الجمهورية الجزائرية الديمقراطية الشعبية
PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
وزارة التعليم العالي و البحث العلمي
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

جامعة محمد بوضياف – المسيلة
كلية التكنولوجيا
قسم الإلكترونيك

MOHAMED BOUDIAF UNIVERSITY OF M'SILA
FACULTY OF TECHNOLOGY
DEPARTMENT OF ELECTRONICS

الرقم : 49/ق.إ/2024 المسيلة في : 2024/03/05

شهادة إدارية

بعد الإطلاع على التقارير الايجابية الواردة من السادة الخبراء أعضاء لجنة دراسة المطبوعة الجامعية والاتية أسماؤهم:

- بلوطي عادل أستاذ محاضر "أ" جامعة محمد بوضياف - المسيلة
- بوخالفة عبد الوهاب أستاذ محاضر "أ" جامعة محمد بوضياف - المسيلة
- بلهوشات نوري أستاذ محاضر "أ" جامعة فرحات عباس – سطيف 01

صادق أعضاء اللجنة العلمية على قبول المطبوعة البيداغوجية المقدمة **باللغة الانجليزية**، مع إمكانية إتخاذها سندا في تدريس طلبة السنة الثانية ماستر إلكترونيك الأنظمة المضمنة، في ميدان علوم و تكنولوجيا و أن تعتمد في أي تقييم المسار العلمي للأستاذ المعني **بريك مراد** (أستاذ محاضر قسم "ب"– جامعة محمد بوضياف - المسيلة) تحت عنوان :

Java language

رئيس القسم
طباخ مصطفى

رئيس اللجنة العلمية
مزعاش عمار

Keywords : Java, OOP, Compiler, interpreter

Abstract

This handout is meticulously crafted to cater to the specific needs of second-year master's students in programming skills, providing a comprehensive exploration of Java's capabilities. From simple concepts to intricate advanced data structures, each chapter is designed to enhance the student's proficiency in Java, equipping them with skills that needed to attain a proficient level in Java programming, which enabling them to effectively tackle the modern technology facet

Democratic and Popular Republic of Algeria

Ministry of Higher Education and Scientific Research

University Mohamed Boudiaf of M'sila

Faculty of Technology

Department of Electronics

Pedagogical handout of:

Java language

Specialty: Embedded systems

Cycle: Master

Level: Second Year -M2-

Elaborated By: Dr. Mourad Brik

MCB - Electronics départment

Email : mourad.brik@univ-msila.dz

Academic Year: 2023/2024

Semestre: 3
Unité d'enseignement: UEM 2.1
Matière 4: Langage JAVA
VHS: 37h30 (Cours : 1h30, TP: 1h00)
Crédits: 3
Coefficient: 2

Objectifs de l'enseignement:

Java est un langage objet pur, c'est le langage le plus approprié à la programmation du réel. Il est présent dans les noyaux logiciels de presque tous les appareils technologiques actuels. Cette matière permet à l'étudiant d'accéder à un niveau appréciable en programmation Java qui lui permet de faire face à cet aspect de la technologie moderne.

Connaissances préalables recommandées:

Programmation objet et langage C.

Contenu de la matière:

Chapitre 1. Introduction à Java **(1 semaine)**
Le concept de Machine Virtuelle, spécificités par rapports à C++ et C#

Chapitre 2. Notions de base **(3 semaines)**
Les types primitifs de variables: Java langage fortement *typé*, Les structures de contrôle *if, for, while*, Les tableaux de types primitifs, Les fonctions arithmétiques intégrées, opérateurs de base

Chapitre 3. Classes et objets **(3 semaines)**
Déclaration de classe, Variables et méthodes d'instance, Définition des méthodes, Droits d'accès et encapsulation, Constructeur et destructeur, Accesseurs, Tableaux d'objets

Chapitre 4. Rappels Héritage et polymorphisme **(2 semaines)**
Héritage, Polymorphisme, Méthodes et classes abstraites, Interfaces, Surcharge, Modificateurs de méthodes.

Chapitre 5. Les structures de données génériques intégrées de Java **(3 semaines)**
Les Collections : Interfaces et implémentations, set, *list, map, queue.* Algorithmes sur les collections : Remplissage, Lecture, Tri, Mélange, etc. Parcours d'une collection : Itérateurs, Opérations massives

Chapitre 6. Les API graphiques **(2 semaines)**
Librairies de l'API standard, JNI Codes natifs, Les librairies AWT et SWINGX.

Objectives

Java technology in embedded systems provides a platform-independent and flexible solution for developing applications across a diverse range of devices. With its "write once, run anywhere" feature, Java code can be executed on any device equipped with a Java Virtual Machine (JVM), allowing for efficient portability. The language owns rich libraries that simplify development tasks and covering areas such as networking and I/O operations. Java represents a valuable choice for developing applications in embedded systems due to its provision of robust security features, active community support, and scalability.

This handout is meticulously crafted to cater to the specific needs of second-year master's students in programming skills, providing a comprehensive exploration of Java's capabilities. From simple concepts to intricate advanced data structures, each chapter is designed to enhance the student’s proficiency in Java, equipping them with skills that needed to attain a proficient level in Java programming, which enabling them to effectively tackle the modern technology facet.

Summary

CHAPTER I

1. Introduction **2**

1.1. Java Virtual Machine **3**

1.2. JVM, JRE and JDK **4**

a) JRE (Java Runtime Environment): **4**

b) JVM (Java Virtual Machine): **4**

c) JDK (Java Development Kit): **4**

1.3.Existing Java Platforms **5**

a) J2SE **5**

b) J2EE: **5**

c) J2ME: **5**

1.4. Key Difference and similarity Between Java and C# **6**

1.5.Key Difference between Java and C++ **7**

1.6.Exercises **7**

CHAPTER II

2. Basic concept **9**

2.1. Primitive type in java **9**

2.2. Control Structures **11**

2.2.1. Blocks **12**

2.2.2. Conditional Control Structure **12**

2.2.2.1. if, else Statement: **12**

a) Simple Condition: **12**

b) Complex Condition: **12**

2.2.2.2. If statement **13**

2.2.2.3. if else statement **14**

2.2.3. Using if with multiple statements **14**

2.2.4. Switch, case, default, break **15**

2.3. Repetition Control Structure (Loops) **17**

2.3.1. While statement **17**

2.3.2. The do-while Statement **18**

2.3.4. For statement **19**

2.4. Using Break and Continue in java **19**

a) break statement: **19**

b) continue statement: **20**

2.5. Array of primitive type **20**

2.5.1. Array Creation: **20**

2.5.2. Accessing Array Elements **21**

a) Length: **21**

2.6. Integrated arithmetic functions **21**

2.7. Java basic operators **22**

2.8. Exercises **23**

CHAPTER III

3. Classes and Objects **25**

3.1. Object-Oriented Programming (O.O.P) **25**

3.2. Declaring Classes **25**

3.3. Variables and method instance **26**

3.3.1. Objects and instances **27**

3.3.2. Java Methods **28**

a) Declaration: **28**

3.3.3. Modifiers **29**

3.3.4. Return value **29**

3.3.5. Parameters **29**

3.4. Static Methods and Variables 29

3.5. Method Overloading 30

3.6. Java access rights 30

3.7. Encapsulation in java 31

3.8. Java Constructor and Destructor 33

3.9. Accessor and mutator 33

3.9.1. Accessor 33

3.9.2. Mutator 34

3.10. Array of objects in java 34

CHAPTER IV

4. Inheritance, polymorphism and Abstract class 36

4.1. Inheritance 36

4.1.1. Syntax 36

4.2. Inheritance and constructor 37

4.2.1. Invoking Superclass Constructors: 37

4.2.2. Calling Superclass Methods: 37

4.3. Polymorphism 38

4.3.2. Method Overriding: 39

4.4. Abstract Class and method 40

4.5. Interface 41

4.6. Overload (method surcharging) 42

4.7. Modifiers 42

CHAPTER V

5. Integrated Generic data structure in java 44

5.1 Collection 44

5.1.1 Interfaces and implementation 44

5.1.2. Generic types in Java 45

5.1.2.1. Set interface **45**

5.1.2.1.1. ***HashSet*** **46**

5.1.2.1.2. ***LinkedHashSet:*** **46**

5.1.2.3. List **47**

5.1.2.3.1 ArrayList **47**

5.1.2.4. Map **48**

5.1.2.5. Queue **49**

- **a) LinkedList** **50**
- **b) PriorityQueue** **50**

5.2. Collection Algorithms **50**

5.2.1. Fill **51**

5.2.3. Sort **51**

5.2.4. Shuffle **51**

5.2.5. Reverse Order **52**

5.2.6. Binary Search **52**

5.3. Collections' iterating **52**

5.3.1. For-each **52**

5.3.2. Iterator **53**

5.3.3. Massive operations in Java **53**

- **a) Concurrency and Parallelism** **53**
- **b) Efficient Algorithms** **53**
- **c) Memory supervision** **53**

CHAPTER VI

6. Graphical API **56**

6.1. Standard Java library API **56**

- **a) Java Standard Edition (Java SE)** **56**

b) Java Enterprise Edition (Java EE) **56**

c) Java Micro Edition (Java ME) **56**

6.2. Java Natives Interface (JNI) **57**

6.3. AWT and SWINGx library **59**

6.3.1. AWT (Abstract Window Toolkit): **59**

6.3.2. Swingx: **60**

Chapter I

Introduction to Java

1. Introduction

Java is a popular programming language that introduced by Sun Microsystems in 1995, now is owned by ***Oracle Corporation*** after the acquisition of Sun Microsystems in 2010. It was designed to be portable, secure, and robust, making it a preferred choice for software developers across various platforms. Java is an object-oriented language, which means it allows modeling the real world using objects and classes. It promotes a modular approach to development, enabling code reusability and the construction of scalable programs.

Java is widely used for desktop applications, web applications, games, mobile applications, and enterprise applications. It provides a vast standard library that facilitates the development of various functionalities, such as file manipulation, network management, GUI creation, and much more.

The Java Virtual Machine (JVM) ensures independence from the execution environment. Java is a compiled language, but the compiler does not produce native machine code. Instead, it generates bytecode, a set of instructions understood by the JVM, which translates it into executable code for the machine at runtime.

In order to run Java program, it is necessary not only to compile the source code but also the runtime environment (including the JVM) must be installed on the target machine.

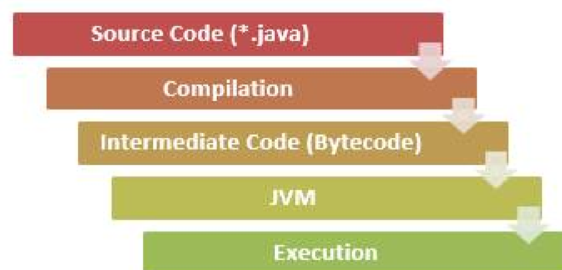


Figure 1 Execution Cycle of Java Program

Java utilizes the Java Virtual Machine (JVM) to enable the execution of Java code on various operating systems that are compatible with the JVM. As a result, Java applications can run seamlessly on Windows, Mac, or Linux computers without the need for OS-specific alterations.

The popularity of Java can be attributed in part to its portability, security, and reliability. It also has a large community of developers and abundant online resources, making it easy to learn and share knowledge.

