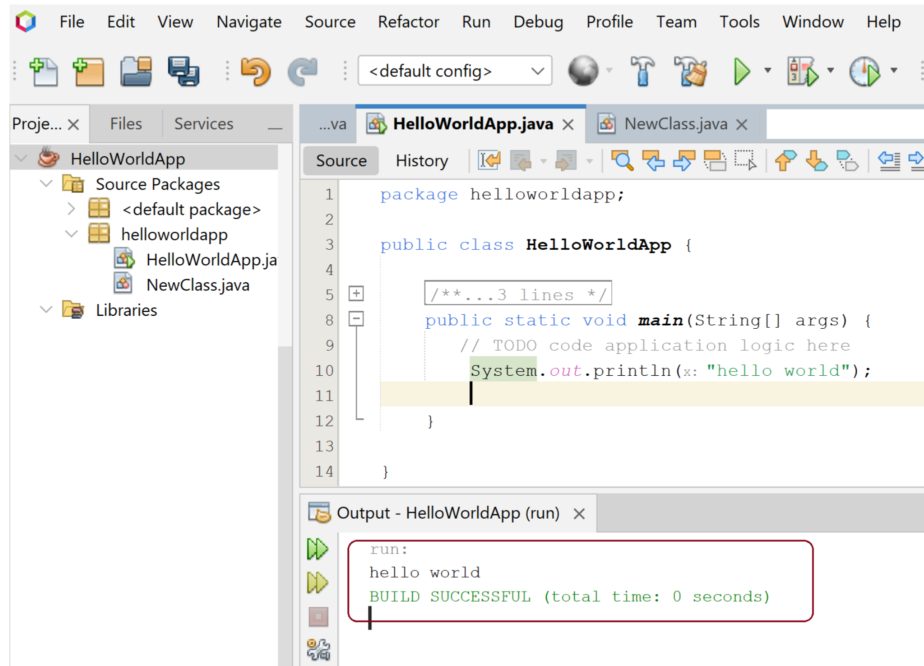


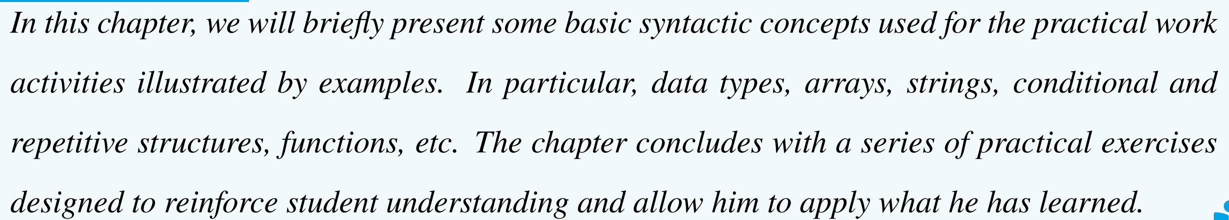
Figure 1.9: Running application in Netbeans

CHAPTER 2

INTRODUCTION TO JAVA

Contents

2.1 What is Java ? 11
2.2 How does Java work ? 11
2.3 Building your First Java Program 11
2.4 Input/Output & Control Statements 12
2.4.1 Primitive (Basic) Data Types 12
2.4.2 Implicit and Explicit Casting 12
2.4.3 Displaying Data 13
2.4.4 Reading Data from Keyboard 14
2.4.5 Control Statements 14
2.5 Array & ArrayList 15
2.5.1 Array 15
2.5.2 Dynamic Array (List) 16
2.6 Characters & String 18
2.7 Static Functions 20
2.8 Quick Check 21
2.9 Activities 24
2.10 Supplementary Activities 31



Key Objectives

In this chapter, we will briefly present some basic syntactic concepts used for the practical work activities illustrated by examples. In particular, data types, arrays, strings, conditional and repetitive structures, functions, etc. The chapter concludes with a series of practical exercises designed to reinforce student understanding and allow him to apply what he has learned.

2.1 What is Java ?

Java is a programming language which helps us to write instructions, called code, to make the computer do a particular task. Java is one of several programming languages we use to instruct the computer. Among these languages, we can cite C, C++, Python, etc. Thus, Java is a mediator between human and computer. Java can be used everywhere and the applications where it can be used are endless. It can be used to instruct robots to carry out a particular task, to create Android apps (almost every anroid phone application uses java), to create desktop applications, and many more.

2.2 How does Java work ?

A java program follows the following steps:

① First we write a list of java instructions (using any Java editors and Integrated Development Environments (IDEs), such as IntelliJ, NteBeans or Eclipse)

② Then, the program is converted into intermediate version called *byte-code* which is not readable by humans.

③ The *byte-code* is then read by a special software called a Java interpreter, which translates it to the *machine language*.

④ The computer reads the translated code and executes the tasks requested.

2.3 Building your First Java Program

Let's start with the first program, called Hello World. It asks the computer to display (print) "Hello World" on screen. The command used to display some text on the screen is : `System.out.print();` and the text which being printed must be inside the curved brackets and enclosed in double quotes. So, the hello world program will look like the following code:

```
System.out.print("Hello World");
```

Now, you can hit the Run button and check if you get the desired output.

- Java is a case sensitive language, the uppercase letters and lowercase letters are different.
- The text is written in quotes to tell Java that it has to be cinsidered as a string.

2.4 Input/Output & Control Statements

2.4.1 Primitive (Basic) Data Types

Recapitulation 2.1

Primitive (basic) types of Java correspond to those of the C language, plus the byte and boolean types. Strings are managed (differently from C) as classes. Table 2.1 lists the basic Java types.

Type	Description	Valeurs
byte	un entier signé sur 8 bits	de –128 à +127
short	un entier signé sur 16 bits	de –32 768 à +32 767
int	un entier signé sur 32 bits	de –2147483648 à 2147483647
long	un entier signé sur 64 bits	de l'ordre de (±) 9 milliards de milliards
float	un réel sur 32 bits	de l'ordre de 1.4E-45 à 3.4E38
double	un réel sur 64 bits	de l'ordre de 4.9E-324 à 1.79E308
boolean	vrai ou faux	true ou false
char	un caractère Unicode entier positif sur 16 bits	entre 0 et 65535

Table 2.1: Primitive data types of Java

2.4.2 Implicit and Explicit Casting

Recapitulation 2.2

Arithmetic operators are only defined when both of their operands are of the same type. But we can write mixed expressions in which operands of different types are involved. Implicit (automatic) type conversions allowed in Java with examples are illustrated by Listing 2.1. However, the reverse is not allowed and we must explicitly force the type conversion. To force the conversion of any expression into a type of choice, we use a special operator called cast.

Its denotation consists of placing in parentheses, before the value to be converted, the type into which it must be converted. Explicit (manual) type conversions in Java with examples are illustrated by Listing 2.2.

```
//converting a smaller type to a larger type size
byte -> short -> char -> int -> long -> float -> double
//Example:
int n = 1, p = 5;
float x = 1.5f;
float b= n * x + p; // Automatic casting: float to double
long a=n;           // Automatic casting: int to long
double d =n+x;      // Automatic casting: int to double
char c='a';
int e=c;            // Automatic casting: char to int
```

Listing 2.1: Automatic casting in Java

```
// converting a larger size type to a smaller size type
double -> float -> long -> int -> char -> short -> byte
//Example:
double d = 9.78d;
int myInt = (int) d; // Manual casting: double to int
float f=3;
f=(float)(d+3);      // Manual casting: double to float
n= (int) f ;         // Manual casting: float to int
char c='a'; c=(char) e;   // Manual casting: int to char
```

Listing 2.2: Manual casting in Java

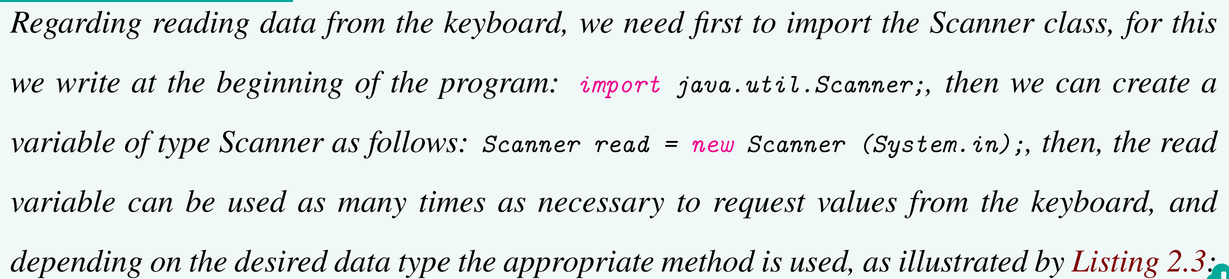
2.4.3 Displaying Data

Recapitulation 2.3

Displaying data on screen in Java is done using the `System.out.print` or `System.out.println` functions. This function allows to display literal strings in quotes (e.g., `System.out.println ("Hello world!");`) or simple or complex Java expressions (e.g., `System.out.println(a+b);`).

2.4.4 Reading Data from Keyboard

Recapitulation 2.4



Regarding reading data from the keyboard, we need first to import the Scanner class, for this we write at the beginning of the program: `import java.util.Scanner;`, *then we can create a variable of type Scanner as follows:* `Scanner read = new Scanner (System.in);`, *then, the read variable can be used as many times as necessary to request values from the keyboard, and depending on the desired data type the appropriate method is used, as illustrated by Listing 2.3:*

```
//import java.util.Scanner; in the preambule of the program
 Scanner read = new Scanner(System.in);
 int n=read.nextInt();
 double d=read.nextDouble();
 float f=read.nextFloat();
 String s=read.nextLine();
 char c=read.next().charAt(0);
 //and so on ...
```

Listing 2.3: Reading data from keyboard

2.4.5 Control Statements

Recapitulation 2.5

Java provides several control statements to manage program flow, including:

- *Conditional Statements: if, if-else, nested-if, if-else-if*
- *Switch-Case: For multiple fixed-value checks*
- *Jump Statements: break, continue, return*

Java's control statements are largely analogous to those found in C. The Listing 2.4 illustrates the syntax of the Java's control statements

```
 //if without else
 if(test) //test is a logic expression
 {  block_of_instructions }

 //if with one instruction
 if(test) one_intruction;
```

```
//if with else
if(test) {  block_of_instructions }
else
{  block_of_instructions }

//if with else if
if(test)
{  block_of_instructions  }
else if {  block_of_instructions }

// switch statement
switch (expression) {
  case value1:// Statements
  break; // Exits the switch block
  case value2:// Statements
  break;

  // ... any number of cases

  default: // Statements executed if no case matches (optional)
}
```

Listing 2.4: Control statements in Java

2.5 Array & ArrayList

2.5.1 Array

Recapitulation 2.6

- *In Java, an array is a data structure designed to store a fixed-size sequence of elements of the same data type, that are accessed using their index, which starts from 0.*
- *In Java, an array can be declared in several ways (see Listing 2.5 for single-dimensional arrays and Listing 2.6 for multi-dimensional arrays).*
- *Using a traditional for loop with an index for iterating through Arrays, as follows:*

```
for (int i = 0; i < t.length; i++) {
```

```
2      System.out.println(t[i]);
3    }
```

- *The length property provides the number of elements in the array (*`int s=t.length`*).*
- *Attempting to access an index outside the valid range (0 to length - 1) will result in an ArrayIndexOutOfBoundsException.*
- *The elements of an array can be of any type, not just a primitive type.*

```
1 int tab1[] = {1, 2, 3};// the same as language C
2 int[] tab2 = {1, 2, 3};
3 int[] tab3 ;
4 tab3=new int[6];/tab3 refers to an array of 6 integers
5 int []t1,t2;//t1 et t2 are references to arrays of integers
6 int t[10] ; // error: we cannot indicate the size like this
```

Listing 2.5: Declaration of simple arrays in Java

```
1 // these three statements are equivalent
2 int t [] [] ;
3 int [] t [] ;
4 int [] [] t ;
5
6 //an array of two rows and a varying number of columns
7 int t [] [] = {new int [3], new int [2]};
```

Listing 2.6: Declaration of multi-dimensional arrays in Java

2.5.2 Dynamic Array (List)

Recapitulation 2.7

Dynamic arrays (lists) have the particularity of being able to change size during program execution. In Java, they are defined using the ArrayList class from the `java.util.ArrayList` *library, as follows:*

```
1   ArrayList <Type> myList;
2   myList = new <Type> ArrayList();
```

The type of the elements of an arraylist cannot be a simple type (int or double for example). It

must be one of the advanced types corresponding to simple data types. For example: Integer for int, Float for float, Double for double, etc. As it can be a specific class (Person for example). The following example defines an arraylist of integer:

```
import java.util.ArrayList; // Import the ArrayList class
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

The ArrayList class offers various methods for manipulating an arraylist elements. Some of the most commonly used ArrayList methods are summarized in Table 2.2. An example of adding and getting elements of an ArrayList of String is given bellow:

```
ArrayList<String> cars = new ArrayList<String>();
cars.add("Mazda");
cars.add("Ford");
cars.add("BMW");
cars.add("Volvo");
for (int i = 0; i < cars.size(); i++) {
System.out.println(cars.get(i));
}
```

R To loop through the elements of an ArrayList we use a for loop with the the size() method to specify how many times the loop should run. If an attempt to access an element at an index that is outside the valid range when looping through an ArrayList in Java, an IndexOutOfBoundsException occurs. The valid indices for an ArrayList are 0 to size() - 1.

Method	Returned result
$size()$	Returns the number of elements in this list.
$add(element)$	Appends the specified element to the end of this list.
$add(int\ index, element)$	Inserts the specified element at the specified position in this list.
$get(int\ index)$	Returns the element at the specified position in this list
$remove(int\ index)$	Removes the element at the specified position in this list
$set(int\ index, element)$	Replaces the element at the specified position in this list with the specified element
$isEmpty()$	Returns true if this list contains no elements
$clear()$	Removes all of the elements from this list
$indexOf(element)$	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element
$contains(element)$	Returns true if this list contains the specified element

Table 2.2: Some of the most commonly used methods of the class ArrayList

2.6 Characters & String

Recapitulation 2.8

- *char is a primitive data type in Java, which directly stores the character value in memory.*
- *A char variable can hold only a single character, enclosed in single quotes (* `char c='B';`*).*
- *The Character class provides utility methods for character manipulation, for example:* `Character.isDigit('5'); Character.toUpperCase('r'); Character.isLowercase ('Z'); Character.isLetter('a').`
- *In Java, converting a char to its numerical representation can be achieved in several ways, depending on whether you want the character's Unicode/ASCII value or its numeric digit value if the character represents a digit.*
- *We can use the Unicode/ASCII Value (Implicit Type Casting):*

```
char c = 'A';
int Value = c; // Value will be 65 (ASCII/Unicode value of 'A')
char d = '5';
```

```
4 int digiteValue = d; // digiteValue will be 53 (ASCII/Unicode value of '5'
```

- *If the char represents a digit ('0'-'9') and we want its actual numeric value, we can use* `Character.getNumericValue()` *method: this method returns the integer value represented by the character:*

```
1 char c = '2';
2 int n = Character.getNumericValue(c); // n will be 2
3 System.out.println("Numeric value of '2': " + n);
4
```

Recapitulation 2.9

- *String is a class in Java, and String variables are objects that stores sequences of characters, that can hold zero or more characters, enclosed in double quotes ("", "Java", "Java Programming", ...).*
- *String in Java are immutable; their content cannot be changed after creation. Any operation that attempts to modify a String actually creates a new String.*
- *The String class provides a variety of methods for manipulating and working with strings, such as concatenation, comparison, searching, and substring extraction. Table 2.3 summarizes some of the most commonly used functions of the String class.*

Method	Returned result
$length()$	Returns the number of characters in the string
$charAt$	Returns the character at the specified index
$compareTo$	Compares two strings lexicographically
$startsWith$	Checks if the string begins with the specified prefix. Returns $true$ ou $false$
$endsWith$	Checks if the string ends with the specified suffix. Returns $true$ ou $false$
$equals$	Compares two strings for equality (case-sensitive)
equalsIgnoreCase	Compares two strings for equality, ignoring case differences. Returns $true$ or $false$
$contains$	Checks if the string contains the specified sequence of characters, returns $true$ or $false$
$indexOf$	Returns the index of the first occurrence of the specified substring
$isEmpty$	Checks if the string is empty (has a length of 0)
$toLowerCase$	Converts all characters in the string to lowercase
$toUpperCase$	Converts all characters in the string to uppercase

Table 2.3: Some of the most commonly used methods of the class String

2.7 Static Functions

Recapitulation 2.10

A static method in Java is a part of the class definition. We can define a static method by adding the `static` keyword to a method. Static methods can be called directly using the class name, without the need to create an object of that class.

