People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

Abderrahmane Mira Bejaia University

جامعة بجاية
Tasdawit n Bgayet
Université de Béjaïa

Faculty of Exact Sciences

Computer Science department

Course handout

Module: Object Oriented Programming

Level: 2nd year license

Specialty: Computer Systems

Established by Dr. Mohamed Mohammedi

mohamed.mohammedi@univ-bejaia.dz

Version 1

2023/2024

Preface

Welcome to the fascinating world of object-oriented programming! In this course handout, I invite you to dive into the heart of this modern programming paradigm that has revolutionized the way we design and develop software.

Object-oriented programming is much more than just a coding technique. It is a conceptual approach that allows us to effectively structure and organize our code by modeling it around autonomous entities called objects. These objects are endowed with characteristics (attributes) and behaviors (methods) which interact with each other to carry out complex tasks.

This course handout was designed for beginners who want to gain a solid understanding of the fundamental concepts of object-oriented programming. I have taken care to present the basic principles in a clear and accessible manner, avoiding technical jargon as much as possible.

Throughout the chapters, you will learn about essential concepts such as encapsulation, inheritance, polymorphism and abstraction. You will learn how to create classes, instantiate objects, and manipulate their attributes and methods. I will guide you in implementing these concepts through real-world examples and practical exercises.

Beyond the syntax and syntax of programming languages, I will emphasize object-oriented thinking. I will help you adopt a problem-solving approach based on modeling the real-world using objects and interactions between them.

It is important to note that object-oriented programming is used in many popular programming languages such as Java, C++, Python and many more. The principles you will learn in this course handout can therefore be applied in different contexts and will help you become a versatile and competent developer.

So, whether you are a computer science student, a self-taught person passionate about programming or a professional retraining, this course handout will be your ideal companion to introduce you to object-oriented programming. Prepare to dive into a world where modularity, reusability and maintainability of code are at the heart of every line you write.

Take the first step toward mastering object-oriented programming and discover how to create robust, flexible, and scalable software. Enjoy reading and enjoy your journey into the wonderful world of object-oriented programming!

Contents

Foreword

To my students of yesterday, today and tomorrow… and to all the others!

Object Oriented Programming (OOP) is a fundamental and essential concept in the field of computer science, and mastering it is essential to becoming a competent developer. Object-oriented programming is a computer programming framework. A fundamental principle of OOP is that computer programs are composed of individual units or objects that can function as subprograms. OOP achieves three major goals of software engineering: reusability, flexibility, and extensibility. OOP = object + class + inheritance + polymorphism + message, where the basic concepts are class and object.

The Object-Oriented Programming method involves simulating the human way of thinking as much as possible. This helps make the software development process as close as possible to the method and process of human understanding of the world and solving real-world problems. In other words, it aims to coherently describe the problem space and the solution space of the problem. This approach abstracts entities from the objective world as objects in the problem domain, thus ensuring a clear and concise structure.

Object-oriented programming takes objects as its core, and this method believes that a program is composed of a series of objects. A class is an abstraction of the real world, including data representing static properties and operations on data, and an object is an instantiation of a class. Objects communicate with each other by transmitting messages to simulate the relationship between different entities in the real world. In object-oriented programming, objects are the basic modules that make up a program.

This course handout has been carefully developed to provide you with a solid foundation in object-oriented programming. It gradually addresses the key concepts of object-oriented programming, starting with the notions of class, object, encapsulation, inheritance and polymorphism. Each concept is explained clearly and concisely, accompanied by relevant examples and practical parts to help you consolidate your knowledge. The main objective of this course material is to enable you to develop the skills necessary to design and implement robust and scalable object-oriented programs. By understanding the principles and best practices of object-oriented programming, you will be able to create high-quality, efficient applications.

Organization of this course handout

From a structuring point of view, this course handout is structured around the following chapters:

- Chapter 1 presents the main concepts of object-oriented programming which must be understood systemically.

- Chapter 2 discusses encapsulation, which is a mechanism for hiding the implementation of objects while making them accessible through a number of services. Because it allows data manipulation services to be defined within an object, encapsulation is a guarantee of data integrity.

- Chapter 3 presents the inheritance mechanism allowing us to make objects from those we have on hand while adding new attributes and methods to them. However, they remain full representatives of all their superclasses: same attributes, same method signatures. This chapter also describes one of the possibilities offered by inheritance and which is the basis of polymorphism: the redefinition in subclasses of methods first defined in the superclass.

Who is this course handout for?

This course handout offers the reader maximum information on learning Object Oriented Programming (OOP) with the Java language. It is intended for 2nd year undergraduate students in Computer Science and developers who already have initial experience of structured programming and who wish to move on to Object Oriented Programming with the Java language, to develop portable applications.

Basics of Object-Oriented Programming

Chapter 1.

Outline	1.1. Introduction 1.2. Fundamental concepts of OOP 1.3. Objects and Classes 1.4. Introduction to Java 1.5. Exercises 1.6. Answers to exercises
Goals	➢ Develop an understanding of Java programming principles; ➢ Learn the key elements of the language, including primitive types, operators (arithmetic, relational, logical, and assignment), expressions, statements, control structures (loops, conditional statements), and arrays; ➢ Understand the concepts of classes, objects, methods, and attributes; ➢ Master string concatenation; ➢ Read data from the keyboard and display the results on the screen; ➢ Become familiar with best practices for producing high-quality code; ➢ Enhance problem-solving skills through practical exercises.

1.1.Introduction

Object-oriented programming is a programming method that maps the real world onto a computer model through objects. OOP designs software architecture around objects rather than functions and logic. An object can be defined as a data type with unique properties and methods.

1.2. Fundamental concepts of OOP

a. A brief history of OOP

T The use of the terms "object" and "object-oriented" in modern programming has its origins at MIT (Massachusetts Institute of Technology) in the late 1950s and early 1960s. In 1960-61, Ivan Sutherland developed Sketchpad[1], a software program that introduced the concepts of "object" and "instance" in its 1963 technical reporting glossary.

In 1962, Kristen Nygaard started a simulation language project at the Norwegian Computer Center, which led to the design of the Simula programming language. Simula introduced fundamental concepts of object-oriented programming, such as classes, objects, inheritance, and dynamic linking.

[1] Sketchpad is a computer program written by Ivan Sutherland in 1963 as part of his doctoral dissertation at MIT.

In 1965, Simula was recognized as the first widely used "object-oriented" programming language. Much like Sketchpad, Simula had object-related capabilities and introduced classes, class inheritance, subclasses, and virtual methods.

Inspired by the simplicity of LISP [2]and the concepts of classes and objects of Simula 67, Alan Kay created Smalltalk around 1966 or 1967. Smalltalk refined the concept of objects, viewing them as minicomputers, combining data (properties) and operations on this data (methods), reflecting the very structure of computers.

b. Procedural programming vs object-based programming

There are two different programming paradigms: procedural programming and object-oriented programming. Procedural programming focuses on procedures or functions, which are blocks of code that perform particular tasks. It often happens that variables are declared outside of functions and passed as parameters to functions that use these variables. For simple, linear tasks that involve a sequence of clearly defined steps, procedural programming is often used.
In contrast, object-oriented programming focuses on objects, which are instances of classes. The properties and methods of objects are defined by their classes, and the objects interact with each other by exchanging messages. Variables are encapsulated in objects and cannot be accessed from outside the object. Complex, modular tasks that require clear organization and reusability of code frequently use object-oriented programming.
Here are some of the advantages of object-oriented programming over procedural programming:

Advantages of procedural programming are indeed the following:

- Ease of learning and understanding
- Efficient in terms of memory and execution time
- Well suited to simple and linear tasks

Procedural programming has disadvantages such as the following:

- Difficulty managing the increasing complexity of programs
- Difficulty reusing code
- Risk of undesirable side effects on global variables

Advantages of object-oriented programming are in fact are as follows:

- Clear and hierarchical organization of the code
- Easy code reuse
- Encapsulation of variables for better security and better modularity

The disadvantages of object-oriented programming include the following:

- Longer learning curve and increased complexity
- Cost in terms of memory and higher execution time

[2] LISP (LISt Processing) is a programming language that was developed in the early 1950s by John McCarthy.

- Difficulty designing and implementing a consistent and efficient class hierarchy

c. Code reuse

A programmer uses all or parts of code already written and tested to create a new program by reusing the code. Reused code can come from a variety of sources, such as other programs created by the programmer, books, or code snippets available online. Developers can benefit significantly from code reuse, saving time and improving the quality of their programs.

d. Introduction to modularity

In object-oriented programming, modularity is a key concept that refers to the ability to divide a program into distinct, self-contained modules, with each module having distinct and consistent functionality.

Modules can be classes or packages that can be reused in other programs or integrated into larger programs. Modularity improves code organization, facilitates maintenance, and allows code reuse.

Each module can be designed to perform a typical, independent task thanks to the modularity of object-oriented programming. It is possible to develop, test and debug each module individually before being integrated into the overall program. Additionally, modules can be reused in other programs, saving development time and effort.

Depending on the specific needs of the program, modularity can be implemented in a variety of ways. For example, classes can be organized into logical modules using packages. By limiting dependencies between classes and using clear, well-defined communication interfaces between modules, classes can be created to be independent and reusable.

Note:

Developers can create more efficient, more robust, and easier to maintain programs by dividing a program into self-contained, reusable modules.

1.3. Objects and Classes

a. Object notions

Programming entities called objects can represent elements of the real or abstract world. They can include individuals, animals, things, concepts or processes. Objects have their own properties and behaviors, and they can interact with other objects or their environment.

b. Class notions

A class is a template used to create objects. A class specifies the properties and behaviors that objects derived from it will have. Classes allow the hierarchical organization of similar objects. For example, a class ''truck'' may have subclasses such as ''electric dump truck'' or ''refrigerated truck''.

c. Attributes

Properties of an object that describe its characteristics or states are called attributes. They can be of different types such as including integers, strings, booleans, arrays, objects, etc. For

example, a truck may have attributes like its make, model, year of manufacture, load capacity, mass, maximum speed, fuel consumption, horsepower, etc.

d. Message notion

Messages are instructions sent to an object to ask it to perform an action or return a value. Messages can contain parameters that inform the object so that it can perform the requested action. For example, a “slow down” message may be sent to a truck asking it to reduce its speed.

e. Problem solving through message exchange

Problem solving based on message exchange is an object-oriented programming approach. It consists of solving a problem by breaking it down into a set of objects which interact with each other by exchanging messages. By using objects that have clearly defined responsibilities and that interact coherently, this approach allows for a clearer and more modular representation of complex problems.

1.4. Introduction of Java

a. A brief history of Java language

Java is a programming language developed in 1991 by James Gosling (the father of Java) and his team at Sun Microsystems. Originally called Oak, the first version was created in just 18 months. The initial idea was to design a platform-independent language, which could be integrated into various consumer electronic devices such as household appliances. However, in its early days, the market response was not very enthusiastic.

With the rise of the Internet in the 1990s, Sun saw the potential of Oak for web application development. Consequently, they decided to rename the language "Java" in 1995, in reference to the Indonesian island famous for its coffee. Since then, Java has gradually become an essential language for the development of web applications.

Today, Java is no longer just a simple programming language, but a real technical system composed of a series of software and computer specifications. Java is the basis for almost all types of network applications, embedded applications, mobile applications, games, and also many enterprises software. Java has thus become a global standard for web content and application development.

Indeed, Java is a very popular programming language and is used in a wide variety of fields. Here are some important points regarding the use of Java:

- According to statistics, about 97% of enterprise desktops run Java. This is due to the popularity of Java in enterprise application development.
- In the United States, approximately 89% of desktop computers (or computers) run Java. This statistic also highlights Java's widespread presence in the personal computing space.

- There are approximately 9 million Java developers worldwide. Java is known to be one of the most used and in-demand programming languages, which explains the high number of developers working with Java.
- Java is considered the number one choice of developers due to its versatility, portability, and large developer community.
- Java is also considered the number one deployment platform, which means it is widely used to deploy applications and services.
- Around 3 billion mobile phones worldwide use Java. Java has been used for many years to develop mobile applications, especially for feature-based cell phones.
- All Blu-ray disc players come with Java. Java provides a platform for developing interactive features for Blu-ray discs.
- Around 5 billion Java cards are used in different electronic devices.
- About 125 million TVs run Java. Java is used to develop interactive applications and advanced features for smart TVs.
- Top five Original Equipment Manufacturers (OEMs) offer Java ME (Micro Edition) for developing embedded applications on devices such as mobile phones, tablets and smart devices.

When it comes to Java, it is naturally inseparable from JDK, JVM and JRE. The relationship between the three is as follows:

- JDK (Java Development Kit): The Java development kit includes the Java language, development tools and the library of predefined classes, called the Java Class Library. It provides the minimum environment needed to develop Java programs.

- JVM (Java Virtual Machine): Java virtual machine runs on different operating systems like Linux, Windows, Solaris, etc. It executes Java class files compiled into bytecode.

- JRE (Java Runtime Environment): The Java runtime environment consists of the JVM and the Java API [3]base class libraries. It is often considered to be the combination of the JVM and the base class libraries of JavaSE.

In Java, several packages are widely used depending on the needs of developers. Here is a list of the most common packages:

- java.lang: Contains fundamental Java classes, such as String, Math, System, etc. This package is imported by default.

- java.util: Includes utility classes for data structures such as ArrayList, HashMap, collections, as well as classes for handling dates and times.

[3] An API (**Application Programming Interface**) is a set of methods, classes, interfaces, and tools that allow developers to create applications or interact with other software. In other words, using an API allows developers to access ready-to-use functionality, which speeds up the development process and facilitates interoperability with other systems.

- java.io: Provides classes for input and output, such as File, InputStream, OutputStream, Reader, and Writer.
- java.net: Contains classes for network programming, including Socket, URL, and URLConnection.
- java.nio: Provides more efficient non-blocking I/O and buffer manipulation capabilities.
- java.sql: Provides classes for interacting with databases via JDBC (Java Database Connectivity).
- javax.swing: Used to create graphical user interfaces (GUIs) with components such as JFrame, JButton, and JPanel.
- java.awt: Contains classes for creating basic graphical user interfaces and for event handling.
- java.security: Provides classes for security, including encryption and certificate management.
- java.time: Introduced with Java 8, this package provides classes for handling dates and times, such as LocalDate, LocalTime, and ZonedDateTime.

These packages cover a wide range of functionality and are essential for developing Java applications.

Java portability

The main feature of Java is its portability. This means that the same program can be deployed on different operating systems without modification. This greatly reduces the difficulty of cross-platform development.

The portability of Java is made possible thanks to the JVM (Java Virtual Machine). During compilation, Java source code is transformed into intermediate code called bytecode, rather than operating system-specific machine instructions (see Figure 1). This bytecode can then be executed on any implementation of the JVM, which is responsible for transforming it into machine code understandable by the underlying operating system.

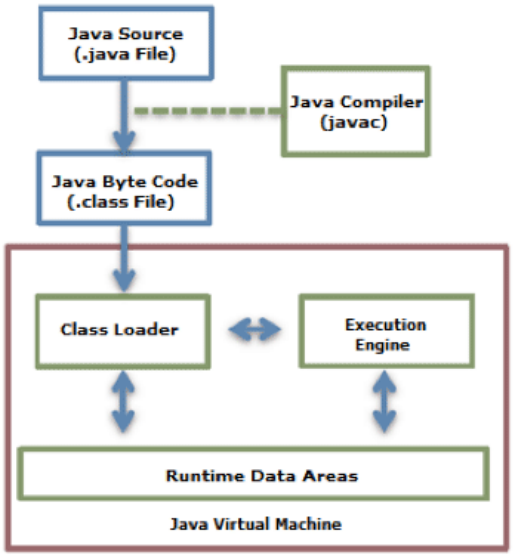


Figure 1: The process followed to execute java code.

Note:

Java is considered a portable language because the bytecode remains the same regardless of the execution environment.

Java Virtual Machine

The Java Virtual Machine (JVM) is a virtual computer run by software. All Java programs can be run if they are compatible with the JVM. Different versions of the JVM exist to accommodate different operating systems, ensuring portability of Java applications.

Compiled Java programs must first be transformed into bytecode and then interpreted into the JVM. The interpretation of all Java programs must be done on a virtual machine. The JVM is responsible for reading and executing the compiled bytecode, independent of the platform.

JVM application execution mechanism

When a Java program is executed, the process occurs as follows: Java source code (.java) is first compiled by the Java compiler, which generates Java bytecode (.class). This Java bytecode is then loaded and executed by the Java virtual machine. The JVM reads the bytecode, checks it to ensure it is valid and secure, and then executes it. The JVM provides an execution environment for the Java program, managing memory, resource allocation, thread execution, etc. It acts as an abstraction layer between the Java program and the underlying operating system. At runtime, the JVM can also compile parts of the bytecode into native machine code on the fly to improve performance, thanks to its JIT (Just-In-Time) compiler. The JVM also implements advanced features such as the garbage collector which automatically manages the memory allocated to Java objects.

Java performance

Although Java runs on a virtual machine rather than directly on the operating system, its performance has improved greatly in recent years. Advances in JVM design and implementation have largely closed the performance gap with native applications.

A key aspect of these advances is the JIT (Just-In-Time) compiler built into the JVM. This analyzes the running Java bytecode in real time and dynamically generates machine code optimized for the target platform. This dynamic compilation makes it possible to achieve performance levels very close to those obtained with native code, without the constraints of static compilation.

In addition, developers can further refine the performance of their Java applications by making fine adjustments to the JVM configuration. For example, by adjusting the heap memory size, garbage collection settings, or enabling some specific optimizations. These adjustments allow the execution of the JVM to be adapted to the specific needs of each project.

Thanks to these constant innovations, performance is no longer a major barrier to Java adoption. The language remains a relevant choice for a wide range of applications, from the most lightweight to the most demanding in terms of resources.

b. Declaration of a class

In Java a class is declared by the **class** keyword followed by the class name. We will return in chapter 2 to the **public** keyword which precedes and which makes it possible to specify the scope of the definition of this class. Then, we open a block with braces to declare the contents of the class.

```
class MyFirstClass {

}
```

Note:

In Java, a class is declared in its own file which must have the same name as the class with the extension **.java**. By convention, each word making up the class name must have the first letter written in capital letters. Therefore, it is necessary to create the **MyFirstClass.java** file.

c. The "System.out.println()" instruction

It is used in Java to display information to the console. Here is an example to display text and the contents of a variable:

Example 1 - Text display:

```
System.out.println("Good morning, how are you doing?");
```

This will display the message to the console: Good morning, how are you doing

Example 2 - Displaying the contents of a variable:

Suppose we have a variable `name` of type `String` containing the name of a person. Here is how we can display the contents of this variable:

```
String name = "DUBOIS Aurélie";
System.out.println("The name is: " + name);
```

This will display the message on the console: The name is: DUBOIS Aurélie

Using the ''+'' concatenation operator, we can combine text with the variable contents to display them together.

d. Overall structure of a program in Java

A Java program can use one or more programmer-defined classes. Executing a Java program consists of executing a method called "main()" which is declared in the main class of the program. The main class of the program must have the same name as the file in which it is declared. The main() method can use the class in which it is declared and other classes that belong to the same program.

Exemple:

```
class MyFirstProg {

	public static void main(String[] args) {
	System.out.println("Hello World");
	}

}
```

- The static keyword specifies that the main method of the MyFirstProg class is not linked to a particular instance (object) of the class.
- The String [ ] args parameter of the main method allows you to retrieve arguments passed to the program when it is launched. You can run a program without providing arguments but the String [ ] indication is mandatory.
- We can write String [ ] args or (String args [ ]) the args parameter is an array of String type objects, used to represent character strings.
- The keyword public in public static void main is mandatory for our program to execute. Here it is no longer really a problem of access rights, but rather a convention which allows the virtual machine to access the main method.

Note:

The main() method of a Java program is the first method executed when the program is launched. It is she who will eventually invoke the other methods.

Note:

The void keyword specifies that a method should not have a return value.

Note:

In Java, blocks of code are delimited by curly braces ''{ }'', while each statement ends with a semicolon (;).

e. Comments

Java allows comments in source programs, like any other advanced language. Java programs recognize three types of comments:

- Single line comment

Single line comment is used to comment out a single line.

Syntax :

```
// This is a single line comment
```

Example:

```
public class Main {
   public static void main(String[] args) {
          // Displaying the message Hello World!
          System.out.println("Hello World!");
   }
}
```

- Multi-line comment

Multi-line commenting is used to comment out multiple lines of code.

Syntax:

```
   /*
   This is a comment
   on several
   lines
   */
```

Example:

```
public class Main {
   public static void main(String[] args) {
          /*
              Declare and display the
              Hello World message!
          */
          String msg = "Hello World!";
          System.out.println(msg);
   }
}
```

- Documentation comment

This type of comments is usually used when writing code for a project, as it helps generate a reference documentation page, which can be used to get information about the methods, its parameters, etc.

Syntax:

```
   /**
   *
   * This is a documentation
   * Comment
   *
   **/
```

Example:

```
   /**
   *
   * Java program to declare and display
```

```
* the message Hello World!
*
* @author  Dr Mohamed Mohammedi
* @version 1.2
* @since   03-08-2023
* @link    www.waytolearnx.com
*
**/
public class Main {
	public static void main(String[] args) {
		String msg = "Hello World!";
		System.out.println(msg);
	}
}
```

Note:

To generate Javadoc documentation comments in NetBeans, start by adding Javadoc comments to your source code using the syntax "**/** ... ***/** ". Then, right-click your project and select "**Generate Javadoc**", configure the generation options, and then click "**Generate**". Once the generation is complete, view the generated documentation in the output folder.

a. Types and Control Structures in Java

- **Data types**

In Java, data types are used to determine the size of data in memory as well as to know its representation (its format). Java offers two types of data, namely:

- Primitive types: include numeric, boolean, and character. These are predefined types.
- Object types: include arrays, strings and classes.

