2025 Edition

Core Java

Your Ultimate Learning Guide

Kushal Prasad Joshi

kushalprasadjoshi@gmail.com

PREFACE

Welcome to this handbook, designed to guide learners, practitioners, and enthusiasts through the fundamentals and advanced concepts of **Java programming**. Curated from textbooks, tutorials, and real-world applications, the material balances clear explanations with hands-on examples and practical tips, ensuring you can apply what you learn from day one.

Each chapter builds on industry-recognized principles and standards, offering step-by-step walkthroughs, relevant code samples, and exercises that invite active problem-solving. Whether you are taking your first steps in Java or aiming to deepen your expertise, you'll find content tailored to your pace and interests.

Throughout the handbook, you'll discover:

- **Conceptual Foundations:** Core syntax, data types, and object-oriented principles
- **Practical Examples:** Realistic scenarios that demonstrate how to translate theory into working code
- **Best Practices:** Insights on writing clean, maintainable, and efficient Java programs
- **Exercises & Challenges:** Opportunities to test your understanding and reinforce key ideas

I encourage you to experiment with the provided examples, explore beyond the exercises, and adapt techniques to your own projects. Mastery comes through curiosity and consistent practice, and this handbook is here to support you at every step.

GitHub Repo: This handbook is part of the core-java GitHub repository, which houses all the source code, notes, and examples you need. To dive in and learn by doing, fork the repository into your own account and start experimenting locally. Repository URL: https://github.com/kushalprasadjoshi/core-java

Thank you for choosing this resource. May it inspire your creativity, strengthen your problem-solving skills, and accelerate your journey toward Java proficiency.

Happy coding!

Kushal Prasad Joshi

kushalprasadjoshi@gmail.com

Author

TABLE OF CONTENTS

Preface 1

Chap 01 – Introduction 5

- How Java Works? 5
- Setting Up Our Computer 6
- Components of Java Program 7
- Basic Java Program 7
- Comments 8
- View Java Documentation 9
- Variables in Java 9
- Data Types in Java 9
- Scanning and Printing 12
- Exercise 14

Chap 02 – Operators 17

1. Arithmetic Operators 17
2. Relational Operators 19
3. Logical Operators 20
4. Bitwise Operators 21
5. Assignment Operators 21
6. Miscellaneous Operators 22

- Precedence and Associativity 23
- Exercise 25

Chap 03 – Control Statements 27

1. Decision Making Statements 27
2. Selection Statement (Switch-case) 29
3. Iteration Statements 31
4. Jump Statements 34

- Exercise 35

Chap 04 – Array and String 41

- Introduction to Arrays 41

Multidimensional Array 43
Introduction to Strings 44
String Methods in Java 45
Exercise 47
Chap 05 – Methods 57
Working with Methods 57
Method Overloading 60
Types of Methods 61
Special Types of Methods 63
Exercise 68
Chap 06 – Object Oriented Programming 77
OOps Terminologies 77
Creating Our Own Class 83
Constructors 90
Inheritance in Java 92
Dynamic Memory Dispatch 96
Abstract Classes in Java 99
Interfaces in Java 100
Anonymous Classes and Lambda Expressions 107
Exercise 109
Chap 07 – Advanced Concepts 113
Errors in Java 113
Exceptions in Java 114
File Handling 120
Annotations 122
Packages 123
Creating Documentation 124
Collections Framework 126
Java Generics 131
Date and Time 133
Multithreading 137

Exercise 146

CHAP 01 – INTRODUCTION

Java is a programming language, created in 1995. It is owned by Oracle, and more than 3 billion devices run Java.

It is used for a variety of applications such as:

- Mobile applications (especially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection

Java is known for its robustness, security, and simplicity. It is an object-oriented class-based programming language. The language is designed to have as few implementation dependencies as possible. The term WORA, write once and run anywhere is often associated with JAVA. It means that whenever we compile a Java code, we get the byte code (.class file), and that can be executed (without compiling it again) on different platforms provided they support Java.

Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.). It is one of the most popular programming languages in the world. There is a large demand for Java developers in the current job market. It is easy to learn and simple to use. It is open source and free. It is secure, fast, and powerful. It has huge community support (tens of millions of developers).

Java is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs. As Java is close to C++ and C#, it makes it easier for programmers to switch Java and vice versa.

HOW JAVA WORKS?

In Java, source code is compiled into the byte code and then it is interpreted into the machine code. The Java compiler, 'javac' compiles the java code into byte code (.class file), which is then interpreted in the JVM (Java Virtual Machine).

Source code ----(Compiled)→ Byte code ----(Interpreted)→ Machine code.

INTERPRETER VS COMPILER

Interpreters translate one statement at a time to machine code but the compiler scans the entire program and translates the whole of it into machine code.

Interpreter	**Compiler**
One statement at a time.	Entire program at a time.
Interpreter is needed every time.	Once compiled compiler is not needed.
Partial execution if error.	No execution if an error occurs.
Usually easy for programmers.	Usually not as easy as Interpreted ones.

IS JAVA INTERPRETED OR COMPILED?

Java is a hybrid language (both compiled as well as interpreted).
Java file (Kushal.java) **---**compiled (using javac)→ Class file (Kushal.class(byte code)) → can be used by Java interpreter

NOTE:

1. Any JVM can be used to interpret this byte code.
2. This byte code can be taken to any platform (Windows/Mac/Linux) for execution.
3. Hence Java is platform independent (write once run everywhere).

EXECUTING A JAVA PROGRAM

Javac Kushal.java → Compile to byte code (Kushal.class)
java Kushal → Interpret.
So far, the execution of our program has been managed by IntelliJ IDEA. We can download a source code editor like VS Code to compile and execute our Java programs manually.

SETTING UP OUR COMPUTER

1. Install JDK (Java Development Kit)
2. Install JRE (Java Runtime Environment). It is not important to install because JDK has inbuilt JRE.
3. Install IntelliJ Idea (community version).