In summary, Java is a powerful and versatile programming language, well suited for software development across different platforms. Its portability, security, and popularity make it a common choice for worldwide developers.

1.1.Java Virtual Machine

At the present time, all software designers' utilize high-level programming languages like Java, Pascal, C++, and others. However, programs written in high-level languages cannot be directly executed on such computer. They need to use a process known as translation, accomplished through a specific program called ***compiler***. The main task of a compiler is to map programs written in a given source language into a target language. Once the translation is complete, the resulting machine-language program can be executed multiple times on particular type of computer. This limitation exists because each computer type has its own unique machine language. To enable the program to run on a different type of computer, it must be retranslated using a different compiler into the appropriate machine language.

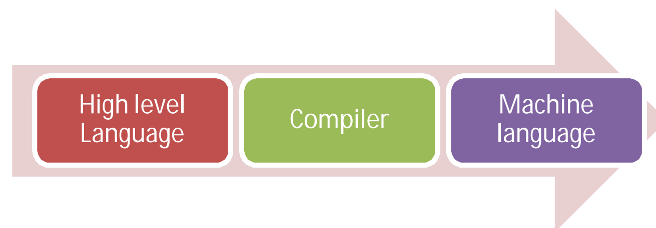


Figure 2 Compiler Life Cycle

Consequently, an alternative to compiling a high-level language program exist which called ***interpreter***. Unlike a compiler, an interpreter translates the program instruction-by-instruction as needed. It function is like fetch-and-execute cycle. To execute a program, the interpreter operates within a loop where it continuously reads one instruction from the program. It then determines the necessary actions required to carry out that instruction and proceeds to execute the corresponding machine-language commands. This allows for a more dynamic and interactive execution process compared to the static translation offered by a compiler.

As shown in figure 3, JVM use a combination of compilation and interpretation. Programs written in Java are compiled into intermediate code called ***java byte code*** (****.class)*** using a specific program called ***Javac*** , this generated code is not a machine language, it needs an interpreter for Java bytecode which guaranteed by **JVM.** Of course, all that the computer requires a specific Java bytecode interpreter to execute Java programs, but once installed, it enables the execution of any Java bytecode program. Furthermore, a single Java bytecode program can be

executed on any computer equipped with such interpreter. This characteristic is a fundamental aspect of Java, allowing a compiled program to be executed on diverse computer platforms.

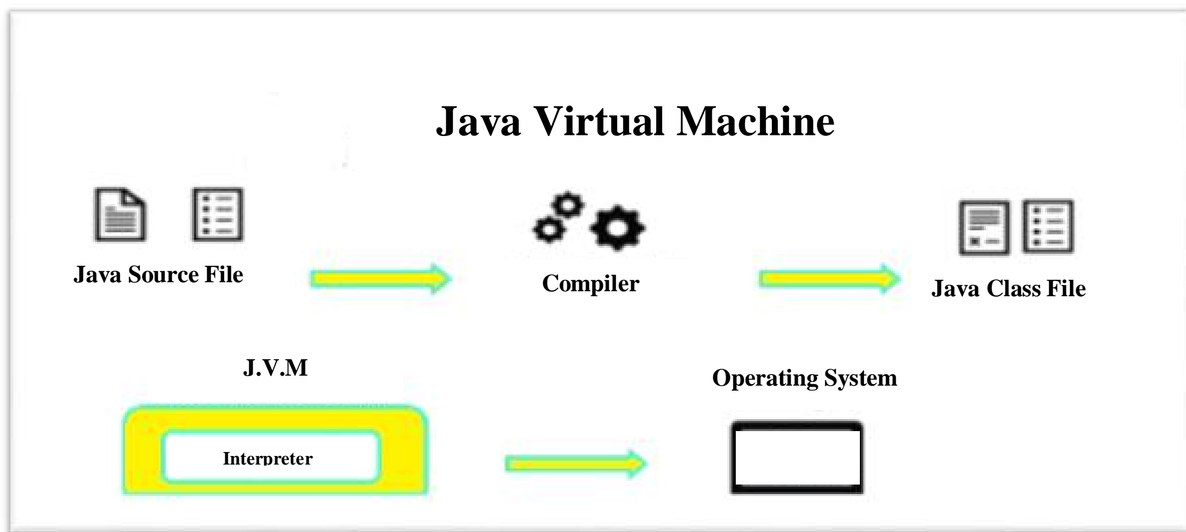


Figure 3 Java Program Execution Cycle

1.2. JVM, JRE and JDK

Three concepts are used widely in java paradigm JVM, JRE and JDK, which represent important components in the Java programming ecosystem

a) ***JRE (Java Runtime Environment):*** JRE is the runtime environment for executing Java applications. It includes the JVM, Java class libraries, and other supporting files required to run Java programs. JRE does not include development tools and is primarily used by end-users to run Java applications on their machines.

b) ***JVM (Java Virtual Machine):*** JVM is a crucial part of the Java platform. It is responsible for executing Java bytecode, which is the compiled form of Java source code. JVM provides an abstraction layer between the Java code and the underlying operating system. It manages memory, handles garbage collection, and provides runtime support for executing Java programs. JVM implementations are platform-specific, meaning different operating systems require different JVMs.

c) ***JDK (Java Development Kit):*** JDK is a software development kit used by Java developers to create, compile, and debug Java applications. It includes the necessary tools, compilers, and libraries to develop Java programs. JDK consists of the JRE along

with additional development tools such as the Java compiler (javac), debugger (jdb), and other utilities. It allows developers to write, test, and package Java applications.

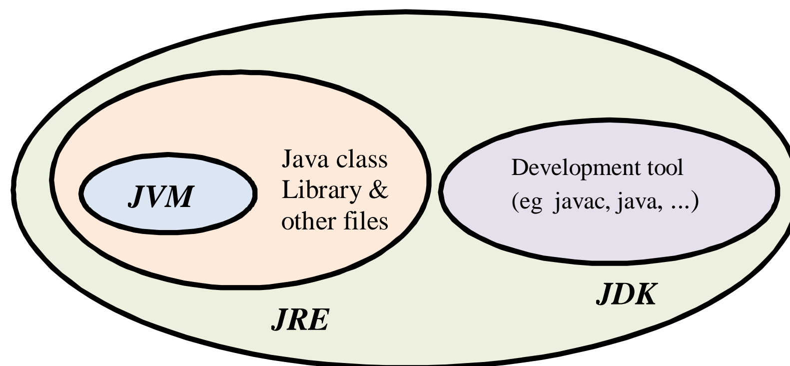


Figure 4 JVM, JDK and JRE

1.3.Existing Java Platforms

a) **J2SE:** Java 2 Standard Edition. This edition consists of application programming interfaces (APIs) needed to build a Java application or applet. It includes the Java Development Kit (JDK) and the Java Runtime Environment (JRE). Java SE is used for desktop applications, command-line tools, and standalone applications.

b) **J2EE:** Java 2 Enterprise Edition. This edition consists of application programming interfaces (APIs) needed to build a application for access server-side systems. It provides a set of APIs and services for building distributed, scalable, and robust applications.

c) **J2ME:** Java 2 Micro Edition. This edition consists of application programming interfaces (APIs) needed to build a applications for small computing devices. It includes a set of libraries and APIs specifically designed for developing applications for resource-constrained devices, such as mobile phones, embedded systems, and ***I.O.T*** (Internet of Things) devices. Java ME enables developers to create mobile applications and embedded systems with limited memory, processing power, and display capabilities.

1.4. Key Difference and similarity Between Java and C#

C# is an object-oriented programming language, developed by Microsoft that runs on .Net Framework.

C# provides modern features and simple codes, it keeps on updating from time to time which allows for numerous integrations and contemporary elements.

Both C# and Java came into being by keeping the C and C++ language in view as they have a similar syntax. C# uses CLR (Common Language Runtime), whereas Java uses JRE (Java Runtime Environment).

C# supports operator overloading for multiple operators. Java still doesn't support operator overloading and pointers. C# can support leads only in unsafe mode (it is not inherently dangerous, just the code not verified by CLR). C# arrays have the specialization of System, whereas Java arrays have the occupation of Objects.

The major differences and similarities between c# and java is summarized in the table below

Table 1 Java and C#

Parameters	C#	Java
Programming Paradigm	Object-oriented, component-oriented, strong typing, functional	Object-oriented
Platform Dependency	Primarily Windows, .NET framework	Platform-independent (runs on JVM)
Application	Web and game development	Complex web-based applications
Tools and IDE	Visual Studio, Mono Develop	Eclipse, NetBeans, ...
Memory Management	Automatic garbage collection	Automatic garbage collection.

1.5.Key Difference between Java and C++

C++ is a versatile and widely used programming language that was developed as an extension of the C programming language. It was created in the late 1970s, which become a popular choice for system programming, game development, embedded systems, and other performance-critical applications

C++ combines both high-level and low-level programming features, allowing developers to write efficient and portable code. It supports object-oriented programming (OOP) principles, including classes, inheritance, polymorphism, and encapsulation

Table 2 Java and C++

Parameters	**Java**	**C++**
Platform	Platform-independent (runs on JVM)	Platform-dependent
Memory Management	Automatic garbage collection	Manual memory management
Object-Oriented	Fully object-oriented language	Supports object-oriented programming
Pointer Arithmetic	No support for pointer arithmetic	Supports pointer arithmetic
Exception Handling	Checked and unchecked exceptions	Exception handling through try-catch blocks
Standard Library	Extensive and robust standard library	Standard library is smaller and less mature
Performance	Generally slower due to JVM overhead	Generally faster due to direct machine code
Compilation	Bytecode compiled and interpreted by the JVM	Direct compilation to machine code
Multi-threading	Built-in support for multi-threading	Supports multi-threading through libraries
Memory Safety	Provides memory safety with JVM	Memory safety depends on programmer's skill
Syntax	Simplified syntax and strict coding conventions	Flexible syntax and allows low-level programming

1.6. Exercises

- *Is C++ a compiler? Cite two advantage of an interpreter?*
- *What is the main purpose of JVM?*
- *What is JDK? Is eclipse a JRE?*
- *Cite two features of java program vs c# program*

Chapter II

Basic Concepts

2. Basic concepts

As mentioned in the previous chapter, the Java language is a fully object-oriented language (O.O.P), which means that any java program must be write in oriented-object programming principles. The central concepts of object-oriented programming is the class and object, which represent a kind of module containing data and methods. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has both static state (variables) and dynamic state (methods).

Therefore, to start writing a Java program, you must define a class that contains the main method. The main method acts as the entry position of your program, from where the execution begins. In conclusion, every Java application has at least one class and at least one main method.

```
public class Test {
      public   static   void   main (String[] args)  {
            //  code to execute or call other methods
                } }
```

Normally, an application consists of many classes and only one of the classes must have a main method.

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

This program is an example that calls the statement named ***System.out.println*** which displays the message ***hello! world***. When you run this program, the message "Hello World!" (Without the quotes) will be displayed on standard output. Because the java is cross platform, the standard output can be different things. for example when using a command-line interface like in windows platform, the Java interpreter run the program, by calling the main() subroutine.

2.1. Primitive type in java

In Java, primitive types are the fundamental building blocks for representing simple data values. Primitive types are not objects and directly hold their values in memory, making them more efficient in terms of memory usage and performance. Java has eight primitive data types designed to represent different kinds of data:

Table 3 primitive type in java.