The different primitive types are represented in the following table:

Elementary type	Variation interval	Number of bits
boolean	false, true	1 bit
char	Unicode characters (valeurs 0 à 65536)	16 bits
double	Double precision floating point ~5.10^{308}	64 bits
float	Single precision floating point ~9.10^{18}	32 bits
int	Integer signed: [-2^{31}, $+2^{31}$ - 1]	32 bits
long	Integer signed long: [-2^{63}, $+2^{63}$ -1]	64 bits
short	Integer signed short: [-2^{15}, $+2^{15}$ -1]	16 bits
byte	[-128, +127]	8 bits

In the table above it is indicated that there is a data type for representing Boolean values, identified by the keyword "**boolean**". A data type for handling characters, identified by the keyword "**char**". Additionally, there are two floating point data types (**float** and **double**) as well as four integer data types (**int**, **long**, **short**, and **byte**).

- Variables

A variable is an object identified by its name that can contain data that can be modified during the execution of the program. Variables in Java must meet certain criteria:

- Variable names must begin with a letter or "_" or with the "$" sign.
- Variable names cannot be Java language keywords.
- With the exception of spaces and punctuation[4] marks, variable names can be composed of letters and/or numbers.
- Java is a case-sensitive language, which means that it distinguishes between a variable written in uppercase and a variable written in lowercase.

1. Variable declaration

To be able to use a variable, you must define it, that is to say give it a name, but above all a type of data to store so that a memory space conforming to the type of data it contains is reserved for it.

The syntax of the declaration:
Type variable_name ;

Example:

```
int x ;
```

In Java, it is possible in a single instruction to declare several variables of the same type by separating each of them with a comma.
Syntax
Type $variable_1$_name , $variable_2$_name, … ,$variable_n$_name;

Example:

```
int x, y, z;
```

2. Assigning data to a variable

To store data in a variable already declared previously, you must make an assignment using the assignment operator "**=**".
variable_name = data;

Example: `x=123;`

A variable can be initialized when it is declared. It is also possible in a single instruction to assign a value to several variables.

Examples:

```
int w=4 ;
char myCharacter='A';
int x=123, y=12, z=3 ;
```

[4] Punctuation marks are: period (.), question mark (?), exclamation mark (!), comma (,), semicolon (;), colon (:), ellipsis (…), parentheses ( ), brackets [ ], quotation marks "", hyphen (-) and slash (/).

3. Constants

A constant is a value of a variable that does not change during program execution.

- **Character strings**

In Java, strings are represented by the String class. The character string must be placed in quotation marks. Here are some examples of using strings in Java:

1. Declaration of a character string:

```
String message = " Hello World !";
```

2. String concatenation:

```
String name = "John";
String message = " Hello, " + name + "!"; // "Hello, John!"
```

In Java, there are several special characters that can be inserted into a string using escape sequences. Here are some of the most commonly used special characters in Java:

- \n: represents a line break
- \t: represents a tab
- \b: represents a step back (backspace)
- \r: represents a carriage return
- \': represents a single apostrophe (')
- \": represents a double quote (")
- \\: represents a backslash (\)

- **Operators**

Java provides a wide range of operators for manipulating variables. All Java operators can be divided into the following groups:

1. Arithmetic operators

They are used on operands to perform arithmetic/mathematical operations:

a) Addition: The ''+'' operator adds two operands. For example, x+y.
b) Subtraction: the operator ''-'' subtracts two operands. For example, x-y.
c) Multiplication: the operator ''*'' multiplies two operands. For example, x*y.
d) Division: the ''/'' operator divides the first operand by the second. For example, x/y.
e) Module: The ''%'' operator returns the remainder when the first operand is divided by the second. For example, (x%y).

Unary arithmetic operators are those that include the following:

– **Increment**: The "++" operator allows you to increment the value of an integer. When it is placed in front of the variable name (also called the pre-increment operator), its value is incremented instantly. For example, ++x. And when it is placed after the variable name (also called post-increment operator), its value is held temporarily until that

statement is executed and is updated before the next statement is executed. For example, x++.

– Decrement: The ''- -'' operator is used to decrement the value of an integer. When it is placed before the variable name (also called the pre-decrement operator), its value is decremented instantly. For example, --x. And when it is placed after the variable name (also called post-decrement operator), its value is held temporarily until that statement is executed and it is updated before the next statement is executed. For example, x--.

2. Relational operators

To compare two values, the following relational operators are used:

a) The operator ''= ='' checks whether the two operands are equal or not. If yes, it returns true. Otherwise, it returns false. For example, 5==5 will return true.
b) The ''!='' operator checks whether the two operands are equal or not. it returns true if the two operands are not equal, otherwise it returns false. This is the exact Boolean complement of the ''=='' operator. For example, 5!=5 will return false.
c) The operator ''>'' checks if the first operand is greater than the second operand. If so, this returns true. Otherwise, it returns false. For example, 6 > 5 will return true.
d) The operator ''<'' checks if the first operand is less than the second operand. If so, this returns true. Otherwise, it returns false. For example, 6 < 5 will return false. <
e) The operator ''>='' checks if the first operand is greater than or equal to the second operand. If so, this returns true. Otherwise, it returns false. For example, 5>=5 will return true. L'opérateur ''**<=**'' vérifie si le premier opérande est inférieur ou égal au deuxième opérande. Si c'est le cas, cela retourne true. Sinon, il retourne false. Par exemple, 5 <=5 retournera également true.

3. Logical operators

They are used to combine two or more conditions or constraints or to complete the evaluation of the initial condition taken into consideration. Here is a description of them:

a) Logical AND: the ''**&&**'' operator returns true when the two conditions considered are met. Otherwise, it returns false. For example, (a && b) returns true when a and b are true (i.e. non-zero).

b) Logical OR: the ''**||**'' operator returns true when one (or both) of the considered conditions is met. Otherwise, it returns false. For example, a (a || b) returns true if either a or b is true (i.e. non-zero). Of course, it returns true when a and b are both true.

c) Logical NON: The operator "!" Returns true if the condition considered is not met. Otherwise, it returns false. For example, (!A) returns true if a is false, that is, when a = 0.

4. Assignment operators

In Java, assignment operators are used to assign values to variables.

Operator	Example	Equivalent to
=	x=5	x=5
+=	x+=5	x=x+5
-=	x-=5	x=x-5
*=	x*=5	x=x*5
/=	x/=5	x=x/5
%=	x%=5	x=x%5
&=	x&=5	x=x&5
>>=	x>>=5	x=x>>5
<<=	x<<=5	x=x<<5

- Conditions

If, if-else, switch and ternary conditions are the different types of conditions used in Java. The syntax for each type is as follows:

- If instruction: The if statement is the most basic testing structure. It allows you to execute a series of instructions if a condition is met.

The syntax of the if statement is as follows:

```
if (condition) {
      // Block of code to execute if the condition is true
}
```

- If-else instruction: The if statement in its basic form only allows you to test a condition, but most of the time you would like to be able to choose which statements to execute if the condition is not met. The if...else statement allows you to execute another set of statements if the condition is not met.

The syntax of the if...else statement is as follows:

```
if (condition) {
      // Block of code to execute if the condition is true
      } else {
      // Block of code to execute if the condition is false
}
```

- Switch instruction: The syntax of the switch statement is:

```
      switch(expression) {
```

```
        case value1: // Code to execute if expression==value1;
        break;
        case value2: // Code to execute if expression==value2;
        break;
        ...
        case valuen: // Code to execute if expression==valuen;
        break;

         default: // Code to execute if none of the previous cases
         match;
        break;
    }
```

The Switch statement checks the value of a variable (expression) and compares that variable with each different value from top to bottom. Each value compared is called a case. When a case is true, the command block for that case will be executed. If all cases are false the default command block will be executed.

Note:

Break is an instruction used to exit a control instruction. In other words, it helps the program exit the Switch statement.

- **Ternary instruction:** In Java, the ternary statement is a concise way of writing a condition with a single expression. It allows you to evaluate a condition and return a value based on this condition. Here is the general syntax:

```
variable =(condition) ? // ValueIfTrue: // ValueIfFalse
```

condition is a Boolean expression that is evaluated. If the **condition** evaluates to **true**, the value of **valueIfTrue** is assigned to the **variable**. If the **condition** evaluates to **false**, the value of **valueIfFalse** is assigned to the **variable**.

Example:

```
public class TernaryCondition {

  public static void main(String[] args) {
      int age = 12;

      String type=(age>18)?" Adult" : "Teenager";

      System.out.println("You are: "+type);
  }
      }
```

The execution result is:

```
You are: Teenager
```

- **Loops**

Loops are structures that allow the same series of instructions to be executed several times until a condition is no longer satisfied. The general overview of the different loops in Java is in fact the following:

The most common syntax is a for loop, which consists of a declaration, a condition and an increment/decrement:

```
for (initialisation; condition; increment/decrement) {
      // Block of code to execute
      }
```

While and do-while loops are also available, which have similar syntax to the for loop:

```
while (condition) {
    // Block of code to execute
}
```

```
do {
    // Block of code to execute
      } while (condition);
```

Note:

The do…while loop always executes the instructions it contains at least once.

There is also a forEach loop which is useful when you want to iterate through all the elements in a collection and perform an action on each one:

```
for (element: collection) {
    // Block of code to execute
      }
```

Instruction continue

The continue statement is used to skip one loop iteration and move on to the next iteration.

Example

```
for (int i=0;i<7;i++){
            if (i == 3){
            // If i is 3, we move on to the next iteration without
            executing the rest of the code
            continue;
            // In all other cases, we execute the rest of the code
            }
            System.out.println(i);
}
```

b. Classes et instantiation

1. Classes

A class represents a model for constructing an object. It is an abstract description in terms of data and behavior of a family of objects. An object class is made up of a static part (attributes) and a dynamic part (methods). The skeleton declaration of a class is as follows:

```
class Myclass{
type₁ a₁; // Attribute a₁
type₂ a₂; // Attribute a₂
```

```
.
.
.
type3 Method1(…){ // Method1
…
}

Type4 Method2(…){ // Method2
…
}
…
}
```

Note:

A method must be declared in a class. It is defined by its signature, followed by its body(). The signature of a method includes the method name and the types of its parameters. The body of a Java method is defined by a syntax identical to that of C methods.

2. Constructor: Default constructor and other constructors

A constructor is a method which is responsible for initializing instances (objects) and which is systematically called when declaring an object, that is to say at the start of the life of each instance.

To declare a constructor, simply add a method that has the same name as the name of the class, followed by the parameters necessary for initializations and followed by the definition of the method in question.

Example of a constructor

```
public class Triangle {
    public Triangle (double base, double height) {
      // Constructor
    }
}
```

Constructors are methods like any other, but they have some differences from traditional methods:

- They do not have a return type, not even the “void” type. That is, a constructor does not return any values.
- They have the same name as the class in which they are located.
- They are called automatically each time an instance of the class is created.

Exemple

```
public class Triangle {
    public double base;
    public double height;
    public Triangle(double b, double h) {
        base = b;
        height = h;
    }
    public double surface() {
```

```
        return (base * height)/2;
    }
}
```

In the previous example, it is possible to observe the presence of two distinct methods. The first method designates the constructor of the class, which has the same name as the class itself, but without specifying a return type. On the other hand, the "area" method returns the value corresponding to the area of the triangle, thus distinguishing itself from the first method.

Initialization by constructor

The declaration with initialization of an object is done according to the following syntax:

Syntax

```
NameClass object = new NameClass(var1,...,varn);
```

For example, to create a Triangle object with specific values, we can use the following line of code:

```
Triangle triangle = new Triangle (5.2, 7.9);
```

The ''new'' keyword followed by the class name makes a call to the constructor to which we pass the corresponding values of the various required parameters in parentheses.

Default constructor

A default constructor is called without providing initialization values, that is, without any parameters. It is simply a constructor that has no parameters. Here is an example of a constructor in the Triangle class:

```
Triangle (){
    base = 7.3 ;
    hauteur = 3.5 ;
}

 Triangle (double x){
    base = x ;
    hauteur = 4.5 * x ;
}

Triangle (double b, double h){
    base = b ;
    hauteur = h;
}
```

The last two constructors are not default constructors because they have a non-empty parameter list.

To invoke the third constructor, you must create a Triangle type variable by calling the constructor using the "new" keyword while passing the necessary parameters.

Example:

```
Triangle triangle = new Triangle (4.2, 7.5);
```

If a constructor is not specified, the compiler automatically generates a minimal version of the default constructor. This default constructor performs minimal initialization, assigning default values such as 0, 0.0, or false to all base type attributes, and initializing objects to null.
Once an explicit constructor is defined, whether default or not, the default constructor is gone.

Note:

A default constructor is produced automatically (implicitly) by the compiler when a class does not explicitly define one. This implicit constructor, which has no parameters, allows attributes to be initialized with default values.

Invoke another constructor

When a class has multiple constructors, Java allows any of those constructors to call another constructor of the same class. This is done using a special syntax: the "this" keyword. By using "this" followed by parentheses, we provide the arguments corresponding to the constructor we wish to call. This approach avoids unnecessary code duplication. However, only one "this" statement is allowed in a constructor, and that statement must be the very first statement invoked. No other instruction can be placed before "this". It is also possible in Java to give default values to attributes without using a constructor, by initializing them when they are declared.

Example

Here is an example that illustrates the use of "this" to invoke another constructor:

```
public class Triangle {

public double base;
double height public;

// First constructor that initializes attributes with default values
public Triangle() {
this(67.8, 23.6); // Call the second constructor with default values
}

// Second constructor that initializes attributes with provided values
public Triangle(double b, double h) {
base = b;
height = h;
}

public doublesurface() {
return (base * height) / 2;
}

public static void main(String[] args) {
Triangle t1 = new Triangle(); // Use the default constructor
Triangle t2 = new Triangle(10.0, 5.0); // Use constructor with parameters

System.out.println("Surface of triangle t1: " + t1.surface());
System.out.println("Surface of triangle t2: " + t2.surface());
}
}
```

Notes:

- For clarity and good programming practice, it is recommended to initialize attributes in constructors rather than using default values directly in the declaration.
- The example provided shows how a class can have a default constructor that calls another constructor to initialize attributes.

Copy constructor

The purpose of the copy constructor is to initialize an instance with a copy of another instance of the same class. Its header is precisely defined because it is a constructor. Therefore, it has the same name as the class, but it only takes one parameter, which is another instance of the same class.

```
public class Triangle {

    private double base = 7.5;
    private double height = 4.7;
    public Triangle (Triangle otherTriangle){
       height = otherTriangle.height;
       base = otherTriangle.base;
    }
}
```

In Java, there is no default version of the copy constructor. This means that if no special measures are taken, it is not possible to create direct copies. Therefore, this constructor does not exist by default and must be explicitly defined if one wishes to make a copy.

The copy constructor is not the only method of making copies in Java. A more commonly used method is the **clone** method.

2.1. Instantiating a class or creating objects

An object is an instance of a class, so creating objects is called instantiation. This instantiation is done using the **new** operator followed by the name of the class to instantiate and parentheses containing the instantiation parameters (empty parentheses if there are no parameters). Necessarily, the object to be created must first be declared with the type of the appropriate class.

The declaration and creation of a **class A** object can be done in the following forms if **class A** defines a constructor which accepts **n parameters** (arg_1, arg_2,...arg_n):

```
classA objet =new classA(varg1,varg2,...,vargn);
```

Or

```
classA objet;
objet=new classA(varg1,varg2,...,vargn);
```

Example: (Person class with explicit constructor)

If the “Person” class has a constructor taking a name and an age as parameters, you can write:

```
Person p1, p2, p3; // Declaration of object p1, p2 and p3
p1=new Person ("DUBOIS Nathalie", 22); // Creation of object p1
```

```
P2=new Person ("MARTIN Alain", 24); // Creation of object p2
P3=new Person ("MOREAU Christophe", 32); //Creation of object p3
```

Or

```
Person p1=new Person ("DUBOIS Nathalie", 22);
Person p2=new Person ("MARTIN Alain", 24);
Person p3=new Person ("MOREAU Christophe", 32);
```

Example: (Person class without explicit constructor)

If the "Person" class does not define a constructor, you can write:

```
Person p1; // Declaration of object p1
p1=new Person (); // Creation of object p1
```

Or

```
Person p1=new Person ();
```

- **References (handles)**

  It is possible to identify each object by a name using an element called a reference (in English handle, literally handle). However, we cannot say that a handle is the name of an object, given that an object can have several handles (a handle can, however, only designate a single object). The concept of handles is very close to that of pointers in C/C++ language, however it is much simpler to understand and implement.

  To create an object reference, simply precede the name you give to an object with the name of the class you are instantiating, for example a reference to the object pers1, instance of the Person class, can be defined as follows:

  Personal person1;

  The pers1 reference does not currently point to any object. To do this, simply initialize it, that is to say in reality assign to it the result of the creation of the object returned by the new operator. With the previous example this could be written as follows:

  ```
  Person pers1= new Person();
  ```

  Note:

  Assigning the new operator to the reference does not mean that the object is stored in the reference; rather, it indicates that the reference points to the recently created object.

- **Access to members of an object**

  Access to the member data (attributes) of an object is done using the name of the reference to the object, followed by a period then the name of the attribute as follows:

  HandleName.AttributeName

  If the member data (attribute) is a reference to another object, its member data can be accessed through the current object as follows:

  HandleName.ObjectAttributeName.OtherObjectAttributeName

Access to an object's methods is done as for access to attributes, that is to say through a point. The method is followed by parentheses, containing the parameters, if any. Access to a method is therefore done as follows:

HandleName.MethodName(arg_1, arg_2,....)

Example:

```
class Person {
    String familyName;
    String firstName;
    int age;

    // Constructor of the Person class
    public Personne(String familyN, String firstN, int a) {
        familyName= familyN;
        firstName= fisrtN;
        age= a;
    }

    // Method displaying person information
    public void displayInfo() {
        System.out.println("FamilyName: " + familyName+ ", "+"FirstName: " +
firstName + ", "+"Age: " + age);
    }
}

// In the pricipal class
public class Main {
    public static void main(String[] args) {
        // Declaration and Creation of an object of Person type
        Person pers1 = new Person("PETIT", "Pierre", 26);
        Person pers2 = new Person("RICHARD", "Nathalie", 23);

        // Access to members of the pers1 object
        // Using the displayInfo method of the pers1 object
        System.out.println(pers1);
        System.out.print("pers1:");
        pers1.displayInfo(); // Displaying “Name: PETIT Pierre” and “Age: 26”
        // Access to members of the pers2 object
        // Using the displayInfo method of the pers2 object
         System.out.println(pers2);
        System.out.print("pers2:");
        pers2.displayInfo (); // Display "Name: RICHARD Nathalie" and "Age:
23"

        Personne pers3=pers2; // pers3 references the same object as pers2
        System.out.println(pers3);
        System.out.print("pers3:");
        pers3.displayInfo(); // Display "Name: RICHARD Nathalie" and "Ag: 23"

        System.out.println("Access to the attributes of the pers1 object");
        System.out.println(pers1.familyname); // Display "PETIT"
        System.out.println(pers1.firstname); // Display "Pierre"
        System.out.println(pers1.age); // Display 26
    }
}
```

The execution result is:

```
Personne@15db9742
pers1: Familyname: PETIT, FirstName: Pierre, Age: 26
Personne@6d06d69c
pers2: Familyname: RICHARD, FirstName: Nathalie, Age: 23
Personne@6d06d69c
pers3: Familyname: RICHARD, FirstName: Nathalie, Age: 23
Access to the attributes of the pers1 object
PETIT
Pierre
26
```

Note:

Person@15db9742 is the reference of the pers1 object, and Person@6d06d69c is the reference of the pers2 and pers3 objects.

c. Methods

A method is a function or procedure that is part of a class. It allows processing to be carried out on (or with) member data. A method can be a **static method** (also called a **class method**) or an **instance method**.

A **static method** belongs to a class and a **non-static method** (**instance method**) belongs to an object of a class. **Static methods** are useful if you only use the method once and don't need multiple objects. **Instance methods** are used if you are going to use your method to create multiple objects.

Example of a static method

```
public class StaticExample {
      public static void display(){
                System.out.println("Calling the static method");
              }

      public static void main(String[] args) {
      // There is no object created here because display() is a static method
            display();
      }
}
```

The execution result is:

```
Calling the static method
```

Example of an instance method

```
public class MyClass {

  public void display(){
                System.out.println("Calling an instance method");
              }

  public static void main(String[] args) {
   MyClass c = new MyClass();
  // The object is created here, because display() is a non-static (instance)
method
    c.display();
```

```
  }
}
```

The execution result is:

```
Calling an instance method
```

d. References and parameter passing

References and parameter passing are important concepts in programming, especially in object-oriented languages like Java. They are used to manipulate objects and pass data between different parts of a program.

In Java, objects are always manipulated by reference. This means that a variable that stores an object does not directly contain the object itself, but a reference to the memory location where that object is stored. Therefore, when a variable is passed as an argument (parameter) to a method, it is the object reference that is passed, not a copy of the object itself.

There are two types of parameter passing in Java: passing by value and passing by reference. By default, primitive types like int, double, char, boolean, etc. are used. are passed by value, that is, the value of the variable is copied and passed to the method. On the other hand, objects are passed by reference, which means that the object reference is passed to the method. To illustrate this, here is an example of Java code:

```
class Car {
    String color;

    public Car(String color) {
        this.color = color;
    }

    public void startEngine() {
        System.out.println("The engine started.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car MyCar = new Car("Red");
        changeColor(MyCar); // Pass the MyCar reference
        MyCar.startEngine(); // Use reference to access methods

        System.out.println("The color is: " + MyCar.color); //Use reference to
access variables
    }

    public static void changeColor(Car vtr) { // Pass reference by parameter
      vtr.color = "Gray"; // Directly modify the original variable
    }
}
```

The execution result is:

```
The engine started.
The color is: Gray
```

e. Inputs/Outputs

Output flow

Commonly used output instructions for displaying information in Java include: System.out.println(), System.out.print(), System.out.format(), and System.err.println(). The main difference between these instructions is that System.out.println() prints a line and System.out.print() prints on the same line, while System.out.format() allows you to format the display and System.err.println() prints an error message to the console.