```
1 public class MathOperations {
2       // Static method to add two numbers
3   public static int add(int a, int b) {return a + b;}
4       // Static method to multiply two numbers
5   public static int multiply(int a, int b) {  return a * b;}
6   public static void main(String[] args) {
7         // Calling static methods:
8 int sum = add(4, 2);
9 System.out.println("Sum: " + sum); // Output: Sum: 6
```

```
10 int product = multiply(5, 3);
11 System.out.println("Product: " + product); // Output: Product: 15
12     }}
```

R We will talk in detail about methods in the next chapter.

2.8 Quick Check

Quick Check 2.1 *Which one of these instructions is correct for printing text on screen ?*

1. `System.Out.print`
2. `System.out.Print`
3. `System.out.print`
4. `system.Out.print`

Quick Check 2.2 *Which of these instructions contain errors?*

1. `System.out.print(Hello World);`
2. `System.out.print("Hello World")`
3. `System.out.Print("Hello World");`
4. `System.out.print("Hello World");`

Quick Check 2.3 *What will be the value of sum after the following nested for loops are executed?*

1.

```
1     int sum = 0;
2     for (int i = 0; i < 5; i++) {
3       sum = sum + i;
4       for (int j = 0; j < 5; j++) {
5         sum = sum + j;
6       }
7   }
```

2.

```
1     int sum = 0;
2     for (int i = 0; i < 5; i++) {
```

```
        sum = sum + i;
        for (int j = i; j < 5; j++) {
          sum = sum + j;
        }
    }
```

Quick Check 2.4 *Which of these statements are invalid?*

1. `Person[25] person;`
2. `Person[ ] person;`
3. `Person person[] = new Person[25];`
4. `Person person[25] = new Person[25];`

Quick Check 2.5 *What is the output of this code?*

```
int[][] table = new int[10][5];
System.out.println(table.length);
System.out.println(table[4].length);

```

Quick Check 2.6 *What is the output from the following code?*

```
List<String> list = new ArrayList<String>();
for(int i = 0; i < 6; i++) {
  list.add("element " + i);
}
list.remove(1);
list.remove(3);
System.out.println(list.get(2));
```

Quick Check 2.7 *What will be the output of the following codes:*

1.
```
    char []C=new char[10]
```

```
2        for(int i=0;i<10;i++)
3        {
4          C[i]='i';  System.out.print(C[i]);
5        }
```

(2)

```
1        int T[]={0,1,2,3,4,5,6,7,8,9};
2        int n=6;
3        n=T[T[n]/2];
4        System.out.println(T[n]/2);
```

Quick Check 2.8 *Which data type is used to create a variable that should store text?*

(1) *text*

(2) *String*

(3) *string*

(4) *myString*

Quick Check 2.9 *Which method can be used to find the length of a string?*

(1) *size()*

(2) *len()*

(3) *getLength*

(4) *length()*

Quick Check 2.10 *Which method can be used to find the length of an ArrayList?*

(1) *size()*

(2) *len()*

(3) *getLength*

(4) *length()*

2.9 Activities

Exercise 2.1 *Rewrite the following program in Java:*

```
#include <stdio.h>
int main() {
  int a, b, sum;
  printf("Enter two numbers: ");
  scanf("%d %d", &a, &b);

  sum = a + b;

  printf("Sum = %d\n", sum);
  return 0;
}
```

Exercise 2.2 *Write a Java program that asks the user to enter his name, family name, and age, and then, these information will be displayed by the computer.*

Test Data:

- *Name : Moustapha*
- *Family name : Mohammed*
- *Age : 21*

Expected Output: ***You are Moustapha Mohammed and you are 21 years old***.

Exercise 2.3 *Write a Java program to print the sum, multiplication, division, and the remainder of the division of two numbers*

Exercise 2.4 *Write a Java program to compute the area and perimeter of a circle.*

Test Data: Radius = 5

Expected Output: Area is 78,539816

Perimeter is 31,415926

R *To get π you can use the predefined function $Math.PI$*

Exercise 2.5 *Write a Java program to compute the distance between two points P, W. The two points are defined, in a two-dimensional space, by their coordinates as follow:*

- P : (x_1, y_1) *and,*
- W : (x_2, y_2).

R

- *All data are real numbers.*
- *Distance*(P, W)=$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- *You can use predefined functions:* $Math.sqrt()$ *et* $Math.pow()$

Exercise 2.6 *Write a program that inputs temperature in degrees Celsius and prints out the temperature in degrees Fahrenheit. The formula to convert degrees Celsius to equivalent degrees Fahrenheit is:*

*Fahrenheit = 1.8 * Celsius + 32*

Exercise 2.7 *Write a program that does the reverse of Exercise 2.6, that is, input degrees Fahrenheit and prints out the temperature in degrees Celsius. The formula to convert degrees Fahrenheit to equivalent degrees Celsius is Celsius =5/9 * (Fahrenheit - 32)*

Exercise 2.8 *Write a program that inputs the year a person is born and outputs the age of the person in the following format:*

You were born in 1990 and will be (are) 18 this year.

Exercise 2.9 *If you invest P dollars at R percent interest rate compounded annually, in N years, your investment will grow to $P(1 + \frac{R}{100})^N$ dollars. Write an application that accepts P, R, and N and computes the amount of money earned after N years.*

Exercise 2.10 *Your weight is actually the amount of gravitational attraction exerted on you by the Earth. Since the Moon's gravity is only one-sixth of the Earth's gravity, on the Moon you would weigh only one-sixth of what you weigh on Earth.Write a program that inputs the user's Earth weight and outputs her or his weight on Mercury, Venus, Jupiter, and Saturn. Use the values in the following table:*

Planet	***Multiply the Earth Weight by***
Mercury	*0.4*
Venus	*0.9*
Jupiter	*2.5*
Saturn	*1.1*

Exercise 2.11 *When you say you are 18 years old, you are really saying that the Earth has circled the Sun 18 times. Since other planets take fewer or more days than Earth to travel around the Sun, your age would be different on other planets. You can compute how old you are on other planets by the formula:*

$$y = \frac{x \times 365}{d}$$

where x is the age on Earth, y is the age on planet Y, and d is the number of Earth days the planet Y takes to travel around the Sun. Write an application that inputs the user's Earth age and print outs his or her age on Mercury, Venus, Jupiter, and Saturn. The values for d are listed in the following table:

Planet	d***: Number of Earth Days for This Planet to Travel around the Sun***
Mercury	*88*
Venus	*225*
Jupiter	*4.380*
Saturn	*10.767*

Exercise 2.12 *Write a program that accepts a person's weight and displays the number of calories the person needs in one day. A person needs 19 calories per pound of body weight, so the formula expressed in Java is calories = bodyWeight * 19;*

Exercise 2.13 *Write a program that accepts the unit weight of a bag of coffee in pounds and the*

number of bags sold and displays the total price of the sale, computed as:

$$totalPrice = unitWeight * numberOfUnits * 5.99;$$

$$totalPriceWithTax = totalPrice + totalPrice * 0.0725;$$

where 5.99 is the cost per pound and 0.0725 is the sales tax. Display the result in the following manner:

```
Number of bags sold:  32
Weight per bag:  5 lb
Price per pound:  $ 5.99
Sales tax:  7.25Total price:  $ 1027.884
```

Exercise 2.14 *A perfect number is a positive integer that is equal to the sum of its proper divisors. A proper divisor is a positive integer other than the number itself that divides the number evenly. For example, 6 is a perfect number because the sum of its proper divisors 1, 2, and 3 is equal to 6. Eight is not a perfect number because* $1+2+4 \neq 8$*. Write a program that accepts a positive integer and determines whether the number is perfect. Also, display all proper divisors of the number. Try a number between 20 and 30 and another number between 490 and 500.*

Exercise 2.15 ***Guess the number***

"Guess the number" is a simple game where the user tries to guess a number randomly generated by the computer within a specified range. The game rules and steps are as follow:

- *The computer generates a random number* n *from a given range, let's* $n \in$ *[1,10];*
- *The user tries to guess the generated number and inter his given number using keyboard;*
- *The user has a limit number of attempts,* k*, and let's* $k = 5$*;*
- *The computer replays if the generated number* n *matches the guesses number or it is lower/higher than it.*
- *If the guessed number is equal to* n *or if the* K *attempts are reached, the program ends with an appropriate message;*
- *For each attempt, the program also displays the number of attempts remaining;*
- *You can also incorporate further details as displaying score and giving points based on the number of attempts.*

R *Use the following lines in order to generate a random integer:*

```
import java.util.Random;
//Par exemple, générer un nombre dans l'intervalle [a,b]:
Random random = new Random();
int nb=a+random.nextInt(b-a);

```

Exercise 2.16 *Let us take exercise 2.5 again.*

① *Represent the coordinates of n points in a two-dimensional space, using a two-dimensional table, named $PointsT$, and fill it randomly;*

② *Let $P(x, y)$ be a point. Calculate and display the distance between P and all points in the table $PointsT$, as well as the minimum distance and the point in the table $PointsT$ that has the minimum distance.*

R *Coordinates of the n points are real numbers. For generating a real random number in a range $[a, b]$:*

```
import java.util.Random;
Random random = new Random();
double nb = a+random.nextDouble(b-a);

```

Exercise 2.17 *Write a program that computes the standard deviation of N real numbers saved in an ArrayList (or array). The standard deviation S is computed as follows:*

$$S = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_N - \bar{x})^2}{N}}$$

The variable $\bar{x}$ is the average of N input values x_1 through x_N. In the case where an array is used, the program first prompts the user for N and then declares an array of size N. Noting that the N real values can be generated randomly as well as N.

Exercise 2.18 *Use an array-list of integers to represent the first n element of the Fibonacci sequence. Then, compute and display their sum.*

R *The Fibonacci sequence is defined as follow:*

$$\begin{aligned} U_0 &= 0, \\ U_1 &= 1, \\ U_n &= U_{n-1} + U_{n-2} \text{ for } n >= 2 \end{aligned}$$

Exercise 2.19

① *Display alphabet in uppercase (A to Z) and lowercase (a to z).*

② *Convert all characters of a given string st to uppercase.*

③ *Display the number of uppercase characters, lowercase characters, vowels, numeric characters in a given string St.*

R *Use the following statements for the second question:*

```
Character.isLowerCase('a')
Character.isUpperCase('a')
Character.isLetter('a')
Character.isDigit('2')

```

Exercise 2.20 *Resume Exercise 2.17. Modularize the code using auxiliary methods to write a method named $Standarddeviation$, that takes as input an array of integers and displays their standard deviation. Each function must has one responsibility (task).*

Exercise 2.21 ***Functions for array manipulation***

We want to define a collection of functions for manipulating tow-dimensional array. In order to do that, start by defining the following static functions:

① *A function named random, that returns as value a random integer $\in [1, 10]$;*

② *A function named $fill$, that fills randomly a table t of integer;*

③ *A function named* $display$*, that displays values of a table* t *of integer;*

④ *A function named minT, that returns as value the min value of a table* t *of integer;*

⑤ *A function named maxT, that returns as value the max value of a table t of integer;*

⑥ *A function named somT, that returns as value the sum of all values of a table t of integer;*

As second step, use the above functions to write the following functions:

① *A function named* $fill$*, that fills randomly a matrix* m *of integer;*

② *A function named* $display$*, that displays values of a matrix* m *of integer;*

③ *A function named minT, that returns as value the min value of a matrix m;*

④ *A function named maxT, that returns as value the max value of a matrix m;*

⑤ *A function named somT, that returns as value the sum of all values of a matrix m;*

Finally, in order to test the above defined functions, use these functions in a main program to fill randomly an irregular table T *and display values of* T*, the min, the max, and the sum of all values of* T*. The size of each line of* T *must be also a random value.*

R *An example of an irregular table is shown in the Exercise 2.27.*

Exercise 2.22 ***Amicable numbers***

In mathematics, two natural numbers n *and* m *are said to be amicable (friendly or amicable) if the sum of the proper divisors of* n *equals* m *and the sum of the proper divisors of* m *equals* n*. The proper divisors of a number* m*, noted* $div_p(m)$*, are all divisors of* m *accept itself. For example, the proper divisors of 6,* $div_p(6)$ *= {1, 2, 3}. So, two numbers* m *and* n *are amiable if:* $\sum div_p(m)$*=* n *and* $\sum div_p(n)$*=*m*.*

For example: 220 and 284 are amicable numbers, because:

$\sum div_p(220)$= $1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$

and

$\sum div_p(284)$= $1 + 2 + 4 + 71 + 142 = 220$

An other definition (manner), the amicable numbers are two different natural numbers related in such a way that the sum of the divisors of each are equals, and equals to their sum. The divisors of a number m *are all divisors of* m*, noted* $\sigma(m)$*. For example,* $\sigma(6)$ *= {1, 2, 3, 6}. So, two numbers* m *and* n *are amiable if:* $\sum \sigma(m)$*=* $\sum \sigma(n)$*=*$n + m$*.*

For example: 220 and 284 are amicable numbers, because:

$\sum \sigma(220)$= $1+2+4+5+10+11+20+22+44+55+110+220 =$ ***504***

and

$\sum \sigma(284)$= $1+2+4+71+142+284 =$ ***504***

and $220 + 284 =$ ***504***

- ***Requested work****:*

  Modularize the code using auxiliary methods to write a method named $FriendNumbers$*, that takes as input an array of integers and displays all pairs of friendly numbers within it. Each pair will be displayed only once.*

R *The first seven amicable pairs are: (220, 284), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368), (10744, 10856), (12285, 14595).*

2.10 Supplementary Activities

Exercise 2.23 ***Closest number to zero***

Write a program that displays the temperature closest to 0 among input data. If two numbers are equally close to zero, positive integer has to be considered closest to zero (for instance, if the temperatures are -4 and 4, then display 4).

- *Input 1:* N*, the number of temperatures to analyze*
- *Input 2:* N *temperatures expressed as integers ranging from -273 to 5526*
- *Output: Display 0 (zero) if no temperatures are provided. Otherwise, display the temperature closest to 0.*
- *Constraints :* $0 < N \leq 10$

Exercise 2.24 *You need to figure out what to do by observing the following input/output:*

- *Input: 5 4*
- *Output:*