Type	Min Value	Max Value	Comment
byte	-128	127.	Used to store small integer values. It has a size of 8 bits
Short	-32,768	32,767	Used for storing short integer values. It has a size of 16 bits
Int	-2^31	2^31 - 1.	The most commonly used integer type. It has a size of 32 bits
Long	-2^63	2^63-1.	Used for long integer values. It has a size of 64 bits
Float	Size : 32 bits		Represents single-precision floating-point numbers, which are used for decimal values with limited precision
Double	Size : 64 bits		Represents double-precision floating-point numbers, which are used for decimal values with higher precision.
Char	Size : 16 bits		:Used for storing a single character
Boolean	Size : 1 bits		Represents a binary value, typically used for conditions like true or false

The next example shows how primitive types are declared and used:

```
int    age  =  30; // Declaring an integer variable named age
double   price   =   19.99; // Declaring a Real variable
char   C  =   'A'; // Declaring a character variable
boolean   isTrue   =   true; // Declaring a Boolean variable
```

Primitive types are essential in Java for their efficiency and simplicity. They are often used to store basic data values in variables, as method parameters, and as return values from

methods. When you need to work with complex data, you can use objects and classes to create custom data type.

Java is known as a ***strongly typed language,*** which implies that all variables types must be explicitly declared and rigorously respected to throughout the program. The strong typing system of Java enhances the language's security and reliability by preventing common type errors that could lead to crash or undesirable behaviors during execution.

a) Example

```
int num1 = 10; // ok
int num2 = "20"; // Compilation Error because "20" is not an int
Char  Ch = "6 ";
int  num = 42;
int  result = Ch + num; // compilation Error   type mismatch
```

In summary, Java's strong typing ensures that data types are strictly defined and respected to throughout programming, which helping us to prevent many potential errors and enhance the reliability of Java programs.

2.2. Control Structures

In programming paradigm, a control structure is a mechanism that enables the control of how instructions are executed within a program. Two types of control structures, loops and branches, can be used to repeat a sequence of statements over and over or to make decision of action by testing conditions. Java supports many flow control statements. We can group these statements in the following manner:

- Conditional Control Structure
- Repetition Control Structure
- Selection Control Structure
- Jump Control Structure

2.2.1. Blocks

The block is the simplest type of structured statement. Its purpose is simply to group a sequence of instructions into a single instruction. it consists of a sequence of statements enclosed between a pair of braces, “{” and “}” . The format of a block is:

```
{     Statements ;  }
```

2.2.2. Conditional Control Structure

These statements are calling also ***decision-making control structure***. These statements enable us to make a decision based on a condition value (true, false). it permit to execute a block of code conditionally depending on whether the condition is **true** or false. We find two's type:

- **if, else**
- **switch, case, default, break**

2.2.2.1. if, else Statement:

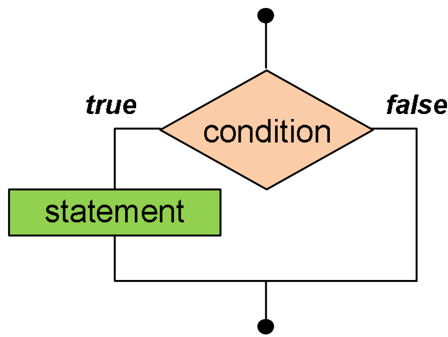
The condition is any expression that take a **true** or **false** value. we found two type of conditional expression : **Simple condition and complex condition**

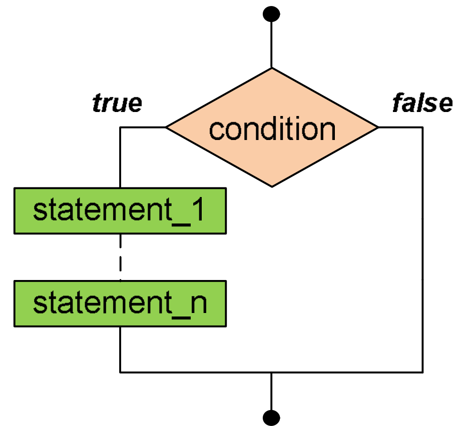
a) **Simple Condition:** A simple condition typically involves a single test or comparison. It evaluates whether a single criterion is met or not. For example, checking if a number is greater than 10 or if a variable is equal to a specific value are examples of simple conditions

b) **Complex Condition:** A complex condition, on the other hand, involves multiple tests or comparisons combined using logical operators (such as AND, OR, NOT). It evaluates whether a combination of criteria are met or not. Complex conditions allow you to create logic that is more complex by considering multiple factors simultaneously. For instance, checking if a number is greater than 10 AND less than 20 is a complex condition because it involves two tests combined with the logical AND operator

2.2.2.2. If statement

This statement is the most **basic** form of a decision-making structure, employed to determine if a code block should be executed or no, depending of a specified condition evaluation.






```
if (condition) statement;
```

```
if (condition)
{
  statement_1;
  . . .
  statement_n;
    }
```

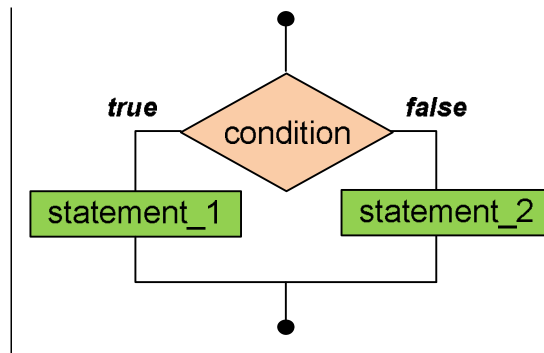
Consider some examples of conditional statements:

`if (x <= 0) { y = x + 2; }`	if x is less or equal to 0, assign to y the value of x + 2
`if (a == b) { y = a + b; }`	if a and b are equal, assign to y the sum of a and b
`if (x != a + 3) { y = a; }`	if x does not equal to a + 3, assign to y the value of a

2.2.2.3. if else statement

Another alternate is the use the **else** statement, which must employed with if statement. Employed to determine whether a code block should be executed, depending of a specified condition evaluation

```
if (condition)
  statement_1;
    else
      statement_2;
```



Consider next example of ***if else*** statement:

```
public class IfElseExample {
  public static void main(String[] args) {
    int age = 18; // Declare and initialize a variable
         // Check if the person is eligible to vote*
    if (age >= 18) {
        System.out.println("You are eligible to vote.");
         // This block will be executed when condition = true.
    } else {
        System.out.println("You are not eligible to vote.");
         // This block will be executed when condition = false.
        }
    }}}
```

This example illustrates how you can use the if-else statement to make decisions based on conditions related to variables in your Java program.

2.2.3. Using if with multiple statements

The ***if statement*** only executes a single statement if the expression is true, and the else only executes a single statement if the expression is false. In order to execute multiple statements, we can use a block statements which enclosed between a pair of braces, "{" and "}"

a) Example : Write a code that finds the maximum of four real numbers a, b, c, d.

Let us consider ***Maxvalue*** assign initially a. Then compare each of the next numbers with ***Maxvalue***. If some number is greater than *Maxvalue* , update Maxvalue.

```
import java.util.Scanner;

  public class FindMaximum {
  public static void main(String[] args) {
   Scanner sc = new Scanner(System.in);
   System.out.print("Give four numbres ? :  ");
        System.out.print(" a=  ");
   double a = sc.nextDouble();
        System.out.print(" b=  ");
   double b = sc.nextDouble();
        System.out.print(" c=  ");
   double c = sc.nextDouble();
        System.out.print(" d=  ");
   double d = sc.nextDouble();
   double Maxvalue = a ;
   if (b > Maxvalue) {  Maxvalue = b;  }  else
   if (c > Maxvalue) {  Maxvalue = c;  }  else
   if (d > Maxvalue) {  Maxvalue = b;  }
   System.out.println("The maximum is: " + Maxvalue);    }}
```

The java.util.Scanner class is used for reading value from input devices like keyboard.

2.2.4. Switch, case, default, break

Another alternate of conditional statement called Switch statement, which employed in some cases. The statement is used when we have many choices of many values. It provides a way to select one of many code blocks to be executed, depending on the value of the expression. It used only with the integral datatypes (e.g., int, char, byte, short) or a String (since Java 7).

```
switch (expression) {
   case value1:
       // Code to be executed when expression matches value1
       break;
```

```
    case value2:
         // Code to be executed when expression matches value2
         break;
    // Additional case statements for other values
    default:
      // Code to be executed if expression doesn't match any case   }
```

a) **Example:** To retrieve the season based on the month number ("Spring," "Summer," "autumn," or "Winter") , we can used switch statement like below:

```
public class SeasonExample {
    public static void main(String[] args) {
        int monthNumber;
        Char season;
        switch (monthNumber) {
            case 1:
            case 2:
            case 3:
                season = "Spring";
                break;
            case 4:
            case 5:
            case 6:
                season = "Summer";
                break;
            case 7:
            case 8:
            case 9:
                season = " autumn  ";
                break;
            case 10:
            case 11:
            case 12:
                 season = "Winter";
                break;
            default:
                season = "Invalid month number"; }
 System.out.println("number " + monthNumber + " in season: " + season);}}
```

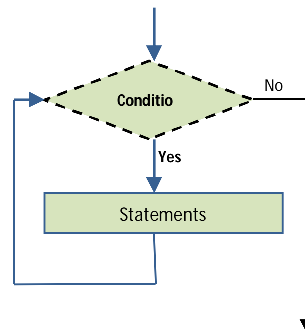
In conclusion, the switch statement in Java is a versatile control flow construct that allows you to make decisions based on the value of an expression. Here are some key points to summarize the switch statement.

2.3. Repetition Control Structure (Loops)

2.3.1. While statement

A while statement is declared using the ***while*** keyword. When a while statement is executed, the expression is evaluated. If the expression evaluates to true (non-zero), the statement executes, it has a definition very similar to that of an ***if*** statement: However, unlike an if statement, once the statement has finished executing, control returns to the top of the while statement and the process is repeated. in a most cases, a "while" statement is used to create a loop that continues executing a block of code as long as a specified condition is true.

```
while (condition) {
    // Code to be executed while
    // when the condition is true
}
```



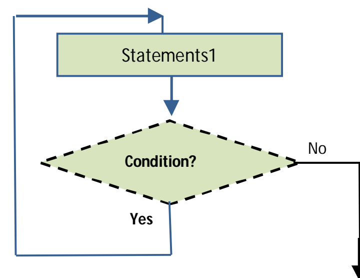
Here's an example of a simple "while" loop in Java that counts from 1 to 5:

```
public class WhileExample {
    public static void main(String[] args) {
        int count = 1;
while (count <= 10) {
    System.out.println(count);
    count = count +1; }}}
```

2.3.2. The do-while Statement

Sometimes it is more suitable to test the condition at the end of a loop, instead of at the beginning, as is done in the while loop in order to assurance that the code block will be executed at least once, even if the condition is initially false. The ***do..while*** statement is very similar to the while statement, except that the word “while,” along with the condition that it tests, has been moved to the end. The word “do” is added to mark the beginning of the loop. A ***do..while*** statement has the form

```
do
{   statements;  }
 while ( condition );
```



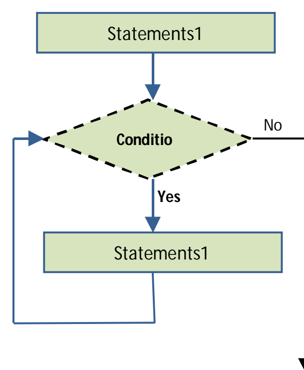
a) **Example**

```
int i=0;
 do { i=i+1;
        System.out.println("counter=" + count);
      while(i<=10); }
```

The computer executes first the body of the loop i.e, the statement or statements inside the loop block, and then it evaluates the condition. If the value of the expression is true, the computer returns to the beginning of the do loop and repeats the process; if the value is false, it ends the loop and continues with the next part of the program.