Example output:

```
import java.util.Scanner;

public class Test {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);
		// TODO Auto-generated method stub
		System.out.println(2023);
		System.out.print(2024);
		System.out.printf("The score is: %d", 20); // Output according to
format
	}
}
```

The execution result is:

```
2023
2024 The score is: 20
```

- Input flow

The java entry must rely on the Scanner class:

```
import java.util.Scanner;
```

If input is required, first declare a Scanner object:

```
Scanner s = new Scanner(System.in);
```

Scanner is attached to the System.in input flow. After declaring the Scanner object, you must use the next() series of methods to specify the input type, such as input int, input string, input real, etc. , when typing.

The series of next() methods commonly used in Java include: e.nextInt() to read an integer, e.nextFloat() to read a float, e.nextDouble() to read a double, e.next() to read a character string without whitespace characters, e.nextLine() to read a character string including the whitespace character between words until the end of the line.

Example input:

```
import java.util.Scanner;
public class Test {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
    Scanner s = new Scanner(System.in); // Declare a Scanner object
    System.out.print("Please enter your name: ");
```

```
    String name = s.nextLine();
    System.out.println(name);

    System.out.print("Please enter your age: ");
    int age = s.nextInt();
    System.out.println(age);

    System.out.print("Please enter your weight: ");
    double weight = s.nextDouble();
    System.out.println(weight);
    System.out.print("Please enter the name of your university: ");
    String university = s.next();
    System.out.println(university);

    s.close();    // Close input flow and warn if not closed
    }
}
```

The execution result is:

```
Please enter your name: Lucas Hernandez
Luke Hernandez
Please enter your age: 21
21
Please enter your weight: 70
70.0
Please enter the name of your university: Harvard
Harvard
```

f. Destroyers

Constructors could be provided to allow the creation of objects, as we saw in the previous section. Similarly, when the object is no longer referenced and Java's garbage collector detects it, the object is destroyed. Therefore, the memory occupied by the object is freed and the object is deleted. In Java, the destructor is represented by the "finalize()" method which is called automatically by the garbage collector. Here is a code example that illustrates the destruction of an object in Java:

Example of object destruction

```
public class DestroyerExample {
    public static void main(String[] args) {
       Triangle p1 = new Triangle ();
        P1 = null; // Reference the object to null to make it eligible for
deletion
        System.gc(); // Manually call the garbage collector

        // The object will be destroyed and its destructor will be called
    }
}

class Triangle {
    @Override
    protected void finalize() throws Throwable {
        try {
            // Code to release resources used by the object
        } finally {
            super.finalize();
```

```
        }
    }
}
```

Through this example, we create a "Triangle" class with a "finalize()" method which is used when the object is deleted. We create an object "p1" in the "main()" method and then reference it to "null" to make it eligible for deletion. After that, we manually call the "System.gc()" method to ask the garbage collector to free the memory used by the object. Therefore, the object will be destroyed and its destructor "finalize()" will be called to free the resources used by the object.

Note:

It is important to note that in most cases it is not recommended to explicitly use the "finalize()" method in Java, as it has limitations and can cause performance issues.

g. Arrays

1. Definition

An array is a data structure containing a group of elements all of the same type. The type of elements can be a primitive type or a class.

2. Creating an array

From the point of view of its creation, an array behaves like an object. Also, the creation of an array takes place in two stages: the declaration of an array type variable, and the creation of the array itself. There are two ways to declare an array type variable, which are strictly equivalent:

Exemple. Declaration of an array

```
int [] arr1; // declaring an array that can contain integers
int arr2 []; // declaration equivalent to the previous one
```

Declaring this variable is like declaring an object. No memory space is reserved, other than that which allows storing this variable. Reserving memory to store an array requires providing the size of that array. Once this size is fixed, it is no longer possible to modify it. That said, the size of an array can be fixed at code execution, unlike C or C++.

Note:

Array names start with a lowercase letter and use uppercase letters to begin subsequent words.

Exemple:

Student [] studentsList;

Example. Creating an array, initializing in a for loop

```
int persNbr=8;  // initial size of the array

      Person [] Pers =  new Person [persNbr];  // creation of the array
```

```
for (int i=0; i<Pers.length;i++) {
      Pers[i] =new Personne();  // initialization of each box of the array
}
```

This example shows how to create an array, and one way to initialize it. Note the use of the public field (it is indeed a field, and not a method) length, available for any array, which gives the length of this array. This field is to be used whenever you want to browse the boxes in this array. Note that you can also browse arrays using the foreach syntax, starting with Java 5.

Example. Traversing an array in a foreach loop

```
for (Person Per : Pers) {
 Per = new Person() ;  // initialization of each box of the array
      }
```

Note:

Generally speaking, creating an array of objects does not initialize the objects it contains.

3. Initializing an array

It is possible to initialize an array with explicit values, whether that array contains base types or objects. Let's see this on an example.

Example. Initializing an array of base types

```
int arr1 [] = {5, 8, 12, 2, 7};  //we can also place the [] just after int

int [] arr2 ;
arr2 = new int [] {5, 8, 12, 2, 7};
```

The first syntax can only be used when declaring the array. The second can be used once the array has been declared. It is possible to use expressions between pairs of braces, as in the following example.

Example. Initializing an array of objects

```
Person Pers [] ;
       Pers =  new Person [] {
            new Person(),
            new Person ("Surcouf"),
            null
    };
```

4. Arrays of arrays

Multidimensional arrays are actually arrays of arrays, as in most languages. The syntax for creating such arrays is easily deduced:

Example. Declaration of a two-dimensional array

```
int [][]twodimArr;
twodimArr =  new  int [8][3] ;
```

The syntax for initializing a two-dimensional array is also deduced:

Example. Initializing a two-dimensional array

```
int twoDimArr [][] ;
twoDimArr =  new  int [][] {
```

```
      {8, 3, 7},
      {4, 12, 5},
      {0, 2, 1}
    } ;
```

It is also possible to initialize an array line by line.

Example. Initialisation d'un tableau bidimensionnel – bis

```
int twoDimArr [][] =  new  int [4][] ;

for (int i =  0 ; i < twoDimArr.length ; i++) {  // twoDimArr.length vaut 4
          arr[i] =  new  int [] {i, i+2, i+3} ;
}
```

Since twoDimArr is an array of arrays, twoDimArr.length is defined, as is twoDimArr[i].length, which in our example is 3 for values of i: 0, 1, 2.

5. Assignment between arrays

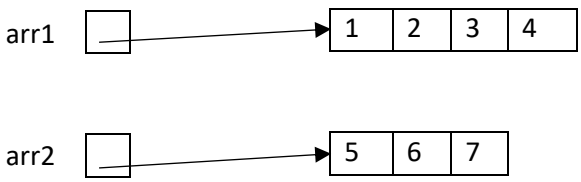
It is possible to assign an array type variable to another variable, provided that they are declared with the same data type. After assignment, the two variables refer to the same and only array in memory: they become synonymous. Any modification to an element of one modifies the same element of the other. Finally, by assigning array references, Java allows them to be manipulated globally.

Example:

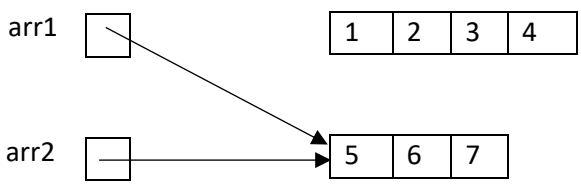
The following instructions can be used to create two arrays of integers by putting their references in arr1 and arr2:

```
int [] arr1 = new int[4];

for (int i=0; i<4; i++) {
      arr1[i] = i+1;}
int [] arr2 = new int[3];

for (int i=0; i<3; i++){
      arr2[i] = i+5;}
```

It is possible to represent the situation in the following way:



If we execute the assignment "arr1 = arr2" we end up with the following situation:



From now on, the two arrays arr1 and arr2 designate the same array. Thus, with: arr1[0] = 12; System.out.println(arr2[0]); We will obtain the display of the value 12.
Both arrays arr1 and arr2 will now refer to the same array in the future. Therefore, using the following code:

```
arr1[0] = 12 ;
System.out.println(arr2[0]);
```

The display result will be the value 12.

When the object that makes up the array of four integers, which was formerly named arr1, is no longer referenced elsewhere, Java's automatic garbage collection system considers it a candidate for garbage collection. It is thus possible to free the associated location, which can be used for something else, when there is no longer a reference to an object.

1.5. Exercises

Exercise 1:

Write a Java program to display the message “Hello friends”.

Exercise 2:

In this exercise, you will work with different types of variables and perform simple operations on them. The goal is to familiarize yourself with the declaration and use of integer, real and boolean variables, as well as division operations and the logical ‘’and’’ and ‘’or’’ operations.

a. Declare two integer variables e_1 and e_2, assign them the values of your choice and then display the result of the division between these two variables.

b. Declare two real variables of the double type, d_1 and d_2, assign them to the values of your choice and then display the result of the division between these two variables. Then, declare two other real variables of the float type, f_1 and f_2, affect them of the values of your choice and then display the result of the division between these two variables.

c. Declare two Boolean variables b_1 and b_2, assign them the values of your choice and then display the result of the logical operations (and, or) between these two variables.

N.B. Be sure to use meaningful variable names and add comments to explain each step of the code.

Exercise 3:

Write a Java program that behaves like a calculator, that is to say, executing a loop on:
1. Reading a line supposed to contain an operator, an integer, and another integer (example: + 11 20), the operators are +, -, *, /, %;
2. Calculation of the value of the expression;

3. Display of the result;
4. The program stops when it encounters one of the characters ''e'' or ''E''.

Exercise 4:

Write a Java program that reads a sequence of numbers. When encountering a number greater than 1000 and even, we stop reading and display the difference compared to 1000, i.e. D this difference. Then, display all the divisors of this integer (D), calculate the sum of all its divisors, and find the largest (except D), and the smallest (except 1) of the divisors of D.

Exercise 5:

Write a Java program that can read the color of a low beam (red, orange or green) and display the message to the arriving vehicle.

Exercise 6:

Considérons la classe **Student** suivante :

```
class Student {
    String name;
    int age;
    double average;

    Student(String n, int a, double m) {
        name = n;
        age = a;
        moyenne = m;
    }

    void displayInformation() {
        System.out.println("Name : " + name);
        System.out.println("Age : " + age);
        System.out.println("Average : " + average);
    }

    void increaseAverage(double value) {
        average += value;
    }
}
```

1. Write a Java program that creates two instances of the `Student` class with different names, ages, and averages.
2. Display each student's information (name, age, average).
3. Increase the average of the first student by 0.5.
4. Display the first student's information again after increasing their average.

Exercise 7:

1. Write the code for a class named “BankAccount” which offers three methods:
   - The first is a method called “deposit” to deposit a sum of money into the account.
   - The second is a method named “withdraw” to withdraw a sum of money from the account.
   - The third is a method named “ShowBalance” to display the final balance of the account.

2. Write the code for a BankAccountTest class which allows you to test the three methods of the BankAccount class by creating a bank account by making a deposit, making a withdrawal then displaying the final balance of the account.

Exercise 8:

In this exercise, we will work with an array of initialized integers:
int[] arr= {7, 2, 9, 2, 6, 10, 2, 8, 10, 13, 9, 2, 9, 20, 15};
- Write a Java program that calculates the number of occurrences of a given element in the array.

Exercise 9:

Write a Java program that reads the elements of an array of ten (10) cells and calculates the average and minimum of the array elements.

1.6. Answers to exercises

Answer to exercise 1:

```
package chapter1;
	class Exercise1 {
		public static void main(String args[]) {
			System.out.println("Hello friends");
		}
	}
```

The execution result is:

```
Hello friends
```

Answer to exercise 2:

```
package chapter1;

public class VariablesExercise {

    public static void main(String[] args) {

        // Question 1: Division between two integer variables

        int e1 = 15;

        int e2 = 4;

        int intDiv = e1 / e2;

        System.out.println("****Integer Division****");

        System.out.println("The result of the division between the integers " + e1
+ " and " + e2 + " is: " + intDiv + "\n");

        // Question 2: Division between two real variables (double)

        double d1 = 15.5;
```

```
        double d2 = 4.2;

        double doubleDiv = d1 / d2;

        System.out.println("****Real Division (double)****");

        System.out.println("The result of the division between the real numbers
(double) " + d1 + " and " + d2 + " is: " + doubleDiv + "\n");

        // Question 2: Division between two real variables (float)

        float f1 = 15.5f;

        float f2 = 4.2f;

        float floatDiv = f1 / f2;

        System.out.println("****Real Division (float)****");

        System.out.println("The result of the division between the real numbers
(float) " + f1 + " and " + f2 + " is: " + floatDiv + "\n");

        // Question 3: Logical operations on two boolean variables

        boolean b1 = true;
        boolean b2 = false;

        boolean andResult = b1 && b2;

        boolean orResult = b1 || b2;

        System.out.println("****Logical AND****");

        System.out.println("The result of the logical '&' operation between " + b1
+ " and " + b2 + " is: " + andResult + "\n");

        System.out.println("****Logical OR****");

        System.out.println("The result of the logical '|' operation between " + b1
+ " and " + b2 + " is: " + orResult);

    }

}
```

The execution result is:

```
****Integer Division****
The result of the division between the integers 15 and 4 is: 3

****Real Division (double)****
The result of the division between the real numbers (double) 15.5 and 4.2
is: 3.6904761904761902

****Real Division (float)****
The result of the division between the real numbers (float) 15.5 and 4.2
is: 3.6904764

****Logical AND****
The result of the logical '&' operation between true and false is: false
```

```
****Logical OR****
The result of the logical '|' operation between true and false is: true
```

Answer to exercise 3:

```
package chapter1;

    import java.util.Scanner;
    public class Exercise3 {

        /**
         * @param args the command line arguments
         */
        public static void main(String[] args) {
            // TODO code application logic here

            Scanner p1=new Scanner (System.in);
            System.out.println("Please enter an operator +, -, *, /,%");
            String op=p1.next();
            do{
            System.out.println("Please enter an integer");
            int x=p1.nextInt();
            System.out.println("Please enter another integer");
            int y=p1.nextInt();

            switch(op.charAt(0)){
                case '+' : System.out.println(x+" + "+y+" = "+ (x + y));break;
                case '-' : System.out.println(x+" - "+y+" = "+ (x - y));break;
                case '*' : System.out.println(x+" * "+y+" = "+ (x * y));break;
                case '/' : System.out.println(x+" / "+y+" = "+ (x / y));break;
                case '%' : System.out.println(x+" % "+y+" = "+ (x % y));break;
                default : System.err.println("ERROR: please enter an operator
    +, -, *, /,%");
            }

            System.out.println("To exit the program type 'e' or 'E'");
            System.out.println("Otherwise, please enter an operator +, -, *,
    /,%");
            op=p1.next();
            } while ((op.charAt(0)!='e')&& (op.charAt(0)!='E'));


        }

    }
```

The execution result is:

```
Please enter an operator +, -, *, /,%
+
Please enter an integer
8
Please enter another integer
2
8 + 2 = 10
To exit the program type 'e' or 'E'
System.out.println("Otherwise, please enter an operator +, -, *, /,%
```

Answer to exercise 4:

```
package chapter1;
    import java.util.Scanner;

    public class Exercise4 {

          public static void main(String[] args) {
                // TODO Auto-generated method stub

                Scanner Sc1=new Scanner(System.in);
                int x;

                do{
                      System.out.println("Please enter a number ");
                      x=Sc1.nextInt();

                }while(x<=1000 || x%2!=0);

                int D=x-1000;
                int sum=0, smallD = D, bigD=1;
                System.out.println("The difference from 1000 is: "+ D);

                System.out.println("The divisors of "+ D +" are as follows:");

                for (int i=1;i<=D/2;i++) {
                      if (D%i==0) {
                            System.out.println(i);
                            sum=sum+i;
                            if(smallD >i && i!=1) {
                                  smallD =i;
                            }

                            if(smallD <i && i!=D) {
                                  bigD=i;
                            }
                      }
                }
          sum+=D;
      System.out.println("The sum of the divisors of "+ D+" is: "+ sum);
      System.out.println("The smallest divisor of "+ D+" except 1 is: "+ smallD);
      System.out.println("The greatest divisor of "+ D+ " except "+D+" is:"+
      bigD);

          }
      }
```

The execution result is:

```
Please enter a number
4000
The difference from 1000 is: 3000
The divisors of 3000 are as follows:
1
2
3
4
5
6
```

```
8
10
12
15
20
24
25
30
40
50
60
75
100
120
125
150
200
250
300
375
500
600
750
1000
1500
The sum of the divisors of 3000 is: 9360
The smallest divisor of 3000 except 1 is: 2
The greatest divisor of 3000 except 3000 is:1500
```

Answer to exercise 5:

```
package chapter1;

import java.util.Scanner;
public class Main {
public static void main(String[] args) {
 Scanner in = new Scanner(System.in);
 System.out.println("Give the color of light:");
 String str=in.nextLine();
 String light = str.toLowerCase();

 switch(light) {
 case("green"):System.out.println("PASS ");break;
 case("red"): System.out.println(" STOP ");break;
 case("orange"): System.out.println("it's orange SLOW DOWN "); break;
 default : System.out.println("ERROR ");
 }

}
}
```

The execution result is:

```
Give the color of light:
red
 STOP
```

Answer to exercise 6:

```
package chapter1;

class Student {
```

```
    String name;
    int age;
    double average;

    Student(String n, int a, double m) {
        name = n;
        age = a;
        average = m;
    }

    void displayInformation() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("average: " + average);
    }

    void increaseAverage(double value) {
        average += value;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating student instances
        Student student1= new Student ("Alice", 20, 15.5);
        Student student2= new Student ("Bob", 22, 18.2);

        // Displaying student information
         System.out.println("Student 1 information:");
         student1.displayInformation();

         System.out.println("\nStudent 2 information:");
         student2.displayInformation();

         // Increase in the average of student 1
         student1.increaseAverage(0.5);

         // Displaying information for student 1 after increasing the average
         System.out.println("\nStudent 1 information after average increase:");
        student1.displayInformation();    }
}
```

The execution result is:

```
Student 1 information:
Name: Alice
Age: 20
average: 15.5

Student 2 information:
Name: Bob
Age: 22
average: 18.2

Student 1 information after average increase:
Name: Alice
Age: 20
average: 16.0
```

Answer to exercise 7:

```
package chapter1;

class BankAccount {
    double balance;

    public BankAccount () {
        this.balance = 0.0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= this.balance) {
            this.balance -= amount;
        } else {
            System.out.println("Insufficient balance.");
        }
    }

    public double ShowBalance() {
        return this.balance;
    }
}

public class TestBankAccount {
    public static void main(String[] args) {
        CompteBancaire cmt = new CompteBancaire();

        cmt.deposer(2000);
        System.out.println("Balance after deposit: " + cmt.ShowBalance() +" €");

        cmt.retirer(300);
        System.out.println("Balance after withdrawal: " + cmt.ShowBalance()+" €");
    }
}
```

The execution result is:

```
Balance after deposit: 2000.0 €
Balance after withdrawal: 1700.0 €
```

Answer to exercise 8:

```
package chapter1;

import java.util.Scanner;
public class Main {
public static void main(String[] args) {

int[] arr= {7, 2, 9, 2, 6, 10, 2, 8, 10,  13 ,9, 2, 9, 20, 15} ;

Scanner in = new Scanner(System.in);
System.out.println ("Give the desired value: ");
int val=in.nextInt();

int Nb_Occ=0;
```

```
for(int i=0;i<arr.length;i++){
 if (arr[i] == val)
 Nb_Occ++;
}
System.out.println("The number of occurrences of the value " +val+" is " +Nb_Occ);
in.close();
}
}
```

The execution result is:

```
Give the desired value:
2
The number of occurrences of the value 2 is 4
```

Answer to exercise 9:

```
package chapter1;

import java.util.Scanner;
public class Main {
public static void main(String[] args) {

int []arr = new int[10];
int min, sum=0;
double average;
Scanner in = new Scanner(System.in);

for(int i=0;i<arr.length;i++){
 System.out.println ("Enter the value of arr["+ i+"] : ");
 arr[i]=in.nextInt();
}

min =arr[1];
for(int i=0;i<arr.length;i++){
 sum =sum+arr[i];
 if (arr[i] < min)
 min= arr[i];
}

average=sum/arr.length;
System.out.println("The average is " + average);
System.out.println("The minimum is " +min);
in.close();
}
}
```

The execution result is:

```
Enter the value of arr[0] :
5
Enter the value of arr[1] :
4
Enter the value of arr[2] :
8
Enter the value of arr[3] :
6
Enter the value of arr[4] :
1
Enter the value of arr[5] :
19
Enter the value of arr[6] :
```

```
15
Enter the value of arr[7] :
30
Enter the value of arr[8] :
20
Enter the value of arr[9] :
14
The average is 12.0
The minimum is 1
```

Encapsulation

Chapter 2.

Outline	2.1. Visibility Levels 2.2. Encapsulation 2.3. Encapsulation in Java 2.4. Exercises 2.5. Answers to exercises
Goals	➢ Know the different visibility levels in Java (public, private, protected and package); ➢ Master the notion of encapsulation; ➢ Know the different types of accessors; ➢ Know when and how to use accessors.

2.1. Visibility levels

In the Java programming language, each attribute or method of a class can be of type public, protected, or private. These are indicated before the header of a class or method or before the type of an attribute to signify the type of visibility authorized for other classes. In the case where a field (attribute, method) is present in a class A:

- If it has **package** visibility, it can only be accessed inside the package where the class is located.
- If it is **public**, this indicates that it can be accessed from anywhere.
- If it is **private**, it means that it is only accessible inside its own class.
- If it is **protected**, this mode of protection will be explained later in the next chapter which will deal with inheritance.