EXTRACTING TAR FILE (LINUX)

1. Open terminal.
2. Go to the file location.
3. Use command "tar -xvzf filename.tar" to extract the file.

CREATING A LAUNCHER (LINUX)

1. Right click on desktop.
2. Choose to create a launcher.
3. Give name to launcher.
4. Give an icon to the launcher.

5. Mark done.

COMPONENTS OF JAVA PROGRAM

- **Keywords:** Predefined words that have special meaning to the compiler.
- **Identifiers:** Names given to different entities such as variables, functions, etc.
- **Constants:** Values that remain fixed and cannot be changed.
- **Literals:** A constant value which can be assigned to the variable.
- **String literals:** Sequence of characters enclosed in double quote.
- **Symbols:** abbreviations of operators.

JAVA LANGUAGE KEYWORDS

abstract	double	int	super
assert***	else	interface	switch
boolean	enum****	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	goto*	public	void
const*	if	return	volatile
continue	implements	short	while
default	import	static	
do	instance of	strictfp*	

These fifty keywords are reserved by Java and are used for special purposes. These keywords can't be used as literals.

BASIC JAVA PROGRAM

```java
package chap01introduction;

public class Eg01HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

ANATOMY OF A JAVA PROGRAM

A Java program is made up of **classes** that contain **methods**, with execution starting from the main() method. Each Java file must match the name of its **public class**, and all code must be enclosed within a class definition.

Following sections can be found on a Java program:

1. Documentation section → Suggested (Name of author, Date of coding, Purpose, etc.)
2. Package segment → Optional.
3. Import statements → Optional.
4. Interface statements → Optional
5. Class definitions → Optional
6. Main method Class {Main method definition} → Essential

COMMENTS

Comments are used to provide explanations or notes in the source code. They make the code easier to read and understand. Java completely ignores the comments.

There are two types of writing comments in Java. They are:

SINGLE LINE COMMENT

In Java, a single line comment starts with '//'. It extends till the end of the line, and we don't need to specify its end.

Example: // This is a single line comment.

MULTI-LINE COMMENT

In Java, there is another type of comment that allows us to comment on multiple lines at once, they are multiline comments. To write multiline comments we use the '/*……*/' symbol.

Example: /* This is a multi-line comment.

This comment can be written in multiple lines. */

```java
package chap01introduction;

public class Eg02Comments {
   // This is a single line comment

   /*
   This is a multiline comment.
   Multiline comments are used to write comment on multiple lines
    */

   public static void main(String[] args) {
      System.out.println("We studied comments successfully!");
   }
}
```

VIEW JAVA DOCUMENTATION

Java documentation, often referred to as **Javadoc**, provides detailed information about Java classes, methods, interfaces, and packages. It helps developers understand how to use built-in libraries and APIs effectively. You can explore the official Java documentation at docs.oracle.com to learn about language features, class hierarchies, and method usage. For your own projects, you can generate similar documentation using the javadoc tool, which parses specially formatted comments in your code to produce HTML documentation which is great for maintaining and sharing your codebase.

VARIABLES IN JAVA

Java is statically typed. (Statically typed → variables must be declared before use.)

WHAT IS A VARIABLE?

- A name given to a memory location.
- Declared by writing: **type variable_name;**
- Initialized and declared by: **type variable_name = value;**
- Multiple variables can be declared by: **type variable_name1, variable_name2;**

RULES FOR DECLARING A VARIABLE

1. It can contain alphabets, digits, and underscores.
2. A variable name can start with an alphabet and underscore only.
3. Can't start with digit.
4. No white spaces and reserved keywords are allowed.
5. Can contain alphabet, $ character, _ character and digits if other conditions are met.

EXAMPLES:

- **Valid Variable Names:** int Kushal, float Kushal1234, char _Kushal, long _Kushal_2_joshi
- **Invalid Variable Names:** int $Kushal, int 345Kushal, char int

DATA TYPES IN JAVA

Java has mainly three types of data types. They are:

- **Primitive:** byte, short, int, long, float, double, char, boolean
- **Reference:** arrays, classes, interfaces, enums, strings
- **Special:** void (used only for method return types)

PRIMITIVE DATA TYPES

Java has **eight primitive data types** that serve as building blocks for data manipulation. These include:

- **byte, short, int, long** → for integer values of varying sizes
- **float, double** → for decimal (floating-point) numbers
- **char** → for a single 16-bit Unicode character
- **boolean** → for logical values: true or false

These types are **not objects**, so they are stored directly in memory and offer better performance. Each has a fixed size and range, ensuring predictable behavior across platforms.

BYTE

- Values range from -128 to 127.
- Take 1 byte.
- The default value is 0.

SHORT

- Values range from $-(2^{16})/2$ to $(2^{16})/2-1$.
- Take 2 bytes.
- The default value is 0.

INT

- Values range from $-(2^{32})/2$ to $(2^{32})/2-1$.
- Take 4 bytes.
- The default value is 0.

LONG

- Values range from $-(2^{64})/2$ to $(2^{64})/2-1$.
- Takes 8 bytes.
- Default value is 0L.

FLOAT

- Values range from ($\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{38}$).
- Take 4 bytes.
- The default value is 0.0f.

DOUBLE

- Values range from ($\pm 4.9 \times 10^{-324}$ to $\pm 1.7976931348623157 \times 10^{308}$).
- Take 8 bytes.
- The default value is 0.0D.

CHAR

- Values range from 0 to 65535 (2^{16}-1).
- Take 2 bytes. (Because it supports Unicode.)
- The default value is 10000.

BOOLEAN

- Value can be true or false.
- Size depends on JVM.
- The default value is false.

REFERENCE DATA TYPES

- Store **references (addresses)** to objects in memory, not the actual data.
- Size depends on the **JVM and architecture** (typically 4 or 8 bytes).
- Default value is null.
- Can be used to create complex structures like:
  - **Arrays** → int[] arr = {1, 2, 3}; or, int[] arr = new int[]{1, 2, 3};
  - **Strings** → String name = "Kushal"; or, String name = new String("Kushal");
  - **Objects** → Student student = new Student();
  - **Enums, Interfaces, Classes**
- Allow access to methods and fields defined in the object/class.