Note that we can use this loop using only the while loop .the main difference between is that in do while loop executes the block statements at least once and while loop executes zero or many time. when using *while loop* to perform a *do while* loop we must repeated the block statement two time as follow

```
statements1;
while (condition) {
     {   statements1;   }  }
```



2.3.4. For statement

The most utilized looping statement in programming language is the ***for*** statement. The for statement (also called a for loop) is ideal when we know exactly how many times we need to iterate, because it lets us easily define, initialize, and change the value of loop variables after each iteration.

The ***for statement*** looks simple in abstract:

```
for (init-statement; condition-expression; end-expression)
       statement;
```

The easiest way to understand a for loop is to convert it into an equivalent while loop:

```
{  init-statement;
  while (condition-expression)
  {
    statement;
    end-expression;
  }} // variables defined inside the loop go out of scope here
```

a) Example:

```
for (int count = 0; count < 10; count++)
         System.out.println("counter=" + count);
```

2.4. Using Break and Continue in java

In Java, the ***break*** and ***continue*** statements are used to control the flow of loops. Here's how you can use them:

a) break statement: The break statement is used to exit a loop prematurely, regardless of whether the loop's condition is still true. It is commonly used to exit a loop when a specific condition is verified.

```
int count = 1;
     while (count <= 5) {if (count == 5) {
    break; // Exit the loop when count is equal to 5
         }
     Count = count +1 ; System.out.println(i); }
```

The loop will print numbers from 1 to 4 and then exit when count is equal to 5.

b) *Continue statement:* The continue statement is used is used to skip the current iteration of a loop and move to the next iteration. It is useful when you want to skip some iterations based on a specific condition.

Here is an example of a simple for loop and continue in Java that shows the even numbers from 1 to 10:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue; // Skip even numbers
                        }

 System.out.println(i); }
```

2.5. Array of primitive type

In Java, arrays of primitive types can hold primitive data types that allow you to store elements of primitive data types such as **int**, **char**, **double**, **boolean**, and so on. Arrays are an efficient way to manage collections of data of the same type. Here's a guide on how to declare, instantiate, and utilize arrays of primitive types in Java:

```
int[] intArray;
```

2.5.1. Array Creation:

After declaring an array, you need to create it by specifying its size (the number of elements it can hold) using the new operator. For example, to create an array of integers with a size of 5, you can do this:

```
int[] intArray = new int[5];
```

You can also initialize an array with values at the time of creation. For example, an array of integers initialized with values:

```
int[] intArray = {1, 2, 3, 4, 5};
```

2.5.2. Accessing Array Elements

You can access array elements by using the index of the element (starting at zero for the first element). For example, to access the first element of the array:

```
int firstElement = intArray[0]; // Accesses the first element (1)
```

a) Length:

You can obtain the length (i.e., the number of elements) of an array using the length property. For example:

```
int length = intArray.length; // Gets the length of the array (5 in this example)
```

Arrays of primitive types in Java have a fixed size once they are created, meaning you cannot add or remove elements.

2.6. Integrated arithmetic functions

In Java, integrated arithmetic functions are built-in mathematical functions available through the standard library. These functions allow you to perform common mathematical operations. These functions are typically accessed through the **java.lang.Math** class. Here are some of the integrated arithmetic functions available in Java:

Math.abs(): Returns the absolute value of a number, making it positive, regardless of its sign.
Math.sqrt(): Returns the square root of a number.
Math.pow(): Calculates a power by raising a number to a given exponent.
Math.round(): Rounds a decimal number to the nearest integer.
Math.ceil(): Returns the smallest integer greater than or equal to a decimal number.
Math.floor(): Returns the largest integer less than or equal to a decimal number.
Math.max() and Math.min(): Returns the greater and lesser of two given numbers.
Math.random(): Generates a random decimal number between 0.0 and 1.0
Math.sin(), Math.cos() : Return trigonometric values for a given number in radians.
Math.exp(): Returns the value of the constant 'e' raised to the power of a specified number.
Math.log(): Returns the natural logarithm (base e) of a specified number.
Math.log10(): Returns the base-10 logarithm of a specified number.
Math.cbrt(): Returns the cube root of a specified number.

2.7. Java basic operators

In Java, you can use a variety of basic operators to perform operations on variables and values. These operators are fundamental for performing various operations, such as arithmetic calculations, assignments, comparisons, logical decisions, and bit-level manipulations. the following table summarize the most commonly used basic operators :

Table 4 Java basic operators

Operator	Meaning
Arithmetic Operators	
+	(Addition): Adds two values together
-	(Subtraction): Subtracts the right operand from the left operand.
*	(Multiplication): Multiplies two values.
/	(Division): Divides the left operand by the right operand.
%	(Modulus): Returns the remainder after division.
Assignment Operators:	
=	(Assignment): Assigns the value on the right to the variable on the left.
+=	(Add and Assign): Adds the right operand to the left operand and assigns the result to the left operand.
-=	(Subtract and Assign): Subtracts the right operand from the left operand and assigns the result to the left operand.
*=	(Multiply and Assign): Multiplies the left operand by the right operand and assigns the result to the left operand.
/=	(Divide and Assign): Divides the left operand by the right operand and assigns the result to the left operand.
%=	(Modulus and Assign): Computes the modulus of the left operand and right operand and assigns the result to the left operand.
Comparison Operators:	
==	(Equal to): Checks if two values are equal.
!=	(Not Equal to): Checks if two values are not equal.
<=	(Less than): Checks if the left operand is less than the right operand.
>=	(Greater than): Checks if the left operand is greater than the right operand
>	(Less than or Equal to): Checks if the left operand is less than or equal to the right operand.
<	(Greater than or Equal to): Checks if the left operand is greater than or equal to the right operand.

<table>
<tr><th colspan="2">Logical Operators:</th></tr>
<tr><td>&&</td><td>(Logical AND): Returns true if both operands are true.</td></tr>
<tr><td>||</td><td>(Logical OR): Returns true if at least one of the operands is true.</td></tr>
<tr><td>!</td><td>(Logical NOT): Inverts the value of the operand.</td></tr>
<tr><td colspan="2">Bitwise Operators (operate at the bit level):</td></tr>
<tr><td>&</td><td>(Bitwise AND)</td></tr>
<tr><td>|</td><td>(Bitwise OR)</td></tr>
<tr><td>^</td><td>(Bitwise XOR)</td></tr>
<tr><td>>></td><td>(Left Shift)</td></tr>
<tr><td><<</td><td>(Right Shift)</td></tr>
<tr><td colspan="2">Increment and Decrement Operators:</td></tr>
<tr><td>++</td><td>(Increment): Increases the value of the variable by 1.</td></tr>
<tr><td>--</td><td>(Decrement): Decreases the value of the variable by 1.</td></tr>
</table>

2.8. Exercises

- *How does write do-while loop with while-do loop?*
- *Cite the primitive types and its range of values?*
- *What is an array of primitive type? Is it an object?*

Chapter III

Classes and Objects

3. Classes and Objects

3.1. Object-Oriented Programming (O.O.P)

Object-oriented programming (OOP) aims to align software design with the way that human people conceptualize and interact with the real world. In contrast to older programming paradigms called procedural, where programmers identify specific computational tasks to address a problem and then formulate a sequence of instructions to accomplish those tasks, OOP centers on the concept of objects. These objects are entities with behaviors and data, capable of interacting with each other. Instead of focusing on tasks, OOP involves the creation of a collection of objects that represent the problem domain in a more intuitive manner. These software objects can correspond to tangible or abstract entities within the problem space.

In our daily lives, we encounter various entities and objects, each with their own characteristics (attributes) and behaviors. For example, people, cars, and books are all objects. These objects have attributes (such as a person's name, age, and address) and can perform actions (such as driving a car, reading a book). Object-oriented programming attempts to abstract the real-world concepts into classes and objects. This abstraction allows programmers to model and manipulate complex systems by breaking them down into manageable components.

The concept of objects and classes in programming is a way to model and represent the entities, attributes, and behaviors of the real world. By using this approach, software can be designed to closely mirror the way we think about and interact with the physical world.

A class in Java is a fundamental template for creating objects, embodying the core principles of object-oriented programming (OOP). It serves as a structural and behavioral model for objects and is a key element in achieving code organization, reusability, and modularity. A class encapsulates data (in the form of fields or instance variables) and behavior (in the form of methods) within a single unit, enabling the creation of objects with shared characteristics and functionalities.

3.2. Declaring Classes

A class is defined using the ***class*** keyword followed by the class name.

```
public class MyClass {
    // Class members (fields and methods) }
```

Inside a class, you can define fields, also known as attributes or variables. These fields represent the properties of objects created from the class declared.

a) Example 1:

```
public class Cars {
    String Model_Name;
    int Seats_Number; }
```

b) Example 2:

```
public class Person {
     String name;
     int age;
     Public void display_Info();
public void display_Info() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
     }}
```

Person	*Class Name*
Name : String **Age** : Integer	*Attributes*
Display_Info()	*methods*

3.3. Variables and method instance

In Java, a class defines the structure and behavior, while objects are instances of that class that hold their own data and state. An object is a self-contained, created based on the blueprint provided by the class.

An object is created using the ***new*** keyword in the following manner:

```
ClassName objectName = new ClassName();
```

`ClassName():` represents a constructor which is a special method that is used to initialize an object(see next section).

a) Example

```
Person person1 = new Person();  // Create the first
person object
Person person2 = new Person();  // Create the second
person object
person1.Name ="Name1";
person1.age = 30;
person2.Name ="Name2";
person2.Name = 32;
person1.display_info();
person2.display_info();
```

Objects are stored in memory as instances of classes accessed by references. These references hold the memory address of the object's location on the heap, which is a region of memory managed by the Java Virtual Machine (JVM). In summary when you work with an object, you are usually using a reference to access the object's data and invoke its methods.

The memory representation of the previous example resembles the following diagram

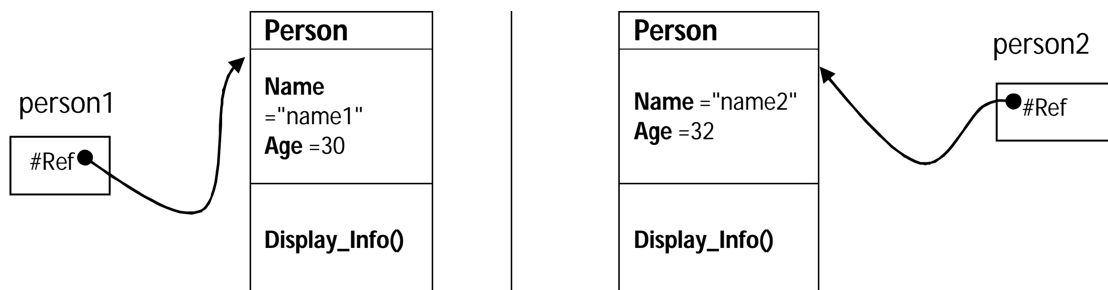


Figure 5 memory representation.

3.3.1. Objects and instances

In Java, a class is a **type**, similar to the primitive types such as **int** and **boolean**. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For **example**, give the class below:

```
public class Student {
public String name;   // Student's name.
public double Exam1, Exam2, Exam3;
public double Average() { // compute the average
   return (Exam1 + Exam2 + Exam3) / 3;}}
```

A program could define a variable named std of type Student with the statement:

```
Student std;
```

However, declaring a variable (***std*** in previous example) does not create an object. A variable can only hold a reference to an object. Therefore, in previous example ***std*** hold a **null** value which mean that there is no reference.