2.2. Encapsulation

If we want to protect information against accidental (unexpected) changes, we must refer to the principle of encapsulation. We distinguish two (2) types of encapsulation in OOP:

a. Data encapsulation (attributes)

Encapsulation of data, also known as attributes encapsulation, means that the attributes of a class or object are hidden or protected from other parts of the code. This means that attributes are only accessible through public methods. This prevents other parts of the code from modifying the state of the attributes directly and guaranteeing data integrity.

b. Code encapsulation (messages)

Code encapsulation means that code is organized into classes and methods, which helps isolate implementation details from other parts of the code. This allows for easier reuse and

maintenance of parts of code. Methods encapsulate the specific behavior of an object and provide a clear interface for interacting with that object.

Example:

Consider an object model: the Animal class, which is an encapsulated class. This encapsulation ensures that attributes of the Animal class cannot be directly accessed or modified by an outside class, preventing any unintended or unexpected changes to the program's behavior.

```
public class Animal{
// Attributes
private String name;
private int age;
private String species;

// Method
public void initialization (String n, int age, String s){
this.name=n;
this.age=age;
this.species=s;
}

// Other methods
public void Display (){
	System.out.println ("Animal("+name+", "+age+" years old, "+ species+")");
}

public void MakesNoise (){
	System.out.println ("The animal makes some noises.");
}

}
```

Through this example, we note that the name, age, and species attributes are of a private type. To modify them we had to create the initialization method which has three entry parameters (String n, int age, String s). To display their values, we will have to call the display method (). Thanks to this, we have therefore acquired better control over data. Therefore, we prevented a situation where the direct modification of the three attributes which can unexpectedly modify the functioning of the instance of the Animal class.

2.3. Encapsulation in Java

a. Access control (public, private)

In object-oriented programming, access control is the use of special keywords to control which parts of the code can access specific members of a class. The public and private keywords define the scope respectively accessible and inaccessible for the members of the class. Public members can be accessible from the outside of the class and private members are not. Public members can be modified by external classes, while private members cannot be modified only inside the class. This gives developers greater control and greater security on the behavior of objects.

c. Access to the instance (this)

The keyword **this** designates, in a class, the current instance of the class itself. It can be used to make the code explicit and unambiguous. For example, if in a method, there is a parameter having the same name as an attribute of the class of which the method is part, we can explicitly designate the attribute using the keyword this.

Let us examine the code for the initialization method of the Animal class:

```
public void initialization (String n, int age, String s){
this.name=n;
this.age=age;
this.species=s;
}
```

In this example, the initialization method assigns the age parameter to the age attribute. The attribute was explicitly designated by the keyword **this**, designating the instance of the class, prefixed in the name of the attribute.

Consider the following **TestAnimal** call program:

```
public class AnimalTest{
public static void main(String[] args) {
	// TODO Auto-generated method stub

	Animal B1=new Animal();
	B1.initialization("Rabbit- Alfie", 2, "Angora dwarf");
	System.out.print("B1=");
	B1.Display();

}
}
```

The compilation and execution of the program below gives the following result:

```
B1 = Animal (Rabbit- Alfie, 2 years old, Angora dwarf)
```

Let's analyze how object B1 is initialized in the following example:

```
B1.initialization("Rabbit- Alfie", 2, "Angora dwarf");
```

In this example, we invoke the initialization method of object B1. When we use the ''this'' keyword in this method, we are referring to the B1 object itself. The method could also be rewritten as follows:

```
public void initialization (String n, int age, String s){
name=n;
this.age=age;
species=s;
}
```

Implicit use of the "this" keyword occurs when an object's method refers to its own x attribute using the this.x notation. We must use it explicitly when there is an identifier conflict. In the instruction: ''age = age'' the word ''age'' designates both an attribute of the current object and a parameter named "age" received by the initialization() method. In this case, in order to

remove the ambiguity, we must use ''this'' to explicitly denote the ''age'' attribute of the current object. This can be fixed by replacing the statement with: ''this.age = age''.

Now approaching the Animal class and examining the usefulness of the initialize() method, it is important to note that the name, age and species attributes are considered private data of the Animal class. Therefore, the following statements present in the TestAnimal class are unauthorized:

```
Animal B1;

B1.name="Rabbit- Alfie";
B1.age=2;
B1.species="Dwarf Angora";
```

To be able to set up the initial values of an Animal type object, we will use a publicly accessible method. The initialization() method plays precisely this role and we will implement it according to the following sequence:

```
Animal B1;
B1=new Animal();
B1.initialisation("Lapin- Alfie", 2, "Nain angora");
```

The notation **B1.initialization("Rabbit-Alfie", 2, "Angora dwarf")** is authorized because the initialization() method has public access and visibility.

c. Accessors (Get: Getters and Set: setters)

Although the variables (attributes) declared private type are only accessible in the same class, it is possible to access it if we provide public methods, namely the get and set method.
The get method is a function to return the value of the variable (attribute), while the set method is a procedure allowing to define the value of the attribute. The name of these two methods begins with get or set, followed by the name of the variable, with the first letter in capital letters.

Exemple:

```
public class Animal{

// Attributes
private String name;
private int age;
private String species;

// Constructor
public Animal(String n, int age, String s){
this.name=n;
this.age=age;
this.species=s;
}

// Getters
public String getName(){return name;}
public int getAge(){return age;}
public String getSpecies(){return species;}
```

```
// Setters
public void setName(String n){this.name=n;}
public void setAge(int age){this.age=age;}
public void setSpecies(String E){this.species=s;}
}
```

The **getName** method returns the value of the **name** variable, on the other hand the **setName** method takes a parameter (n) and affects it to the name variable. The keyword **this** is used to refer to the current object. However, as the **name** variable is declared as private, we cannot access it from the outside of this class:
If the variable was declared as public, we would touch on an output that will display the value of the **name** variable. However, when we try to access a private variable, we get an error. Instead, we use **getName ()** and **setName ()** methods to access and update the variable.

Note:

To make sure that sensitive data is hidden from users via encapsulation, you must: (1) Declare the variables / attributes of class as private, and (2) Providing get and set public methods to access and update the value of a private variable.

d. Variables and class methods (static)

Although Java language is an object -oriented language, there are cases where class instance is not essential (useless). The keyword **static** therefore allows a method of executing without having to install the class that contains it.
If a field (variables, methods, blocks and classes nested) is declared **static**, then a unique copy of this field is created and shared between all the instances of this class. Even if you create a million instances of the class or if you create none, there will always be a unique copy of the static field that belongs to it. The value of this **static** field will be shared by all the instances (by all objects of the same class or different class). Finally, the call to a method declared with the word **static** is made with the name of the class, and not with the name of the object. In Java, we can keep track of the number of objects that have been created in a class using a static variable.

Since a static variable is linked to a class and not to an object, it is perfect to create a static variable to follow the number of objects created. Here is an example of the Java code, which shows how to create a program that counts the number of objects created by the class animal.

Example:

```
public class Animal{
//Attributes
private String name;
private int age;
private String species;
private static long counter = 0;   // Number of animals created

//Constructor
public Animal(String n, int age, String s){
```

```
this.name=n;
this.age=age;
this.species=s;
System.out.println("Constructor call.");
counter++;
}

// Class method
public static long getNbAnimals(){
return counter;
}

public static void main(String[] args) {
            // TODO Auto-generated method stub
Animal a,b,c; // Declaration of objects a, b, c
a = new Animal("dog-percy",8," poodle"); // Creation of the object a
Animal e = a; // The object a and the object e share the same memory space
Animal f = new Animal("Chat-Ollie", 4," Siamese"); // Declaration and creation of
the object f
Animal m = new Animal("parrott-pongo", 12," cockatoles"); // Declaration and
creation of the object m

System.out.println();
System.out.println("Animal("+a.name+", "+a.age+" years old, "+a.species+")");
System.out.println("Animal("+e.name+", "+e.age+" years old, "+e.species+")");
System.out.println("Animal("+f.name+", "+f.age+" years old, "+f.species+")");
System.out.println("Animal("+m.name+", "+m.age+" years old, "+m.species+")");
System.out.println();

System.out.println("Number of animals created is: "+ Animal.getNbAnimals());

      }
}
```

We obtain the following result:

```
Constructor call.
Constructor call.
Constructor call.

Animal(dog-percy, 8 years old,  poodle)
Animal(dog-percy, 8 years old,  poodle)
Animal(Chat-Ollie, 4 years old,  Siamese)
Animal(parrott-pongo, 12 years old,  cockatoles)

Number of animals created is: 3
```

We initially put the "counter" variable which counts the number of animals created at 0. Then, in the constructor of the Animal class, we put this variable with an increasing operator so that it increases by 1 in the constructor. Since this constructor is called each time an object is instantiated, this code works. Each time an object is instantiated, this constructor (Animal) is called, the object is created and the static counter variable which retains the number of objects increases by 1.

Thus, each time we institute a new object, we can follow it thanks to a static property. Thus, we can count the number of instantial objects for a given class.

Note:

The instance variables are declared in a class, but apart from a method, a constructor or any block. Class variables, also called static variables, are declared with the keyword ''**static**'' in a class, but apart from a method, a constructor or a block.

2.4. Exercises

Exercise 1:

Write a Student class that represents a student with its attributes and behaviors. This class will be used to manage student information, respecting certain constraints.

Student Class Specifications:

1. Attributes:

- name (String): the student's full name.
- age (int): the student's age, which must be a positive integer.
- averageGrade (float): the student's average grade, which must be between 0 and 20 inclusive.

2. Methods:

- getName(): returns the student's name.
- getAge(): returns the student's age.
- getAverageGrade(): returns the student's average grade.
- setAverageGrade(float grade): Updates the student's average grade, ensuring it respects the specified limits (between 0 and 20).

In addition to the Student class, you must create a StudentTest class whose role will be to test the functionality of the Student class. This class will:

- Instantiate two objects of the Student class with various values.
- Test the methods of the Student class and display the results of each operation.
- Verify that the age and average grade constraints are respected when creating and updating objects.

Make sure to handle exceptions or potential errors related to invalid values. Remember to use clear comments in your code to explain each method and how it works.

Exercise 2:

Create a Point class to represent a point in a graph. A point has the following coordinates: an x abscissa and a y ordinate. The following operations must be possible:

1. Obtain the abscissa x of the point (getX).
2. Obtain the ordinate y of the point (getY).
3. Move the point to new coordinates (moveTo).

Write the code for a PointTest class that allows you to test the methods of the Point class by creating a Point. Make sure that the x and y coordinates are integer values and that the access methods do not allow changing the point coordinates directly.

Exercise 3:

Create a Circle class that represents a circle in a Cartesian plane. A circle is defined by a radius and a center, the latter being a point in the plane.

You must create the Circle class with attributes including a radius, which must be positive, and a center, which must be an object of the Point class. The methods to implement in the Circle class include getRadius(), which returns the radius of the circle, getCenter(), which returns the center, calculateCircumference(), which calculates and returns the circumference, calculateArea(), which calculates and returns the area, and moveTo(newX, newY), which moves the center of the circle to new coordinates.

In addition, you must create a CircleTest class to test the methods of the Circle class to validate its proper functioning. These tests must verify that the radius is positive, confirm that the center is a Point object with the correct coordinates, test the circumference calculation, verify the area calculation, and ensure that the circle moves correctly.

Exercise 4:

Considering a class representing a bank account named account. Any account is defined by a number and a balance.

1. Define the Account class with its constructor, its getters and setters.
2. We want to define three methods in the Account class as follows:

   a) A method called deposit: allowing to make deposits on an account, the deposit amount will be passed in parameter to the Depot method and this amount will be added to the balance of the account.
   b) A method named withdrawal: which will be exactly the opposite operation of the deposit, the amount past in parameter will be subtracted from the balance of the account
   c) A method called Display to display the current balance.

3. Define an executable class called TestAccount as follows:

   a. Give the code which allows to install two accounts: CP1 and CP2 as follows:
      - CP1: account number 11156765, balance at € 45,000.
      - CP2: account number 11167523, balance at € 20,000.

   b. Give the code that allows the following operations to be made:
      - A withdrawal of 5 000 € from the CP1 account and 7 000 € from the CP2 account.
      - A deposit of 8 000 € in the CP1 account and 12 000 € in the CP2 account.

- Display of the two accounts balance.

Exercise 5:

1. Write the code for a class named ''Calculation'' which offers two methods: a class method named "product" and an instance method named ''sum''.
   - The "product" class method takes two integers as parameters and returns the product of these two numbers.
   - The "sum" instance method takes no parameters and returns the sum of these two numbers.
2. Write the code of a TestCalcul class allowing you to test the two methods of the "Calculation" class by entering two numbers by calling the methods to calculate the product and the sum and displaying the calculated results.

2.5. Answers to exercises

Answer to exercise 1:

```
package chapter2;

class Student {
private String name;
private int age;
private float AverageGrade;;

// Constructor
public Student(String name, int age, float AverageGrade) {
this.name = name;
setAge(age);
setAverageGrade(AverageGrade);
}

// Getters
public String getName() {
return name;
}

public int getAge() {
return age;
}

public float getAverageGrade() {
return AverageGrade;
}

// Setters
public void setAge(int age) {
if (age > 0) {
this.age = age;
} else {
throw new IllegalArgumentException("Age must be positive.");
}
}

public void setAverageGrade(float grade) {
if (grade >= 0 && grade <= 20) {
this.AverageGrade = grade;
} else {
```

```
throw new IllegalArgumentException("The grade must be between 0 and 20.");
}
}
}

public class StudentTest {
public static void main(String[] args) {
// Instantiate students
Student student1 = new Student("Alice Smith", 20, 15.5f);
Student student2 = new Student("Bob Martin", 22, 18.0f);

// Test methods
System.out.println("Name: " + student1.getName());
System.out.println("Age: " + student1.getAge());

System.out.println("Average Grade: " + student1.getAverageGrade());

// Update Grade
student1.setAverageGrade(17.0f);

System.out.println("New Average Grade: " + student1.getAverageGrade());
}
}
```

The execution result is:

```
Name: Alice Smith
Age: 20
Average Grade: 15.5
New Average Grade: 17.0
```

Answer to exercise 2:

```
package chapter2;
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void moveTo(int newX, int newY) {
        x = newX;
        y = newY;
    }
}

public class PointTest {
    public static void main(String[] args) {
        Point point = new Point(2, 3);
```

```
        System.out.println("Abscissa x : " + point.getX());
        System.out.println("Ordinate y : " + point.getY());

        point.moveTo(5, 7);
        System.out.println("New Abscissa x : " + point.getX());
        System.out.println("New ordinate y : " + point.getY());
    }
}
```

The execution result is:

```
Abscissa x : 2
Ordinate y : 3
New Abscissa x : 5
New ordinate y : 7
```

Answer to exercise 3:

```
package chapter2;
//Point Class
class Point {
private double x;
private double y;

public Point(double x, double y) {
this.x = x;
this.y = y;
}

public double getX() {
return x;
}

public double getY() {
return y;
}

public void moveTo(double newX, double newY) {
this.x = newX;
this.y = newY;
}
}

//Circle Class
class Circle {
private double radius;
private Point center;

public Circle(double radius, Point center) {
if (radius <= 0) {
throw new IllegalArgumentException("Radius must be positive.");
}
this.radius = radius;
this.center = center;
}

public double getRadius() {
return radius;
}

public Point getCenter() {
```

```
return center;
}

public double calculateCircumference() {
return 2 * Math.PI * radius;
}

public double calculateArea() {
return Math.PI * radius * radius;
}

public void moveTo(double newX, double newY) {
center.moveTo(newX, newY);
}
}

//Test Class
public class CircleTest {
public static void main(String[] args) {
//Test the Point class
Point pointCenter = new Point(0, 0);
System.out.println("Center of circle: (" + pointCenter.getX() + ", " +
pointCenter.getY() + ")");

//Test the Circle class
Circle circle = new Circle(5, pointCenter);

//Check the radius
System.out.println("Radius of the circle: " + circle.getRadius());

//Check the center
System.out.println("Center coordinates: (" + circle.getCenter().getX() + ",
" + circle.getCenter().getY() + ")");

//Calculate the circumference
System.out.println("Circumference of the circle: " +
circle.calculateCircumference());

//Calculate the area
System.out.println("Area of the circle: " + circle.calculateArea());

//Move the center
circle.moveTo(2, 3);
System.out.println("New center of the circle: (" +
circle.getCenter().getX() + ", " + circle.getCenter().getY() + ")");
}
}
```

The execution result is:

```
Center of circle: (0.0, 0.0)
Radius of the circle: 5.0
Center coordinates: (0.0, 0.0)
Circumference of the circle: 31.41592653589793
Area of the circle: 78.53981633974483
New center of the circle: (2.0, 3.0)
```

Note:

Using the if-else statement to handle errors helps warn the user, but does not guarantee that exceptions will be explicitly thrown and caught. For more robust error handling, it is better to use specific exceptions such as IllegalArgumentException.

Answer to exercise 4:

```
package chapter2;

class Account {
    private long number;
    private double balance;
      public Account(long x, double y) {
             this.number=x;
             this.balance=y;
      }

      public long getNumber() {
             return this.number;
      }

      public double getBalance() {
             return this.balance;
      }

      public void setNumber(long x) {
             this.number=x;
      }

      public  void setBalance(double y) {
             this.balance=y;
      }

      public  void Deposit(double amount) {
             setBalance(getBalance()+amount);
      }

      public  void Withdrawal(double amount) {

         if (amount <= getBalance()) {
            setBalance(getBalance()-amount);
         } else {
            System.out.println("Insufficient balance.");
         }
      }

      public double Display() {
             return getBalance();}
}

public class TestAccount {

      public static void main(String[] args) {
             // TODO Auto-generated method stub


             Account Cp1= new Account (11156765, 45000);
             Account Cp2= new Account (11167523, 20000);
```

```
            Cp1.Withdrawal(5000);
            Cp2.Withdrawal(7000);

            Cp1.Deposit(8000);
            Cp2.Deposit(12000);

System.out.println(Cp1.Display()+" €");
            System.out.println(Cp2.Display()+" €");
                  }
}
```

The execution result is:

```
48000.0 €
25000.0 €
```

Answer to exercise 5:

```
package chapter2;
class Calculation {
        int a, b;
        Calculation (int a, int b) {
              this.a=a;
              this.b=b;

        }
        // “Product” class method
        public static int product (int a, int b) {
          return  a * b;
        }

        // Instance method "sum"
        public int sum() {
          return a + b;
        }
}
        public class TestCalculation {
        // Main method for testing methods
        public static void main(String[] args) {
          Calculation object = new Calculation (5,9);

          // Calling the “product” class method
          int resultProduct= Calculation.product(5, 9);
          System.out.println("Product result is:" + resultProduct);

          // Calling the "sum" instance method
          int resultSum = object.sum();
          System.out.println("The result of the sum is: " + resultSum);
        }
      }
```

The execution result is:

```
Product result is: 45
The result of the sum is: 14
```

Inheritance

Chapter 3.

Outline	3.1. Subclasses and inheritance 3.2. Single inheritance, multiple inheritance 3.3. Class hierarchy 3.4. Polymorphism 3.5. Inheritance and Polymorphism in Java 3.6. Exercises 3.7. Answers to exercises
Goals	➢ Know the different types of inheritance; ➢ Learn to do encapsulation in inheritance; ➢ Understand the difference between static and dynamic polymorphism; ➢ Know what an abstract class is; ➢ Learn Java syntax to define an interface.

3.1. Subclasses and inheritance

Subclasses and inheritance refer to how classes in an object can be related and inherited from each other. A subclass is a class that is derived (also called an inherited class, child class or child class) from an existing class, generally called a parent class, base class, parent class, original class, or superclass. A subclass inherits all the attributes and methods of the parent class and, in addition, defines its own attributes and methods. Inheritance is a mechanism for creating a child class from a parent class. This is done by creating a new class that is based on the properties and methods of a parent class. So, to define a new class, simply inherit from an existing class and add new properties and methods to it.

3.2. Single inheritance, multiple inheritance

In computer science, single inheritance is an object-oriented programming mechanism in which a class can only inherit behaviors and features from a single superclass. It is opposed to multiple inheritance, in which a class can inherit behaviors and features from more than one superclass. In Java, a class cannot be inherited from more than one superclass. Therefore, multiple inheritance is prohibited. However, a class can implement one or more interfaces, which allows multiple inheritance to be overridden.

Note:

The reason behind prohibiting multiple inheritance in java is to avoid ambiguity. Consider a case where the **Teacher** class inherits from the **Employee** class and the **Person** class. The **Employee**

and Person classes have the same ShowInfo() method. For the avoidance of doubt, multiple inheritance is not supported in Java.

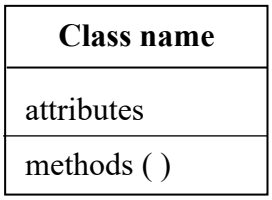
3.3. Class hierarchy

In Java, each object is an instance of a class, and each class is a member of a class hierarchy. This means that each class has a parent, and each parent has a parent class, and so on up to the top of the class hierarchy, which is usually the Object class.