```
package chap01introduction;

public class Eg03VariablesAndDataTypes {
  public static void main(String[] args) {
    // Primitive Data Types
    byte number = 44;
    int number1 = 45;
    short number2 = 126;
    char character = 'K';
    float floatingNumber = 5.67f;
    float floatingNumber1 = 5.67F;
    double doubleNumber = 4.98d;
    double doubleNumber1 = 4.98D;
    long ageOfEarth = 5400000000L;
    boolean learnJava = true;

    // Reference Data Types
    int[] arr1 = {1, 2, 3};
    int[] arr2 = new int[]{1, 2, 3};
    String name = "Kushal";
    String myName = new String("Kushal Prasad Joshi");
  }
}
```

SCANNING AND PRINTING

Scanning in Java is done using the Scanner class to read user input from the keyboard. **Printing** is handled using System.out.print(), println(), or printf() to display output to the console.

READING DATA FROM KEYWORD

To read data from the keyboard Java has a Scanner class. Scanner class has a lot of methods to read the data from the keyboard.

Syntax:
Scanner scan = new Scanner(System.in); // System.in is used to read from the keyboard.
int a = scan.nextInt(); // Method to read from keyboard. (Integer in this case.)

```
package chap01introduction;

import java.util.Scanner;

public class Eg04ReadingDataFromKeyboard {
  public static void main(String[] args) {
    System.out.println("Taking input from user.");

    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter first number.");
    int number1 = scanner.nextInt();
    System.out.println("Enter second number.");
    int number2 = scanner.nextInt();

    int sum = number1 + number2;

    System.out.println("The sum of numbers is " + sum );
  }
}
```

PRINTING METHODS IN JAVA

1. **System.out.print()** → No new line at the end.
2. **System.out.println()** → Prints new line at the end.
3. **System.out.printf()** → Uses format specifiers and escape sequence characters to create a formatted string.
4. **System.out.format()** → Uses format specifiers and escape sequence characters to create a formatted string. [same as System.out.printf()]

FORMAT SPECIFIERS

Format specifier is a way to tell compiler what type of data is in a variable during taking input and displaying output to user.

Format Specifier	Data Type
%c	char
%d	Int, short, byte
%f	float
%l	long
%lf	double
%b	boolean

EXAMPLE:

System.out.printf("The sum is %a.bf", var); // Here in place of '%a.bf' compiler will print the floating number in 'var' by consuming 'a' character space with 'b' decimal accuracy.

NOTE: If you use '%+a.b', characters will occupy space from the right side but if you use '%-a.b', characters will occupy space from left side.

ESCAPE SEQUENCE CHARACTERS

The escape sequence characters in Java are a combination of characters that begin with backslash (\ ) which represent certain special characters within string literals.

Escape Sequence Character	Purpose
\b	Backspace. It is used to move the cursor one place backward.
\f	Form feed. It is used to move the cursor to the start of the next logical page.
\n	New line. It moves the cursor to the start of the next line.
\r	Carriage return. It moves the cursor to the start of the current line.
\t	Horizontal tab. It inserts some white spaces to the left of the cursor and moves the cursor accordingly.
\\	Backslash. Used to insert backslash character in string literals.
\'	Single quote. Used to insert single quote in string literals.
\"	Double quote. Used to insert double quote in string literals.
\?	Question mark. Used to insert question mark in string literals.

\0	Null character. It is used to terminate the string.
\uXXXX	Unicode character.
\XXXX	Unicode value in hexadecimal.

NOTE: Escape sequence characters are used for special purposes in string literals.

```
package chap01introduction;

public class Eg05PrintingMethods {
   public static void main(String[] args) {
      String name = "Kushal Prasad Joshi"; // String class has a special support in Java so can be used simply like data types

      System.out.print(name); // Prints name
      System.out.println(); // Prints a new line
      System.out.println(name); // Prints name and a new line
      System.out.printf("%s\n", name); // Prints name and a new line
      System.out.format("%s", name); // Prints name
   }
}
```

EXERCISE

1. Write a Java program to add three numbers and print sum.

```
package chap01introduction;

public class Qn01AddThreeNumbers {
   public static void main(String[] args) {
      int number1 = 45;
      int number2 = 55;
      int number3 = 65;

      int sum = number1 + number2 + number3;

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

2. Write a java program to check whether the input is an integer or not.

```
package chap01introduction;

import java.util.Scanner;

public class Qn02CheckDataTypeOfInput {
   public static void main(String[] args) {
      System.out.println("Enter an integer.");
```

```
        Scanner scanner = new Scanner(System.in);
        boolean answer = scanner.hasNextInt();

        System.out.println("You entered " + answer);
    }
}
```

3. **Write a program to calculate the percentage of a given student in an exam. His marks from seven subjects out of 100 must be taken from the keyboard.**

```
package chap01introduction;

import java.util.Scanner;

public class Qn03CalculatePercentage {
    public static void main(String[] args) {

        System.out.println("Enter the marks obtained by students in seven subjects (out of
100).");

        Scanner scan = new Scanner(System.in);
        float marks1 = scan.nextFloat();
        float marks2 = scan.nextFloat();
        float marks3 = scan.nextFloat();
        float marks4 = scan.nextFloat();
        float marks5 = scan.nextFloat();
        float marks6 = scan.nextFloat();
        float marks7 = scan.nextFloat();

        float sumOfMarks = marks1 + marks2 + marks3 + marks4 + marks5 + marks6 +
marks7;
        float percentage = sumOfMarks / 7; // Calculate percentage

        System.out.println("The percentage obtained by student is " + percentage);
    }
}
```