```
Student std = new Student();
```

This statement would create a new object, which is an instance of the class Student, and it would store a reference to that object in the variable ***std.*** in other way, the variable ***std*** refers to the object created when calling the ***new*** statement

3.3.2. Java Methods

In Java, the word method refers to the same of function in other languages. A function is a reusable portion of a program, sometimes called a procedure or subroutine.

A Java method is a group of statements that are collected together to carry out an operation. For example, When you call the System.out.print() method, the system actually performs several hidden instructions in order to display a message on the console.

a) **Declaration:** A method is declared with a method signature, which includes the method's name, return type, and parameters (if any). The basic syntax is as follows:

```
<modifiers> <return-type> <subroutine-name> ( parameter-list ) {
<statements>
}
```

b) **Example:**

```
public double Average() { return (Exam1 + Exam2 + Exam3) / 3;}}
```

modifier   Return-type   Method-Name   Method-body

3.3.3. Modifiers

Modifiers consist of words that set certain characteristics of the methods. It can be: **public**, **private**, **protected** (see next section) that control their visibility. They can also have other modifiers like **static** and **final**. For example transferring the previous method into static method:

```
public static double Average(){return(Exam1 + Exam2 + Exam3) / 3;}}
```

3.3.4. Return value

The return type specifies the type of data that the method will return. If the method is a procedure, then the return-type is replaced by the special value **void**, which means that no value is returned.

3.3.5. Parameters

Parameters are variables that are used to pass values into a method. They are known as method signature and represent the values passed when the method is called. The parameter list in a method can be empty, or it can consist of one or more parameter declarations of the form type parameter-name. If there are several declarations, they are separated by commas.

In Java, parameters are always passed by value (java do not implement the pass by address technique). This means that when you pass a variable to a method, a copy of the variable's value is passed to the method, not the actual variable itself. It is essential to understand the distinction between "pass by value" and "pass by reference."

3.4. Static Methods and Variables

The ***static keyword*** in Java is used for both variables and methods. We can also apply static keyword with blocks and nested classes. The static keyword belongs to the class than an instance of the class.

In the Java programming language, the keyword static means that the particular member belongs to a type itself, rather than to an instance of that type which means that only one instance of that static member is shared across all instances of the class.

Static members are common for all the instances(objects) of the class but non-static members multiple copy for each instance of class

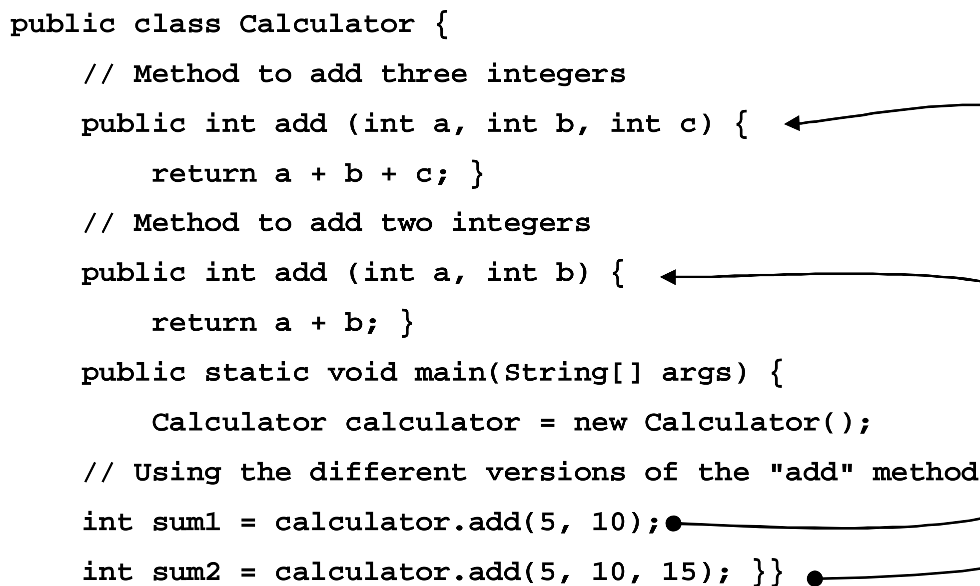
a) Example:

```
private static double variable_Name;    // a static variable
public static double method_Name() { .. };// a static method
```

3.5. Method Overloading

Method Overloading occurs within a class when there are multiple methods sharing the same name, each with distinct functions or parameter types. When a call to an overloaded method is made, the compiler is tasked with selecting the appropriate version based on the number and types of arguments provided. The distinguishing factor in method overloading lies in the parameter list of the methods; the return type alone is insufficient to differentiate between two overloaded methods. If the compiler identifies two method declarations with the same name and parameter list but differing return types, an error will be generated.

a) Example



```
public class Calculator {
    // Method to add three integers
    public int add (int a, int b, int c) {
        return a + b + c; }
    // Method to add two integers
    public int add (int a, int b) {
        return a + b; }
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
    // Using the different versions of the "add" method
    int sum1 = calculator.add(5, 10);
    int sum2 = calculator.add(5, 10, 15); }}
```

3.6. Java access rights

Java provides four types of Java access modifiers which insures a robust way to accessing members of a class. The use of modifiers in Java is fundamental for building a secure and flexible software, they serve several crucial purposes like controlling access, enhancing encapsulation, supporting inheritance, organizing code, etc.

Access level modifiers decide whether other classes can use a particular attribute or calling a particular method. The table below summarizes the access rights used in java

Table 5 Java modifiers

Modifier	**in Class**	**in Package**	**Sub-class**	**Outside**
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
no modifier(default)	Y	Y	N	N
Private	Y	N	N	N

The **public** modifier in Java can be applied to classes, methods, and data fields, indicating that they are accessible from any other classes. In the absence of a visibility modifier, classes, methods, and data fields are, by **default**, accessible by any class within the same package.

Apart from the public and default visibility modifiers, Java includes the **private** and protected modifiers for class members. The private modifier restricts access to methods and data fields exclusively within their own class while **protected** modifier denotes that the member can only be accessed within its own package and, by a subclass of its class in another package.

3.7. Encapsulation in java

In Java, encapsulation is a robust technique that consolidates a class's data members and methods into a single entity. This functionality allows the protecting of a class's internal implementation details from external access, exposing solely a public interface for interaction with the class.

To implement encapsulation in Java, it involves declaring the instance variables of a class as private, restricting their access to within the class. If external access to these variables is desired, you can establish public methods, commonly known as getters and setters, to retrieve and modify the variable values, respectively. These getter and setter methods not only facilitate external interaction with the class but also enable the class to enforce its data validation rules, ensuring the maintenance of a consistent internal state.

a) Example

```
public class Class_1 {
    private int  x ;
    public int  getX () { return x ; }
    public void  setX (int v) { x = v  ; } }
    public class Class_2 {
    public static void main(String[] args) {
     Class_1  C  =  new Class_1();
          C.x = 2 ;        // caused an error since x is private
          C.setX(2);     // the correct form
       }}
```

3.8. Java Constructor and Destructor

In object programming field, primitive types and object types has important distinctions. The act of declaring a primitive variable is just reserving a memory case in the heap after then working with it using her name assigned explicitly before, however with a class as type of variable generate an instance of that class. So, the objects must be constructed before using it. In the second step and after working with the object, other mechanism comes to destroy the object.

The process of constructing an object passes by first, finding some available memory in the heap that can be used to store the object and, second, putting in the variable object (called instance) the reference that refers to the reserved memory.

Constructors in java are similar to methods, which are used to initialize an object. They have not any return value (even void return). The using of ***new*** statement create an object and implicitly the constructor is called. The constructor method is always exist in given class, which means when we do not create a constructor, then Java compiler create a default constructor with empty parameters. when the programmer create its own constructor, the default constructor do not exist anywhere.

In order to create constructor you must assign to it the same name of the class with out return type.

a) **Example 1**

```
public class Class_1 {
private int  x ;
public int  getX () { return x ; }
public void  setX (int v) { x = v  ; } }
public class Class_2 {
public static void main(String[] args) {
Class_1  C  =  new Class_1(); // default
constructor was called
C.setX(2); }}
```

b) **Example 2**

```
public class Class_1 {
private int  x ;
public int  Class_1 () {  ...  } // Constructor defined explicitly
public int  getX () { return x ; }
public void  setX (int v) { x = v  ; } }
public class Class_2 {
public static void main(String[] args) {
Class_1  C  =  new Class_1(); // the constructor created was called
C.setX(2); }}
```

3.9. Accessor and mutator

Accessor and mutator methods are two significant concepts associated to encapsulation in Java. In object-oriented paradism, encapsulation stands as a fundamental principle, it denotes to the act of hiding the implementation of an object and providing one way to access on its attributes and methods. Other major benefits to the using these methods is to add validation to the process of get-method and set-method. For example, ensure that the age value is within a certain range or add text formatting tothe person's name.

3.9.1. Accessor

Accessor methods, commonly known as getter methods, are methods that allow you to get the value of an object's private attributes. These functions exclusively offer a read-only

access to the object's attribute. The use of accessor methods ensure that the object's variable not modified accidentally by outside code.

In the following Example the getter method GetName() ensure that the returned name is converted to capital letter

```
public class Class_1 {
private int  age ;
private String  Name ;
public String  GetName () { return name.toUpperCase() ; }}
```

3.9.2. Mutator

Mutator methods, commonly known as setter methods, are methods that permit you to change the value of an object's private variables. These methods offer write-only access to the object's state. The use of mutator methods ensure that the object's state is changed only by a controlled interface.

Let's take the previous example with extending it by following setter method ***GetAge***. This method ensures that the validate value is between 0 and 100.