A class corresponds to a description of a family of objects having the same data structure and the same behavior. It is symbolically represented by a rectangle divided into three (3) basic compartments, namely:

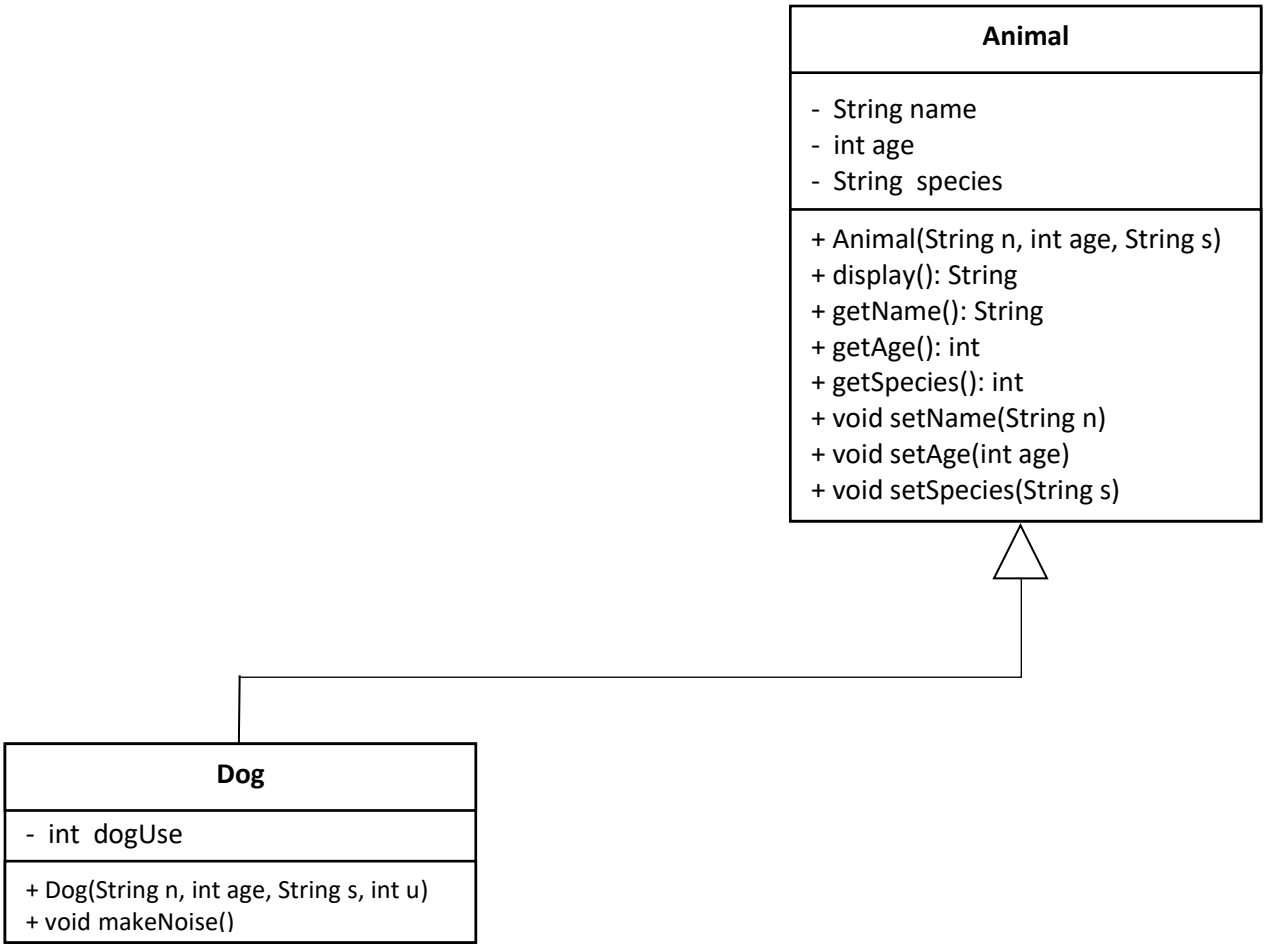
- Class Designation: This is the compartment that contains the class name, which must start with a capital letter.
- Description of attributes: This is the static part of the class, representing a set of attributes that can have a value.
- Description of methods: This is the dynamic part of the class, representing a set of operations or methods that define the common behavior of objects of the same class.

Graphical notation:



3.3.1. Class diagram

The class hierarchy of the example presented in subsection a of section 3.5 is symbolically represented in the form of a class diagram, as follows:



3.4. Polymorphism

The word polymorphism is a combination of two words, namely **poly** and **morphs**. The word **poly** means many and **morphs** means different shapes. In short, polymorphism is a mechanism by which we can perform the same action in different ways. Let's take a concrete example to understand what polymorphism is.

Example:

A **person** in a store is a **customer**, in an office he is an **employee**, at home he is a **husband/father/son**, at a party he is a **guest**. Consequently, the same person has different roles in different places. This is called polymorphism.

There are two types of polymorphism in Java:

- Static polymorphism (**method overloading**)
- Dynamic polymorphism (**method overriding**)

3.5. Inheritance and Polymorphism in Java

a. Single inheritance (extends)

Let's say we want to create a dog class which is a very specific animal. A dog has attributes that other animals may not have, such as dog use. But it also has attributes of any other animal, namely: name, age and species. A dog is therefore a full member of the Animal class, despite this it has additional attributes. Therefore, instead of writing a Dog class from scratch, we would

prefer to take the achievements of the Animal class which we would adapt to the particular character of dogs. It is the notion of inheritance that allows us to do this. In order to express that the Dog class inherits the properties and methods of the Animal class, we will write: **public class** Dog **extends** Animal
Animal is designated as the Base class (or superclass) and Dog as the derived class (or subclass). A dog object has all the qualities of an animal object: it has the same attributes and the same methods. These attributes and methods of the base class are not repeated in the definition of the child class. We just specify the attributes and methods added by the child class.
We assume that the Animal class is defined as follows:

```
public class Animal{
// Attributes
private String name;
private int age;
private String species;
// Constructor
public Animal(String n, int age, String s){
this.name=n;
this.age=age;
this.species=s;
}

public String display (){
      return "Animal("+name+", "+age+" years old, "+ species +")";
}
// Getters
public String getName(){return name;}
public int getAge(){return age;}
public String getSpecies(){return species;}

// Setters
public void setName(String n){this.name=N;}
public void setAge(int age){this.age=age;}
public void setSpecies(String s){this.species=s;}
}
```

The **Dog** class is defined as follows:

```
class Dog extends Animal{
// Attributes
private String dogUse; // long distance dog/military dog/assistance dog/sled dog
// Constructor
public Dog(String n, int age, String s, String u){
      super(n, age, s);
      this.dogUse=u;
}

      public void makeNoise(){
            System.out.println("bark!");
      }
}
```

Here the Dog class adds to the methods and attributes of the Animal class the following: (1) A dogUse attribute which is necessary to define the specific use for which the dog is prepared, such as a companion dog, a hunting dog, a guard dog, long-distance racing dog, military dog,

assistance dog, sled dog, etc., and (2) A new constructor allowing all the attributes of a dog to be initialized.

- Construction of an object from the Dog class

The constructor of the Dog class which is a specific method is defined as follows:

```
public Dog(String N, int age, String s, String u){
      super(n,age,s);
      this.dogUse=u;
}
```

The super(n,age,s) instruction allows you to call the constructor of the parent class here, the Animal class. So, we don't need to re-enter the parent attributes in the constructor of your Dog class since the constructor of the parent class already exists. We know that the constructor that super refers to takes three parameters. Consequently, super must take these parameters namely name, age, and species. If you don't put any parameters in it, super() will return the default constructor of the Animal class. This seems very convoluted and it would be better to write the following:

```
public Dog(String n, int age, String s, String u){
      this.name=n;
      this.age=age;
      this.species=E;
      this.dogUse=U;
}
```

However, this is not possible. The Animal class has declared its three fields namely name, age and species as private. Thus, only objects of the same class can directly access these fields. All other objects, including child objects like here, must go through methods declared public to have access to them. It would have been different if the Animal class had declared its three fields as protected. It then allowed derived classes to have direct access to the three fields. In this circumstance, using the parent class constructor was the correct solution and this is the usual method used during the construction of a child object. First, we call the constructor of the parent object and then we complete the initializations specific to the child object (dogUse in our example). The program testing the Dog class is as follows:

```
public class TestDog{
public static void main(String arg[]){
System.out.println(new     Dog("Dog-     Rex",3,"German     Shepherd",     "military
dog").display());
            }
            }
```

The above program makes it possible to create a dog object (new) and display it. The Dog class does not have any method called display(), but its parent Animal class has one which is also public: it becomes by inheritance a public method of the Dog class.

The execution result will be:

```
Animal(Dog- Rex, 3 years old, German Shepherd)
```

b. Encapsulation in inheritance

Encapsulation in inheritance refers to the ability to hide the implementation details of a class from its subclasses. This means that subclasses do not have direct access to the private variables and methods of a base class. However, the base class provides public methods for manipulating these variables and methods. Here is example code in Java for encapsulation in inheritance with the base class Animal and the derived class Cat:

Example:

```
class Animal{
// Encapsulated instance variables
private String name;
private int age;
private String type;
// Constructor
public Animal(String name, int age, String type){
this.name=name;
this.age=age;
this.type=type;
}

// Accessors for accessing encapsulated data
public String getName(){return name;}
public int getAge(){return age;}
public String getType(){return type;}

public void setName(String n){this.name=n;}
public void setAge(int age){this.age=age;}
public void setType(String type){this.type=type;}
// Method to display animal information
public void showInfo (){
	System.out.println("Name: "+name);
	System.out.println("Age: "+age);
	System.out.println("Type: "+type);
}
}

class Cat extends Animal{
	// Encapsulated instance variables
	private boolean clawsFolded;
	private String color;

	// Constructor
	public Cat(String name, int age, String type, boolean clawsFolded, String
color){
		super(name,age,type);
		this.clawsFolded=clawsFolded;
		this.color=color;
	}
	// Accessors for accessing encapsulated data
	public Boolean getClawsFolded(){return clawsFolded;}
	public String getCouleur(){return color;}

	public void setClawsFolded(boolean clawsFloded){this.clawsFolded =
clawsFloded;}
	public void setColor(String color){this.color= color;}
```

```
        // Method to display cat information
        public void showInfo () {
                System.out.println("Name: "+ getName());
                System.out.println("Age: "+ getAge());
                System.out.println("Type: "+getType());
                System.out.println("ClawsFolded: "+ clawsFolded);
                System.out.println("Color: "+getColor());
        }
}

public class TestCat {
            public static void main(String[] args) {
                Cat minouchette = new Cat("Minouchette", 4, " Domestic cat", true,
"Black");
                minouchette.showInfo ();
            }
        }
```

The results of the execution will be:

```
Name: Minouchette
Age: 4
Type: Domestic cat
Claws Folded: true
Color: Black
```

In this example, the **Animal** base class has three private attributes including name, age, and type with public accessors (getters and setters) to access these attributes. The **Cat** subclass inherits from the **Animal** class and adds two additional private attributes, namely clawsFolded, and color with corresponding accessors. This encapsulation helps control access to the base attributes of the **Animal** class while allowing subclasses like **Cat** to add additional attributes.

This approach is necessary because it isolates the implementation details from the base class. It also allows you to delegate responsibility to subclasses to implement their own variations of behavior while still using the functionality of the base class. It also helps maintain the code and makes it easier to manage future changes. Because changes to the base class have no direct impact on the subclasses.

i. Protection of members (protected)

An attribute or method declared **protected** in a class becomes accessible in the methods of its own class and in all subclasses. It also becomes accessible in all classes (and types) defined in the same package as the one that defines the protected member.
Here is a code example that demonstrates how to use the protected keyword to protect access to members of a class.

Example:

```
public class Person {
// Encapsulated instance variables
protected String name;
protected int age;

// Constructor
public Person (String name, int age){
```

```
this.name=name;
this.age=age;
}

// Encapsulated method
protected void sayHello (){
	System.out.println("Hello, my name is "+ name);
	}
}

public class Student extends Person{
	// Encapsulated instance variables
	private int level;

	// Constructor
	public Student (String name, int age, int level){
		super(name,age);
		this.level=level;
		}

	public void introduce (){
		System.out.println("I am a student in level "+ level +" rd
year");
	}

	public void greet (){
		System.out.println("I'm "+ age + " years old");

	}
}

// Example of use

public class TestStudent {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
	Student stud= new Student ("Pierre",21, 3);
	stud.sayHello();
	stud.greet();
	stud.introduce();

	/*The following code can be compiled because the name and age, which
are protected in the Person class,
	* also become accessible in the StudentTest class, defined in the
same package.
	*/

	 System.out.println(stud.name);
	 System.out.println(stud.age);

	}
}
```

The result of this program is:

```
Hello, my name is Pierre
I'm 21 years old
I am a student in level 3 rd year
Pierre
21
```

In this example, we have a **Person** class that has two protected attributes: name and age. This means that these members are only accessible from the Person class itself or by classes that inherit from the person class (like the **Student** class above). The **sayHello()** method is also protected. Which means it can only be called from the Person class or its subclasses.
The student class extends Person and adds a private attribute 'level' as well as two public methods: **introduce()** and **greet()**. In **greet()**, we use the **sayHello()** method of the Person class to greet using the person's name.
In the main code, we create an instance of the **Student** class and call its public methods. We cannot directly access the protected members of Nobody from the main code, which shows how the protected keyword can be used.

ii. Class constructors (this(), super())

In Java language, each instance of a subclass is subject to two special references, namely **this**, and **super()**: (1) the keyword **this** refers to the instance of the object on which the method is called. It can be used to access class members, including attributes and methods, and (2) the **super** keyword is a reference variable that is used to reference objects of the parent class. The **super** keyword is mainly used in the following contexts: (1) to reference an instance variable of the parent class, (2) to call the method of the immediate parent class, and (3) to call the constructor of the immediate parent class.

iii. ''Object'' Class

In this chapter, we have already seen that Java is a language that only supports single inheritance. The inheritance tree is a tree whose root is the Object class. If the developer does not specify a parent class in a class declaration, then the class implicitly inherits from the Object class.
The Object class provides methods common to all classes. Some of them need to be redefined in derived classes in order to work correctly.

Note:

Nothing prevents the creation of an Object class instance.
Object MyObj = **new** Object();

The Object class code is as follows:

```
public  class Object {
     public Object() {...}  // Contructor

     public String toString() {...}

     protected  native Object clone()  throws CloneNotSupportedException {...}

     public equals(java.lang.Object) {...}
     public  native  int hashCode() {...}

     protected  void finalize()  throws Throwable {...}

     public  final  native Class getClass() {...}
```

```
    // Methods used in thread management
    public  final  native  void notify() {...}
    public  final  native  void notifyAll() {...}

    public  final  void wait(long)  throws InterruptedException {...}
    public  final  void wait(long,  int)  throws InterruptedException {...}
}
```

The Object class includes several methods, including equals(), hashCode(), toString(), clone(), finalize(), and getClass(). One of the advantages of inheritance is that any object in a derived class can use the methods of the parent class. As a result, all Java objects can potentially use all methods of the Object class under certain conditions. Some of these methods are very specialized, while others are commonly used. Finally, some of these methods have a special status because they are used in a particular way by the Java machine. Each method has its own specific principle:

equals() method

In Java, the ''=='' operator is used to compare references. It should therefore never be used to compare objects. Object comparison is done using the equals method inherited from the Object class.

```
Vehicle v1 = new Car ("Bugatti");
Vehicle v2 = new Motorcycle ("Porsche");

if (v1.equals(v1)) {
  System.out.println("v1 is identical to itself.");
}

if (v1.equals(v2)) {
  System.out.println("v1 is the same as v2.");
}
```

In this example, we have two objects: v1 of type Car with the brand "Bugatti" and v2 of type Motorcycle with the brand "Porsche". We use the equals method to compare these objects. If v1.equals(v1), it means v1 is the same as itself. If v1.equals(v2), it means v1 is the same as v2.

The default implementation of the equals method provided by the Object class compares object references. In other words, it checks if the objects refer to the same location in memory. The default implementation of equals is therefore simply:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

However, in some cases it may be necessary to redefine the equals method to take into account specific comparison criteria. For example, if we add the registration attribute to our Vehicle class to identify vehicles, it makes sense to consider two vehicles equal if they have the same registration.

```
public class Vehicle {

  private String registration;
  private final String brand;
  public Vehicle (String registration, String brand) {
    this.registration= registration;
    this.brand= brand;
  }

  // ...

}
```

In this case it is necessary to redefine the equals method as follows:

```
public class Vehicle {

  private String registration;
  private final String brand;

  public Vehicle (String registration, String brand) {
    this.registration= registration;
    this.brand= brand;
  }

  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Vehicle)) {
      return false;
    }
    Vehicle vehicle = (Vehicle) obj;
    return this.registration != null &&
           this.registration.equals(vehicle.registration);

  }

  // ...
}
```

In the previous example, the Vehicle class was modified to include the registration attribute. The equals method has been redefined to compare the registration of two Vehicle objects. Using the instanceof operator allows you to check if the object parameter is compatible with the Vehicle type before comparing it. If the object is not of the correct type, the method returns false. Additionally, implementing the equals method ensures that two vehicles without registration are not considered identical.

Note:

The @Override annotation tells the compiler that a given method is an override of a method in the superclass.

hashCode() method

The `hashCode()` method of `Object` class in Java is used to generate a unique hash value for an object. This hash value is used in various data structures such as hash tables to speed up lookup operations. Here is an example showing the use of the `hashCode()` method:

```
public class Vehicle {
    private String brand;
    private int registration;
    public Vehicle(String brand, int registration) {
        this.brand = brand;
        this.registration = registration;
    }

    @Override
    public int hashCode() {
        int result = 17; // Choose an initial prime number
        result = 31 * result + brand.hashCode(); // Multiply by another prime
number and add the hash result of the brand
        result = 31 * result + registration; // Multiply again by a prime number
and add the registration
        return result;
    }

    public   static void   main   (String      args[]) {
      Vehicle v = new Vehicle("Mercedes Benz", 12345123);
        int hashCode = v.hashCode();
        System.out.println("HashCode: " + hashCode);
    }
}
```

The result of this program is:

```
HashCode: 1360372155
```

toString() method

The toString method is a very useful method, especially for debugging and logging. It allows you to obtain a representation in the form of a character string of an object. It is implicitly called by the compiler when you concatenate a character string with an object.

By default the implementation of the toString method in the Object class returns the type of the object followed by @ followed by the hash code of the object. Simply redefine this method to obtain the desired representation.

Example:

```
class Vehicle {
  private final String brand;

  public Vehicle(String brand) {
    this.brand= brand;
  }

  @Override
  public String toString() {
    return brand + " brand vehicle";
  }

  //...

}
public class Main {
    public static void main(String[] args) {
      Vehicle obj = new Vehicle("Ferrari");
```

```
        String msg = " Object created: " + obj;
        System.out.println(msg); // Object created: Ferrari brand vehicle

    }
}
```

The result of this program is:

```
Object created: Ferrari brand vehicle
```

getClass method

The getClass method provides access to the object representing the class of the instance. This means that a Java program can programmatically access the class definition of an instance. This method is particularly widely used in advanced uses involving reflexivity. The example below displays the full name (i.e. including its package) of the class of an object:

```
Vehicle obj = new Vehicle("Ferrari");
System.out.println(obj.getClass().getName());
```

clone() method

The `clone()` method of the `Object` class is used to create a shallow copy of an object. Here is an example to illustrate its use:

```
class Person implements Cloneable {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person pers1 = new Person("Françoise SIMON");


            // Using the clone() method to create a copy of the object
            Person pers2 = (Person) pers1.clone();

            System.out.println("person1 : " + pers1.getName());
            System.out.println("person2 : " + pers2.getName());
```

```
            pers2.setName("Philippe SMITH");

            System.out.println("person1 : " + pers1.getName());
            System.out.println("person2 : " + pers2.getName());
    }
}
```

The result of this program is:

```
person1 : Françoise SIMON
person2 : Françoise SIMON
person1 : Françoise SIMON
person2 : Philippe SMITH
```

finalize() method

The finalize method is called by the garbage collector before the object is deleted and memory reclaimed. Redefining this method therefore gives the developer the opportunity to trigger processing before the object disappears. However, we have already seen in Chapter 1 that the operation of the garbage collector gives no guarantee that this method will be called.

Competition methods

The Object class provides a set of methods that are used for exchanging signals in concurrent programming. These are the **notify()**, **notifyAll()** and **wait()** methods. Thread synchronization is performed using these methods. They allow a thread to wait until a certain condition is met before continuing execution.

- **notify()** wakes up a thread waiting on the object.
- **notifyAll()** wakes up all threads waiting on the object.
- **wait()** puts the current thread on hold until another thread calls it with notify() or notifyAll().

In managing concurrent threads, these methods are of great importance because they help avoid problems with race conditions and deadlocks. To ensure the security of concurrent access to shared resources, it is essential to use these methods carefully and in a "synchronized" block.

iv. Implicit and explicit casting

Type casting involves assigning a value from one primitive data type to another type. In Java, there are two types of casting:

- **Implicit casting to a “larger” type**

  Implicit casting is done automatically when changing from a smaller type to a larger type.

  Example:

```
public class ImplicitCasting {

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            int c=4;
            double d=c; // Implicit casting
```

```
                System.out.println("c= "+c);
                System.out.println("d= "+d);

        }
}
```

The result of this program is:

```
c= 4
d= 4.0
```

- Explicit casting to a "smaller" type

  Explicit casting must be done manually by placing the type in parentheses in front of the value.

  Example:

```
public class ExplicitCasting {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

                double d=4.85;
                int c=(int)d; // ExplicitCasting
                System.out.println("d= "+d);
                System.out.println("c= "+c);

        }

}
```

The result of this program is:

```
d= 4.85
c= 4
```

Note:

The only elementary type that cannot be cast is the Boolean type, while all other elementary types can.

Note:

The "**Integer.parseInt()**" method allows you to convert a string into an integer.

Example :

```
String zero = "0";
int n = Integer.parseInt(zero);
```

Class casting in Java allows you to convert an object from one class type to another compatible class type. This can be useful when you need to access specific methods or attributes of a child class from a parent class, for example. In Java, there are two common methods of converting between a parent class and a child class:

- Down casting (Explicit casting):

  Explicit casting, also called Down casting, occurs when you have an instance of the parent class and you want to treat it as an instance of the child class. This allows you to access specific methods and members of the child class. To cast down safely, you must

use the casting operator (childClassName) followed by the object you want to convert. Here is an example:

```
Animal animal = new Dog(); // Parent class Animal, object of the
daughter class Dog
Dog dog = (Dog) animal; // Down cast
```

In this example, we have a parent class "Animal" and a child class "Dog". We create an object of type "Dog" and assign it to a variable of type "Animal". Next, we cast down to treat the "animal" object as a "Dog" object. This allows us to access specific methods and members of the "Dog" class.