4. **Write a program to sum up three numbers in java. Take the numbers as input from the user.**

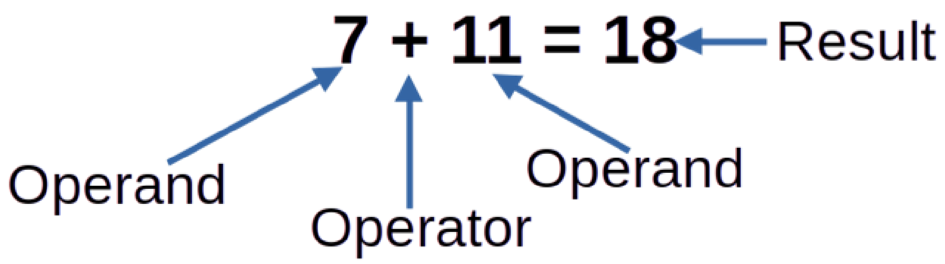
```
package chap01introduction;

import java.util.Scanner;

public class Qn04SumOfThreeNumbers {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter first number.");
        int num1 = scan.nextInt();
        System.out.println("Enter second number.");
        int num2 = scan.nextInt();
        System.out.println("Enter third number.");
        int num3 = scan.nextInt();

        int sum = num1 + num2 + num3; // Find the sum

        System.out.println("The sum is " + sum);
    }
}
```

5. **Write a program which asks the user to enter his/her name and greet them with "Hello <|name|>, have a good day!" text.**

```
package chap01introduction;

import java.util.Scanner;

public class Qn05GreetTheUser {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("What is your name?");
        String name = scan.next(); // String class has special support in Java

        System.out.println("Hello " + name + ", have a good day!");
    }
}
```

CHAP 02 – OPERATORS

Operators are the symbols used to perform operations on variables and values.



There are six types of operators in Java. They are:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators

1. ARITHMETIC OPERATORS

Java provides a set of built-in operators for performing basic arithmetic on numeric types (byte, short, int, long, float, double). These operators follow the usual mathematical rules and Java's own rules for type promotion and integer division.

NOTE: Arithmetic operators are used to perform arithmetic operations. Arithmetic operators can't work with Booleans. Modulus (%) operator can work on floats and doubles as well.

Remember: In Java, the '+' operator can be used for both addition and string concatenation. When used with strings it concatenates them. When one operand is a string and another is a number, the number is converted to string and then concatenated. So, if we have the statement *System.out.println("sum = " + a + b);*, *"sum = " + a* is evaluated first which results in a string and *add 'b'* similarly. To fix this you can use parentheses to change the order of operations like *System.out.println("sum = " +(a + b));*.

Operators	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

++	Increment
--	Decrement

```
package chap02operators;

public class Eg01ArithmeticOperators {
    public static void main(String[] args) {
        int a = 5, b = 3;

        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));
        System.out.println("++a = " + ( ++ a));
        System.out.println("--b = " + (--b));
    }
}
```

INCREMENT AND DECREMENT OPERATORS

a++, ++a → increment operators

a--, --a → decrement operators

// Data type remains the same in both cases.

NOTE:

- These will operate on all data types except Booleans.
- a++ → First use the variable and then increment. Same for decrement.
- ++a → First increment the value then use it. Same for decrement.
- char a = 'B';
  a++; // Now a = C because A++ = B, B++ = C, C++ = D, ...

```
package chap02operators;

public class Eg02IncrementDecrementOperators {
    public static void main(String[] args) {
        int a = 5, b = 3;

        System.out.println("++a = " + ( ++ a));
        System.out.println("--b = " + (--b));
    }
}
```

TYPE PROMOTION RULE

When mixing different numeric types, the resulting data type after arithmetic calculations in Java follows following rules:

1. Widening conversion happens automatically (e.g., int + double → both become double).
2. byte, short, and char are first promoted to int before any arithmetic.
3. No automatic *narrowing*, you must cast explicitly if you want to convert a double back to an int.

Some conversions after arithmetic operations are as follows:

byte + short → int	short + int → int
long + float → float	int + float → float
char + int → int	char + short → int
long + double → double	float + double → double

NOTE:

- integer type → integer
- 0.0 and 0 → 0.0
- other → bigger datatype

```java
package chap02operators;

public class Eg03TypePromotionRule {
   public static void main(String[] args) {
      char character= 'A';
      int integer = 5;
      float floatingNumber = 6.0f;
      byte byteNumber = 34;
      short shortNumber = 234;
      long longNumber = 4L;
      double doubleNumber = 789.78D;

      System.out.println(byteNumber + shortNumber); // Prints int
      System.out.println(shortNumber + integer); // Prints int
      System.out.println(longNumber + floatingNumber); // Prints float
      System.out.println(integer + floatingNumber); // Prints float
      System.out.println(character + integer); // Prints int
      System.out.println(character + shortNumber); // Prints int
      System.out.println(longNumber + doubleNumber); // Prints double
      System.out.println(floatingNumber + doubleNumber); // Prints double
   }
}
```

2. RELATIONAL OPERATORS

Relational operators compare two operands and yield a Boolean result (true or false). They're fundamental for decision-making (e.g., if, while) and work on numeric and character types.

Operators	Description
==	Is equal to
!=	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

NOTE: Relational operators always return false (0) or true (1) i.e. binary values.

```
package chap02operators;

public class Eg04RelationalOperators {
   public static void main(String[] args) {
      int num1 = 5, num2 = 6;

      System.out.println(num1 == num2); // false
      System.out.println(num1 != num2); // true
      System.out.println(num1 > num2);  // false
      System.out.println(num1 < num2); // true
      System.out.println(num1 >= num2); // false
      System.out.println(num1 <= num2); // true
   }
}
```

3. LOGICAL OPERATORS

Logical operators work on Boolean values and are essential for controlling program flow (e.g., if, while). They evaluate Boolean expressions and return true or false.