```
public class Class_1 {
private int  age ;
private String  Name ;
public String  GetName () { return name.toUpperCase() ; }
public void   SetAge (int a) {  if ( a  <= 100) ||  (a > 0)
                               { age = a;
                                 }}
```

3.10. Array of objects in java

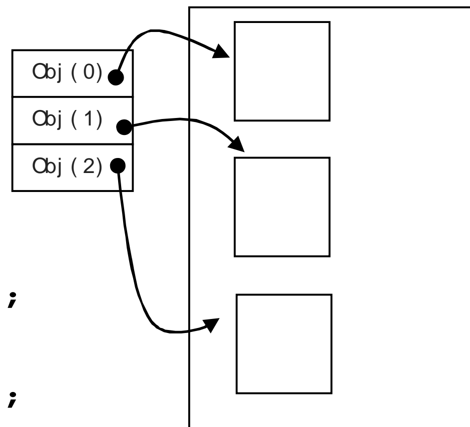
In java, an array is a assembly of variables that have similar type. An array can consist of both primitive (int, char, etc.) data-type elements and non primitive (Object) references of a class. An object is an instance of a class that describe methods and variables. Any object created holds a space memory referenced by a memory adress.

By the same way array creation In java, We can create an array of objects, the example below illustrates this idea

```
class TestObject{
int a;
int b;
public void setValue(int c, int d){ a=c;  b=d; }
public void showValue(){
System.out.println("Value of a ="+a);
System.out.println("Value of b ="+b); }}
```

Now, Create a class that makes three instances of TestObject which stored in an array

```
class TestArray{
public static void main(String args[]){
TestObject obj[] = new [3] ;
obj[0].setValue(1,2);
obj[1].setValue(3,4);
obj[2].setValue(5,6);
System.out.println("For Array Element 0");
obj[0].showValue();
System.out.println("For Array Element 1");
obj[1].showValue ();
System.out.println("For Array Element 2");
obj[2].showValue(); }}
```



Chapter IV

Inheritance, polymorphism and Abstract class

4. Inheritance, polymorphism and Abstract class

A class denotes a set of objects that encapsulates static characteristics (variables) and dynamic characteristics (methods), serving as a model for creating objects. It acts as a template, defining the structure of objects by specifying variables and governing their actions through methods. In the context of oriented object programming, a class can have a superclass, which leads to create a hierarchical relationship where it inherits attributes and behaviors from another class.

4.1. Inheritance

Inheritance is an important object-oriented programming concept in which one class gets the properties and behavior of another class. It establishes a IS-A relationship between two classes. This relationship is common known as a parent-child relationship. Inheritance is a mechanism wherein one class (called **child-class**) inherits the attributes and methods of another class (called **super-class**) with the gain that this class has the ability to update the behavior of her super class.

In Java, classes are organized into a hierarchy, where a superclass can pass on its attributes and methods to a subclass. Java supports single inheritance and the multiple inheritance (inheritance from more than one class) is not supported in the traditional sense. In java, a class can extend only one superclass. This design choice was made to avoid the complications and ambiguities associated with multiple inheritance. However, Java supports the multiple inheritance through interfaces. An interface in Java is a collection of abstract methods, and a class can implement multiple interfaces. By implementing multiple interfaces, a class can inherit the abstract methods declared in each interface.

4.1.1. Syntax

The syntax for inheritance in Java can be made by using ***extends*** keyword in order to indicate that a class is inheriting from another class.

```
class ChildClass extends ParentClass {
    // Fields and methods specific to the child class
}
```

a) Example :

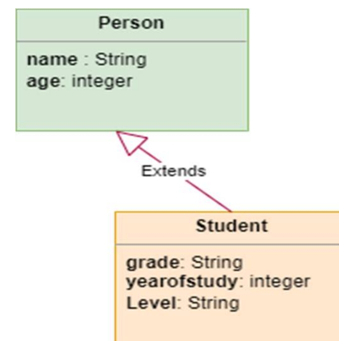
Let's consider a "Person" class that serves as a representation of real-life humans.

```
class person {
```

```
public String name;
public int Age; }
```

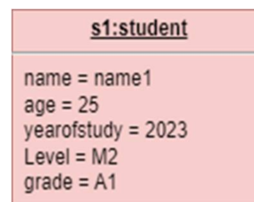
Now, we want to create another class, which named ***student*** that represents a student in such university. We know that a student is a person, implying that this new class can inherit all characteristics of person class

```
class student extends person {
    public String grade;
    public int yearOfStudy;
    public String Level; }
```



An objects of student class is fulfilled as shown below:

```
class Main{
  public static void main(String[] args) {
    student s1 = new student();
    s1.name = "name1";
    s1.age = 25;
    s1.grade = "A1";
    s1.yearofstudy = 2023;
    s1.Level = "M2" ;}
```



4.2. Inheritance and constructor

If one class (subclass) extends another class (superclass), the ***super*** keyword is used as reference variable when a class (subclass) extends another class (superclass). It plays an important role in situations concerning inheritance. Here is a brief overview of how the ***super*** keyword is used:

4.2.1. Invoking Superclass Constructors: to ensure an appropriate initialization of the object with both own attributes and inherited attributes, the ***super*** keyword is used to invoke the constructor of the superclass from the constructor of the subclass.

4.2.2. Calling Superclass Methods: to allow the execution of the superclass method version before or after performing additional actions in the subclass, the **super** keyword used for this purpose while calling methods from the superclass.

Let us extend our previous example to demonstrate the use of constructors and inheritance in Java. We extend the person class with a constructor that have two parameters and invoking it in student class to have a proper initialization of any object of student class.

```
class person {
  public String name;
  public int Age;
  person(String name, int Age){
          this.name = name; this.age = age}}
class student extends person {
public String grade;
public int yearOfStudy;
public String Level; }
student(string name, int Age, String grade, int yearofstudy, String
Level){
super(name,age); // invoking the constructor of superclass
this.grade = grade ; this.yearofstudy = yearofstudy;
this.Level=Level }}
```

NB: The keyword ***'this'*** is employed to indicate the current instance, it acts as a reference variable that is used to refer to the current instance of the object within which it appears. It is often used to distinguish instance variables from local variables when they use the same name, similarly it is used to invoke current object's method.

4.3. Polymorphism

Polymorphism is an important concept of object-oriented programming. It just means the ability to take more than one form by an object. Polymorphism allows us to create flexible code. We can achieve polymorphism in Java using: ***overloading and overriding*** techniques. The example below explains why we need for polymorphism model, its shows a generic class "Shape" and the its child's classes "Circle and rectangle"

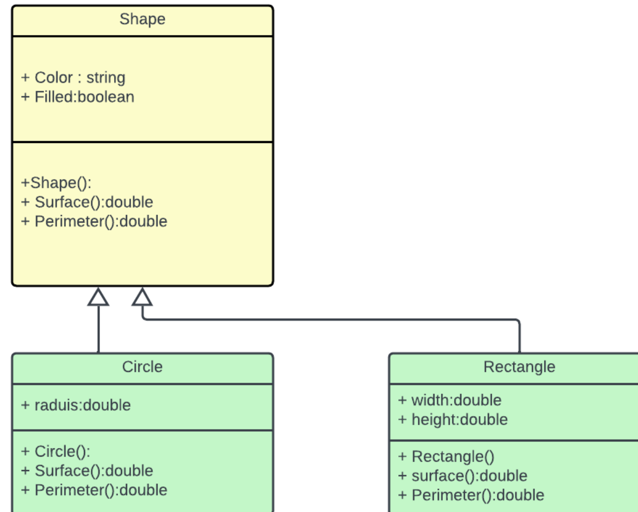


Figure 6 inheritance and polymorphism

4.3.1. Method overloading:

Method overloading which is introduced in previous chapter, its permits to create more than method with the same name and different parameter lists within cl ass. We explain how compiler determines which method to invoke based on the arguments passed during a method call.

4.3.2. Method Overriding:

Another crucial approach for achieving polymorphism is method overriding, wherein a subclass redefines the implementation of a method that already exists in its superclass. This permits the modification of a method in a subclass while maintaining its existence in the superclass. In general, any subclass can override any method from a superclass unless the method is explicitly marked with the ***final*** or ***static*** keywords. It is essential to note that the overriding method must retain the same name and parameter list as the overridden method.

In previous example the inherited methods ***surface()*** and ***perimeter()*** for both circle and rectangle which representing the calculations for the surface area and perimeter, respectively, have not the same implementation, each class had the own version of these methods. So, in order to suit the specific formulas for both circles Rectangles. Similarly, these classes can override these methods with formulas appropriate for any object. This approach allows for a specialized implementation of geometric calculations tailored to the characteristics of each shape while following to the common definition in the Shape superclass.

```
Class shap{  …
      Public surface() { retrun …}
      Public perimeter() { retrun …}}
Class Circle extends shape{ …
      Public surface() { retrun Math.PI * Math.pow(radius, 2);;}
      Public perimeter() { retrun 2 * Math.PI * radius;}}
Class rectangle extends shape { …
      Public surface() { retrun width * height}
      Public perimeter() { retrun (width + height)*2 }}
```

4.4. Abstract Class and method

An abstract class in Java serves as a model for other classes that cannot be instantiated, it is destined to be a super class. Generally, it is designed to provide a common structure for its subclasses. An abstract class can hold both abstract methods (methods without any code) and concrete methods (methods with code), and can have attributes (fields or member variables) just like regular classes.

The main purpose of abstraction is to define a common interface and behavior for a group of related classes. It allows developers to focus on the relevant aspects of a system without paying attention to unnecessary complexities. This promotes code reusability, as the common code is centralized in the abstract class, and it establishes a clear hierarchy among related classes.

Abstraction is a major key factor in achieving polymorphism (previous section). Through abstract classes and interfaces, different objects can be treated consistently based on their common abstractions.

An abstract method is a method that have not any code and is defined using the abstract keyword (like abstract class), it is designed for implementation in subclasses which their implementations must be provided by any class that extends the abstract class.

In the previous example, the surface and perimeter methods of the shape class must be abstract method because we cannot calculate the surface and perimeter of unknown form. This method must be left blank. However, their existence is vital because all geometric from derived from shape class have these characteristics and can be calculated using the appropriate formulas for each kind of shape.

a) **Example:**

```
abstract class Shape { // abstract subclass (Circle)
   abstract double surface();
   void display() { System.out.println("This is a shape.");}}

class Circle extends Shape {
    // Implementation of abstract method for calculating area
    @Override
    double surface() { return Math.PI * Math.pow(radius, 2); } }
class Rectangle extends Shape {
    // Implementation of abstract method for calculating area
    @Override
    double surface() { return length * width; }}
```

Note: You cannot instantiate an object of the shape class using ***new shape()***. This will result a compilation error because an abstract class cannot be directly instantiated.

4.5. Interface

Interfaces are an essential part of object-oriented paradigm, they serve several purposes particularly in realizing abstraction (see previous section). Straightforwardly, an interface is a collection of abstract methods. It allows defining a set of abstract methods that must be all implemented by the class that implements the interface. An interface can also include constants (public, static, final fields) and default methods (methods with a default implementation) note that the data variables are not allowed in an interface.

a) **Syntax:** Writing an interface is similar to writing a class; the only difference is replacing the ***"class"*** keyword with the ***"interface"*** keyword.