Note: If you have a variable of type ''Animal'' that actually contains an object of type ''Dog'', you can cast to convert it to ''Dog'' in order to access the specific methods and properties of the ''Dog'' class. However, the problem arises when the object in the ''Animal'' variable is not really a ''Dog'', but for example a ''Cat''. In this case, casting to ''Dog'' will not work and will throw an exception ''ClassCastException'' at runtime.
To avoid this exception, the best practice is to check the type of the object before casting, using the ''instanceof'' operator. Here is an example in pseudo-code:

```
if (animal instanceof Dog) {
      // At this stage, we know that the animal is indeed a Dog, we
can castrate it in complete safety
      Dog dog = (Dog) animal;
      // Use the dog object
      } else {
      // animal is not a Dog, we cannot cast
      System.out.println("Object is not a Dog");
      }
```

- Upward casting (Implicit casting):

  Implicit casting, also called upward casting, occurs when you assign an instance of a child class to a variable of type parent class without performing an explicit conversion. This happens automatically when you assign an instance of the child class to a variable of the parent class. Here is an example:

  ```
  Dog dog = new Dog(); // Girl class Dog
  animal animal = dog; // Implied casting (towards the top)
  ```

  In this example, we have a parent class ''Animal'' and a child class ''Dog''. We create an object of type ''Dog'' and assign it to a variable of type ''Animal''. As the "Dog" class inherits from the ''Animal'' class, the assignment is done automatically without requiring explicit casting.
  Implicit casting treats the object as an instance of the parent class, meaning you can only access methods and members of the parent class. If you want to access specific methods of the daughter class, you will need to cast down.

Here is an example to illustrate class casting in Java:

```
//Declaration of parent class
class Animal {
public void eat() {
    System.out.println("The animal is eating.");
}
}

//Declaration of a child class that inherits from the parent class
class Dog extends Animal {
public void bark() {
    System.out.println("The dog is barking.");
}
}

public class Main {
public static void main(String[] args) {
    // Creation of an object of the Dog class
    Dog myDog = new Dog();

    // Casting the myDog object to Animal type (implicit casting)
    Animal myAnimal = myDog;

    // Call the methods of the cast object
    myAnimal.eat();

    //myAnimal.bark(); <- This line would generate a compilation error because the
bark() method is not defined in the Animal class

    // Reciprocal casting of the myAnimal object to type Dog (explicit casting)
    Dog myDog2 = (Dog) myAnimal;

    // Call the bark() method of the Dog class
    myDog2.bark();

    // Call the eat() method of the Animal class
    myDog2.eat();
}
}
```

```
The result of this program is:
The animal is eating.
The dog is barking.
The animal is eating.
```

v. Limitation of inheritance

In Java language, the **final** keyword indicates that an element cannot be changed in the rest of the program. It can apply to the **methods** and **attributes** of a class and to the **class itself**. Likewise, it can be applied to the **parameters of a method** and to **local variables**. Depending on the context, **final** is used for design or optimization purposes.

- **Methods**

  A method indicated as **final** cannot be redefined in a child class.

  Exemple :

```
public class MaClass {
    public final void uneMethode() {
```

```
    System.out.println("I have a method marked as final");
}
   }
```

It can be used to force the behavior of a method in subclasses.

- Class or object attributes

  Applied to a primitive type attribute, the latter becomes a constant. The assignment must be made, at the latest, in the class constructor. The operation is a little different on an object type: it is the reference to the object which becomes constant and not its value. This last point also applies to tables which are also references.

  Example:

```
final class MaClass {
    public final double PI = 3.14159; // Unable to change value
    public final double[] arr = {90.4, 41.7, 30.36};

    public void uneMethode() {
        arr[1] = 25; // Works because tab is a reference to an array
    }
   }
```

  Note:

  Constants defined with the final keyword cannot change their values once they are initialized.

- Classes

  A class specified as final is a class that cannot be extended. This means that we cannot create a subclass from a class defined with the final keyword.

  Example:

```
public final class ExampleFinalClass {
    ...
   }
```

  This may be done for efficiency reasons or to prevent the user from making inappropriate use of it in the derived class. Many classes in the Java Standard Library are declared as final such as java.lang.System or java.lang.String.

- Method parameters

  The final keyword can also be used for the parameter of a method which we are sure will not change value inside the method. Here is an example of a Java method with a final parameter:

```
public void myMethod(final int myParameter) {
       // Here, myparameter cannot be modified
        System.out.println("Parameter value: " + myParameter);
```

```
        }
```

The operation is a little different on a parameter of an object type method: it is the reference to the object which becomes constant and not its value. Here is a Java example:

```
public class Person {
        private String name;
        public Person(String name) {
        this.name = name;
        }

        public void changeName(final Person p) {
         // Person cannot be reassigned to a new instance
        // p=new Person("New Name");//This would not be possible because p is
final

         // But its properties can modified
        p.name = "Nouveau Nom";
        }
       }
```

- Local variables

  Another use of the final keyword is in local variables. It follows the same rule, that is to say we are sure that the variable will not change.

  Example:

```
public static int add (final int x){
          final int y = 10;
          return x + y;
            }
```

c. Polymorphism

Polymorphism means that the same service, also called an operation or method, can behave differently depending on the situation. Two expressions of polymorphism can be distinguished:

i. Method overloading (static polymorphism)

In a given class, several methods can have the same name, but with different parameters (the signatures of the overloaded methods must necessarily be different) and can possibly have different visibility levels.

Take for example, class "A" below, having three (3) methods with the same name myMethod, they are overloaded in the class based on three different signatures:

```
public class A {

        protected void myMethod( ){
              ...
       }
       public void myMethod(a, b) {
```

```
            ...
        }
        private void myMethod(x, y, z){
             ...
        }
}
```

The first overload of myMethod is without parameters, the second has three (3) parameters, and the last finally has two (2) parameters. It is the language compiler who will have to choose to select the code for the correct method to use.

Example:

```
public class Dog{

	public void bark(){
		System.out.println("bark!");
	}

	public void bark(String msg){
		System.out.println("Woof, "+msg+" !"); // Overload
	}

	public void bark(int num){
		for(int i=0;i<num;i++) {
		System.out.print("Woof!"); // Overload
		}
	}
}

public class TestDog {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Dog x=new Dog();
		x.aboyer();

		Dog y=new Dog();
		y.bark("aouuuuu");

		Dog z=new Dog();
		z.bark(2);
	}

}
```

Here, we have an overload of the **bark()** method by the **bark(String msg)** method and the **bark(int num)** method.

The result of this program is:

```
Aboyer!
Woof, aouuuuu !
Woof!Woof!
```

ii. Method overriding (dynamic polymorphism)

B is considered a class derived from class A. Redefining a method of class A in class B means defining in B a method with the same name and the same parameters as a method contained in class A.

The override action provides a new definition of the method that will be called based on the type of the object at run time, not the type of the reference variable known at compile time.

Exemple:

```
public class Animal{

public void makeNoise (){
	System.out.println("Make noise");
}
}

public class Dog extends Animal{

	public void makeNoise (){
		System.out.println("Bark!"); // Overriding (redefinition)
	}
}

public class Cat extends Animal{


	public void makeNoise (){
		System.out.println("Meow!"); // Overriding (redefinition)
	}
}

public class TestAnimal {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Animal a=new Animal();
		a.makeNoise();

		Animal b=new Dog();
		b.makeNoise();

		Animal c=new Cat();
		c.makeNoise();

	}
}
```

The result of this program is:

```
Make noise
Bark!
Meow!
```

Here, the **makeNoise()** methods of the **Cat** and **Dog** class are a redefinition of the **makeNoise ()** method of the **Animal** class.

d. Abstract Classes

Abstract classes are classes that cannot be instantiated directly. However, they can serve as templates for other classes that inherit from these abstract classes. These may contain abstract methods that are not implemented in the abstract class itself. Derived classes have the flexibility to choose which abstract methods they want to implement based on their specific needs. It is not mandatory for derived classes to provide an implementation for all abstract methods of the abstract class. However, if a derived class does not provide an implementation for an abstract method, it must itself be declared as an abstract class. This allows derived classes to customize and specialize the behavior of the abstract class according to their individual needs.

Creating an abstract class is generally useful when multiple classes have a common part of logic, but with slight differences. By defining an abstract class as a template, it is possible to reuse code and avoid duplicating code across these classes.

Example:

```
public abstract class Staff {

	public abstract double grossSalary();  // Abstract method
	public abstract double NbrMonthsWorked();  // Abstract method

	public double leaveCompensation () {	// Defined method (non-abstract)
		return grossSalary() * NbrMonthsWorked()/12;
	}
}
```

An **abstract** method, indicated by the **abstract** modifier, has only one prototype, that is to say its return type, followed by its name, followed by the list of its parameters in parentheses, followed by a **semicolon**. Finally, an **abstract** method must necessarily be declared public.

Examples:

```
1. public abstract double NbrMonthsWorked (); // Abstract method
2. public abstract double grossSalary (); // Abstract method
3. public abstract void bonusEndYear (String msg); // Abstract method
```

Once a class contains an abstract method, it must also be declared abstract, with the abstract modifier placed at the start of its header. A class derived from an abstract class is not required to define all the abstract methods of its base class. Consequently, it remains simply abstract, and one must still mention abstract in its declaration. Consider the following **Frame** class which inherits from the abstract **Staff** class:

Example 1:

```
public class Frame extends Staff {

	public double grossSalary () {
		return 4000; // Value taken as an example
	}
	public double NbrMonthsWorked () {
		return 12;   // Value taken as an example
	}
```

```
	public double leaveCompensation () {// Defined method
		return grossSalary () * NbrMonthsWorked ()/12;
	}

	public String toString() {// Defined method
		return " The Frame salary "+ grossSalary () + ", number of months
worked " + NbrMonthsWorked ();
	}
}
```

Here, the Frame class defines the grossSalary() method, and the NbrMonthsWorked () method. Consequently, the Frame class is no longer abstract.

Exemple 2:

```
public abstract class Frame extends Staff { // abstract is required here

	public double grossSalary () {
		return 4000; // Value taken as an example
	}
	public abstract double NbrMonthsWorked ();// Undefined method

	public double leaveCompensation () { // Undefined method
		return grossSalary () * NbrMonthsWorked ()/12;
	}

	public String toString() {  // Undefined method
		return " The Frame salary "+ grossSalary () + ", number of months
worked " + NbrMonthsWorked ();
	}
}
```

Here, the Frame class defines the grossSalary() method, but not the NbrMonthsWorked() method. The Frame class remains abstract.

Note 1:

Abstract classes are classes that cannot be instantiated directly because they are not complete. They serve as templates for concrete classes, which means that they contain methods and fields that must be implemented or defined by a class that inherits from them. Since the abstract class is not complete, it makes no sense to instantiate it.

Note 2:

A class declared with the final keyword cannot contain abstract methods since it cannot be subclassed.

e. Interfaces

By definition, an interface is just an abstract class that can include constants and/or methods that have no body and must be implemented by the classes that implement that interface. Intrinsically, the methods of an interface are public and abstract (because we do not give a definition). However, it is not obligatory to mention the keywords ***public*** and ***abstract***. The

declaration of an interface resembles that of a class but, using the keyword **interface** instead of **class**:

```
public interface I {
	//List of constants
	//List of method prototypes
	}
```

Example:

```
public interface Human {

    void breath(); // breathe() method header (optional public abstract)
    void eat();  // eat() method header (optional public abstract)
    void sleep();  // sleep() method header (optional public abstract)
}
```

In Java language, we declare that a class implements an interface with the **implements** keyword. An interface defines a type (like a class) and the classes that implement this interface are therefore subtypes.

Example:

```
public class Boy implements Human {

    public void breath() {
       System.out.println("I breathe");
    }

    public void eat() {
       System.out.println("I'm eating");
    }


    public void sleep() {
       System.out.println("I sleep");
    }


}
```

Here, the **Boy** class necessarily defines the methods: **breathe()**, **eat()** and **sleep()** provided in the **Human** interface.

```
public class TestBoy {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Boy Jean=new Boy();
		Jean.breath();
		Jean.eat();
		Jean.sleep();

	}
}
```

```
The result of this program is:
I breathe
I'm eating
I sleep
```

A single class can implement an unlimited number of interfaces. However, each interface must be unique and offer unique methods and properties.

Example:

```
public interface Father {

	public static final int mark =100;
	int roll=121;
	 void display ();

}

public interface Mother {
	int mark1=200;
	 void add();
}

public class Son implements Father, Mother {

	int sum= Father.mark + Mother.mark1;

	public void display () {
		System.out.println("roll= "+ Father.roll);
	}

	public void add() {
		System.out.println("Total mark = "+ sum);
	}

}


public class TestSon {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Son Jean=new Son();
		Jean.display();
		Jean.add();
	}
}
```

The result of this program is:

```
roll= 120
Total mark= 300
```

Here, the Son class necessarily defines the display() method and the add() method provided in the Father interface and the Mother interface.

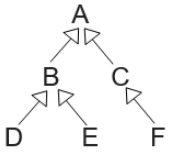
Note:

Although an interface can include attributes, these attributes must be defined as constants because they cannot be changed after they are initialized. We define attributes in an interface in the same way as variables in a class, except that their scope is public, static and final by default. This means that they can be accessed in other classes without instantiation and cannot be modified.

3.6. Exercises

Exercise 1:

Considering the following class graph, corresponding to an inheritance hierarchy, answer the following questions:



a) Which classes should define protected members (attributes and methods) and why?
b) Which classes can directly access protected members?
c) How would adding a new class G inheriting from B affect the answers to questions a) and b)?

Exercise 2 :

Consider the following program:

```
package chapter3;
class Book {
String title = "Unknown";
void display(String author)
{
System.out.println(title +
" by " + author);
}
}
class Novel extends Book {
String title = "Novel";
void display(String author)
{
super.display(title + " - "
+ author);
}
}
public class ScienceFiction
extends Novel {
String title = "Science
Fiction";
ScienceFiction(String
title) {
this.title = title;
}
void display(String author)
{
super.display(title + " - "
+ author);
```

Give the execution result of the program opposite:

.....
.....
.....

Explain very briefly the result obtained in terms of method calls:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

```
}
public static void
main(String[] args) {
Book book = new
ScienceFiction("Dune");
book.display("Frank
Herbert");
}
}
```

.....
.....
.....
.....
.....
.....
.....

Exercise 3:

For each code snippet below, provide the result of its execution, if applicable (explain the result). If an error occurs, justify it.

(a)

```
package chapter3;

class Animal {
Animal() {
System.out.print("X"); }
}

class Dog extends Animal {
public Dog(String name) {
System.out.print("Y"); }
public Dog() {
this("Rex");
System.out.print("Z"); }
}

public class AnimalTest {
public static void
main(String[] args) {
new Dog();
}
}
```

Final display:

.....

Explanation:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

(b)

```
package chapter3;
abstract class Fruit {
protected abstract void getFruitName();
public double getFruitPrice(int
quantity) { return quantity * 1.5; }
}

public class Apple extends Fruit {
protected void getFruitName() {
System.out.println("I am an Apple"); }
public static void main(String args[])
{
Fruit f = new Apple();
f.getFruitName();
System.out.println(f.getFruitPrice(3));
}
}
```

Final display:

.....

Explanation:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Exercise 4:

In modern aquaculture, fish farm management is a rapidly expanding sector, requiring a rigorous and methodical approach. Farmers must be able to monitor and evaluate the growth of their fish while adapting to market fluctuations. This exercise aims to develop a simple application to manage a fish farm, with an emphasis on creating classes representing the different fish species.

Required tasks:

a. Write an abstract class "Fish." In aquaculture, fish are raised for their meat or eggs. Each fish is characterized by an "ID number" and a "weight." Add a constructor and a "ChangeWeight" method to this class to change its weight. Also write two abstract methods: one to return its selling price per kilogram, and the other to determine whether the fish has reached the required weight for harvesting. These criteria can vary depending on the type of fish (salmon, trout, tilapia, etc.).

b. Write a "Salmon" class that inherits from the "Fish" class. All farmed salmon have a standard selling price per kilo and a standard harvest weight. However, due to market fluctuations, these two values can be adjusted. Include attributes for the salmon variety (e.g., Atlantic salmon, King salmon).

c. Write a "Main" class that will serve as an entry point for testing the functionality of the "Salmon" class. This class will allow you to instantiate Salmon objects, modify their weight, change their variety, and display their selling price based on the defined parameters.

Exercise 5:

Consider a video game streaming platform with a “GameDeveloper” class representing a video game developer. The "GameDeveloper" class has the following attributes:

- "DeveloperID": a character string attribute which represents the developer's ID and which is only accessible within its class.
- "games": an attribute of type array of strings (String[]) which represents the games developed by the developer and which is accessible only within its class.

1. Create the “GameDeveloper” class.
2. Write accessors and mutators for both attributes.
3. Define a constructor with two parameters for the “GameDeveloper” class (String DeveloperID, String[] games).
4. Write the “calculateNumberGames” method which returns the number of games developed by an object of type “GameDeveloper”.
5. Write the "displayGames" method which displays the list of games developed by an object of type "GameDeveloper".

Now consider a class called ‘’StudioDeGeux’’ representing a video game development studio, with the following attributes:

- ‘’Studioname’’: a private string type attribute which represents the name of the studio.
- ‘’developers’’: an array of objects of type ‘’GameDeveloper’’ which represents the studio's developers.

6. Create the "GameStudio" class, write accessors and mutators for its attributes, and define the ‘’GameStudio(String StudioName,GameDeveloper[] developers)’’ constructor.
7. Write a ‘’calculateTotalNumberGames’’ method that returns the total number of games developed by all developers in the studio.
8. Write a ‘’showGamesByDeveloper’’ method that displays the list of games for each developer in the studio.
9. Write a main class to test the ‘’showGamesByDeveloper’’ and ‘’calculateTotalGames’’ methods by creating three objects of the ‘’GameDeveloper’’ class and assigning them to the same game studio (for example, studio ‘’StudioX’’).

Exercise 6:

Consider the two Java classes “Vehicle” and “Compound_Car”. The attributes of the “Vehicle” class are:

- engine: a Boolean type attribute.
- maxSpeed: an attribute of integer type.

The attributes of the “Compound_Car” class are:

- numberOfDoors: an attribute of integer type.
- ve: an object type attribute of the “Vehicle” class.

1. Create the two classes “Vehicle” and “Compound_Car” in two separate files. Don't forget to generate the getters/setters and the constructor(s) (give the default constructor(s) and those with parameters knowing that the default values are "false" for engine, ‘’0’’ for maxSpeed, “2’’ for numberOfDoors, and ‘’null’’ for ve).

2. In the “Vehicle” class, write the Vehiclecharacteristics() method to return the characteristics of the vehicle.
3. In the “Vehicle” class, write the maxSpeed() method to display the maximum speed of the vehicle.
4. In the “Compound_Car” class, write the CompoundVehicle() method to return the characteristics of the Composite Car.
5. Create a third class “Drift_Car” deriving from the class “Vehicle” having a single added attribute named numberOfDoors which is an attribute of integer type. Generate the getter/setter for the added attribute and the constructor allowing all the attributes of this class to be initialized (give the default constructor and the one with parameters knowing that the default values are “false” for engine, “0” for maxSpeed, “2” for numberOfDoors).
6. In the “Drift_Car” class, write the DriftVehicleCharacteristics() method to return the characteristics of the Drifted Car.
7. In the main method of the Main main class, carry out exhaustive tests to verify the correct functioning of all the methods implemented in the "Vehicle", "Compound_Car" and " Drift _Car" classes. Make sure to include different test scenarios to cover all possible cases, and display the results in a clear and understandable way for easy verification.

Exercise 7:

Considering a “Score” class with the following attributes:

- moduleName visible to the Note class and its derived class;
- scorePW visible to the Note class and its subclass;
- scoreExam visible to the Note class and its child class;

1. Create the “Score” class with its constructor assigning values to the moduleName, scorePW, and scoreExam attributes.

2. In the “Score” class, write the average () method to calculate and return the average of an evaluation as follows: average = (scorePW × 0.4) + (scoreExamen ×0.6).

3. Create a second “Evaluation” class deriving from the “Score” class having a single added attribute named evaluation year which is an integer type attribute visible only in the “Evaluation” class. Generate the getter/setter (of the added attribute) and the constructor to initialize all the attributes of this class.

4. In the “Evaluation” class, write two instance methods as follows:
The first method named deliberate(): is a method that displays ‘’Admitted’’ if the average of the evaluation is greater than or equal to 10, ‘’Adjourned’’ otherwise.
The second method named compareScore(Evaluation ev): is a method for comparing the PW score of two assessments as follows:

If the two Assessments are from the same module: the compareScore(Evaluation ev) method returns the best TP score between the two Assessments.

If the two Evaluations are not of the same module: the compareScore(Evaluation ev) method returns 'Please note, these two Evaluations do not concern the same module'.

5. Create an executable class "TestEvaluation", then provide the code that instantiates three evaluations, namely ev_1 ev_2 and ev_3 as follows:

- ev_1: $module_1$ OOP, $scorePW_1$ 14, $scoreExam_1$ 12, year of $assessment_1$ 2012.
- ev_2: $module_2$ OOP, $scorePW_2$ 9, $scoreExam_2$ 10.5, year of $assessment_2$ 2013.
- ev_3: $module_3$ Network, $scorePW_3$ 10, $scoreExam_3$ 11.75, year of $assessment_3$ 2012.

➢ Give the code that allows you to do the following operations:

a) Calculation of the average of the three evaluations ev_1, ev_2, and ev_3.
b) Deliberation for the three evaluations ev_1, ev_2, and ev_3.

Comparison of PW scores between $scorePW_1$ and $scorePW_2$, then between $scorePW_2$ and $scorePW_3$.

Exercise 8:

A fleet consists of cars and trucks that have common characteristics grouped into the "Vehicle" class.

1. Write an abstract class "Vehicle". Each vehicle is characterized by its registration number, model year, maximum speed, and price. When creating a vehicle, its number is incremented according to the number of vehicles created. All attributes of the "Vehicle" class are assumed to be accessible only in their own class. This forces the creation of getters and setters.