Operators	Description
&&	Logical AND operator. If both the operands are non-zero, then the condition is true.
||	Logical OR operator. If any of the two operands is non-zero, then the condition is true.
!	Logical NOT operator. It reverses the logical state of its operand i.e. true (1) to false (0) and vise verse.
^	Logical XOR operator. If exactly one operand is non-zero, then the condition is true.

NOTE: Logical operators only operate on binary operands i.e. 0 or 1 (other values than 0 are considered as 1).

```
package chap02operators;

public class Eg05LogicalOperators {
   public static void main(String[] args) {
    boolean b1 = true, b2 = false;

    System.out.println(b1 && b2); // false
    System.out.println(b1 || b2); // true
    System.out.println(!b2); // true
  }
}
```

4. BITWISE OPERATORS

Bitwise operators work directly on the individual bits of integer types (byte, short, int, long). They're useful for low-level programming tasks such as masking, setting, clearing, or toggling specific bits, and for efficient arithmetic on powers of two.

Operators	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Binary one's complement operator
<<	Binary left shift operator
>>	Binary right shift operator
>>>	Unsigned right shift operator

NOTE: Bitwise operators operate on each bit of operand.

```
package chap02operators;

public class Eg06BitwiseOperators {
  public static void main(String[] args) {
    System.out.println(2 & 3); // Bitwise and
      /*
      2 -----> 10
      3 -----> 11
      ............
      2 -----> 10
       */
    System.out.println(2 | 3); //Bitwise or
      /*
      2 -----> 10
      3 -----> 11
      ............
      3 -----> 11
       */
  }
}
```

5. ASSIGNMENT OPERATORS

Assignment operators assign values to variables. Java provides the simple assignment operators as well as compound assignment operators that perform an operation and assignment in one step.

Operators	Description
=	Simple assignment operator. Assign value from right side operand to left side operand.
+=	Add and assign operator. It adds the right operand to the left operand and assigns the result to the left operand.
-=	Subtract and assign operator. It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	Multiply and assign operator. It multiplies the right operand with the left operand and assigns the result to the left operand.
/=	Divide and assign operator. It divides the left operand with the right operand and assigns the value to the left operand.
%=	Modulus and assign operator. It divides the left operand with right operand and assigns the remainder to the left operand.
>>=	Binary right shift and assign.
<<=	Binary left shift and assign.
&=	Bitwise AND and assign.
|=	Bitwise OR and assign.
^=	Bitwise XOR and assign.
>>>=	Unsigned right shift assignment.

NOTE: Assignment value does arithmetic values with operands and finally assign result to left operand.

```java
package chap02operators;

public class Eg07AssignmentOperators {
   public static void main(String[] args) {
      int i = 2; // Simple assignment
      i += 2; // Add and assign
      i -= 2; // Subtract and assign
      i *= 2; //Multiply and assign
      i /= 2; // Divide and assign
   }
}
```

6. MISCELLANEOUS OPERATORS

Beyond arithmetic, relational, logical, bitwise, and assignment operators, Java includes a handful of “miscellaneous” operators that support concise conditionals, type checks, casting, and member access.

Operators	Description

?:	Conditional operator. Used as alternative of if-else.
,	Comma operator. Allows multiple expressions to be evaluated in a sequence and returns the result of the last expression.
->	Arrow operator. Used in lambda expressions to separate the parameters from the body of expression.
.	Member access operator. Used to access the members (methods and fields) of an object or class.
[]	Array index operator. This operator is used to access elements of array.
new	New operator. This operator is used to create new objects.
instanceof	Instance of operator. This operator is used to check whether an object is the instance of a particular class.
(<|data type|>)	Typecast operator. This operator is used to cast an object from one type to another.

NOTE: Miscellaneous Operators are operators used for specific purposes.

```
package chap02operators;

import java.util.Scanner;

public class Eg08MiscellaneousOperators {
   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in); // Creating an object of Scanner class

      System.out.println("Enter a character.");
      char character = scan.next().charAt(0);
      System.out.println((int) character); // Prints an integer
   }
}
```

PRECEDENCE AND ASSOCIATIVITY

PRECEDENCE OF OPERATORS

The operators are applied and evaluated based on precedence. For example -> (+, -) has less precedence compared to (*, /). Hence, * and / are evaluated first.

In case we like to change the order, we use parenthesis since the parenthesis has highest order of precedence and get evaluated first.

ASSOCIATIVITY OF JAVA

Associativity tells the direction of execution of operations. It can either be left to right or right to left.

PRECEDENCE ASSOCIATIVITY TABLE

The precedence associativity table is a table with set of rules related to precedence and associativity that guide the operations sequence in Java.

Operators		Associativity	Precedence
() [] . ->	Function call Array subscript Dot (Member of structure) Arrow (Member of structure)	left-to-right	14 (highest)
! - _ ++ -- & * (type) size of	Logical NOT One's complement Unary minus (Negation) Increment Decrement Address-of Indirection Cast Size of	right-to-left	13
* / %	Multiplication Division Modulus (Remainder)	left-to-right	12
+ -	Addition Subtraction	left-to-right	11
<< >>	Left shift Right shift	left-to-right	10
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	left-to-right	9
== !=	Equal to Not equal to	left-to-right	8
&	Bitwise AND	left-to-right	7
-	Bitwise XOR	left-to-right	6
|	Bitwise OR	left-to-right	5
&&	Logical AND	left-to-right	4
||	Logical OR	left-to-right	3
?:	Conditional	right-to-left	2
=, +=, etc.	Assignment operators	right-to-left	1