```
public interface Name_Of_Interface {
 //constant, final, static fields
 // abstract method declarations   }
```

b) **Example:**

Writing an interface that contains a constant and two abstract methods

```
public interface Example_Of_Interface {
     int MY_CONST = 42; // Implicitly public, static, and final
     void myMethod1();
     void myMethod2(); }
```

Now, writing a class that implements this interface. This class must implement all abstract methods of the interface. To do this, we use the ***“implements”*** keyword like below:

```
public class Example_Of_class implements Example_Of_Interface  {
     public void myMethod1() { System.out.println(“method1”)};
     public void myMethod2() { System.out.println(“method1”)};  }
```

4.6. Overload (method surcharging)

In Java, the term "surcharge" is usually used to denote to method overloading (see chapter 3). Method overloading allows a class to have multiple methods with the same name but with different signature (parameter lists). The different versions of the method are called "overloaded."

4.7. Modifiers

Modifiers are keywords that you used to control the access of variables, methods and other program elements of classes (seen in chapter 3):

Here is some modifiers that can used in access control:

- **public**: Accessible from anywhere. There is no restriction on access.
- **protected**: Accessible within the same package and by subclasses, even if they are in a different package.
- **default** (no modifier): Accessible within the same package only. This is also known as package-private.
- **private**: Accessible only within the same class.
- **static :** Used to specify that a is a variable class and belongs to the class itself, or method is a method class
- **final:** in Java, the final modifier is used for variable, method or class. it act like a constant in variable which means that you cannot changing the value of this variables, for method, the final modifier is used to indicate that this method cannot be override and finally this modifier is used with class to specify that the class cannot be inherited. In summary, this modifier is used to apply restrictions on the use of a class, method, or variable.

Chapter V

Integrated Generic data structure in java

5. Integrated Generic data structure in java

A data structure is an organized collection of data that not only stores information but also facilitates operations for accessing and manipulating the stored data.

In object-oriented programming, everything must be depicted as objects (except java primitive type), including data structures, which are objects designed to store other objects. Defining a data structure essentially involves defining a class. The class representing a data structure should use the data fields for storing information and provide methods to facilitate operations such as searching, insertion, and deletion.

5.1 Collection

Java's generic data structures can be categorized into two classes: ***collections*** and ***maps***. A collection represents a group of objects. A map is a key-value association between objects in one set with objects in another set.

Collections and maps are represented by the interfaces Collection<T> and Map<T,S>. With, "T" and "S" stand for a given object type.

5.1.1 Interfaces and implementation

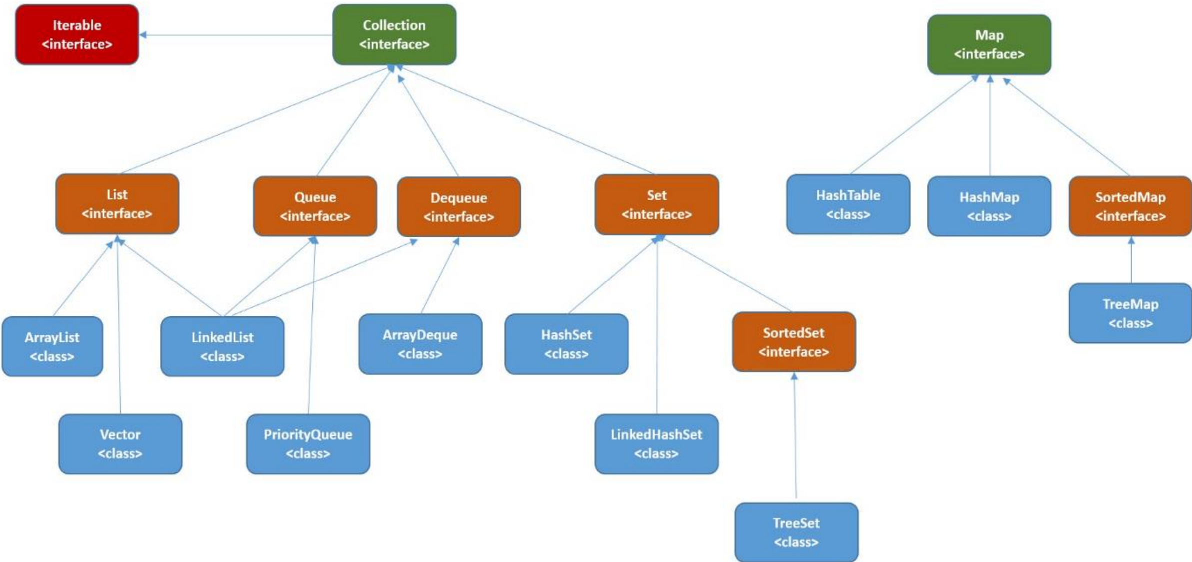


Figure 7 data collection

As seen in the previous chapter, an interface is a set of abstract methods that must be implemented by any class intending to implement this interface. Additionally, the interface can be contains some predefined methods and variables.

In Java, generic data structures like **map** and **collection** are represented as interfaces in a package called **util**. So, to use them, you must import this package in your worksheet (import all components) by writing ***import java.util.****. You can specify what you would like to import. If you want to import one thing. Example **import java.util.map** for using only the map interface.

As we see in above figure, this family of related interfaces and classes is called ***Java* collections *framework***

5.1.2. Generic types in Java

Defining classes that utilize the generic type construct introduced in Java 5.0 requires adopting a new syntax for referencing the class name. These classes and interfaces, which include those in the collections framework, utilize angle brackets that enclose one or more variables (separated by commas) to denote unspecified type names. For instance, you might employ <*E*> or <*K,V*> to signify unspecified type names. Consequently, the names of classes or interfaces implemented with generic types are expressed using the syntax ClassName<*E*>.

5.1.2.1. Set interface

In java, the Set interface is a part of the Java Collections Framework, which defined in the java.util package. It provides a way to store and manipulate non-duplicate elements with any guarantee of the element order. Java provides three implementations of the Set interface in the java.util package: ***HashSet, LinkedHashSet*** and ***TreeSet.***

The Common Methods provided in set interface are:

boolean add(E e): add method perform the addition of the specified element to the set.

boolean remove(Object o): Removes the specified element from the set.

boolean contains(Object o): is boolean method that returns true if the set contains the element.

int size(): it determine the number of elements in the set.

boolean isEmpty(): Returns true if the set is empty.

void clear(): perform a deletion of all elements.

5.1.2.1.1. *HashSet*

A ***HashSet*** implements the Set interface, like a hash table assuming a good hash function. It does not allow duplicate elements, and it does not guarantee the order of elements.

```
import java.util.HashSet;
import java.util.Set;
public class Example {
public static void main(String[] args) {
Set<String> cars = new HashSet<>();
cars.add("Renault");
cars.add("BMW");
cars.add("peugeot");
cars.add("BMW"); // Duplicate, will not be added
cars.add("Mazda");
// Displaying elements (order may vary)
System.out.println(cars);
// Removing an element
cars.remove(5);
```

The result of this example is `[Renault, Mazda, BMW, peugeot]`

5.1.2.1.2. *LinkedHashSet:*

linkedHashSet is a class in Java that extends HashSet and implements the Set interface. It is part of the java.util package and a member of the Java Collections Framework. LinkedHashSet is intended to preserve the order of insertion, which means that it guarantee the order in which elements were added to the set.

Let us consider the previous example:

```
import java.util.HashSet;
import java.util.Set;
public class Example {
public static void main(String[] args) {
Set<String> cars = new LinkedHashSet<>();
cars.add("Renault");
```

```
cars.add("BMW");
cars.add("peugeot");
cars.add("BMW"); // Duplicate, will not be added
cars.add("Mazda");
// Displaying elements (order may vary)
System.out.println(cars);
```

The output of this example is `[Renault, BMW, Peugeot, Mazda]` which illustrates that the order in which elements were inserted into the LinkedHashSet is maintained.

5.1.2.3. List

A list is a common data structure for storing data in sequential order—for example, a list of available books, a list of students, list of cities …etc. Java provides several operations on lists based on their index positions such as insertion, deletion and retrieval.

To implement a list, you can use one of two methods: dynamic or static. Static method is to use an array to store the elements with fixed size. Dynamic approach is to use a linked structure, which means that each node is dynamically created to store an element. All the nodes are related to form a list.

There are several implementation of list interface such as ArrayList, and LinkedList. We will use ArrayList for example.

5.1.2.3.1 ArrayList

The ArrayList<***E***> class, which is the most popular generic type of the Java collections framework, has a generic type implementation in Java 5.0. The ArrayList class covers the methods for accessing, retrieving and storing objects by using their index position within the ArrayList form. The ***E*** refers to the name of an interface or class. This type must be specified before calling the constructor.

a) Example

```
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo {
public static void main(String args[]) {
// declaration of arraylist of Strings
List<String> stringlist = new ArrayList<>();
```

```
stringlist.add("Un");
stringlist.add("deux");
String str = stringlist.get(0);
System.out.println("The 0th element is " + str);
// print the ArrayLists without a loop!
System.out.println(stringlist);}}
```

5.1.2.4. Map

Interface map is presented in ***java.util.Map*** package, which includes the common methods (get and put) for working with maps as in other programming languages. The main purpose of this structure is to define the key-value relationship. The key must be unique, and it is used to recover the corresponding value.

The map interface, is parameterized by two objects T and S Map<T,S>. The first parameter, T, indicates the objects type that is considered as a key in the map; the second parameter, S, specifies the objects type that is considered as a value in the map.

Java offers some implementations of the Map interface, with different characteristics. Some common implementations are:

HashMap: Similar to a ***HashSet***, this implementation does not ensure any specific order. It is designed for fast access and insertion of key-value pairs.

TreeMap: keeps its keys in a sorted order. It provides special methods that make it easy to navigate through the keys.

LinkedHashMap: keeps keys in the order of putting them in, which allows a fast access and known iteration order.

Consequently, java provides several methods that working with the map such as:

- ***values():*** Returns all Collection values in the map
- ***get(key):*** Returns the value associated with the specified key.
- ***put(key, value)***: sets the specified value with the specified key.
- ***remove(key):*** Removes the key and its associated value from the map.

a) Example

```
import java.util.*;
public class NumberCityMapExample {
    public static void main(String[] args) {
        Map<Integer, String> numberCityMap = new HashMap<>();
        numberCityMap.put(1, "Adrar");
        numberCityMap.put(28, "Msila");
        numberCityMap.put(19, " setif");
        numberCityMap.put(5, " batna");
        String cityForNumber2 = numberCityMap.get(5);
        System.out.println("City for number 5: " + cityForNumber2);
        // result = City for number 5: batna
        System.out.println("All willaya are:");
        for      (Map.Entry<Integer,      String>      entry      :
numberCityMap.entrySet()) {
            int number = entry.getKey();
            String city = entry.getValue();
            System.out.println("Number " + number + ": " + city);
        }}}// iterating over key-value pairs
```

5.1.2.5. Queue

In Java, the abstraction of FIFO (First-In-First-Out) structure is represented as staructure that called **queue.** The queue is a data structure that follows this principle, which mean that the first element stored in the queue will be the first one to be deleted.

Here are some Basic Methods provided by Java Queue interface:

add(element) or ***offer(element)***: Adds an element to the end of the queue.

remove() or ***poll()***: Removes and returns the element from the front of the queue.

element() or ***peek()***: Retrieves, but does not remove, the element at the front of the queue.

Java presents the Queue interface in the java.util package, and provides several classes implementing this interface, such as LinkedList and PriorityQueue.

a) **LinkedList:** a LinkedList is a class that implements the **List** interface (as seen in the previous section) and ***Deque*** interface. It is a linked list data structure, also known as a chaining list. This means that each element in the list contains two references: one to the previous element and another to the next element.

b) **PriorityQueue:** A ***PriorityQueue*** is a data structure in Java that organizes elements based on their priority. Elements with higher priority are dequeued before those with lower priority. It is typically implemented as a binary heap, ensuring efficient insertion and removal of the highest-priority element.

c) **Example**

```
import java.util.LinkedList;
import java.util.Queue;
public class SimpleQueueExample {
    public static void main(String[] args) {
        // Creating a Queue using LinkedList
        Queue<String> myQueue = new LinkedList<>();
        // Adding elements to the queue using add
        myQueue.add("Element1");
        myQueue.add("Element2");
        myQueue.add("Element3");