```
*****
*   *
*   *
```

*****

Exercise 2.25 ***Prime numbers***

Display the sum of the first N prime numbers. For example, if $N = 4$, we get four prime numbers: $2, 3, 5, 7$ and the sum of these numbers is 17.

Exercise 2.26 *Use an array-list of integers to represent the first n element of the Syracuse sequence. Then, compute and display their sum as well as the min and the max values.*

R *The Syracuse sequence is defined as follow:*

$$
\begin{aligned}
U_0 &\in N^*, \text{ we choose } U_0 = 7 \\
U_{n+1} &= \frac{U_n}{2}, \text{ if } u_n \text{ is even} \\
U_{n+1} &= 3U_n + 1, \text{ if } u_n \text{ is odd}
\end{aligned}
$$

Exercise 2.27 ***Irregular tables***

Irregular arrays are multidimensional arrays such that each row has different size. Write a Java program that declares, initializes, and displays the following irregular array:

1

2 3

4 5 6

7 8 9 10

Exercise 2.28 *The purpose of this exercise is to write a program to identify the most frequently occurring element in an array of integers. This program should also display the number of occurrences in the array of this most frequent element. For example, for the following array:*

{2, 7, 5, 6, 7, 1, 6, 2, 1, 7}

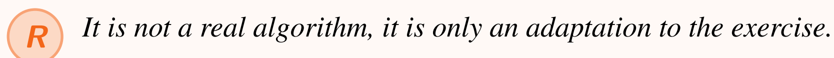
The program should indicate that the most frequent element is 7 and that its frequency of appearance is 3.

Exercise 2.29 ***ASCII Code***

Assuming a string represents the address of a website, for example:`St="www.univjijel.dz"`*. To calculate the IP address, we need to convert all the characters of St to ASCII code and then calculate the sum of the obtained values. For xample, "abc" $\Longrightarrow$ 97 + 98 + 99 = 294*

The first IP number will be the result mod 256 (Because there is no IP number greater than 256). The second IP number will be double of the sum mod 256, the third will be triple of the sum mod 256 and the fourth will be quadruple of the sum, mod 256. For example:

- *input: "www.univjijel.dz"*
- *output: Ip adress: 111.222.77.188*



CHAPTER 3

BASICS OF POO

Contents

3.1 Classes and Objects 35

3.2 Attributes and Methods 35

3.3 Constructors 38

3.4 Static Attributes and Methods 39

3.5 Visibility and Accessor Methods 40

3.6 Quick Check 42

3.7 Activities 46

Key Objectives

The key objectives of this chapter are to introduce the fundamental concepts of object-oriented programming (OOP) and to help student understand how to create and use classes and objects in Java. By the end of the chapter, student should be able to define classes with attributes and methods, create objects from these classes, and understand the relationship between classes and their instances. To reinforce these concepts and give student hands-on experience, the chapter concludes with a series of practical activities, allowing him to apply what he has learned and build confidence in working with classes and objects.

3.1 Classes and Objects

Recapitulation 3.1

- *Classes and objects are the two main aspects of object-oriented programming. A class is a template for objects, and an object is an instance of a class.*
- *To create a class, we use the keyword* `class`*, as illustrated by the follwing example, where we define a class named Student. Inside the class, we declare variables (attributes) and methods.*

```
public class Student {
      int age;
      String name;
      void display(){System.out.println(name+" "+age);}
}
```

Listing 3.1: Student class in Java

- *An object is created from a class. We have already created the class named Student, so now we can use this to create objects. To create an object of Student, specify the class name, followed by the object name, and use the keyword* `new`*:*

```
public static void main(String[] args) {
   Student myObj = new Student();
   System.out.println(myObj.name);
   myObj.display();
}
```

Listing 3.2: Creation of an object of the Student class in Java

3.2 Attributes and Methods

Recapitulation 3.2

- *In the class Student (Listing 3.1), we declared two variables, age and name. They are actually attributes of the class (class attributes are variables within a class). They represents the state of an object and reflects the properties of an object.*

- *Class methods are functions defined within a class that describe the behaviors or actions that objects of the class can perform. These methods can manipulate object data, perform operations, and return results. As example, in Listing 3.1, the display() method is an instance method that displays the information of a student (age and name).*
- *We can access attributes and methods by creating an object of the class, and by using the dot syntax, as used in the previous example (Listing 3.8).*

Recapitulation 3.3

- *In a class Java we can have multiple methods with the same name, this is called method* ***overloading****.*
- *The key distinction between overloaded methods lies in their parameter lists. This difference can be achieved in different ways:*
  - ***Different Number of Parameters****: Methods can have the same name but accept a varying number of arguments.*
  - ***Different Data Types of Parameters****: Methods can have the same name and the same number of parameters, but the data types of those parameters must differ.*
  - ***Different Order of Parameters*** *(with different types): If the data types are different, changing the order of parameters can also lead to method overloading.*

Recapitulation 3.4

Java always uses "pass by value" for method parameters, but this has different consequences depending on whether the parameter type is simple (int, double, etc.) or advanced (object).

① *When a primitive type variable is passed to a method, a copy of its value is made and assigned to the method's formal parameter. Any modifications to this parameter within the method will only affect the local copy, and the original variable in the calling code remains unchanged. Let's consider the Listing 3.3, values of variables x and y, remained the same after calling modify methods.*

② *When an object is passed to a method (such as array, Arraylist, Person, Student, ...), a copy of the object's reference (memory address) is passed by value. This means both the original variable and the method parameter refer to the same object in memory. If any modification of the state of the object (change a field's value) through the passed reference, these changes will be reflected in the original object because both references*

point to the same object. Let's consider the Listing 3.3, value of the first element of the array tab changed after calling modify method. However, changes on tab itself (reassign the parameter to a new object within the method `a=new int[5]`) does not affect the original variable outside the method. This is because changes made within the method to the reference itself are not visible outside the method.

```
public class PassingArguments {
  static void modify(int a){a=12;}
  static void modify(int a,int b){
    int z = a;
    a= b;
    b = z;}
  static void modify(int [] a){a[1] = 0; a = new int[5];}
  public static void main(String[] args)
  {
    int x=10;  int y=8;
    int tab[]={3,4,5};
    System.out.print("Before modification: ");
    System.out.println(x+" "+y+" "+tab[1]+" "+tab.length);
    modify(x); modify(x,y);
    modify(tab);
    System.out.print("After modification : ");
    System.out.println(x+" "+y+" "+tab[1]+" "+tab.length);
    // Execution output:
    // Before modification: 10 8 4 3
    // After modification : 10 8 0 3
}}
```

Listing 3.3: Example of passing parameters in Java

3.3 Constructors

Recapitulation 3.5

- *Constructor is a special method used to initialize new objects of a class. It has the same name as the class. The constructor is called when an object of a class is created. Constructors can also take parameters, which is used to initialize attributes.*
- *Constructors can also take parameters, which is used to initialize attributes with specific values provided during object creation.*
- *A class can have multiple constructors, as long as they have different parameter lists (different number of parameters, different types of parameters, or different order of parameter types). This allows for various ways to initialize an object.*
- *Unlike methods, constructors do not have a return type, not even void.*
- *If a class does not explicitly define any constructors, Java automatically provides a default, no-argument constructor. This default constructor initializes instance variables with their default values (e.g., 0 for numeric types, null for object references, false for booleans).*
- *If any custom constructor is defined (with or without arguments) in a class, Java will not automatically provide the default constructor. If we need a no-argument constructor, in such a case, we must explicitly define it.*
- *We can add contructors to the Student class as follows:*

```java
 public class Student {
   int age;
   String name;
   Student(int a, String n){age=a; name=n;} // Custom constructor
   Student(){age=20; name="Ahmed";}  // a no-argument constructor
   void display(){System.out.println(name+" "+age);}
}
```

Listing 3.4: Student class in Java with two constructors

3.4 Static Attributes and Methods

Recapitulation 3.6

- *The static keyword means that a member of a class (attribute or method) belongs to the class itself, rather than to any specific instance of that class. Thus, we can access static members without creating an instance of an object.*
- *When we declare an attribute static, exactly a single copy of that attribute is created and shared among all instances of that class (The value of this static attribute is shared across all objects of the same class).*
- *A static attribute can be used, for example, to track the number of objects created. This allows the counter to be incremented with each new object. As we can see in the Listing 3.5, the static variable numberOfStudent will be incremented each time we instantiate the Student class.*

```java
public class Student {
  int age;
  String name;
  static int numberOfStudent; // static variable
  Student(int a, String n){age=a; name=n; numberOfStudent++;}
  Student(){age=20; name="Ahmed"; numberOfStudent++;}
  void display(){System.out.println(name+" "+age);}
}
```

Listing 3.5: Student class in Java with a static attribute

Let's create two Student objects and expect the counter to have a value of two:

```java
public static void main(String[] args) {
  Student s1 = new Student();
  Student s2 = new Student();
  System.out.println(Student.numberOfStudent); // 2
  myObj.display();}
```

Listing 3.6: Creation of two objects of Student class in Java

- *Static variables can be accessed through an instance (`s1.numberOfStudent`) or directly from the class (`Student.numberOfStudent`).*

- *Similar to static attribute, static methods also belong to a class instead of an object. So, we can invoke them without instantiating the class. Generally, static methods are used to perform an operation that's not dependent upon instance creation.*
- *Static methods can be used to create utility or helper classes. A popular example is Math utility class (`Math.sqrt(), Math.pow()`).*

R

- Instance methods can directly access both instance methods and instance variables;
- Instance methods can also access static variables and static methods directly
- Static methods can access all static variables and other static methods
- Static methods can't access instance variables and instance methods directly. They need some object reference to do so.

3.5 Visibility and Accessor Methods

Recapitulation 3.7

- *The visibility of methods and properties within a class determines where they can be accessed. This is controlled by access modifiers such as `public` (accessible from anywhere), `private` (only accessible within the class that defines them), `protected` (accessible within the defining class and by any class that inherits from it), and default (accessible within the same package).*
- *To retrieve and update the value of a private attribute of a class, getter and setter methods are used.*
- *The get method returns the variable value, and the set method sets the value. Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case. Consider Student class again, but this time with private instance variables for name and age:*

```
1     public class Student {
2       privare int age;
3       private String name;
4       static int numberOfStudent; // static variable
5       Student(int a, String n){age=a; name=n; numberOfStudent++;}
```

```
6        Student(){age=20; name="Ahmed"; numberOfStudent++;}
7
8        // Getter method for name
9        public String getName() {return name;}
10        // Setter method for name
11       public void setName(String name) {this.name = name;}
12       // Getter method for age
13       public int getAge() {return age;}
14       // Setter method for age with check of validation
15       public void setAge(int age) {
16         if (age > 0) { // Ensure age is a positive value
17           this.age = age;
18         } else {
19           System.out.println("Invalid age, age must be a positive number.");
20         }
21       }
22
23       void display(){System.out.println(name+" "+age);}
24   }
```

Listing 3.7: Student class with private attributes

In this case, because Name attribute is private within the class Student, the access to this attribute outside of class Student is doing by using get method for retrieving its value and using set mathod for updating its value. As example, Listing 3.8, becomes as follows:

```
1 public static void main(String[] args) {
2   Student myObj = new Student();
3   // Because Name attribute is private within the class, the access to it is  by
    using get and set methods
4   myObj.setName("Mohammed");
5   System.out.println(myObj.getName());
6   myObj.display();
7 }
```

Listing 3.8: Retrieve and update value of age field outside of class Student

3.6 Quick Check

Quick Check 3.1 *Which of the following constructors, of a class named ClassA, are invalid?*

①
```java
public int ClassA(int one) {
    ...
}
```

②
```java
public ClassA(int one, int two) {
    ...
}
```

③
```java
void ClassA( ) {
    ...
}
```

Quick Check 3.2 *Which statement about instance methods is correct?*

① *They can only be accessed from static methods.*

② *They require an object to be called.*

③ *They are automatically synchronized*

④ *They must be declared final*

Quick Check 3.3 *What will happen when calling a static method using an object?*

① *Runtime error*

② *Compilation error*

③ *Runs without errors*

④ *NullPointerException*

Quick Check 3.4 *What is the correct way to create an object called myObj of MyClass?*

① `class myObj= new MyClass()`

② `class myObj= new myObj()`

③ `new myObj=MyClass`

④ `MyClass myObj= new MyClass()`

Quick Check 3.5 *What is method overloading?*

① *Two methods with the same name and parameters*

② *Two methods with the same name but different parameters*

③ *Two methods with the same name but different return types*

④ *Defining methods inside a loop*

Quick Check 3.6 *Which of the following is true about Java constructors?*

① *Constructors have a return type of void.*

② *Constructors must always be public.*

③ *A constructor is automatically called when an object is created.*

④ *Constructors can only be defined explicitly.*

Quick Check 3.7 *What happens if you do not define a constructor in a Java class?*

① *The program will not compile.*

② *The compiler provides a default constructor.*

③ *The class cannot be instantiated.*

④ *The constructor from another class will be used.*

Quick Check 3.8 *Can a Java constructor be private?*

① *No, it must be public.*

② *Yes, but the class cannot be instantiated outside the class.*

③ *Yes, and it can be accessed from anywhere.*

④ *No, Java does not allow private constructors.*

Quick Check 3.9 *How can one constructor call another constructor within the same class?*

① *Using super()*

② *Using this()*

③ *Using new()*

④ *Constructors cannot call each other*

Quick Check 3.10 *What will be the output of this program?*

```java
class Ab {
  int num;
  Ab() {
    this(100);
    System.out.println("Default Constructor");
    }
  Ab(int n) {
    num = n;
    System.out.println("Parameterized Constructor");
      }
}
public class Main {
  public static void main(String[] args) {
    Ab obj = new Ab();
  }
}
```

Quick Check 3.11 *Consider the following class declaration:*

```java
class QuestionOne {
    public final int A = 345;
    public int b;
    private float c;
    private void methodOne( int a) { b = a;}
    public float methodTwo( ) { return 23;}
}
```

Identify invalid statements in the following main class. For each invalid statement, state why it is invalid.