.		left-to-right	0 (lowest)

```
package chap02operators;

public class Eg09PrecedenceAndAssociativity {
   public static void main(String[] args) {
      int a = 6*5-34/2;
        /*
        = 6*5-34/2
        = 30-34/2
        = 30-17
        13
         */
      int b = 60/5-34*2;
        /*
        = 60/5-34*2
        = 12-34*2
        = 12-68
        = -56
         */
      //Highest Precedence goes to * and /.
      // They are then evaluated on the basis of left to right associativity.

      System.out.println(a);
      System.out.println(b);
   }
}
```

EXERCISE

1. **Find the results of the following expressions:**
   a. **float a = 7/4*9/2**
   b. **float b = (float) (7/4*9/2)**

```
package chap02operators;

public class Qn01FindResultsOfExpressions {
   public static void main(String[] args) {
      float a = 7/4*9/2; // Arithmetic calculation among integers is always integer
        /*
        = 7/4*9/2
        = 1*9/2;
        = 9/2
        = 4
         */
      System.out.println(a);

      float b = (float) (7/4*9/2); // Typecasting is done after the calculation is finished.
So same result.
      System.out.println(b);

      float c = 7/4f*9/2; // We also have a floating number so float and int arithmetic is a
float
        /*
        = 7/4*9/2
        = 1.75*9/2
        = 15.75/2
        = 7.875
         */
      System.out.println(c);
   }
}
```

2. **Write a java program to encrypt a letter by adding a number and decrypt it to display (use type casting).**

```
package chap02operators;

import java.util.Scanner;

public class Qn02EncryptLetterUsingNumber {
   public static void main(String[] args) {
      int number = 67;

      Scanner scan = new Scanner(System.in);
      System.out.println("Enter a character.");
      char character = scan.next().charAt(0);

      // Encrypt character using number
      character = (char) (character + number);
      System.out.println("Character encrypted successfully! to " + character);

      // Decrypt the character
      character = (char) (character - number);
      System.out.println("The character is " + character);
   }
}
```

CHAP 03 – CONTROL STATEMENTS

Control statements in Java determine the order in which individual statements, blocks, and methods execute. They include **selection statements** (if, switch) for branching logic, **iteration statements** (for, while, do-while) for looping, and **jump statements** (break, continue, return) for altering flow within those structures. Together, they let you build complex, conditional, and repeated behaviors in your programs.

There are two types of control statements in Java:

1. Conditional Control Statements
   i. Decision Making Statements
      a. If Statements
      b. Else-if Ladder
   ii. Selection Statement (switch-case)
   iii. Iteration Statements
      a. While Loop
      b. Do-while Loop
      c. For loop
      d. For-each loop
2. Unconditional Control Statements (Jump Statements)
   i. Break Statement
   ii. Continue Statement

NOTE: Conditional Control Statements are used to perform operations based on some conditions. They execute instructions on a condition being met. Condition may be true (1) or false (0). But Unconditional Control Statements require no condition to work. So, they are also called jump statements.

1. DECISION MAKING STATEMENTS

Decision making control statements decide which statement to execute and when. These statements check the conditions one by one and select one among these which is true. Note that the conditions must be in Booleans.

IF STATEMENTS

This statement checks the condition; if it is true the code inside if is executed, if false it goes to the next else-if or else. Remember that we can use else for the false condition, but it is not compulsory.

Syntax:

```
if (condition_to_be_checked){
        // Statements if conditions are true.
}
else {
        // Statements if conditions are false.
}
```

Example: A program to find if the person can vote or not.

```java
package chap03controlstatements;

import java.util.Scanner;

public class Eg01CanVoteOrNot {
   public static void main(String[] args) {
      System.out.print("Enter your age: ");

      Scanner scan = new Scanner(System.in);
      byte age = scan.nextByte();

      // Check condition for voting
      if (age >= 18) {
         System.out.println("You can vote.");
      }
      else {
         System.out.println("You cannot vote.");
      }
   }
}
```

ELSE-IF LADDER

Instead of using multiple if statements, we can also use else-if along with if thus following an if-elseif-else ladder. Using such kind of logic reduces indents. Last else is executed only if all the conditions fail.

Syntax:

```
if (conditions) {
        // Statements;
}
else if (conditions) {
        // Statements;
}
else {
        // Statements;
}
```

Example: A program to determine divisions of students based on the percentage they have obtained.

```java
package chap03controlstatements;

import java.util.Scanner;

public class Eg02DivisionsOfStudentsBasedOnPercentage {
   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in);

      System.out.print("Enter the percentage of student: ");
      float percentage = scan.nextFloat();

      // Give divisions according to percentage
      if (percentage > 80)
         System.out.println("Distinction");
      else if (percentage > 70)
         System.out.println("First Division");
      else if (percentage > 60)
         System.out.println("Second division");
      else if (percentage > 50)
         System.out.println("Third division");
      else if (percentage > 40)
         System.out.println("Passed");
      else
         System.out.println("Failed");
   }
}
```

2. SELECTION STATEMENT (SWITCH-CASE)

Switch case is used when we have to make a choice between number of alternatives for a given variable. It directly jumps to the required case, if not found jumps to default.

Syntax:

```
switch (variable) {
        case c1:
                // code;
                break;
        case c2:
                // code;
                break;
        case c3:
                // code;
                break;
        ………………………………
        default:
                //code;
}
```

NOTE:

- var can be an integer, character, or a string in Java.
- A switch case can occur in another but in practice, this is rarely done.