        // Removing elements from the front of the queue using remove
        while (!myQueue.isEmpty()) {
          String removedElement = myQueue.remove();
          System.out.println("Removed from the front: " +
removedElement);}}}
```

5.2. Collection Algorithms

The Java Collections Framework provides certain algorithms as static methods that can be applied to collections and map objects. These collection algorithms are proposed in a class called ***Collections*** within the ***java.util*** package. All these algorithms are better to use them to enhance the coding process.

5.2.1. Fill

The Collections.fill() method is particularly useful when you want to initialize or reset the values of a collection, which it doesn't return any value (void). It is used to replace all elements of a specified collection with a given object value.

a) Example

Let us consider a vector, which is a collection type that can be used in java

```
import java.util.Collections;
import java.util.Vector;
public class VectorInitializationExample {
    public static void main(String[] args) {
        // Creating a Vector
        Vector<integer> myVector = new Vector<>();
        // Specifying the size of the vector
        int vectorSize = 5;
        // Initializing the Vector with a default value
        String defaultValue = 0;
        // Filling the Vector with the default value
        Collections.fill(myVector, defaultValue);}}
```

5.2.3. Sort

sort(List<T> list): Sorts the specified list into ascending order.

a) Example:

```
List<Integer> nbers = Arrays.asList(5, 7, 4, 5, 9, 6, 2, 0, 7);
Collections.sort(nbers);
System.out.println("Sorted List: " + nbers);
```

5.2.4. Shuffle:

shuffle(List<?> list): Randomly permutes elements of a list

a) Example:

```
List<String> Dayname = Arrays.asList("Saturday", "Monday",
"Friday",);
 Collections.shuffle(Dayname);
System.out.println("Shuffled List: " + Dayname);
```

5.2.5. Reverse Order:

reverse(List<?> list): Reverses the order of the elements in the specified list.

a) Example:

```
List<String> names = Arrays.asList("name1", " name2", " name3");
Collections.reverse(names);
System.out.println("Reversed List: " + names);
```

5.2.6. Binary Search:

binarySearch(List<? extends Comparable<? super T>> list, T key): Searches for the specified element in the specified sorted list using a binary search algorithm.

a) Example:

```
List<Integer> nbers = Arrays.asList(5, 7, 4, 5, 9, 6, 2, 0, 7);
int key = 2;
Collections.sort(nbers); // sorting the collection
int index = Collections.binarySearch(nbers, key);
System.out.println("Sorted List: " + nbers);
```

5.3. Collections' iterating

In Java, iterating a collection (such as a set, list, or map) can be assured by using numerous methods. The wish of operations and the type of collection are two factors that can be influencing on a selected method. The common used techniques for iterating a collection are:

5.3.1. For-each

The ***for-each*** loop is an easy and elegant way for iterating a collections. It is applicable to classes that implement the ***Iterable*** interface

a) Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachLoopExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
```

```
fruits.add("Orange");
// Using for-each loop to iterate over the elements of list
for (String fruit : fruits) {
    System.out.println(fruit); } } }
```

5.3.2. Iterator

The ***Iterator interface*** allow us a sequential iteration of elements in a collection. You can use the ***iterator()*** constructor to obtain an Iterator instance and iterate through the collection.

a) Example

```
List<String> list = Arrays.asList("hello", "world", "!");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String elm = iterator.next();}
```

5.3.3. Massive operations in Java

Massive or Large-scale operations refer to the management and processing of massive quantities of data within Java programs. When performing with Large-scale operations or datasets, some considerations and strategies must be adhered in order to guarantee efficiency performance. Some practices are:

a) **Concurrency and Parallelism:** the parallelism and concurrency programming paradigms are often used to accelerate the processing of vast amounts of data. Java provides structures such as ***threads*** and classes of ***java.util.concurrent*** package to work with parallel execution.
b) **Efficient Algorithms**: In vast amounts of data, some algorithms are not efficient, especially when working to sorting or selecting data. The developers should be chosen the algorithms based on specific requirements. Some algorithms are more efficient for large datasets that provided by ***java.util.Collections*** package like ***sort*** that use a confirmed efficiency algorithms.
c) **Memory supervision**: manage usage memory is essential when dealing with vast amounts of data. For example, the creation of unnecessary temporary objects can reduce pressure on the garbage collector, which leading to improve the performance.

Chapter VI

Graphical API

6. Graphical API

6.1. Standard Java library API

The Java ***application program interface (API)*** also known as ***library*** provides classes for developing predefined classes and interfaces for developing Java applications, it consists of a set of separate programs. Java is a powerful language that can be used in many domains: you can use it to develop applications for servers, desktop computers, and small devices. It comes in three editions:

a) **Java Standard Edition (Java SE):** used to develop client-side applications. The applications can run as a standalone program.
b) **Java Enterprise Edition (Java EE) :** used for server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
c) **Java Micro Edition (Java ME)** : designed for mobile devices applications, such as sensors, cell phones.

Java SE also known as ***Java Standard Library*** is a highly utilized platform for creating diverse applications relating to diverse domains like desktop applications, web applications and enterprise systems. It remains a popular choice among developers due to its portability, reliability, and wide community support.

Java SE includes many features such as:

- **Java Virtual Machine (JVM):** provides a robust executing environment for running Java code.
- **Versatile Libraries:** Java SE offers in a comprehensive way a set of standard libraries, known as the Java API (Application Programming Interface). These libraries provide essential functionality for tasks such as input/output, networking, data structures, utilities, and more.
- **Security:** Java SE provides a solid security model to ensure the safe execution of java code.
- **Development Tools:** Java SE delivers a set of development tools such as Java Development Kit (JDK) and Integrated Development Environment (IDE).
- **Updates and Versions:** regular updates is a key feature that offered by Java SE which receives updates and new versions, enhancements, bug fixes, and more.

The libraries in Java (standard API) encompass a comprehensive set of predefined classes and packages for a large variety of programming tasks. Here are some key features of the Core libraries:

java.lang: includes classes for basic data types, exceptions, threads, and the Object class. This package contains classes that are fundamental to the design of the Java programs. This package is automatically imported into any Java program.

java.util: (as seen in previous chapter) The **java.util** package provides classes, packages and data structures such as lists, sets, maps, queues, and more data collection. Developers can also working with dates and times by using the classes ***Date*** and ***Calendar*** provided in this package.

java.security: includes classes for cryptography and more security model like secure random number. The ***java.security*** package contains classes related to Java's security model.

java.text: includes classes for parsing and formatting text, dates, and numbers. This package is suitable for internationalization of Java programs.

java.io: basics input and output operations are included in this package, which allow developers to read and write files, streams, and other input/output sources. It includes file I/O, stream I/O, and object serialization.

6.2. Java Natives Interface (JNI)

Native code means that we can use other language code like C in our java programs. The need for using native code is capital essentially in:

- An existing library in other language that we want to reuse without to rewrite it in java
- The need to manage some hardware
- When performance optimization is required …etc.

To use the native code, JDK offers a bridge between the bytecode and the native code (usually written in C or C++).

The bridge provided by java has three steps that must be respected to achieve the correctness of running code:

Step 1: Write a Java Class that include a Native Method

Step 2: Generate Header File (*.h)

Step 3: Compile the C Code

Example:

Let us take an example that use the *add library* written in c language

```
// AddNum.c
#include <stdio.h>
int add(int a, int b) {
    return a + b;  }
```

Step1: Write a Java Class:

```
public class AddJNI {
    static {
        // Load the native library
        System.loadLibrary("Addnum"); }
    // Declare the native method
    public native void add();
    public static void main(String[] args) {
        // Create an instance of the class
        AddJNI addnumbers = new AddJNI();
        // Call the native method
        Float result = addnumbers.add(2,3);}}
```

Step2: Generate Header File

Use the ***javac*** to compile the Java class and the ***javah*** tool to generate the header file.

```
javac AddJNI.java
javah -jni AddNum
```

This will generate a header file named `AddNum`.***h***

Step 3: Compile the C Code

Use the gcc compiler to compile the addNum.c. This will result a file named addNum.dll

Note: we use the path `C:\jniexample\` as example :

```
C:\ gcc -shared -o C:\jniexample\addnum.dll
```

Running the Java Program

Use the java application to run our example

```
C:\ java addJni
```

This command will load the native library (addNum.dll) and call the native method.

6.3. AWT and SWINGx library

In java, the Abstract Window Toolkit (AWT) and Swing are two important package that provide the main API for graphics programming. These libraries offer classes and interfaces for creating graphical user interfaces (GUIs), drawing graphics, and handling events.

6.3.1. AWT (Abstract Window Toolkit): in this library, we found the most used classes for working in graphics mode like color, font …etc.

java.awt.Graphics: provides methods for drawing shapes, text, and images on the screen. It is used when calling the paint() method of components.

java.awt.Color: utilized for working with colors in the RGB model (red, green, blue).

java.awt.Font: Represents fonts for rendering text.

java.awt.Canvas: A space that can be drawn.

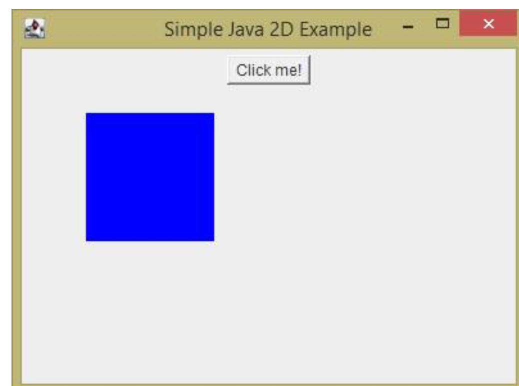
java.awt.Frame and **java.awt.Panel**: Basic containers for organizing components in a GUI.

a) **Example**

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Button;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class cl1 extends JFrame {

    public cl1() {
        super("Simple Java 2D Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        add(new MyPanel());   }

    public static void main(String[] args) {
         cl1 example = new cl1();
        example.setVisible(true); }

    class MyPanel extends JPanel {
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
             Button button = new Button("Click me!");
              add(button);
             Graphics2D g2d = (Graphics2D) g;
             g2d.setColor(Color.BLUE);
             g2d.fillRect(50, 50, 100, 100);}}}
```

This example result:



6.3.2. Swingx:

SwingX offers several components, utilities, and enhancements to the core Swing library, which developers can create rich and complex user interfaces (GUI).

javax.swing.JFrame: A top-level container for Swing applications.

javax.swing.JPanel: A container that can hold other components and can be used for grouping components.

javax.swing.JButton, javax.swing.JLabel, etc.: Several components for creating GUI elements like buttons, labels, and other.

javax.swing.JComponent: The parent class for all Swing components.

To import Swing library use the line command below:

```
import javax.swing.*;
```

Alternatively, if you want to import only one class, you must indicate the name of the package being imported:

```
import javax.swing.JFrame;
```

a) Example

Simple Java program that creates a window that contains a button using Swing:

```
import javax.swing.JFrame;
import javax.swing.JButton;

public class SimpleSwingExample {

    public static void main(String[] args) {
        // Create a JFrame (window)
        JFrame frame = new JFrame("Simple Swing Example");

        // Set the default close operation
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a JButton
        JButton button = new JButton("Click Me");

        // Add the button to the content pane of the frame
        frame.getContentPane().add(button);

        // Set the size of the frame
        frame.setSize(300, 200);

        // Set the frame to be visible
        frame.setVisible(true);
    } }
```