```
class Q1Main {
  public static void main(String[] args) {
    QuestionOne q1;
    q1 = new QuestionOne( );
    q1.A = 12;
    q1.b = 12;
    q1.c = 12;
    q1.methodOne(12);
    q1.methodOne( );
    System.out.println(q1.methodTwo(12));
    q1.c = q1.methodTwo( );
  }
}
```

Quick Check 3.12 *Consider the following class:*

```
class QuestionTwo {
 private int count;
 public void init( ) { count = 1; }
 public void increment( ) { count = count + 1; }
 public int getCount( ) { return count; }
}
```

What will be the output from the following code:

```
class Q2Main {
  public static void main(String[] args) {
    QuestionTwo q2;      q2 = new QuestionTwo( );
    q2.init();
    q2.increment();
    q2.increment();
    System.out.println(q2.getCount());
  }}
```

Quick Check 3.13 *Consider the following class:*

```
class QuestionThree {
 public int count;
 public void init( ) { count = 1; }
 public int increment( ) {
    count = count + 1;
    return count;
    }
}
```

What will be the output from the following code:

```
class Q3Main {
 public static void main(String[] args) {
    QuestionThree q3;
    q3 = new QuestionThree( );
    q3.init();
    q3.count = q3.increment() + q3.increment();
    System.out.println(q3.increment());
 }
}

```

3.7 Activities

Exercise 3.1 ***Creation of Student class***

A student will be modeled here by the $Student$ class of a package named $StudentmanagerGx$, as follow:

The Student class has three attributes:

- *Name: String*
- *Fname: String*
- *Marks: an arraylist of integers*

The Student class has the following methods:

- *SetName: to enter the name of the student*
- *SetFname: to enter the fammily name of the student*
- *AddMark: to enter a mark of the student*
- *RemoveMark: to delete the mark i of the student*
- *Average: compute and returns as value the average of marks of the student*
- *Display: to display information of the student (name, fname, and average of marks)*
- *DisplayMarks: to display marks of the student*
- *SetMarks: to update the mark i of the students if exist.*

  The Student class has the following constructors:
  - *a constructor that allows you to initialize the student's name and Fname (from arguments).*
  - *a constructor that allows you to initialize the student's name and Fname and a mark (from arguments).*
  - *a default constructor that allows you to initialize the student's name as "unknown Name" and Fname as "unknown FName" and a mark value 0.*

The student class has the following static functions:

- *a function named Randm, allows to generate a random integer.*
- *a boolean function named Compare, allows to compare two students according to their average. The function returns true if the average of the first student is greater than the average of the second one.*

In a main class:

1. *use the constructors defined above to create two new students $S1$ and $S2$ named "Ahmed Ismail" and "Khadidja Mouhammed". Then, give to each one three marks. Finally, display these students with their averages and marks.*
2. *create a list $StudentList$ of students. Then, add four students to this list. Give to each one three random marks. Finally, display these students with their averages and marks.*
3. *Display the student with the max average, the passed and failed students.*

Solution

```
package studentmanager;
import java.util.ArrayList;
import java.util.Scanner;
public class Students {
   String Name;
   String Fname;
   ArrayList <Integer> marks=new ArrayList <Integer>();

   void setName(){
       Scanner in=new Scanner(System.in);  Name=in.nextLine(); }
   void setFname(){ Scanner in=new Scanner(System.in); Fname=in.nextLine();
   }
   void setName(String s){  Name=s;}
   void setFname(String s){Fname=s;   }
   void addmark(int m){ marks.add(m); }
   void removemark(int i){ if (i<marks.size()) marks.remove(i);}
   double average(){
       int s=0;
       for(int i=0;i<marks.size();i++) s=s+marks.get(i);
       return s/marks.size();
   }
   void Display(){System.out.println(Name+" "+ Fname+" : "+average());}
   void DisplayMarks(){
       for(int i=0;i<marks.size();i++) System.out.println(i+" : "+ marks.get(i));
   }}
```

```
package studentmanager;
package studentmanager;
import java.util.Random; import java.util.ArrayList;
public class StudentManagerG3 {
    static int rand(){ Random rand=new Random();  return rand.nextInt(20); }
    public static void main(String[] args) {
        Students A= new Students();
        A.Name="Mohammed";
        A.Fname="Ibrahim";
        for(int i=0;i<3;i++) A.addmark(rand());
```

```
11
12        Students B= new Students();
13        B.setName("Amina");
14        B.setFname("Ali");
15        for(int i=0;i<3;i++) B.addmark(rand());
16
17        A.Display();   A.DisplayMarks();
18        B.Display();   B.DisplayMarks();
19        ArrayList <Students>StudentList= new ArrayList<Students>();
20        StudentList.add(B);  StudentList.get(0).Display();
21    }}
```

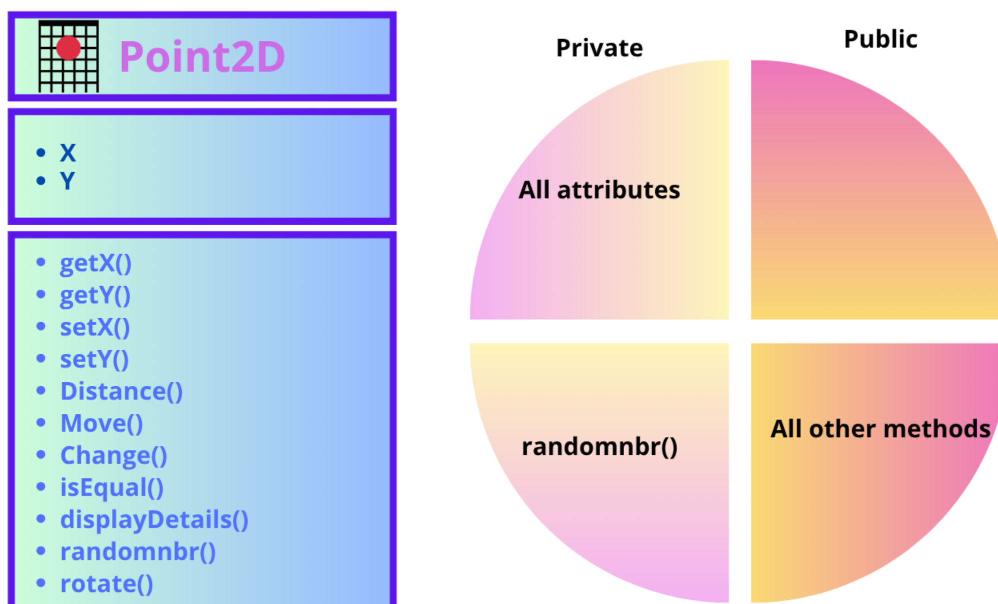


Figure 3.1: Point2D class.

Exercise 3.2 ***Point2D class***

Define a class named Point2D to model a two dimensional point characterized by its coordinates (x, y). The point2D class contains the following constructors and methods (see Figure 3.1):

1. *the getters and the setters for the class attributes.*
2. *a default constructor that allows to initialize the two attributes randomly, and a constructor that allows to initialize a point from two float parameters.*
3. *a method that returns the distance between the current point and the origin of the coordinate*

system.

4. *a method that returns the distance between the current point and another point passed as parameters.*
5. *a method named move to move the location of the point by $(d_x; d_y)$ on the plane.*
6. *a method named change to change the coordinates of the current point.*
7. *a display method, which displays, in a well-organized way, the properties and distance between the point and the origin point.*
8. *a method named rotate to rotate the point by 90 degrees clockwise around the origin. When point is getting rotated 90 clockwise around the origin the following changes happen to its coordinates: new value of $x \leftarrow y$; new value of $y \leftarrow -x$.*
9. *a static function named $isEqual$ takes as arguments two points and returns $true$ if the two points have the same distance from the origin point, otherwise $false$.*

In a main class:

1. *Create a table T of $Point2D$ and initialize randomly all points of T and display them.*
2. *Create a point P and display the distances between P and all elements of T.*
3. *Rotate P and display the distances between P and all elements of T.*

R *All attributes and the random function must be private.*

Solution

```java
package tp_poo;
import java.util.Random;
public class Point2d {
  private  int x;
  private  int y;

    public int getX() { return x; }
    public void setX(int x) {this.x = x;}
    public void setY(int y) {this.y = y;}
    public int getY() {return y;}
```

```
11     private int rand(){ Random r=new Random(); return r.nextInt(10); }
12     public Point2d(){
13         setX(rand());
14         setY(rand());
15     //System.out.print("A new   point is created : x= "+x+" y= "+y+" ");
16     }
17   public Point2d(int a,int b){
18       x=a;    y=b;
19     //System.out.print("A new   point is created : +x+" "+y+" ");
20     }
21  double distance(){ return Math.sqrt(x*x+y*y);}
22  double distance(Point2d p){return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));}
23 
24   void print(){System.out.println(" x="+x+" y= "+y+" ");}
25   void printAll(){System.out.print(" x="+x+" y= "+y+" distance ="+ distance());}
26 
27     void change(int a,int b){x=a;y=b;}
28     void move(int dx,int dy){x=x+dx;y=y+dy;}
29     void rotate(){ int t=x; x=y;y=-x; }
30 
31   static boolean isEquel(Point2d a,Point2d b){return a.distance()==b.distance();}
32   Point2d Merge(Point2d p){ return new Point2d(x+p.x,Math.max(y,p.y));}
33  }
```

Exercise 3.3 *Class Circle*

Using the class of Exercise 3.2 (Point2D) and in the same project (application), create a new class named Circle characterized by its center point (which is a Point2D) and a radius.

R *All attributes must be private.*

The Circle class contains the following constructors and methods:

① *the getters and the setters for the class attributes.*

② *a default constructor that allows to initialize attributes randomly, and a constructor that*

allows to initialize a circle from two parameters (center and radius), a constructor that allows to initialize a circle from one parameter (radius) and the center as the origin point.

3. *Area method that returns the area of the circle ($Area = \Pi * R^2$)*
4. *perimeter method that returns the perimeter of the circle ($perimeter = 2 * \Pi * R$)*
5. *$isInside$ method that returns true if a point is inside the circle, false otherwise.*
6. *isEqual method that compares the current circle with another circle passed as parameter. The isEqual method returns true if the two circles have the same area, false otherwise.*
7. *display method, which displays, in a well-organized way, the properties, the area and the perimeter of the Circle.*
8. *merge method that takes as parameter a circle C and returns as value a new circle which is the result of merging the current circle with C, such that the new circle's center is the point center that has the max distance from the origin and the radius is the max one also.*

In a main class:

1. *Create two circles and display them*
2. *Create a table Tp of $Point2D$ and initialize it randomly, then diplay all points that are inside the previous circles.*

Exercise 3.4 ***Complex numbers***

We want to implement a class that represents complex numbers and supports basic operations. Complex numbers are numbers that can be written in the following form $z = a + bi$, where, a represents the real part, i b represents the imaginary part, a and b are real numbers and i is an imaginary unit called "iota" that represents $\sqrt{-1}$ and $i^2 = -1$. Create a Java class named $Complex$ that models complex numbers. $Complex$ class must include necessary attributes, constructors, getters/setters and provides display method to print a complex number in clear format as well as $iSequal$ method that allows to compare two complex numbers. In addition, $Complex$ class must supports the following operations:

- *Addition: $(a + bi) + (c + di) = (a + c) + (b + d)i$ ($Addition$ method returns the sum of the current complex number and another complex number).*

- *Subtraction:* $(a + bi) - (c + di) = (a - c) + (b - d)i$ *(Subtraction method returns the subtraction of the current complex number and another complex number).*
- *Multiplication:* $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$ *(Multiplication method returns the multiplication of the current complex number and another complex number).*

Exercise 3.5 *Pet class*

Write a Java program to create a class called "Pet" with attributes and methods as illustrated by Figure 3.2:

- *method grow increases the age's pet by* 1.
- *method speak prints "I'm your cuddly little pet".*
- *method humanAge returns the age of the pet and displays "Pets can age differently depending on their species, breed, and size".*
- *method size returns as value the pet size (small, medium, large or Giant), depending on the weight of the pet.*

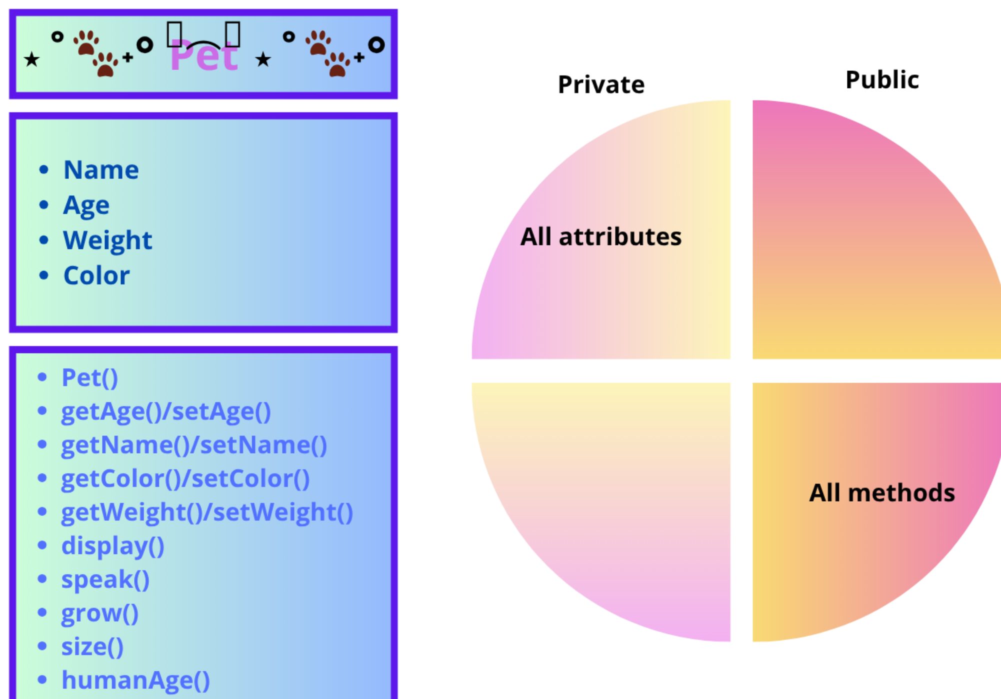


Figure 3.2: Pet class

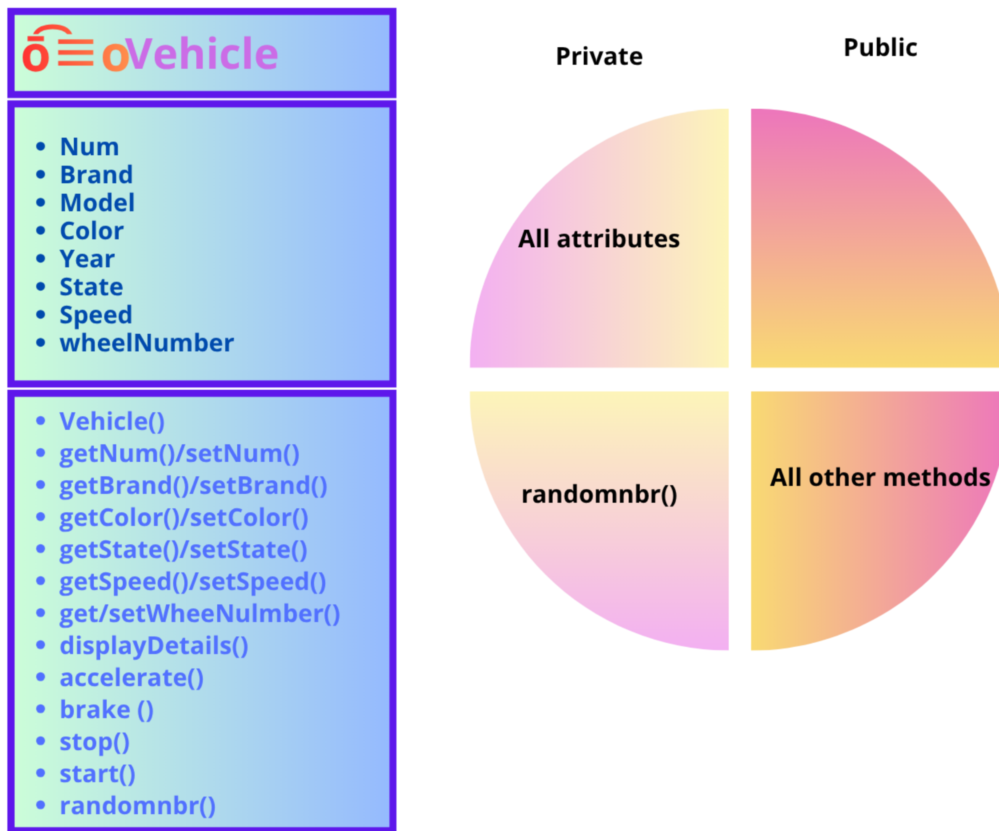


Figure 3.3: Vehicle class

Exercise 3.6 *Class vehicle*

Write a Java program to create a class called "Vehicle" with attributes and methods as illustrated by Figure 3.3:

- *method accelerate increases the vehicle's speed, and takes the increment value as parameter, as it can use a random one.*
- *method brake decreases the vehicle's speed, and takes the decrement value as parameter, as it can use a random one.*

Exercise 3.7 *Driver class*

Write a Java program to create a class called "Driver". The Driver class uses a Vehicle object as part of its data (A Driver has a Vehicle). Class Driver contains:

- *Driver's name*
- *A Vehicle object*
- *A constructor that initializes both the driver's name and the vehicle.*

- *A method displayDriverInfo() that displays the driver's name and the vehicle information.*

In a test program (Main class):

- *Create a Vehicle object.*
- *Create a Driver object that uses that vehicle*
- *Display all the information about the driver and their vehicle*
- *Create multiple Driver and Vehicle objects and store them in an array or list.*

Exercise 3.8 ***Polynome2D class***

Create a class $Polynome2D$ that represents a second-degree polynomial: $ax^2 + bx + c$ where $a, b, c \in R, a \neq 0$, by breaking down the code and methods into auxiliary sub-methods in all possible cases. The class must contain private fields, necessary constructors and necessary getters and setters. In addition, $Polynome2D$ class provides the following methods:

- *Display: prints the polynomial in clear format.*
- *solve() which computes the two roots of the polynomial in R.*
- *solveComplex() which computes the two roots of the polynomial in C (refer to Exercise 3.4).*

Reminder:

- $\Delta = b^2 - 4 * a * c$,
- $x_1 = \frac{-b-\sqrt{\Delta}}{2*a}$,
- $x_2 = \frac{-b+\sqrt{\Delta}}{2*a}$,
- $z_1 = \frac{-b-i\sqrt{|\Delta|}}{2*a}$,
- $z_2 = \frac{-b+i\sqrt{|\Delta|}}{2*a}$.

Exercise 3.9 ***Temperature class***

Suppose you want to maintain the temperatures for each day of a month. To do this:

1. *Write a **Temperature** class that allows to store a temperature in Fahrenheit with the necessary getters/setters and constructors that allow to initialize the temperature to 20 or accept a temperature in Fahrenheit (in double) as a parameter.*
2. *The Temperature class allows basic conversions, to Celsius (getCelsius) and to Kelvin*

(getKelvin), such that:

$$Celsius = (5/9) * (Fahrenheit - 32)$$

$$Kelvin = ((5/9) * (Fahrenheit - 32)) + 273$$

③ *The **Temperature** class provides a method to display the temperature in Fahrenheit.*

④ *The **Temperature** class provides a method to display the temperature based on a parameter: 'C' in Celsius, 'F' in Fahrenheit, and 'k' in Kelvin.*

⑤ *Write a class **TemperatureMonth** that stores the temperatures of a month as an array of **Temperature** values. The **TemperatureMonth** class has constructors that initialize the month and its temperatures. By default, the temperatures for the month are set to 20, however, they may be initialized to the value specified by the parameter.The month and its number of days can be passed as parameters; by default is 30, and the month name is June.*

⑥ *The temperatures are accessed using two methods:* `setTemperature(double tmp, int day)` *and* `getTemperature(int day)`*. If the day value passed as a parameter is negative or greater than the number of days in the month, an error message should be displayed.*

⑦ *The **TempertureMois** class has a display method that displays the temperatures of the month day by day in Celsius.*

⑧ *Write a test class, named **TestTemperature**, whose purpose is to validate the correct functioning of the **Temperature** and **TemperatureMois** classes:*

① *Create an instances of the **Temperature** class, named $T1$, using the default constructor (initialized with the default temperature value). Then, display it in kalvin, in Celsius, and in Fahrenheit.*

② *Create an instances of the **Temperature** class, named $T2$, using a parameterized constructor (initialized with the temperature value provided as an argument). Then, display it in kalvin, in Celsius, and in Fahrenheit.*

③ *Create an instances of the **TemperatureMois** class m using the default constructor (initialize all monthly temperature values to the default value). Then, modify values of TM using double random values. Display these values in Celsius.*

Solution