Example: A command line calculator that performs addition, subtraction, multiplication, and division.

```java
package chap03controlstatements;

import java.util.Scanner;

public class Eg03CommandLineCalculator {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);

    System.out.println("Enter your operation.");
    double operand1 = scan.nextDouble();
    char operator = scan.next().charAt(0);
    double operand2 = scan.nextDouble();

    // Use switch case to switch operation
    switch (operator) {
      case '+':
        System.out.println("Your result is " + (operand1 + operand2));
        break;
      case '-':
        System.out.println("Your result is " + (operand1 - operand2));
        break;
      case '*':
        System.out.println("Your result is " + (operand1 * operand2));
        break;
      case '/':
        System.out.println("Your result is " + (operand1 / operand2));
        break;
      default:
        System.out.println("Your input is wrong!");
        break;
    }
  }
}
```

The above style of using switch is called old style switch statement in Java since it has an enhance switch statement now. The enhance switch statements look like below.

```
package chap03controlstatements;

import java.util.Scanner;

public class Eg04EnhancedSwitch {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter your operation.");
        double operand1 = scan.nextDouble();
        char operator = scan.next().charAt(0);
        double operand2 = scan.nextDouble();

        // Use switch case to switch operation
        switch (operator) {
            case '+' -> System.out.println("Your result is " + (operand1 + operand2));
            case '-' -> System.out.println("Your result is " + (operand1 - operand2));
            case '*' -> System.out.println("Your result is " + (operand1 * operand2));
            case '/' -> System.out.println("Your result is " + (operand1 / operand2));
            default -> System.out.println("Your input is wrong!");
        }
    }
}
```

3. ITERATION STATEMENTS

These statements are used to repeatedly execute a block of code. Java provides four types of looping statements.

WHILE LOOP

- This loop keeps executing as long as the condition is true.
- If the condition becomes false, the while loop keeps getting executed. Such a loop is called infinite loop.

Syntax:

```
while (Boolean condition) {
        // Statements
}
```

Example: A program to display natural numbers from 1 to 100.

```
package chap03controlstatements;

public class Eg05NaturalNumbersFrom1To100 {
    public static void main(String[] args) {
        // Print natural numbers from 1 to 100 using while loop
        int i = 1;
    while (i <= 100) {
      System.out.println(i);
      i++;
    }
  }
}
```

DO-WHILE LOOP

This loop is like while loop except the fact that it is guaranteed to be executed at least once.

NOTE:

1. while → check the conditions and execute the code.
2. do-while → execute the code then check the conditions.

Syntax:

```
do {
    // Code
} while (conditions);   → Note this semi-colon
```

Example: A program to display even numbers from 1 to 100.

```
package chap03controlstatements;

public class Eg06EvenNumbersFrom1To100 {
  public static void main(String[] args) {
    int i = 1;

    // Use do while loop to print the even numbers
    do {
      if (i % 2 == 0)
        System.out.println(i);
      i++;
    }while (i <= 100);
  }
}
```

FOR LOOP

A for loop is usually used to execute a piece of code a given number of times.

Syntax:

```
for (Expression1; Expression2; Expression 3) {
    // Code to be executed
}
```

PROPERTIES OF EXPRESSION1

- The expression represents the initialization of the loop variable.
- This is executed only once no matter how many times it iterates.
- We can initialize more than one variable in expression1.
- Expression1 is optional.

PROPERTIES OF EXPRESSION2

- It is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.
- It can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.
- It is optional.
- Expression2 can perform the task of expression1 and expression3 i.e. we can initialize the variable as well as update the loop variable in expression2 itself.
- We can pass zero or non-zero values in expression 2. However, in Java, any non-zero value is true and zero is false by default.

PROPERTIES OF EXPRESSION3

- Expression3 is used to update the loop variables.
- We can update more than one variable at the same time.
- Expression3 is optional.

Example: A program to print odd numbers from 1 to 100.

```java
package chap03controlstatements;

public class Eg07OddNumbersFrom1To100 {
   public static void main(String[] args) {
      // Use for loop to print odd numbers from 1 to 100
      for (int i = 1; i <= 100; i++) {
         if (i % 2 != 0)
            System.out.println(i);
      }
   }
}
```

FOR-EACH LOOP

The for-each loop, also known as enhanced for loop, is a simplified syntax for iterating over arrays and collections in Java.

Syntax:

```
for (data_type variable_name : array_name) {
```

```
            // Code to be executed
    }
```

Example: Iterating an array to print all the elements in it.

```java
package chap03controlstatements;

public class Eg08IterateAnArray {
   public static void main(String[] args) {
      int[] arr = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};

      // Use for-each loop to print the elements in arr
      for (int element: arr) {
         System.out.println(element);
      }
   }
}
```

We will learn in detail about this loop while learning about arrays.

4. JUMP STATEMENTS

Jump statements provide explicit control over loop and method execution flow. They can be also used inside the switch case, especially break statement is used.

We have three jump statements in Java. They are:

1. break → send control outside the loop.
2. continue → send control to next iteration.
3. return → send control to calling function.

BREAK STATEMENT

The break statement can be used inside loops or switch statement to bring program control out of the loop. It exits the loop whether the test condition of loop is true or false.

```java
package chap03controlstatements;

public class Eg09BreakStatement {
   public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
         if (i == 5) {
            break;
         }
         System.out.println(i);
      }
   }
}
```

CONTINUE STATEMENT

The continue statement can be used to skip some lines of code for particular condition and bring the program control to the next iteration. It sends the control immediately to the next iteration.

```java
package chap03controlstatements;

public class Eg10ContinueStatement {
   public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
         if (i == 5) {
            continue;
         }
         System.out.println(i);
      }
   }
}
```

EXERCISE

1. **Write a program to find whether a student is passing or failing. Assume that the students give exams of 3 subjects (Physics, Chemistry and Mathematics) each of full marks 100 and pass marks 33 and overall percentage must be 40 to pass. Take the marks as input from user.**

```java
package chap03controlstatements;

import java.util.Scanner;

public class Qn01StudentPassedOrFailed {
   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in);

      System.out.print("Enter marks in Physics: ");
      byte markInPhysics = scan.nextByte();
      System.out.print("Enter marks in Chemistry: ");
      byte markInChemistry = scan.nextByte();
      System.out.print("Enter marks in Mathematics: ");
      byte markInMathematics = scan.nextByte();

      // Calculate average of marks
      float average = (markInPhysics + markInChemistry + markInMathematics) / 3.0f ;

      // Check if the student pass or fail
      if (markInPhysics >= 33 && markInChemistry >= 33 && markInMathematics >= 33
&& average >= 40)
         System.out.println("Congratulations! You passed the exam.");
```

```
    else
      System.out.println("Work hard! You failed in the exam.");
  }
}
```