2. In the "Vehicle" class, write the method characteristicVehicle() which returns a character string which contains the characteristics of the vehicle, namely the registration number, the model year and the price.

3. In the "Vehicle" class, write the MaxSpeed() method which displays the maximum speed of the vehicle.

4. The "Vehicle" class also has three abstract methods namely start() method, accelerate() method and loadCargo() method which can be defined in derived classes and will display custom messages.

5. Write the "Car" and "Truck" classes which extend the "Vehicle" class. The "Car" class has a single added attribute named numberOfDoors which is an attribute visible only in the definition class. The "Truck" class also has a single added attribute called cargo which is an attribute visible only in the "Truck" class.

6. Concretely define the accelerate() and start() methods in derived classes, displaying personalized messages.

7. In the "Truck" class, implement the loadCargaison() method to display the nature of cargo transported by a truck.

8. In the "Car" class, redefine the Vehiclecharacteristic() method to return the characteristics of a car.

9. In the main method of the Main class, test all the methods performed in the previous questions.

Exercise 9:

Consider a class representing a material object, which is intended to satisfy a need in a market, called "Product". Any product is defined by a reference, a Stock quantity, and a price.

1. Write the abstract class "Product" by applying instance variable encapsulation, then define the constructors (give the default constructor and the one with parameters knowing that the default values are "null" for reference, "0" for Stock quantity, and "0.0" for price).

2. In the "Product" class, write an abstract productCharacteristic() method, which returns the characteristics of a product.

3. Considering two classes derived from the "Product" class, the "Computer" and "Smartphone" classes which inherit from the "Product" class as follows:
   - The "Computer" class has specific attributes: Ram and Hard disk.
   - The "Smartphone" class has specific attributes: Resolution and nbChips.

4. A. Write the two classes "Computer" and "Smartphone", with their constructors, their necessary getters and setters.

5. B. We want to have a goodDeal () method that informs us whether a Smartphone is worth buying or not, as follows:
   - If the resolution of the Smartphone is greater than 50 Mpx and its price is less than 500 euros, the goodDeal () method must return "Buy it, it's worth it".
   - Otherwise the goodDeal () method must return "Avoid buying, it's a scam".
   - Give the code for the Bonne Affaire () method. In which class should we declare this method?

   C. In the "Smartphone" class, implement the productCharacteristic() method which returns the characteristics of a Smartphone.

4. In the main method of the Main class, test all the methods performed in the previous questions.

Exercise 10:

A tennis tournament is a competition in which several players compete to determine the champion. Each player plays several matches with other players. Tournament rules determine how players earn points:

- A player does not accumulate any points if he loses a match.
- Two points are accumulated by a player who wins a match.
- A draw, in which both players have the same number of sets won, means that both players have accumulated one point. At the end of all matches, the player who has collected the greatest number of points will be declared champion.

We want to set up a tracking program for a tennis tournament, which will require the creation of two classes, namely the TennisMatch class and the Tournament class.

1. Create the TennisMatch class which models a match between players A and B. The following members characterize this class:

- playerA, playerB of character string type representing the names of the two players;
- setsGagnesA, setsGagnesB, two integers which respectively represent the number of sets won by playerA and playerB;
- a constructor with two character string parameters, namely the name of player A and that of player B;
- the necessary getter methods (no setter);
- the markSet(Player String) method: two possible values for player: “A” to add a set won for player A and “B” to add a set won for player B.

2. Create the Tournament class which will be distinguished by the following characteristics:

- tabMatches: array of TennisMatch type objects;
- nbrMatchs: the number of matches in tabMatches;
- tabPlayersPts: two-dimensional array of String type which contains in each line the name of a player and the number of points of this player in the tournament.

Example:

Hint	Player Name	Number of points
0	Federer	9
1	Nadal	6
2	Djokovic	3
3	Thiem	0

The Tournament class will define the following methods:

- a constructor without parameters which initializes the tabMatches table (40 matches maximum) and tabJoueursPts (10 players maximum);
- a void method addMatch(TennisMatch mt): adds a match object to tabMatchs. This method should handle the maximum size of the tabMatches array;
- a void listPlayers() method: which fills the first column of the tabPlayersPts table with the players participating in the tournament without repetition;
- an int pointsPerPlayer(String player) method: which, for a given player, calculates the number of points accumulated in the tournament (following the rules cited at the start of the exercise);
- a void method calculateNumberPoints(): which calls the pointsParJoueur method to fill the second column of the tabJoueursPts table;

- a toString() method which returns a character string representing the table tabJoueursPts once filled.

Example:

Player Name	Points
Federer	9
Nadal	6
Djokovic	3
Thiem	0

3. Write a main class to test the proposed solution. The following elements must be arranged in the main method:

a. A Tournament object that starts out empty.
b. Six (6) TennisMatch objects between the four players: "Federer", "Nadal", "Djokovic" and "Thiem", with the scores (result of the six matches making up the tournament) in the following table:

Match	Player A	Player B	Sets won by A	Sets won by B
1	Federer	Nadal	2	0
2	Djokovic	Nadal	1	1
3	Federer	Djokovic	2	0
4	Thiem	Nadal	0	2
5	Federer	Thiem	1	1
6	Djokovic	Thiem	2	0

c. Displaying the points table of players who participated in the tournament.

Exercise 11:

Create a Shape interface that has the area() and circumference() methods. This interface must be implemented by the Rectangle and Circle classes.
The Rectangle class must have the length and width attributes and the area() and circumference () methods which return the area and perimeter of the rectangle, respectively. The Circle class must have the radius attribute and the area() and perimeter() methods which return the area and circumference of the circle, respectively.

Then create a TestShape class that creates an object of each class (a rectangle and a circle) and displays their areas and perimeters.

Exercise 12:

Create an Animal interface that has the sleep() and eat() methods. This interface must be implemented by the Mammal and Bird classes.
The Mammal class must have the name and age attributes and the sleep() and eat() methods which display messages saying that the animal is sleeping or eating. The Bird class must have the name and habitat attributes and the sleep() and eat() methods which display messages saying that the bird is sleeping or eating.

Then create a Zoo class that creates an array of 5 animals (2 mammals and 3 birds), stores them in an array and calls the sleep() and eat() method on each of them.

Exercise 13:

The following program contains errors. Find these errors and correct them:

```
package Chapter3;

1.    class A {
2.    private int a;
3.    A(int a) {  a = a; }
4.    private int setA(int value) { a = value;  }
5.    private void getA() {  return a;   }
6.    }
7.    class B extends A {
8.    protected int b;
9.    B(int b) {
10.
11.   this.b = b; }}
12.
13.   class C extends A {
14.   private Final double k;
15.   C() {
16.
17.   k = 1; }
18.   private double getC() {
19.   return k;    }}
20.   class D extends A {
21.   protected Final double d=5;
22.
23.   protected int operation(int a) { d= a * 2;
24.   return d;    }
25.   protected int calculation (int b) { return b + 1;   }
26.   protected void display () {
27.   System.out.println("Method display in D"); }
28.
29.   public class test {
30.   public Static void main(string[] arg) {
31.   A x = new A(8);
32.   B y = new B(2);
33.   C z = new C(6);
34.   x.a=2;
35.   y.b = 2;
36.   z.k = 3; 37.      }
37.
```

N.B. indicate the line number where the error is located and provide a corresponding explanation.

Exercise 14:

Consider the following Java program, which uses classes, interfaces, and method overriding:

```
package Chapter3;
interface J {
        static int k = 3;
        int n() ;}

class A {                    class B extends A implements J    class C implements J {
static int e = 2;            {                                 static int e = 9;
```

```
public static int m () {
return (e+4) ; }
public static int s (int x) {
return (e+x);}
public int w (int x) {
return (x+1) ; }
}
```

```
static int e = 6;
public static int m () { return
(e+3) ; }
public int w (int r) { return (e+2);
}
public int n() { return k+4;}
}
```

```
public int n(){ return
k+5;}
}
```

Question 1: What will running the following code produce?

```
public class Test{
public static void main (String [] args){
A p=new B() ;
System.out.println (p.w(1));
System.out.println (p.m());
System.out.println (p.s(2));
}}
```

Result:

Question 2: What will running the following code produce?

```
public class Test{
public static void main (String [] args){
B y = new B() ;
J q = new C();
System.out.println (y.n());
System.out.println (q.n());
}}
```

Result:

Question 3: Fix errors in the following code:

```
1. public class Test{
2. //Do not change the type of variables
3. public static void main (String args []){
4. A d = new J () ;
5. System.out.println (d.n());
6. J [] tab = [new A(), new C(), new B()];
```

Correction + Resul:

Result :

Question 4 : What happens if we add the following line at the start of class B?

```
private int k = 10;
```

3.7. Answers to exercises

Answer to exercise 1:

a) The classes that should define protected members are A, B, and C. In general, higher-level classes in the inheritance hierarchy should define protected members so that their subclasses can access them while hiding them from unrelated classes.
In this case, class A should define protected members because it is the top-level class and its subclasses B and C will need to access them. Similarly, classes B and C should

define their own protected members because their subclasses (D, E, and F) will need to access them.

b) The classes that can directly access protected members are B, C, D, and E. Since B and C inherit directly from A, they can access the protected members defined in A. Similarly, D and E inherit directly from B, so they can access the protected members defined in B.

c) If a new class G inheriting from B is added, the answers to questions a) and b) would be as follows:

   a. The classes that should define protected members remain the same: A, B, and C. Class G does not need to define its own protected members because it already inherits from those defined in B.

   b. The classes that can directly access protected members expand to include G. Since G inherits from B, it can access the protected members defined in B.

So, the addition of class G does not affect the answers regarding the definition of protected members, but it expands the number of classes that can directly access them.

Answer to exercise 2:

Consider the following program:

```
package chapter3;
class Book {
String title = "Unknown";
void display(String author) {
System.out.println(title + " by "
+ author);
}
}
class Novel extends Book {
String title = "Novel";
void display(String author) {
super.display(title + " - " +
author);
}
}
public class ScienceFiction
extends Novel {
String title = "Science Fiction";
ScienceFiction(String title) {
this.title = title;
}
void display(String author) {
super.display(title + " - " +
author);
}
public static void main(String[]
args) {
Book book = new
ScienceFiction("Dune");
book.display("Frank Herbert");
}
}
```

Give the execution result of the program opposite:

Unknown by Novel - Dune - Frank Herbert

Briefly explain the result in terms of method calls:

1. Object Creation: The **book** object is created using **new ScienceFiction("Dune")**. This calls the **ScienceFiction** constructor, which assigns "**Dune**" to the **title** attribute of **ScienceFiction**.

2. Method Call: When **book.display("Frank Herbert")** is called, the ScienceFiction **display** method is executed.

3. Super Call: In the ScienceFiction **display** method, **super.display(title + " - " + author)** is called. Here, **title** refers to the **title** attribute of **ScienceFiction**, which is "**Dune**". This

	passes "**Dune - Frank Herbert**" to the **Novel display** method. 4. **Novel** Method Call: In the Novel **display method**, **super.display(title + " - " + author)** is called. Here, **title** refers to the **title** attribute of **Novel**, which is "**Novel**". This calls Book **display** with the argument "**Novel - Dune - Frank Herbert**". 5. Final display: In the Book display method, **System.out.println(title + " by " + author)** is executed, where title refers to the **title** attribute of **Book**, which is "**Unknown**". So, the final display is: **Unknown by Novel - Dune - Frank Herbert**.

Answer to exercise 3:

For each code snippet below, provide the result of its execution, if applicable (explain the result). If an error occurs, justify it.

(a)	**Final display:** XYX **Explanation:** When executing 'new Dog();': 1. The constructor for Dog() is activated. 2. Inside, this("Rex") is called. 3. This triggers the constructor Dog(String name), which executes 'System.out.print("Y");'. 4. Before this, the constructor of the parent class Animal is invoked, printing 'X'. 5. Then, 'Y' is printed, followed by 'Z' at the end of 'Dog()'.

```
package chapter3;

class Animal {
Animal() { System.out.print("X");
}
}

class Dog extends Animal {
public Dog(String name) {
System.out.print("Y"); }
public Dog() { this("Rex");
System.out.print("Z"); }
}

public class AnimalTest {
public static void main(String[]
args) {
new Dog();
}
}
```

(b)	**Final display:**

```
package chapter3;

abstract class Fruit {
protected abstract void getFruitName();
public double getFruitPrice(int
quantity) { return quantity * 1.5; }
}

public class Apple extends Fruit {
protected void getFruitName() {
System.out.println("I am an Apple"); }
public static void main(String args[]) {
Fruit f = new Apple();
f.getFruitName();
System.out.println(f.getFruitPrice(3));
}
}
```

I am an Apple
4.5

Explanation:

- When running main:

1. Fruit f = new Apple(); creates an instance of Apple.

2. f.getFruitName(); calls the Apple class's getFruitName() method, displaying "I am an Apple."

3. f.getFruitPrice(3); calls getFruitPrice(int quantity) and returns 3 * 1.5, or 4.5.

Answer to exercise 4:

```
package chapter3;

//Abstract Class Fish

abstract class Fish {
private String id;
private double weight;

public Fish(String id, double weight) {
this.id = id;
this.weight = weight;
}

public void changeWeight(double newWeight) {
this.weight = newWeight;
}

public double getWeight() {
return weight;
}

public abstract double sellPricePerKilogram();
public abstract boolean isReadyToHarvest();
}

//Salmon Class
class Salmon extends Fish {
private String variety;
private double sellPricePerKg;
private double harvestWeight;

public Salmon(String id, double weight, String variety, double
sellPricePerKg, double harvestWeight) {
super(id, weight);
this.variety = variety;
```

```
this.sellPricePerKg = sellPricePerKg;
this.harvestWeight = harvestWeight;
}

public void changeVariety(String newVariety) {
this.variety = newVariety;
}

@Override
public double sellPricePerKilogram() {
return sellPricePerKg;
}

@Override
public boolean isReadyToHarvest() {
return getWeight() >= harvestWeight;
}

@Override
public String toString() {
return "Salmon{" +
"id='" + super.toString() + '\'' +
", variety='" + variety + '\'' +
", weight=" + getWeight() +
", salePricePerKg=" + sellPricePerKg +
'}';
}
}

//Main Class

public class Main {
public static void main(String[] args) {
Salmon salmon1 = new Salmon("S001", 4.5, "Atlantic Salmon", 15.0, 3.0);

// Display salmon details
System.out.println(salmon1);

// Change the salmon's weight
salmon1.changeWeight(5.0);
System.out.println("New weight: " + salmon1.getWeight() + " kg");

// Check if the salmon is ready for harvest
System.out.println("Ready for harvest: " + salmon1.isReadyToHarvest());

// Change the salmon's variety
salmon1.changeVariety("King Salmon");
System.out.println("New variety: " + salmon1);
}
}
```

The execution result is:

```
Salmon{id='chapter3.Salmon@1cd072a9', variety='Atlantic Salmon',
weight=4.5, salePricePerKg=15.0}
New weight: 5.0 kg
Ready for harvest: true
New variety: Salmon{id='chapter3.Salmon@1cd072a9', variety='King Salmon',
weight=5.0, salePricePerKg=15.0}
```

Answer to exercise 4:

```
package chapter3;
```

```
class GameDeveloper {
    private String developerID;
    private String[] games;

    public String getDeveloperIdentifier() {
        return developerID;
    }

    public void setDeveloperIdentifier(String developerID) {
        this.developerID = developerID;
    }

    public String[] getGames() {
        return games;
    }

    public void setGames(String[] games) {
        this.games = games;
    }

    public GameDeveloper(String DeveloperID, String[] games) {
        this.developerID = DeveloperID;
        this.games = games;
    }

    public int calculateNumberGames() {
        return games.length;
    }

    public void displayGames() {
        System.out.println("Games developed by developer " + developerID + ":");
        for (String game: games) {
            System.out.println(game);
        }
    }
}

class GameStudio {
    private String studioName;
    private GameDeveloper[] developers;

    public String getStudioName() {
        return studioName;
    }

    public void setStudioName(String StudioName) {
        this.studioName = StudioName;
    }

    public GameDeveloper[] getDevelopers() {
        return developers;
    }

    public void setDevelopers(GameDeveloper[] developers) {
        this.developers = developers;
    }

    public GameStudio(String StudioName, GameDeveloper[] developers) {
        this.studioName = StudioName;
```

```
        this.developers = developers;
    }

    public int calculateNumberTotalGames() {
        int totalGames = 0;
        for (GameDeveloper developer: developers) {
            totalGames += developer.calculateNumberGames();
        }
        return totalGames;
    }

    public void displayGamesByDeveloper() {
        System.out.println("Games developed by studio developers " + studioName +
":");
        for (GameDeveloper developer: developers) {
            developer.displayGames();
            System.out.println();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        GameDeveloper developer1 = new GameDeveloper("Dev1", new String[]{"Game1",
"Game2"});
        GameDeveloper developer2 = new GameDeveloper("Dev2", new String[]{"Game3",
"Game4"});
        GameDeveloper developer3 = new GameDeveloper("Dev3", new
String[]{"Game5"});

        GameStudio studioX = new GameStudio("StudioX", new
GameDeveloper[]{developer1, developer2, developer3});

        studioX.displayGamesByDeveloper();

        int TotalGames = studioX.calculateNumberTotalGames();
        System.out.println("Total number of games developed by the studio " +
studioX.getStudioName() + ": " + TotalGames);
    }
}
```

The execution result is:

```
Games developed by studio developers StudioX:
Games developed by developer Dev1:
Game1
Game2

Games developed by developer Dev2:
Game3
Game4

Games developed by developer Dev3:
Game5

Total number of games developed by the studio StudioX: 5
```

Answer to exercise 5:

```
package chapter3;

//**Vehicle**
class Vehicle {
     private boolean engine;
     private int maxSpeed;

     public Vehicle() {
         this.engine = false;
         this.maxSpeed = 0;
     }

     public Vehicle(boolean engine, int maxSpeed) {
         this.engine = engine;
         this.maxSpeed = maxSpeed;
     }

     // Getters and Setters
     public boolean getEngine() {
        return engine;
     }

     public void setEngine(boolean motor) {
         this.engine= engine;
     }

     public int getMaxSpeed() {
         return maxSpeed;
     }

     public void setMaxSpeed(int maxSpeed) {
         this.maxSpeed = maxSpeed;
     }

     public void VehicleCharacteristics() {
         System.out.println("Engine: " + engine);
         System.out.println("Maximum speed: " + maxSpeed);
     }

     public void MaxSpeed() {
         System.out.println("Maximum vehicle speed: " + maxSpeed);
     }
}

//**Compound_Car**

class Compound_Car {
     private int numberOfDoors;
     private Vehicle ve;

     public Compound_Car() {
         this.numberOfDoors = 2;
         this.ve = null;
     }

     public Compound_Car(int numberOfDoors, Vehicle ve) {
         this.numberOfDoors = numberOfDoors;
         this.ve = ve;
```

```
    }

    // Getters and Setters
    public int getNumberOfDoors() {
        return numberOfDoors;
    }

    public void setNumberOfDoors(int numberOfDoors) {
        this.numberOfDoors = numberOfDoors;
    }

    public Vehicle getVe() {
        return ve;
    }

    public void setVe(Vehicle ve) {
        this.ve = ve;
    }

    public void compoundVehicleCharacteristic() {
        System.out.println("Number of doors: " + numberOfDoors);
        System.out.println("Vehicle characteristics:");
        ve.VehicleCharacteristics();
    }
}

//**Drift_Car**

class Drift_Car extends Vehicle {
    private int numberOfDoors;

    public Drift_Car() {
        super();
        this.numberOfDoors = 2;
    }

    public Drift_Car(boolean engine, int maxSpeed, int numberOfDoors) {
        super(engine,maxSpeed);
        this.numberOfDoors = numberOfDoors;
    }

    // Getter and Setter for the added attribute
    public int getNumberOfDoors() {
        return numberOfDoors;
    }

    public void setNumberOfDoors(int numberOfDoors) {
        this.numberOfDoors = numberOfDoors;
    }

    public void DriftVehicleCharacteristics() {
        System.out.println("Number of doors: " + numberOfDoors);
        System.out.println("Vehicle characteristics:");
        VehicleCharacteristics();
    }
}
```

```
//**Main.java**
public class Main {
     public static void main(String[] args) {
         // Test of the Vehicle class
         Vehicle vehicle = new Vehicle();
         vehicle.VehicleCharacteristics();
         vehicle.MaxSpeed();
         System.out.println();

         // Test of the Compound_Car class
         Vehicle compoundVehicle = new Vehicle(true, 200);
         Compound_Car compoundCar = new Compound_Car(4, compoundVehicle);
         compoundCar.compoundVehicleCharacteristic();
         System.out.println();

         // Test of the  Drift_Car class
         Drift_Car drift_Car = new Drift_Car(true, 250, 2);
         drift_Car.DriftVehicleCharacteristics();
     }
}
```

The execution result is:

```
Engine: false
Maximum speed: 0
Maximum vehicle speed: 0

Number of doors: 4
Vehicle characteristics:
Engine: true
Maximum speed: 200

Number of doors: 2
Vehicle characteristics:
Engine: true
Maximum speed: 250
```