```java
public class Temperature {
  double t;
  public Temperature() { t=20; }
  public Temperature(double t) {this.t = t; }
  public double getT() {return t; }
  public void setT(double t) { this.t = t; }
  public double getTcelus() { return (0.5555*(t-32)); }
  public double getTKelven() {return ((0.5555*(t-32))+273); }
  public void Print() { System.out.println("Fernheit= "+t); }
  public void Print(char c) {
    switch (c) {
      case 'c':
      System.out.println("Celius= "+getTcelus()); break;
      case 'k':
      System.out.println("Kelven= "+getTKelven()); break;
      case 'f':
      System.out.println("Fernheit= "+getT()); break;
      default:System.out.println("Please inter c, k or f  ");
} } }
```

Listing 3.9: Temperature class

```java
public class Tmois {
  Temperature []m;
  String mois;

  public Tmois() {
    m= new Temperature[30]; mois="June";
    for(int i=0;i<m.length;i++)m[i]=new Temperature();
  }

  public Tmois(String mm,int n,double t) {
    m= new Temperature[n]; mois=mm;
    for(int i=0;i<m.length;i++)m[i]=new Temperature(t);
  }

```

```
15      public void setTemperteur(int i, double t) {
16        if((i>=m.length)|| i<0) System.out.println("error");
17        else this.m[i].setT(t);
18      }
19      public  double getTemperteur(int i) {return m[i].getT();}
20
21      public void print(){
22        for(int i=0;i<m.length;i++){
23          System.out.print(i+" ");
24          m[i].Print('c');}
25      }
26  }
```

Listing 3.10: Tmois class

```
1  public class TestTemperature {
2  public static void main(String[] args) {
3    Temperature T1=new Temperature();
4      T1.Print('c');
5      T1.Print('k');
6      T1.Print('f');
7    Temperature T2=new Temperature(45);
8      T2.Print('c');
9      T2.Print('k');
10     T2.Print('f');
11
12   Random nb=new Random();
13   Tmois m=new Tmois();
14   for(int i=0;i<m.m.length;i++)
15     m.setTemperteur(i, nb.nextDouble(100));
16   m.print();
17 }
18 }
```

Listing 3.11: TestTemperature class

CHAPTER 4

INHERITANCE

Contents

4.1 What is Inheritance? 60

4.2 Overriding Methods 61

4.3 Accessing an Overridden Method 62

4.4 Constructors 63

4.5 Quick Check 65

4.6 Activities 69

Key Objectives

The key objectives of this chapter are to introduce the concept of inheritance in OOP and to demonstrate how it is implemented in Java. By the end of the chapter, student should understand how a class can inherit attributes and methods from a parent class, how to create subclasses, and how to use the super keyword to access and extend the behavior of parent classes. The chapter also explains method overriding, and the benefits of code reuse, providing a foundation for building more complex and maintainable object-oriented programs. To reinforce these concepts, the chapter finishes with practical activities that allow student to apply what he has learned.

4.1 What is Inheritance?

Recapitulation 4.1

Inheritance is one of the main concepts of object-oriented programming paradigms. It allows to create a new class by reusing code from an existing class. The new class is called a subclass, and the existing class is called the superclass:

- *A superclass contains the code that is reused and customized by the subclass. It is said that the subclass inherits from the superclass. A superclass is also known as a base class or a parent class.*
- *A subclass is also known as a derived class or a child class. It may use, directly or indirectly, everything from its superclass. Inheritance guarantees that whatever behavior (method) is present in a class will also be present in its subclass.*
- *The relationship that must exist between the superclass and the subclass in order for inheritance to be effective is called an "is-a" relationship.*
- *To inherit a class from another class we use the keyword* `extends` *followed by the superclass name in the class declaration of a subclass. The general syntax is as follows:*

```
class subclassName extends superclassName {
     // Code for the subclass goes here
}
```

Recapitulation 4.2

- *Let's consider a class Pet (Listing 4.1) that models a pet. There are many different types of pets, so we really can't expect one class to be capable of modeling them all. We all know how different dogs, cats, and reptiles are, for example.*

```
class Pet {
   private String name;
   public String getName( ) {  return name; }
   public void setName(String petName) { name = petName; }
   public String speak( ) { return "I'm your cuddly little pet."; }
 }
```

Listing 4.1: Pet class

So, let's define the individual Cat and Dog classes to model them a little more precisely than the generic Pet class. Now, instead of defining the two new classes independently, we will define them based on the Pet class. Although they are different, they share common traits of being a pet, so it makes sense to derive the two classes from the Pet class. Listing 4.2 shows how we might define the Cat class by using inheritance:

```
class Cat extends Pet { // Cat is a subclass of Pet (Cat is a Pet)
  public String speak() { return  "Don't give me orders.\n" +
                              "I speak only when I want to."; }
 }
```

Listing 4.2: Cat class derived from Pet class

Data members and methods of a superclass are inherited by its subclasses. So, for example, the following code is valid:

```
Cat myCat = new Cat( );
myCat.setName("Cha Cha");
System.out.println("Hi, my name is " + myCat.getName( ));
```

R We can disable inheritance of a class by declaring the class final. The `final` keyword, when applied to a class, serves to prevent that class from being subclassed or inherited. This means that no other class can extend a final class.

4.2 Overriding Methods

Recapitulation 4.3

Redefining an instance method in a class, which is inherited from the superclass, is called method overriding. Consider the following declarations of class A and class B:

```
public class A {
    public void display() { System.out.println("A"); }
  }
public class B extends A {
    @Override
    public void display() { System.out.println("B"); }
  }
```

Class B inherits the display() method from its superclass and redefines it. It is said that the display() method in class B overrides the display() method of class A.

Recapitulation 4.4

There are rules when a class overrides a method, which it inherits from its superclass :

- *The method must be an instance method. Overriding does not apply to static methods.*
- *The overriding method must have the same name as the overridden method.*
- *The overriding method must have the same number of parameters of the same type in the same order as the overridden method.*
- *The access level of the overriding method must be at least the same or more relaxed than that of the overridden method (see Table 4.1). The three access levels are public, protected, and package level that allow for inheritance.*
- *The return type must be the same in the overriding and the overridden methods. However, if the return type is a reference type, overriding a method's return type could also be a subtype (any descendant) of the return type of the overridden method.*

Overridden Method Access Level	Allowed Overriding Method Access Level
public	Public
protected	public, protected
Package level	public, protected, package level

Table 4.1: Allowed Access Levels for an Overriding Method

4.3 Accessing an Overridden Method

Recapitulation 4.5

Sometimes, a subclass has overridden a method from its superclass needs to execute the superclass's implementation of that method. In this case, the `super` *keyword is used as follows:* `super.methodName(arguments)`*. Let's create the following class Dog (Listing 4.3) inherited from existing class Pet of the Listing 4.1. Dog class overrides the speak() method in its own way, and call super.speak() to include the parent behavior. This allows Dog to add new behavior while still keeping the original one.*

```
class Dog extends Pet {
      @Override
      void speak() {
        System.out.println("The dog says: Woof!");
        super.speak(); // call the parent (Pet) version
      }
    }
```

Listing 4.3: Dog class inherited from Pet class

4.4 Constructors

Recapitulation 4.6

- *In Java, constructors are not inherited by subclasses. Each class is responsible for defining its own constructors. However, a subclass can invoke a constructor of its superclass using the* `super()` *keyword.*
- *If a subclass constructor does not explicitly call super(), Java automatically inserts a call to the superclass's no-argument constructor (* `super()` *;) as the first statement.*
- *If the superclass does not have a no-argument constructor, and the subclass does not explicitly call another superclass constructor, a compile-time error will occur.*
- *Listing 4.4 illustrates the implicit constructor calling without usage of* `super()` *keyword. In this case, super class constructor will be called first, and then derived(subclass) constructor will be called.*
- *Listing 4.5 illustrates the explicit constructor calling with usage of* `super()` *keyword.*

```
class A {// Super class
  A() // Constructor of super class
  { System.out.println("Constructor of superclass A Called "); }
}
class B extends A {// Subclass
  B()// Constructor of subclass
  { System.out.println("Constructor of subclass B Called "); }
}
```

```
 9   class Test {// Main class
10     public static void main(String[] args){
11       // Creating an object of subclass inside main() method
12       B d = new B();
13       // Output:
14       //Constructor of superclass A Called
15       //Constructor of subclass B Called
16 }}
```

Listing 4.4: Constructor calling without usage of super keyword.

```
 1 class A { // Super class
 2     int x;
 3     A(int _x) { x = _x; } // Constructor of super class
 4 }
 5 class B extends A { // Sub class
 6     int y;
 7     B(int x, int y)// Constructor of subclass
 8     { super(x); y = y; // super keyword refers to superclass constructor}
 9     void Display() { System.out.println("x = " + x + ", y = " + y); }
10 }
11 public class Test { // Main class
12   public static void main(String[] args){
13       // Creating object of subclass B inside main() method
14       B d = new B(10, 20);
15       d.Display();// Invoking method inside main() method
16     }
17 }
18 //Output : x = 10, y = 20
```

Listing 4.5: Constructor calling with usage of super keyword.

- Always provide a constructor for every class you define. Don't rely on default constructors.
- If a class B has a superclass A, then include an explicit call to a constructor of the superclass A in the constructor of the class B.

4.5 Quick Check

Quick Check 4.1 *Which is the subclass and which is the superclass in this declaration?*

```
class X extends Y {
   ...
    }
```

Quick Check 4.2 *Which visibility modifier allows the data members of a superclass to be accessible to the instances of subclasses?*

Quick Check 4.3

1. *If X is a private member of the Super class, is X accessible from a subclass of Super?*
2. *If X is a protected member of the Super class, is X of one instance accessible from another instance of Super? What about from the instances of a subclass of Super?*
3. *How do you call the superclass's constructor from its subclass?*
4. *What statement will be added to a constructor of a subclass if it is not included in the constructor explicitly by the programmer?*

Quick Check 4.4 *Consider the following class definitions. Identify invalid statements.*

```
class Car {
  public String make;
  protected int weight;
  private String color;
  ...
}

class ElectricCar extends Car {
  private int rechargeHour;
  public ElectricCar() { ... }

  public ElectricCar (ElectricCar car) {
    this.make = car.make;
```

```
        this.weight = car.weight;
        this.color = new String(car.color);
        this.rechargeHour= car.rechargeHour;
    }
    ...
}
class TestMain {
    public static void main (String[] args) {
        Car myCar;
        ElectricCar myElecCar;
        myCar = new Car();
        myCar.make = "Chevy";
        myCar.weight = 1000;
        myCar.color = "Red";
        myElecCar = new ElectricCar();
        myCar.make = "Chevy";
        myCar.weight = 500;
        myCar.color = "Silver";
    }
}
```

Quick Check 4.5 *Which of the following cannot be overridden?*

1. *public methods*
2. *protected methods*
3. *private methods*
4. *final methods*

Quick Check 4.6 *True / False Questions*

- *Constructors can be overridden in Java?*
- *Final methods cannot be overridden.*
- *Method signature must be the same for method overriding.*
- *Access level of an overridden method can be more restrictive*

Quick Check 4.7 *Consider the following class definitions. Identify which calls to the constructor are invalid.*

```
class Car {
  public String make;
  protected int weight;
  private String color;
  private Car (String make, int weight, String color) {
    this.make = make;
    this.weight = weight;
    this.color = color;
  }
  public Car () {
    this("unknown", -1, "white");
  }
  class ElectricCar extends Car {
    private int rechargeHour;
    public ElectricCar() {
      this(10);
    }
    private ElectricCar(int charge) {
      super();
      rechargeHour = charge;
    }
  }
class TestMain {
    public static void main (String[] args) {
      Car myCar1, myCar2;
      ElectricCar myElec1, myElec2;
      myCar1 = new Car();
      myCar2 = new Car("Ford", 1200, "Green");
      myElec1 = new ElectricCar();
      myElec2 = new ElectricCar(15);
      }
}
```

Quick Check 4.8 *Consider the following code. identify the mistake and correct it.*

```
class A {
  final void show() { System.out.println("A"); }
}

class B extends A {
  void show() { System.out.println("A"); }
}

public class Test {
  public static void main(String[] args) {
    A obj = new B();
    obj.show();
  }
}
```

Quick Check 4.9 *What will be the output of the following code?*

```
class Base {
  Base() { System.out.println("Base Constructor");}
}
class Derived extends Base {
  Derived() {
    System.out.println("Derived Constructor");}
}
public class Main {
  public static void main(String[] args) {
    Derived obj = new Derived();
  }
}
```

1. *Runtime Error*
2. *Compilation Error*
3. *Derived Constructor*
   Base Constructor

④ *Base Constructor*

Derived Constructor

Quick Check 4.10 *Constructor Call Order*

```
class A {
  A() { System.out.println("A constructor");}
}

class B extends A {
  B() { System.out.println("B constructor"); }
}

class C extends B {
  C() { System.out.println("C constructor"); }
}

public class Test {
  public static void main(String[] args) {
    C obj = new C();
  }
}
```

4.6 Activities

Exercise 4.1 *Using the class of Exercise 3.2 ($Point2D$) and in the same project (application), create a new class named $Point3D$ as illustrated by Figure 4.1 and create different necessary constructors, getters, and setters. Then, redefine (Override) all methods that needs to be overrided, knowing that:*

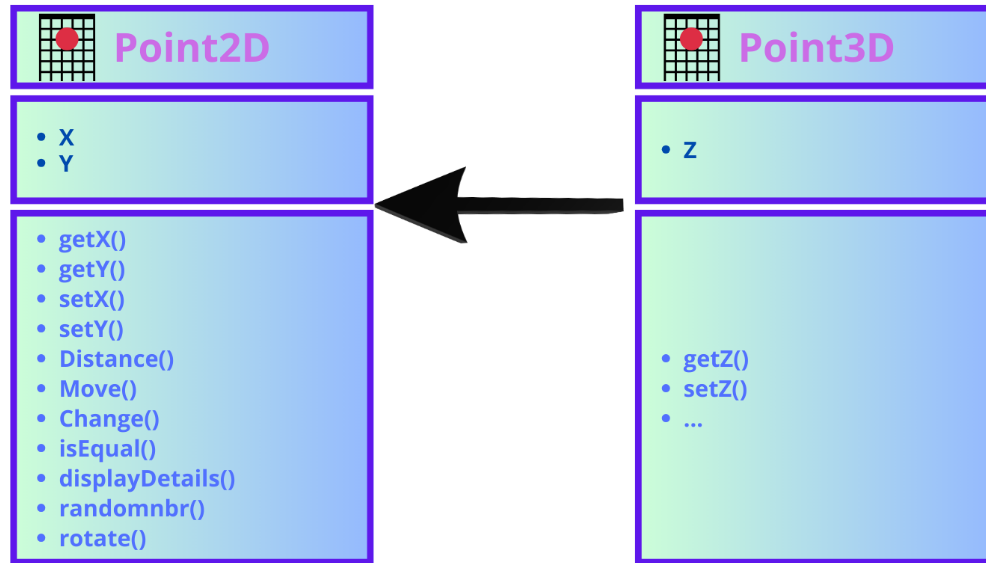


Figure 4.1: Class Point3D inherits from class Point2D

1. *the distance between two three -dimensional points is:* $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$
2. *When a point is getting rotated 90 clockwise around the origin the following changes happen to its coordinates: new value of* $x \leftarrow y$*; new value of* $y \leftarrow -x$ *and* z *remains the same.*

In a main class:

1. *create a matrix* m *of point3D and display them*
2. *compute and display the distance between the first point of* m *and the other points.*

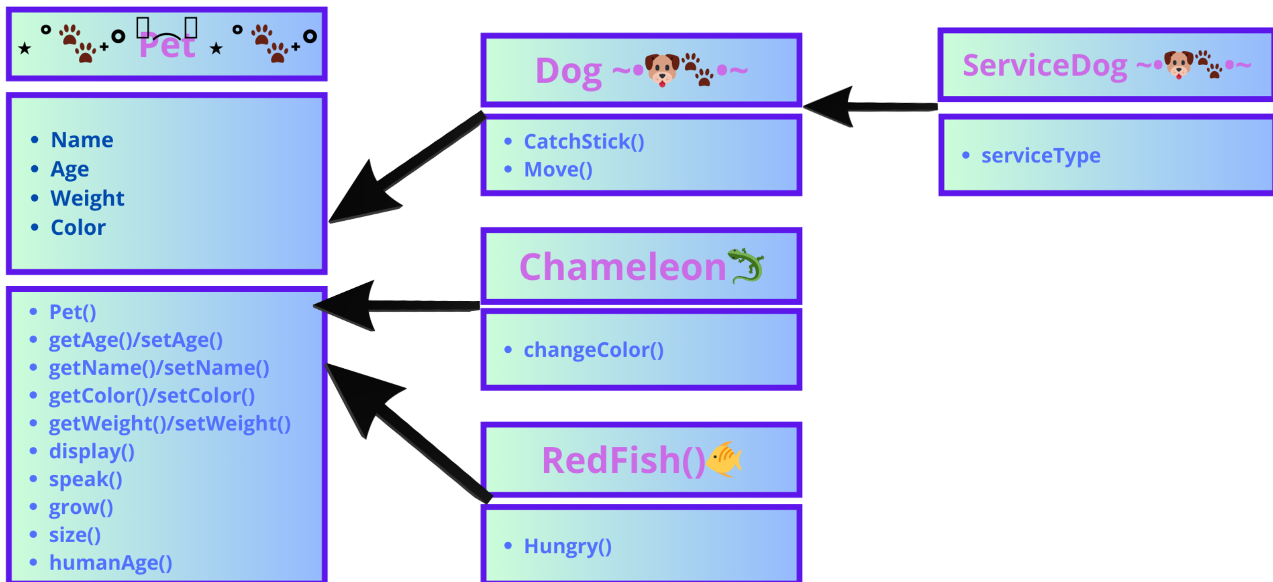


Figure 4.2: Pets classes

Exercise 4.2 *Pet classes*

Let's consider class of Exercise 3.5. Create classes of Figure 4.2 with the necessary methods and constructors. Knowing that, Dog, Chameleon, and RedFish classes inherit from Pet class and ServiceDog class inherits from the class Dog.

1. *Hungry method generates and returns a random boolean*
2. *The ChangeColor() method generates an integer $i \in \{0,1,2,3\}$ and change the color of the pet and returns the name of the color corresponding to the generated number, such as: 0 Red, 1 Green, 2 Yellow, 3 Blue.*
3. *Override the speak method in the service Dog class so that it prints: "Always ready for my task."*
4. *Redefine (Override) the speak method in the dog class so that it prints: "I'm your cuddly little pet" "I speak whenever you want to"*
5. *Redefine the speak method in the Chameleon class so that it prints: "Silent all time"*
6. *Override the Display method in the Redfish class so that it also displays if it is hungry or not.*
7. *Override the Display method in the service dog class so that it also displays the other characteristics.*

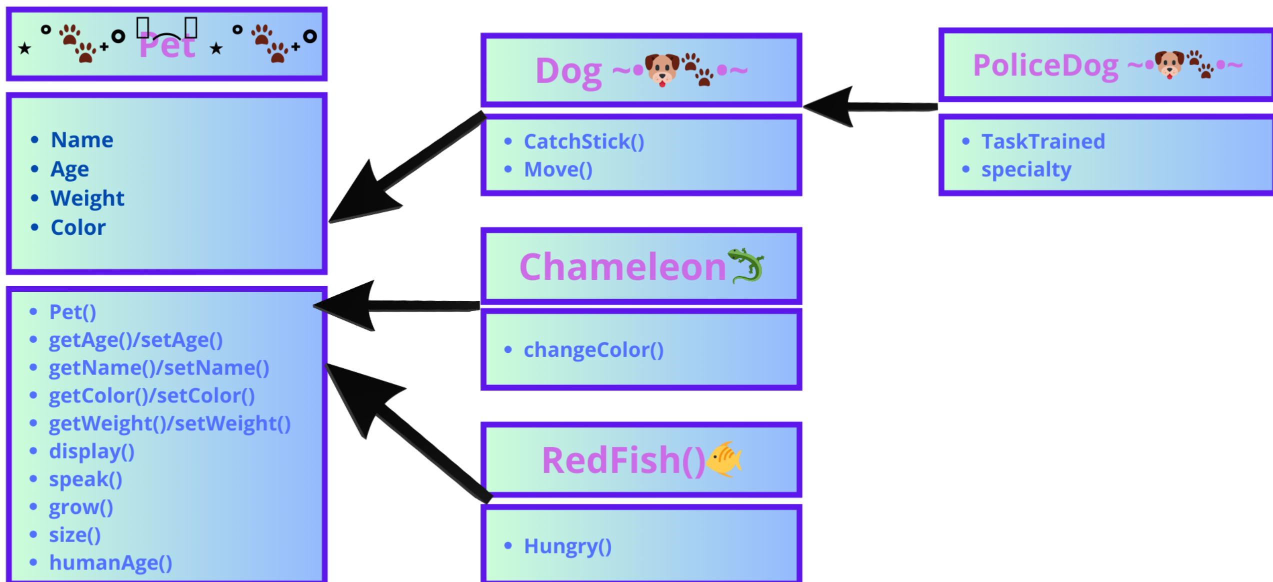


Figure 4.3: Pet Class and its subclasses

Exercise 4.3 *Pet class*

Create classes of Figure 4.3 with the necessary methods and constructors.

1. *Sleep method generates and returns a random boolean*
2. *The ChangeColor() method generates an integer $i \in \{0,1,2,3\}$ and change the pet color of the pets and returns the name of the color corresponding to the generated number, such as: 0 Red, 1 Green, 2 Yellow, 3 Blue.*
3. *Override the speak method in the Police Dog class so that it prints: "Always ready for my task."*
4. *Redefine (Override) the speak method in the dog class so that it prints: "I'm your cuddly little pet" "I speak whenever you want to"*
5. *Redefine the speak method in the Chameleon class so that it prints: "Silent all time"*
6. *Override the Display method in the Chameleon class so that it also displays if it is asleep or not.*
7. *Override the Display method in the Police dog class so that it also displays the other characteristics.*
8. *Set the color of the RedFish to be always Red (whatever, the color must be Red).*

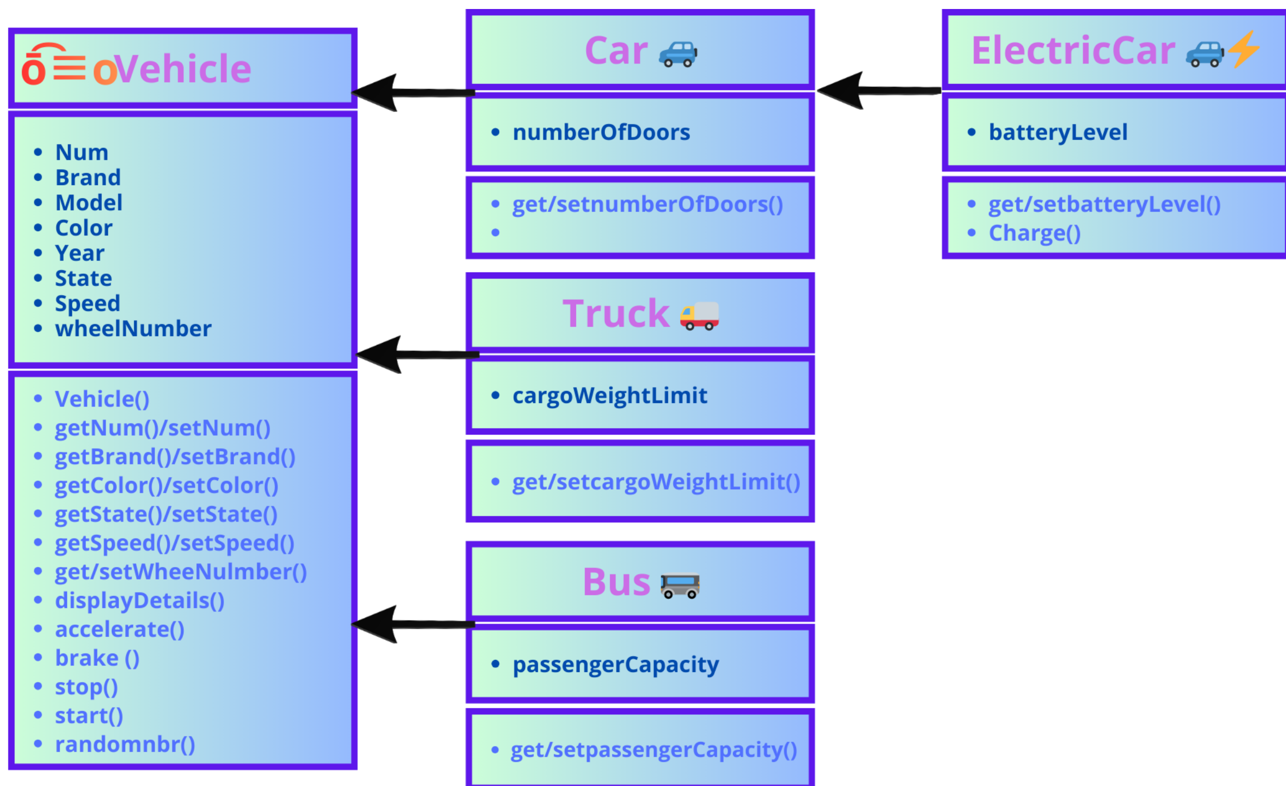


Figure 4.4: Subclasses of vhicle Class

Exercise 4.4 *Create a Java application that demonstrates inheritance using a super class called Vehicle (Exercise 3.6) and several subclasses, as illustrated by Figure 4.4:*

1. *Create a class named Car that inherits from Vehicle. The Car class should add an attribute for the number of doors and include a method such as honk().*
2. *Create a class named Truck that inherits from Vehicle. The Truck class should add an attribute for cargo weight limit and include a method such as haulCargo().*
3. *Create a class named Bus that inherits from Vehicle. The Bus class should add an attribute for passenger capacity and include a method such as pickUpPassengers().*
4. *Create a class named ElectricCar that inherits from Car. The ElectricCar class should add an attribute for battery level and include a method such as chargeBattery().*
5. *Override display method in all subclasses so they display their own attributes too.*
6. *Create a Main class to create objects of Car, Truck, Bus, and ElectricCar, then call both inherited and specific methods to demonstrate inheritance.*

Exercise 4.5 *Rectangle and square*

① *Create a class named Rectangle that represents a rectangle. The Rectangle class should have as attributes:*

- *length (double)*
- *width (double)*

The Rectangle class should include the following methods:

- *area() : returns the area of the rectangle (length * width)*
- *perimeter() : returns the perimeter (2 * (length + width))*
- *display() : prints the length, width, area, and perimeter*

② *Create a class named Square that inherits from Rectangle. The Square class should :*

- *have a constructor that sets both length and width to the same value;*
- *override display() to show it is a square.*

CHAPTER 5

POLYMORPHISM

Contents

5.1 What is Polymorphism 76
5.2 Types of Polymorphism 76
5.3 Benefit of Polymorphism 77
5.4 Polymorphism by Inheritance 77
5.5 Keyword *instanceof* 78
5.6 Quick Check 80
5.7 Activities 84

Key Objectives

Learning objectives of this chapter are to introduce the concept of polymorphism in OOP and to demonstrate how it is implemented in Java. By the end of the chapter, student should understand how objects of different classes can be treated through a common parent class, how method overriding enables dynamic behavior, and how polymorphism promotes flexibility and code reuse in program design. To reinforce these concepts, the chapter finishes with practical activities that allow student to apply what he has learned.

5.1 What is Polymorphism

Recapitulation 5.1

Polymorphism is one of the core concepts in OOP that allows objects to behave differently based on their specific class type. The word polymorphism means having many forms, and it comes from the Greek words poly (many) and morph (forms), this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently based on the context.

5.2 Types of Polymorphism

Recapitulation 5.2

Java primarily supports two types of polymorphism:

1. ***Compile-Time Polymorphism*** *: This occurs when the compiler determines which method to call during the compilation phase. It is achieved through:*
    - ***Method Overloading****: Defining multiple methods in the same class with the same name but different parameters (number, type, or order).*
    - ***Operator Overloading****: For example, Java allows the + operator to perform both numeric addition and string concatenation.*
2. ***Run-Time Polymorphism*** *: This occurs when the Java Virtual Machine determines which method to call at runtime based on the actual object's type, not the reference type. It is achieved through:*
    - ***Method Overriding****: A subclass provides a specific implementation of a method that is already defined in its superclass, using the same name, return type, and parameters (method signature). This relies on inheritance and a superclass reference variable pointing to a subclass object (upcasting).*

5.3 Benefit of Polymorphism

Recapitulation 5.3

Polymorphism has many benefits which are listed below:

- ***Code Reusability**: Polymorphism allows the same method or class to be used with different types of objects, which makes the code more reusable.*
- ***Flexibility**: Polymorphism enables object of different classes to be treated as objects of a common superclass, which provides flexibility in method execution and object interaction.*
- ***Abstraction**: It allows the use of abstract classes or interfaces, enabling working with general types (like a superclass or interface) instead of concrete types (like specific subclasses), thus simplifying the interaction with objects.*
- ***Dynamic Behavior**: With polymorphism, the appropriate method to call can be selected at runtime, giving the program dynamic behavior based on the actual object type rather than the reference type, which enhances flexibility.*

5.4 Polymorphism by Inheritance

Recapitulation 5.4

- *Polymorphism by subtyping (or inheritance) is a key concept in OOP allowing derived (child) classes to share a common interface with their base (parent) class while specializing their own behavior, notably through method overriding.*
- *When a variable is declared to be of class S, the variable can be a reference to an instance of S or any of its subclasses. The inverse is not valid. Let's consider the class Pet and its subclasses Cat and Dog, the following code is valid:*