Answer to exercise 6:

```
package chapter3;
class Score {
protected String moduleName;
protected double pwScore;
protected double examScore;

public Score(String m, double pw, double e){
this.moduleName=m;
this.pwScore=pw;
this.examScore=e;

}

public double average() {
return (this.pwScore *0.4) + (this.examScore *0.6);
}

}

class Evaluation extends Score {
private int entryyear;
```

```
public Evaluation(String m, double pw, double e, int a){
super(m,pw,e);
this.entryyear=a;

}

public int getYearOfEntry() {
return entryyear;
}

public void setYearOfEntry(int a) {
this.entryyear=a;
}


public void deliberate() {

if(super.average()>=10) {
System.out.println("Admitted");
}else {
System.out.println("Postponed");
}
}

public String compareRating(Evaluation ev) {

if (this.moduleName==ev.moduleName) {
if (this.pwScore>ev.pwScore) {
return "The best TP note between the two Evaluations is: " +this.pwScore;
}else {
if(this.pwScore<ev.pwScore) {
return "The best TP grade between the two Assessments is: " +ev.pwScore;
}else {
return "The two TP scores of the two evaluations are equal";
}
}
}else {
return "Please note, these two Evaluations do not concern the same module";
}
}

}

public class TestEvaluation {
public static void main(String[] args) {
// TODO Auto-generated method stub

Evaluation ev1= new Evaluation ("OOP", 15, 12, 2012);
System.out.println("The average obtained in OOP is "+ ev1.average());
ev1.deliberate();

Evaluation ev2= new Evaluation ("OOP", 9, 10.5, 2013);
System.out.println("The average obtained in OOP is "+ ev2.average());
ev2.deliberate();

Evaluation ev3= new Evaluation ("Network", 14, 11.75, 2012);
System.out.println("The average obtained in Network is "+ ev3.average());
ev3.deliberate();
```

```
System.out.println("Comparison of PW scores notes between noteTP1 and noteTP2: "+
ev1.compareRating(ev2));
System.out.println("Comparison of PW scores between pwScore2 and pwScore3: "+
ev2.compareRating(ev3));
}
}
```

The execution result is:

```
The average obtained in OOP is 13.2
Admitted
The average obtained in OOP is 9.9
Postponed
The average obtained in Network is 12.65
Admitted
Comparison of PW scores notes between noteTP1 and noteTP2: The best TP note
between the two Evaluations is: 15.0
Comparison of PW scores between pwScore2 and pwScore3: Please note, these
two Evaluations do not concern the same module
```

Answer to exercise 7:

```
package chapter3;
abstract class Vehicle {
private long registrationNumber;
private int modelYear;
private int maxSpeed;
private double price;
private static long numberOfVehicles = 0;

public Vehicle(int modelYear, int maxSpeed, double price){
numberOfVehicles++;
registrationNumber = numberOfVehicles;
this.modelYear = modelYear;
this.maxSpeed = maxSpeed;
this.price = price;
}

public long getMatricule() {
return registrationNumber; }


public int getModelYear() {
return modelYear; }

public void setModelYear(int value) {
modelYear = value; }


public int getMaxSpeed() {
return maxSpeed;
}

public void setMaxSpeed(int maxSpeed) {
this.maxSpeed=maxSpeed;
}


public double getPrice(){
return price; }
```

```
public void setPrice (double value) {
price = value; }


public static long getNb_vehicles() {
return numberOfVehicles; }


public String vehicleCharacteristic()
{
return "Registration number: " + registrationNumber + " Model year: " + modelYear
+ " Maxspeed: " + maxSpeed + " km/h" + " Price: " + price + "DA";
}


public void MaxSpeed(){
System.out.println("\nmax speed: "+maxSpeed+" km/h\n");
}

public abstract void Start();
public abstract void Accelerate();
public abstract void loadCargo();

}

abstract class Car extends Vehicle {
private int numberOfDoors;
public Car(int modelYear, int maxSpeed, double price, int numberOfDoors) {
super(modelYear, maxSpeed, price);
this.numberOfDoors=numberOfDoors;
}

public int getNumberOfDoors() {
return numberOfDoors;
}

public void setNumberOfDoors (int numberOfDoors) {
this.numberOfDoors=numberOfDoors;
}


public void Start() // Implementation of the abstract Start() method
{
System.out.println("The car starts....");
}

public void Accelerate() // Implementation of the abstract accelerate() method
{
System.out.println("The car is accelerating....");
}


public String vehicleCharacteristic()
{
return "Matricule: " + getMatricule() + "Model Year: " + getModelYear() +
"Maxspeed: " + getMaxSpeed() +" km/h" + " numberOfDoors: " + getNumberOfDoors() +"
Price: " + getPrice( ) + "DA";
}
```

```
}

class Truck extends Vehicle {
private String cargo;
public Truck(int modelYear, int maxSpeed, double price, String cargo) {
super(modelYear, maxSpeed, price);
this.cargo=cargo;
}

public String getCargo() {
return cargo;
}

public void setCargo(String cargo) {
this.cargo=cargo;
}


public void Start() // Implementation of the abstract Start() method
{
System.out.println("The truck starts....");
}

public void Accelerate() // Implementation of the abstract accelerate() method
{
System.out.println("The truck accelerates....");
}

public void loadCargo() // Implementation of the abstract loadCargo() method
{
System.out.println("The cargo load carried by this truck is: "+cargo);
}

}

public class VehicleTest {

public static void main(String[] args) {
// TODO Auto-generated method stub

Truck c = new Truck(2003,240, 45600000,"Food products");
System.out.println(c.vehicleCharacteristic());
c.Start();
c.Accelerate();
c.loadCargo();

}

}
```

The execution result is:

```
Registration number: 1 Model year: 2003 Maxspeed: 240 km/h Price: 4.56E7DA
The truck starts....
The truck accelerates....
The cargo load carried by this truck is: Food products
```

Answer to exercise 8:

```
package chapter3;
abstract class Product {
private String reference;
private long quantityStock;
private double price;

public Product(){
this.reference = null;
this.quantityStock = 0;
this.price = 0.0;
}

public Product(String reference, long quantityStock, double price){
this.reference = reference;
this.quantityStock = quantityStock;
this.price = price;
}

public String getReference() {
return reference; }

public long getQuantityStock() {
return quantityStock; }


public double getPrice() {
return price; }

public void setReference(String ref) {
this.reference=ref; }

public void setStockQuantity(long qts) {
this.quantityStock=qts; }


public void setPrice(double price) {
this.price=price; }

public abstract String productCharacteristic();

}

abstract class Computer extends Product {
private int ram;
private int harddisk;

public Computer(String reference, long quantityStock, double price, int ram, int
harddisk){
super(reference,quantityStock,price);
this.ram = ram;
this.harddisk = harddisk;
}

public int getRam() {
return ram; }

public int getHardDisk() {
return harddisk; }
```

```
public void setRam(int r) {
this.ram=r; }

public void setHardDisk (int d) {
this.harddisk=d; }

}

class Smartphone extends Product {
private int Resolution;
private int nbrChips;

public Smartphone(String reference, long quantityStock, double price, int
Resolution, int nbrChips){
super(reference,quantityStock,price);
this.Resolution = Resolution;
this.nbrChips = nbrChips;
}

public int getResolution() {
return Resolution; }

public int getNbBullets () {
return nbrChips; }

public void setResolution(int r) {
this.Resolution=r; }

public void setNbBullets (int n) {
this.nbrChips=n; }


public String goodDeal() {
if ((Resolution>=50)&&(getPrice()<500)) {
return "Buy it, it's worth it"; }else {
return "Avoid buying, it's a scam";
}
}

public String productCharacteristic()
{
return "Reference: " + getReference() + "StockQuantity: " + getStockQuantity() + "
price: " + getPrice() +" €" + " Resolution: " + Resolution +" Mpx"+" NbrChips: " +
nbrChips;
}

private String getStockQuantity() {
	// TODO Auto-generated method stub
	return null;
}
}

public class ProductTest {

public static void main(String[] args) {
// TODO Auto-generated method stub
Smartphone S = new Smartphone("SM-G988",45,450,49,2);
System.out.println(S.productCharacteristic());
System.out.println(S.goodDeal());
```

```
}
}
```

The execution result is:

```
Reference: SM-G988StockQuantity: null price: 450.0 € Resolution: 49 Mpx
NbrChips: 2
Avoid buying, it's a scam
```

Answer to exercise 9:

```
package chapter3;

class TennisMatch {

    private String playerA;
    private String playerB;
    private int setsWonByA;
    private int setsWonByB;

    public TennisMatch(String playerA, String playerB) {
        this.playerA = playerA;
        this.playerB = playerB;
        this.setsWonByA = 0;
        this.setsWonByB = 0;
    }

    public String getPlayerA() {
        return playerA;
    }

    public String getPlayerB() {
        return playerB;
    }

    public int getSetsWonByA() {
        return setsWonByA;
    }

    public int getSetsWonByB() {
        return setsWonByB;
    }

    public void scoreSet(String player) {
        if (player.equals("A")) {
            setsWonByA++;
        } else if (player.equals("B")) {
            setsWonByB++;
        }
    }
}

// Tournament Class
class Tournament {

    private TennisMatch[] matches;
    private int numOfMatches;
    private String[][] playerPoints;

    public Tournament() {
```

```
        matches = new TennisMatch[40];
        playerPoints = new String[10][2];
        numOfMatches = 0;
    }

    public void addMatch(TennisMatch match) {
        if (numOfMatches < matches.length) {
            matches[numOfMatches] = match;
            numOfMatches++;
        } else {
            System.out.println("Maximum number of matches reached!");
        }
    }

    public void listPlayers() {
        for (int i = 0; i < numOfMatches; i++) {
            TennisMatch match = matches[i];
            String playerA = match.getPlayerA();
            String playerB = match.getPlayerB();

            if (!playerExists(playerA)) {
                addPlayer(playerA);
            }
            if (!playerExists(playerB)) {
                addPlayer(playerB);
            }
        }
    }

    private boolean playerExists(String player) {
        for (int i = 0; i < playerPoints.length; i++) {
            if (playerPoints[i][0] != null && playerPoints[i][0].equals(player)) {
                return true;
            }
        }
        return false;
    }

    private void addPlayer(String player) {
        for (int i = 0; i < playerPoints.length; i++) {
            if (playerPoints[i][0] == null) {
                playerPoints[i][0] = player;
                break;
            }
        }
    }

    public int pointsPerPlayer(String player) {
        int points = 0;
        for (int i = 0; i < numOfMatches; i++) {
            TennisMatch match = matches[i];
            if (match.getPlayerA().equals(player)) {
                points += calculatePoints(match.getSetsWonByA(),
match.getSetsWonByB());
            } else if (match.getPlayerB().equals(player)) {
                points += calculatePoints(match.getSetsWonByB(),
match.getSetsWonByA());
            }
        }
```

```
        return points;
    }

    private int calculatePoints(int setsWonByPlayer1, int setsWonByPlayer2) {
        if (setsWonByPlayer1 > setsWonByPlayer2) {
            return 2;
        } else if (setsWonByPlayer1 == setsWonByPlayer2) {
            return 1;
        } else {
            return 0;
        }
    }

    public void calculateNumberPoints() {
        for (int i = 0; i < playerPoints.length; i++) {
            String player = playerPoints[i][0];
            if (player != null) {
                int points = pointsPerPlayer(player);
                playerPoints[i][1] = String.valueOf(points);
            }
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Player Name | Points\n");
        sb.append("------------------\n");
        for (int i = 0; i < playerPoints.length; i++) {
            String player = playerPoints[i][0];
            String points = playerPoints[i][1];
            if (player != null) {
                sb.append(player).append(" | ").append(points).append("\n");
            }
        }
        return sb.toString();
    }
}

// Main class to test the solution
public class Main {
    public static void main(String[] args) {
        Tournament tournament = new Tournament();

        // Add matches to the tournament
        TennisMatch match1 = new TennisMatch("Federer", "Nadal");
        match1.scoreSet("A");
        match1.scoreSet("A");
        tournament.addMatch(match1);


        TennisMatch match2 = new TennisMatch("Djokovic", "Nadal");
        match2.scoreSet("A");
        match2.scoreSet("B");
        tournament.addMatch(match2);

        TennisMatch match3 = new TennisMatch("Federer", "Djokovic");
        match3.scoreSet("A");
        match3.scoreSet("A");
```

```
        tournament.addMatch(match3);

        TennisMatch match4 = new TennisMatch("Thiem", "Nadal");
        match4.scoreSet("B");
        match4.scoreSet("B");
        tournament.addMatch(match4);

        TennisMatch match5 = new TennisMatch("Federer", "Thiem");
        match5.scoreSet("A");
        match5.scoreSet("B");
        tournament.addMatch(match5);

        TennisMatch match6 = new TennisMatch("Djokovic", "Thiem");
        match6.scoreSet("A");
        match6.scoreSet("A");
        tournament.addMatch(match6);

        tournament.listPlayers();
        tournament.calculateNumberPoints();

        System.out.println(tournament.toString());
    }
}
```

The execution result is:

```
Player Name | Points
------------|-------
Federer     | 5
Nadal       | 3
Djokovic    | 3
Thiem       | 1
```

Answer to exercise 10:

```
package chapter3;
interface Shape {
    double area();
    double Circumference();
}

class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this. width = width;
    }

    public double area() {
        return length * width;
    }

    public double Circumference() {
        return 2 * (length + width);
    }
}

class Circle implements Shape {
```

```
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public double Circumference () {
        return 2 * Math.PI * radius;
    }
}

public class TestShape {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle(5, 8);
        System.out.println("Rectangle :");
        System.out.println("Area: " + rectangle.area());
        System.out.println("Circumference: " + rectangle.Circumference());

        Shape circle = new Circle(4);
        System.out.println("Circle:");
        System.out.println("Area: " + circle.area());
        System.out.println("Circumference: " + circle.Circumference());
  }
}
```

The execution result is:

```
Rectangle :
Area: 40.0
Circumference: 26.0
Circle:
Area: 50.26548245743669
Circumference: 25.132741228718345
```

Answer to exercise 11:

```
package chapter3;
interface Animal {
    void sleep ();
    void eat();
}

class Mammal implements Animal {
    private String name;
    private int age;

    public Mammal (String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void sleep() {
        System.out.println("The mammal " + name + " sleeps.");
    }
```

```
    public void eat() {
        System.out.println("The mammal " + name + " eats.");
    }
}

class Bird implements Animal {
    private String name;
    private String habitat;

    public Bird(String name, String habitat) {
        this.name = name;
        this.habitat = habitat;
    }

    public void sleep() {
        System.out.println("The bird " + name + " sleeps.");
    }

    public void eat() {
        System.out.println("The bird " + name + " eats.");
    }
}

public class Zoo {
    public static void main(String[] args) {
        Animal[] animals =new Animal[5];
        animals[0] = new Mammal ("Monkey", 3);
        animals[1] = new Mammal ("Elephant", 5);
        animals[2] = new Bird ("Parakeet","cage");
        animals[3] = new Bird ("Canary", "cage");
        animals[4] = new Bird ("Eagle","aviary");

        for (Animal animal : animals) {
            animal.sleep();
            animal.eat();
        }
    }
}
```

The execution result is:

```
The mammal Monkey sleeps.
The mammal Monkey eats.
The mammal Elephant sleeps.
The mammal Elephant eats.
The bird Parakeet sleeps.
The bird Parakeet eats.
The bird Canary sleeps.
The bird Canary eats.
The bird Eagle sleeps.
The bird Eagle eats.
```

Answer to exercise 12:

The changes made are as follows:

Code	Correction of errors
1. class A { 2. private int a;	- Line 3: public A(int a) { this.a = a; }

3. A(int a) { a = a; }
4. private int setA(int value) { a = value; }
5. private void getA() { return a; }
6. }
7. class B extends A {
8. protected int b;
9. B(int b) {
10.
11. this.b = b; }}
12.
13. class C extends A {
14. private Final double k;
15. C() {
16.
17. k = 1; }
18. private double getC() {
19. return k; }}
20. class D extends A {
21. protected Final double d=5;
22.
23. protected int operation(int a) { d= a * 2;
24. return d; }
25. protected int calcul(int b) { return b + 1; }
26. protected void afficher() {
27. System.out.println("Méthode afficher dans D"); }
28.
29. public class test {
30. public Static void main(string[] arg) {
31. A x = new A(8);
32. B y = new B(2);
33. C z = new C(6);
34. x.a=2;
35. y.b = 2;
36. z.k = 3;
37. }
	- Line 4: public void setA(int value) { a = value; }
	
	- Line 5: public int getA() { return a; }
	
	- Line 9: public B(int a, int b) {
	
	- Line 10: super(a);
	
	- Line 14: private final double k;
	
	- Line 15: public C(int a) {
	
	- Line 16: super(a);
	
	- Line 18: public double getC() { ….}
	
	- Line 21: protected double d=5;
	
	- Line 22: public D(int a) {
	
	- Line 22: super(a); }
	
	- Line 23: protected double operation(int a) { d= a * 2;
	
	- Line 28: A closing brace (}) to add to properly close the class 'D'.
	
	- Line 29: public class Test { ... }
	
	- Line 30: public static void main(String[] args)
	
	- Line 32: B y = new B(8, 2);
	
	- Line 34: x.setA(2); a is declared private and cannot be modified directly.
	
	- Line 35: y.b = 2; (no error)
	
	- Line 36: //z.k = 3; K is declared final and cannot be modified.
	
	- Line 37: A closing brace (}) to be added at the end to properly close the 'Test' class.

Answer to exercise 13:

Consider the following Java program, which uses classes, interfaces, and method overriding:

```
package Chapter3;
interface J {
        static int k = 3;
        int n() ;}
```

```
class A {
static int e = 2;
public static int m () {
return (e+4) ; }
public static int s (int x) {
return (e+x);}
public int w (int x) {
return (x+1) ; }
}
```

```
class B extends A implements J
{
static int e = 6;
public static int m () { return
(e+3) ; }
public int w (int r) { return (e+2);
}
public int n() { return k+4;}
}
```

```
class C implements J {
static int e = 9;
public int n(){ return
k+5;}
}
```

Question 1: What will running the following code produce?

```
public class Test{
public static void main (String [] args){
A p=new B() ;
System.out.println (p.w(1));
System.out.println (p.m());
System.out.println (p.s(2));
}}
```

Result:

8
6
4

Question 2: What will running the following code produce?

```
public class Test{
public static void main (String [] args){
B y = new B() ;
J q = new C();
System.out.println (y.n());
System.out.println (q.n());
}}
```

Result:

7
8

Question 3: Fix errors in the following code:

```
7.  public class Test{
8.  //Do not change the type of variables
9.  public static void main (String args []){
10. A d = new J () ;
11. System.out.println (d.n());
12. J [] tab = [new A(), new C(), new B()];
```

Correction:

```
public class Test{
// Do not change the type of
variables
public static void main (String
args []){
A d = new B () ;
System.out.println (((B) d).n());
J [] tab = {new C(), new C(), new
B()};
}
}
```

Result :

```
7
```

Question 4 : What happens if we add the following line at the start of class B?

```
private int k = 10;
```

If we add the line private int k = 10; at the beginning of class B, this will have no impact on program execution, because the variable k of interface J is static and will not be hidden by this declaration.

Conclusion

In conclusion, this course handout represents an invaluable resource for 2nd year Computer Science students and developers wishing to deepen their knowledge of object concepts related to the Java language. It offers comprehensive and in-depth content on Object Oriented Programming (OOP), providing readers with the information needed to master this essential programming approach.

One of the strengths of this handout is its clear and organized structure. The three chapters, devoted respectively to the basics of object-oriented programming, encapsulation and inheritance, guide readers progressively and logically through the different concepts. Each chapter begins with an overview of the fundamental concepts required to solve the exercises, allowing developers to gain a solid understanding before moving on to more advanced concepts.

The collection of corrected exercises included in this first edition constitutes an ideal complement to learning programming in Java. The exercises are conveniently classified by theme, which facilitates targeted practice of the concepts covered. Some exercises focus on specific concepts mentioned in the statement, while others, the synthesis exercises, require more in-depth thinking and the simultaneous application of several concepts. These varied exercises offer 2nd year undergraduate students and developers' valuable opportunities to consolidate their skills and broaden their understanding of OOP.

In addition, the presence of detailed solutions after each exercise statement is a major asset of this handout. 2nd year undergraduate students and developers can thus check their answers, understand the solution approaches and strengthen their mastery of programming techniques and good practices.

I hope that the course handout on Object Oriented Programming has been beneficial to you, and that it will allow you to deepen your knowledge in programming and software development. The author of this course handout, Dr. Mohamed Mohammedi, would like to express his gratitude for the time you will spend using it. Please do not hesitate to contact me if you require further information. Enjoy your reading and assimilate the concepts well.

Bibliography

[1]. Apprendre la Programmation Orientée Objet avec le langage Java. Luc Gervais. Eni. 2ème édition.

[2] https://openclassrooms.com/courses/apprenez-a-programmer-en-java

[3] Java 8 - Apprendre la Programmation Orientée Objet et maîtrisez le langage. Thierry GROUSSARD Luc GERVAIS. Edition ENI. 2015.

[4] La programmation objet en Java. Michel Divay. Edition DUNOD. 2006.

[5] Claude Delannoy. “Programmer en Java”,Eyrolles, 3 éme édition, 2006.

[6] Hughes Bersini. “La programmation Orientée Objet, Cours et Exercices en UML2, Java, C#, C++, Python, PHP et Linq”, Eyrolles, 4éme édition, 2009.

Summary

This handout will be an enriching study companion for students who integrate object programming into their curriculum, particularly in the Java language. It must help them, if necessary, to evolve from procedural programming to object-oriented programming. The objective of this course is the introduction of the basic concepts of object-oriented programming (OOP) through practice using the Java language. This document dissects all the mechanisms of object programming (classes and objects, interactions between classes, sending messages, encapsulation, inheritance, polymorphism, abstract classes (use and importance), interfaces (use and importance), etc.). Each chapter includes certain concepts which are translated at the end into Java so that the reader can translate the theoretical concepts acquired into practice. Consequently, this course support accompanies the reader in a progressive learning process based on numerous examples. Finally, each chapter ends with practical exercises with their answers.

This document has the greatest merit in enabling the student to acquire the following skills:

- The essence of object programming and its transformation into Java language;
- Acquire intuitive reasoning to give a solution to a simple problem using the object-oriented approach;
- Write a functional Java language program;
- The essence and importance of Object-Oriented reasoning and OOP.

Who is this course handout for?

• This course handout will be read with benefit by all 2nd year undergraduate students in computer science, all students in computer science disciplines linked to the object approach and can be used by their teachers as course support.

• It is also intended for all computer scientists wishing to develop in Java. Whether the reader is a beginner or already has first experience with another language, he will find in this document all the necessary bases to quickly become familiar with one of the most used languages in the world.

Dr. Mohamed Mohammedi