```
1    Pet petA = new Dog( );
2    Pet petB = new Cat( );
```

However, the following code is not valide:

```
1    Dog DogA = new Pet( );
```

- *The fact that the same variable can be referring to an instance of a different class results in polymorphism.*

- *The following two output statements will produce different results, depending on whether p is a Dog or a Cat:*

```
Pet p;
p = new Dog( );
System.out.println(p.speak( ));
p = new Cat( );
System.out.println(p.speak( ));
```

The $speak$ method is called a polymorphic method.

- *If a variable is declared of type S and is referring to an instance of a subclass of S, then we must typecast the variable to the subclass when calling noninherited methods of the subclass. For example, the fetch method is defined in the Dog class only. So the followind code is invalid:*

```
Pet p;
p = new Dog( );
System.out.println(p.fetch( ));
```

To make this code valid, we need to typecast p to Dog, as follow:

```
Pet p;
p = new Dog( );
System.out.println( ((Dog)p).fetch( ) );
```

5.5 Keyword *instanceof*

Recapitulation 5.5

The `instanceof` keyword in Java is a binary operator used to check whether an object is an instance of a specific class or implements a particular interface. It returns `true` if the object is an instance of the specified type and `false` otherwise. Its syntax is as follows:

```
objectName instanceof className
```

`a instanceof B` *returns* `true` *if:*

- *the variable a stores a reference to an object of type B;*
- *the variable a stores a reference to an object whose class inherits from B;*

- *The variable a stores a reference to an object that implements interface B.*

Otherwise, it returns `false`*. If* `<reference-variable>` *is* `null`*,* `instanceof` *always returns* `false`*.*

Let's consider the class Vehicle and its subclass car and truck. The following Listing 5.1 shows an example of using the `instanceof` *operator. The code begins by creating instances of different classes (Vehicle, Truck, and Car) and placing them in the array T. The code then displays whether each element of T is an instance of the Vehicle, Truck, or Car class. The code concludes by counting the number of trucks in T.*

```
Vehicle v = new Vehicle ();
Vehicle v1 = new Truck ();
Vehicle v2 = new Car ();
Car car1 = new Car ();
Truck truck1 = new Truck ();

Vehicle [] T = {v,v1 ,v2 ,car1 , truck1 };

for (int i=0;i<T. length ;i++)
 System.out.print (T[i] instanceof Vehicle +" ");
 // true true true true true

for (int i=0;i<T. length ;i++)
 System.out.print (T[i] instanceof Car +" ");
 // false false true true false

for (int i=0;i<T. length ;i++)
 System.out.print (T[i] instanceof Truck +" ");
 // false true false false true

int k =0;
for (int i=0;i<T. length ;i++)
 if(T[i] instanceof Truck ) k++;
System .out. println (" Number of trucks = "+k);// 2
```

Listing 5.1: Example of using instanceof operator

5.6 Quick Check

Quick Check 5.1 *Polymorphism in Java means:*

① *One class having many constructors*

② *One method having many implementations*

③ *One variable having many values*

④ *One object having many references*

Quick Check 5.2 *Runtime polymorphism is achieved using:*

① *Method overloading*

② *Method overriding*

③ *Constructors*

④ *Static methods*

Quick Check 5.3 *What will be the output of the following code?*

① *A*

② *B*

③ *Compile-time error*

④ *Runtime error*

```
class A {
  void show() { System.out.println("A"); }
}
class B extends A {
  void show() { System.out.println("B"); }
}
public class Test {
  public static void main(String[] args) {
    A obj = new B();
    obj.show();
  }}
```

Quick Check 5.4 *Which one of the following statements is valid?*

```
Pet p = new Cat();
Cat c = new Pet();
```

Quick Check 5.5 *Is the following code valid?*

```
Pet p = new Dog();
System.out.println(p.fetch( ));
```

Quick Check 5.6 *Suppose $Truck$ and $Motorcycle$ are subclasses of $Vehicle$. Which of these declarations are invalid?*

```
Truck t = new Vehicle();
Vehicle v = new Truck();
Motorcycle m1 = new Vehicle();
Motorcycle m2 = new Truck();
```

Quick Check 5.7 *What is the purpose of the `instanceof` operator? Can we use it in polymorphism to check object type?*

Quick Check 5.8 *What will be the output of the following code?*

```
class A {
  static void show() { System.out.println("A"); }
}

class B extends A {
  static void show() { System.out.println("B"); }
}

public class Test {
  public static void main(String[] args) {
    A obj = new B();
```

```
        obj.show();
      }
  }
```

Quick Check 5.9 *What will be the output of the following code?*

```
class Vehicle {
  void start() { System.out.println("Vehicle starts");}
}

class Car extends Vehicle {
  void start() { System.out.println("Car starts"); }
}

class Bike extends Vehicle {
  void start() { System.out.println("Bike starts"); }
}

public class Test {
  public static void main(String[] args) {
    Vehicle[] v = { new Car(), new Bike() };
    for (i=0;i<v.length; i++) { v[i].start(); }
  }
}

```

Quick Check 5.10 *What will be the output of the following code?*

```
    class A {
      int x = 10;
    }

    class B extends A {
      int x = 20;
    }
```

```
8
9  public class Test {
10   public static void main(String[] args) {
11     A obj = new B();
12     System.out.println(obj.x);
13   }
14 }
```

Quick Check 5.11 *What will be the output of the following code?*

```
1  class A {
2    void display() { System.out.println("A"); }
3  }
4
5  class B extends A {
6    void display() {
7      super.display();
8      System.out.println("B");
9    }
10 }
11
12 public class Test {
13   public static void main(String[] args) {
14     A obj = new B();
15     obj.display();
16   }
17 }
```

Quick Check 5.12 *Overloading vs Overriding*

What will be the output of the following code? Why?

```
1 class A {
2   void show(int x) { System.out.println("A int"); }
3 }
4
```

```
5 class B extends A {
6   void show(double x) { System.out.println("B double"); }
7 }
8
9 public class Test {
10   public static void main(String[] args) {
11     A obj = new B();
12     obj.show(10);
13   }
14 }
```

5.7 Activities

Exercise 5.1 *Resuming Exercise 4.2. Check whether the statements in the following code are valid and justify your answers:*

```
1
2     Pet pet1 = new Dog(); Pet pet2 = new RedFish();   Pet pet3 = new Chamellon();
3     Dog myDog = new Pet();
4
5     Pet p;
6     p = new Dog(); p.speak();
7     p = new ServiceDog(); p.speak();
8
9     Pet p4;  p4 = new RedFish( );   System.out.println(p4.Hungry());
10
11    Pet p5; p5 = new Dog( ); System.out.println( ((RedFish)p5).Hungry( ) );
```

Exercise 5.2 *Resuming Exercise 4.2.*

1. *Add a class named yellowSnake inherited from the class pet and set the color of the YellowSnake to be always Yellow (whatever, the color must be Yellow).*
2. *Create a Box class that is characterized by a color and contains a Pet. The Box class has two methods: addPets to place a Pet in the Box and display to display the color of the box and*

the Pet in the box, as well as the type of Pet (dog, chameleon, ...).

③ *In a main program, create an array-list (or a table) of Pets. Each element of the list must be Dog, RedFish, Chameleon, or yellowSnake. To fill the list (array), generate a random number from 1 to 4. If the generated number = 1, add a Dog, 2 add a RedFish, 3 add a Chameleon, 4 add a yellowSnake.*

(a). *Display each element of the array-list*

(b). *Display the number of yellowSnake, dogs, RedFish and chameleons in the list.*

(c). *Create an array of boxes, fill it and display its elements.*

Exercise 5.3 *Write a program that creates an ArrayList of pets. An item in the list is either a Dog or a yellowSnake. For each pet, enter its name and type ('c' for cat and 'd' for dog). Stop the input when the string STOP is entered for the name. After the input is complete, output the name and the type of each pet in the list.*

Exercise 5.4 *Repeat Exercise 5.3, but this time group the output by printing out the names of all dogs first and then the names of all chameleons.*

Exercise 5.5 ***Mini-project: Multimedia Library***

Imagine a small-town library named "Maplewood Stacks". For decades, they used paper cards tucked into the back of books to track their inventory. However, as they've expanded to offer local indie films on DVD and jazz albums on CD, the paper system is failing. They need a Java-based digital system. The librarian wants the system to treat "everything as an item" while still remembering that a book has an author, but a movie has a director. Here is the technical breakdown of how to build this using OOP with Java:

① *The Class **LibraryItem**: Every object in the library shares basic characteristics:*

- ***Fields**: private String itemId (e.g., "LIB-101"), private String title, protected boolean isAvailable (Defaults to true)*
- ***Methods** to implement:*
  - *checkOut(): Should check if isAvailable is true. If yes, set it to false. If no, print an error or " Already Loaned".*
  - *returnItem(): Sets isAvailable back to true.*

- *displayStatus(): Prints the ID, Title, and whether it is "In Library" or "Loaned Out."*

② *Class **Book**:*

- ***Fields**: String author, int pageCount, edition, year*
- ***Methods** to implement: overrides displayStatus() to also mention the author and the other information.*

③ *Class **MediaItem**: CDs and DVDs can be grouped together. You might create a MediaItem class that adds a duration field and create:*

- *Subclass **DVD**: Inherits from MediaItem; adds String resolution (4K, HD). This class overrides displayStatus() to also mention the author and the other information.*
- *Subclass **CD**: Inherits from MediaItem; adds String artist. This class overrides displayStatus() to also mention the author and the other information.*

④ ***EBook** extends Book: Inherits author from Book and itemId from LibraryItem. Note: You might decide EBooks are always available since they are digital.*

⑤ *The true power of this system is that the librarian can handle a list of items without knowing exactly what they are at first:*

- *Create an array or list of LibraryItem.*
- *Inside that array, mix a Book, a CD, and a DVD.*
- *The Polymorphic Loop: Iterate through the array and call displayStatus() on each.*

⑥ *In your Main class, create a CompactDisc object. Call checkOut() twice. The second time should display an "Already Loaned" message.*

⑦ *Store Books and CDs in a LibraryItem array. Loop through it and print only the items where isAvailable is true.*

CHAPTER 6

ABSTRACT CLASS AND INTERFACE

Contents

6.1 Abstract Classes 88

6.2 Inheritance of Abstract Classes 90

6.3 Interfaces 92

6.4 Quick Check 95

6.5 Activities 97

Key Objectives

This chapter introduces the concepts of abstract classes and interfaces in a simple and progressive manner. We will examine their syntax in Java, rules, and practical applications. Through theoretical explanations and practical examples, students will gain a comprehensive understanding of how these abstraction mechanisms contribute to effective object-oriented application design and support best practices in modern Java development.

6.1 Abstract Classes

Recapitulation 6.1

An abstract class in Java is a class that is not complete on its own and cannot be instantiated (you cannot create objects directly from it). It is used as a base class that provides common structure and behavior for other classes. The main purpose of an abstract class is to:

1. *Represent a general concept*
2. *Share common code among related classes*
3. *Force subclasses to implement specific methods*

Recapitulation 6.2

An abstract class is declared using the `abstract` *keyword and may contain:*

- *Abstract methods (methods without a body)*
- *Concrete methods (methods with a body)*
- *Fields and constructors*

The main characteristics are:

- *It cannot be instantiated.*
- *It may contain abstract and non-abstract methods.*
- *A subclass must implement all abstract methods, unless it is also abstract.*
- *It can have constructors, variables, and access modifiers.*

Let's consider the code of Listing 6.1:

1. *Animal is an abstract class that defines a general behavior.*
2. *makeSound() is abstract and must be implemented by subclasses.*
3. *Dog provides its own implementation of makeSound()*
4. *Although the Animal class has been declared abstract, we cannot create an instance of it even if it has a public constructor.*

```java
abstract class Animal {
  Animal() {
    System.out.println("Animal created");
  }
  abstract void makeSound();   // abstract method

```

```
7     void sleep() {                  // concrete method
8       System.out.println("The animal is sleeping");
9     }
10  }
11
12  class Dog extends Animal {
13    void makeSound() {
14      System.out.println("The dog barks");
15    }
16  }
```

Listing 6.1: Example of an abstract class

The following are some other rules related to abstract classes:

- *Abstract methods cannot be private (Subclasses must be able to override them);*
- *Abstract methods cannot be final or static (final methods cannot be overridden and static methods belong to the class, not to objects);*
- *An abstract class can extend only one class;*
- *Java does not allow constructors to be abstract;*
- *If a class has a method declared abstract, the class must be declared abstract;*
- *An abstract method is declared in the same way as any other method, except that its body is indicated only by a semicolon (An abstract method has only the method header).*

R Not being able to create an instance of a class does not conclusively imply that this class is abstract. Notice that an abstract class is a non-instantiable class by definition, but the reverse is not always correct. There are non-instantiable classes that are not abstract, for example, the Math class. If we want create a non-instantiable class, then we simply declare a private no-argument constructor and providing no additional constructors for the class.

6.2 Inheritance of Abstract Classes

Recapitulation 6.3

An abstract class has no practical use on its own. For instance, until the abstract methods of the Animal class (Listing 6.1) are implemented, its remaining components (such as instance variables, concrete methods, and constructors) cannot be effectively utilized. Consequently, subclasses must be created to provide implementations of the abstract methods and make the abstract class functional.

Recapitulation 6.4

Let's consider the following example:

- *Animal is abstract and it cannot be instantiated directly.*
- *makeSound() is abstract and Subclasses must provide their own implementation.*
- *sleep() is concrete: Subclasses inherit it directly.*
- *Dog and Cat are subclasses that implement the abstract method and can be instantiated.*

```
1  // Abstract class
2  abstract class Animal {
3    String name;
4
5    // Constructor
6    Animal(String name) {
7      this.name = name;
8    }
9
10   // Abstract method (no body)
11   abstract void makeSound();
12
13   // Concrete method (has a body)
14   void sleep() {
15     System.out.println(name + " is sleeping");
16   }
17 }
18
19 // Subclass that extends the abstract class
20 class Dog extends Animal {
```

```java
    Dog(String name) {
        super(name);
    }

    // Implementation of abstract method
    void makeSound() {
        System.out.println(name + " says: Bark!");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    // Implementation of abstract method
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Whiskers");

        dog.makeSound();  // Output: Buddy says: Bark!
        dog.sleep();      // Output: Buddy is sleeping

        cat.makeSound();  // Output: Whiskers says: Meow!
        cat.sleep();      // Output: Whiskers is sleeping
    }
}
```

Listing 6.2: An abstract class and Subclasses that extend it

6.3 Interfaces

Recapitulation 6.5

An interface is a completely abstract class used to group methods associated with empty bodies, providing only their signature. A class can then implement one or more interfaces. The main purpose of declaring an interface is to create an abstract specification (or concept) by declaring zero or more abstract methods. Interfaces are used to:

- *Achieve full abstraction.*
- *Support multiple inheritance (a class can implement several interfaces).*
- *Define a common behavior for unrelated classes.*

In Java, an interface is declared using the `interface` *keyword. An interface is declared in its own file with the same name as the interface. The syntax for declaring an interface in Java is as follows:*

```
[modifiers] interface interfaceName {

  <constant-declaration>
  <method-declaration>
  <nested-type-declaration>
}
```

Recapitulation 6.6

An interface can have public or be package visibility. An interface declaration is implicitly abstract, i.e., an interface declaration is always abstract whether we declare it abstract explicitly or not. An interface can have three types of members:

1. *Constant fields: the interface attributes are by default public, static, and final. Although the interface declaration syntax allows the use of these keywords in a field declaration, their use is redundant.*
2. *Abstract, static, private, and default methods: if a method does not have the static, default, or private modifier, it is considered abstract.*
   - *An abstract method in an interface does not have an implementation.*
   - *A static method's declaration contains the static modifier. They are implicitly public. Static methods in an interface are not inherited by implementing classes or subin-*

terfaces (An interface that inherits from another interface). They can be called only by using the interface name. A static method $f()$ of an interface $Inface$ must be invoked using $Inface.f()$.

- *A private method is not inherited and, therefore, cannot be overridden and it can be called only by default or static methods within the same interface.*
- *A default method in an interface is declared with the modifier `default`. A default method provides a default implementation of the method for the class that implements the interface. Unlike concrete method, a default method does not have access to instance variables of the class implementing the interface. The default method can access other members of the interface, for example, other methods and constants.*

The following example shows an interface containing all four types of methods:

```
interface AnInterface {
  // An abstract method
  int m1();
  // A static method
  static int m2() {
    // The method implementation goes here
  }
  // A default method
  default int m3() {
    // The method implementation goes here
  }
  // A private method
  private int m4() {
    // The method implementation goes here
  }
}
```

③ *Nested interfaces and classes: a nested type is a class, interface, enum, or record declared inside another type. When declared inside an interface, nested types are automatically public and static. They are accessed using the interface name.*

Recapitulation 6.7

A class that implements one or more interfaces specifies them using an implements clause. The implements clause uses the keyword `implements` *followed by one or more interface names separated by commas. A class can implement multiple interfaces. Let's consider a class that implements a single interface. The general syntax for such a class declaration is shown below:*

```
[modifiers] class <class-Name> implements <comma-separated-list-of-interfaces> {
  // Class body goes here
}
```

Let's consider the following interface:

```
public interface Swimmable {
  void swim();
}
```

The following example demonstrates the Fish class declaring that it implements the Swimmable interface.

```
public class Fish implements Swimmable {
  private String name;
  public Fish(String name) {
    this.name = name;
  }
  @Override
  public void swim() {
    System.out.println(name + " (a fish) is swimming.");
  }
}
```

There is no limit on the maximum number of interfaces implemented by a class. If a class implements multiple interfaces, its objects is to acquire as many new types as the number of implemented interfaces. L et's consider the following interface Walkable:

```
public interface Walkable {
  void walk();
}
```

The following code demonstrates the Turtle class declaring that it implements the Swimmable and Walkable interfaces.

```java
public class Turtle implements Walkable, Swimmable {
  private String name;
  public Turtle(String name) {
    this.name = name;
  }
  // Adding a bite() method to the Turtle class
  public void bite() {
    System.out.println(name + " (a turtle) is biting.");
  }
  // Implementation for the walk() method of the Walkable interface
  @Override
  public void walk() {
    System.out.println(name + " (a turtle) is walking.");
  }
  // Implementation for the swim() method of the Swimmable interface
  @Override
  public void swim() {
    System.out.println(name + " (a turtle) is swimming.");
  }
}
```

6.4 Quick Check

Quick Check 6.1 *In Java, we can make a class abstract by*

① *Declaring it* `abstract` *using the abstract keyword.*

② *Making at least one method final.*

③ *Declaring all methods as static.*

④ *Declaring at least one method as abstract.*

Quick Check 6.2

① *Can you create an instance of an abstract class?*

② *Must an abstract class include an abstract method?*

③ *What is wrong with the following declaration?*

```
class Vehicle {
  abstract public getVIN();
  ...
}
```

Quick Check 6.3 *Which of the following statement(s) in JAVA is/are correct ?*

① *An abstract class is one that is not used to create objects.*

② *An abstract class is designed only to act as a base class to be inherited by other classes.*

Quick Check 6.4 *Which of the following is incorrect in Java*

① *If we derive an abstract class and do not implement all the abstract methods, then the derived class should also be marked as abstract using* `abstract` *keyword.*

② *Abstract classes can have constructors.*

③ *class can be made abstract without any abstract method.*

④ *A class can inherit from multiple abstract classes.*

Quick Check 6.5 *Which keyword is used by a class to implement an interface in Java?*

① `uses`

② `extends`

③ `implements`

④ `interface`

Quick Check 6.6 *How many interfaces can a Java class implement?*

① *Only one*

② *Unlimited*

③ *Two*

④ *None*

Quick Check 6.7 *Which of the following is correct in java.*

① An interface can contain following type of members: public, static, final fields, default and static methods with bodies.

② An instance of interface can be created.

③ A class can implement multiple interfaces.

④ Many classes can implement the same interface.

6.5 Activities

Exercise 6.1 *Create an interface named Swimmable with the following methods:*

① void swim(): to define how the creature swims.

② `default void breatheUnderwater() { System.out.println("Breathing underwater"); }`.

③ Create two classes, Fish and Dolphin, that implement the Swimmable interface.

④ In the swim() method, print a message specific to the creature, e.g., "Fish is swimming with fins" or "Dolphin is swimming fast".

⑤ Use the default breatheUnderwater() method for one class, and override it in the other class with a custom message if desired.

⑥ In the main method, create objects of Fish and Dolphin, then invoke swim() and breatheUnderwater().

⑦ Add a static method speciesInfo() in the Swimmable interface that prints "All swimmable creatures can move in water" and call it using the interface name.

Solution

```
1  package swimmable;
2
3  // Interface definition
4  interface Swimmable {
5
6    // Abstract method
7    void swim();
```

```
8
9      // Default method
10     default void breatheUnderwater() {
11       System.out.println("Breathing underwater");
12     }
13
14     // Static method
15     static void speciesInfo() {
16       System.out.println("All swimmable creatures can move in water");
17     }
18   }
19
20   // Fish class implementing Swimmable
21   class Fish implements Swimmable {
22
23     @Override
24     public void swim() {
25       System.out.println("Fish is swimming with fins");
26     }
27
28     // Use default breatheUnderwater() from interface
29   }
30
31   // Dolphin class implementing Swimmable
32   class Dolphin implements Swimmable {
33
34     @Override
35     public void swim() {
36       System.out.println("Dolphin is swimming fast");
37     }
38
39     // Overriding default method
40     @Override
41     public void breatheUnderwater() {
42       System.out.println("Dolphin holds its breath while swimming");
43     }
```

```
44  }
45
46  // Main class to test
47  class Main {
48    public static void main(String[] args) {
49      // Create objects
50      Fish fish = new Fish();
51      Dolphin dolphin = new Dolphin();
52
53      // Call methods
54      fish.swim();
55      fish.breatheUnderwater();
56
57      dolphin.swim();
58      dolphin.breatheUnderwater();
59
60      // Call static method from interface
61      Swimmable.speciesInfo();
62    }
63  }
```

Exercise 6.2

1. *Create a new interface called Walkable with the following methods:*
   1. *void walk() that describes how the creature walks.*
   2. *default method rest() that prints:* `System.out.println("Resting");` .
2. *Create a Turtle class that implements both Swimmable (Exercise 6.1) and Walkable.*
3. *Override swim() from Swimmable*
4. *Override walk() from Walkable*
5. *Override the breatheUnderwater() default method from Swimmable*
6. *In the main method, create a Turtle object and invoke swim(), breatheUnderwater(), walk(), and rest().*

Solution

```
// Walkable interface
interface Walkable {
  void walk(); // abstract method

  default void rest() {
    System.out.println("Resting");
  }

  static void walkingInfo() {
    System.out.println("All walkable creatures can move on land");
  }
}

// Turtle class implementing both Swimmable and Walkable
class Turtle implements Swimmable, Walkable {
  @Override
  public void swim() {
    System.out.println("Turtle swims slowly");
  }

  @Override
  public void walk() {
    System.out.println("Turtle walks slowly on land");
  }

  @Override
  public void breatheUnderwater() {
    System.out.println("Turtle holds its breath while swimming");
  }

  // Uses default rest() method from Walkable
}

// Main class
public class Main {
```

```
36   public static void main(String[] args) {
37     // Turtle actions
38     turtle.swim();
39     turtle.breatheUnderwater();
40     turtle.walk();
41     turtle.rest();
42
43     // Static interface methods
44     Swimmable.speciesInfo();
45     Walkable.walkingInfo();
46   }
47 }
```

BIBLIOGRAPHY

[1] Iuliana Cosmina. *Java for Absolute Beginners: Learn to Program the Fundamentals the Java 9+ Way*. Apress, 2018.

[2] Samanta Debasis and Monalisa Sarma. *Joy with Java: Fundamentals of Object Oriented Programming*. Cambridge University Press, 2023.

[3] Claude Delannoy. *Programmer en Java: Couvre Java 10 à Java 14*. 11e. Groupe Eyrolles, 2020.

[4] Claude Delannoy. *S'initier à la Programmation et à l'Orienté Objet avec des Exemples en C, C++, C#, Python, Java et PHP*. Second. Groupe Eyrolles, 2016.

[5] Michel Divay. *La Programmation Objet en Java: Cours et exercices corrigés*. Dunod, 2006.

[6] Amany Fawzy Elgamal. *Object-Oriented Programming*. Cambridge Scholars Publishing, Newcastle, 2024.

[7] Jamila Sam and Jean-Cédric Chappelier and Vincent Lepetit. *Introduction à la Programmation Orientée Objet (en JAVA)*. EPFL Press, 2016.

[8] Vaskaran Sarcar. *Interactive Object-Oriented Programming in Java: Learn and Test Your Programming Skills*. Second. Apress, 2020.

[9] Kishori Sharan and Adam L. Davis. *Beginning Java 17: Fundamentals Object-Oriented Programming in Java 17*. Third. Apress, 2022.