2. **Write a program to input total units consumed and display total charges. Electricity board charges are according to the following rules:**
   For the first 100 units → Rs.100 (minimum charge)
   For the next 50 units → Rs.8.5 per unit
   For next 100 units → Rs. 9.5 per unit
   For beyond 250 units → Rs.10.5 per unit
   Additional charges → 13% tax and Rs.100 for maintenance

```
/*
Electricity board charges are according to the following rules:
  For first 100 units ◊ Rs.100 (minimum charge)
  For next 50 units ◊ Rs.8.5 per unit
  For next 100 units ◊ Rs. 9.5 per unit
  For beyond 250 units ◊ Rs.10.5 per unit
  Additional charges ◊ 13% tax and Rs.100 for maintenance
 */

package chap03controlstatements;

import java.util.Scanner;

public class Qn02CalculateElectricityBill {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    float charges;

    System.out.print("Enter the total units consumed: ");
    float unitsConsumed = scan.nextFloat();

    if (unitsConsumed <= 100)
      charges = 100;
    else if (unitsConsumed <= 150)
      charges = 100 + (unitsConsumed - 100) * 8.5f;
    else if (unitsConsumed <= 250)
      charges = 100 + 50 * 8.5f + (unitsConsumed - 150) * 9.5f;
    else
      charges = 100 + 50 * 8.5f + 100 * 9.5f + (unitsConsumed - 250) * 10.5f;

    float chargeToPay = charges + (13.0f / 100) * charges + 100;

    System.out.println("Total charges = " + chargeToPay);
    }
}
```

3. Write a program to find the day of the week according to the given number. (1 for Sunday, 2 for Monday and so on.)

```
package chap03controlstatements;

import java.util.Scanner;

public class Qn03DayOfWeekAccordingToNumber {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter the day according to number assuming Sunday as 1: ");
        byte day = scan.nextByte();

        // Use switch case to select day according to number
        switch (day) {
            case 1 -> System.out.println("Sunday");
            case 2 -> System.out.println("Monday");
            case 3 -> System.out.println("Tuesday");
            case 4 -> System.out.println("Wednesday");
            case 5 -> System.out.println("Thursday");
            case 6 -> System.out.println("Friday");
            case 7 -> System.out.println("Saturday");
            default -> System.out.println("Enter number between 1 and 7.");
        }
    }
}
```

4. Write a program to find whether the entered year is a leap year or not.

```
package chap03controlstatements;

import java.util.Scanner;

public class Qn04FindLeapYearOrNot {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter a year");
        int year = scan.nextInt();

        // Checking if the year is leap year or not
        if (year % 400 == 0)
            System.out.println("The year is leap year.");
        else if (year % 100 == 0)
            System.out.println("The year is not a leap year.");
        else if (year % 4 == 0)
            System.out.println("The year is leap year.");
        else
            System.out.println("The year is not leap year.");
    }
}
```

5. Write command-based Scissors, Paper, Rock game with infinite loop.

```
package chap03controlstatements;

import java.util.Random;
import java.util.Scanner;

public class Qn05ScissorsPaperRockGame {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        do {
            System.out.println("Enter 0 for Scissors, 1 for Paper and 2 for Rock.");
            int userInput = scan.nextByte();

            Random random = new Random();
            int botInput = random.nextInt(3);

            if (userInput == botInput) {
                System.out.println("Draw!");
            } else if (userInput == 0 && botInput == 2 ||
                    userInput == 1 && botInput == 0 ||
                    userInput == 2 && botInput == 1) {
                System.out.println("You win!");
            } else {
                System.out.println("Computer win!");
            }
            System.out.println("Computer choice: " + botInput + "\n");

        } while (true);

    }
}
```

6. WAP to find sum of first n even numbers.

```
package chap03controlstatements;

import java.util.Scanner;

public class Qn06SumOfFirstNEvenNumbers {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the last number: ");
        int n = scan.nextInt();

        int sum = 0;
        for (int i = 0; i <= n; i += 2) {
            sum += i;
        }
        System.out.println("The sum is " + sum);

    }
}
```

7. WAP to print the multiplication table of user input number.

```
package chap03controlstatements;

import java.util.Scanner;

public class Qn07_DisplayMultiplicationTable {
    public static void main(String[] args) {
        System.out.print("Enter a number to find multiplication table of: ");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();

        for (int i = 1; i < 10; i++)
            System.out.printf("%d x %d = %d\n", n, i, n * i);
    }
}
```

8. Write a program to find factorial of a number.

```
package chap03controlstatements;

import java.util.Scanner;

public class Qn08FindFactorial {
    public static void main(String[] args) {
        System.out.print("Enter a number to find factorial of: ");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        long factorial = 1L;

        while (n != 0) {
            factorial *= n;
            n --;
        }

        System.out.println("The factorial of given number is " + factorial);
    }
}
```

9. Write a program to print the following star pattern.

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

```
/*
Write a program to print the following star pattern.
  *
  *  *
  *  * *
  *  * * *
  *  * * * *

 */

package chap03controlstatements;

public class Qn09TriangularStarPattern {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j <= i; j++)
                System.out.print("*    ");
            System.out.println();
        }
    }
}
```

CHAP 04 – ARRAY AND STRING

In Java, an **array** is a fixed-length, indexed collection of elements all the same type (e.g., int[] nums = {1,2,3}), allowing fast, constant-time access by index. A **String** is an immutable sequence of characters (java.lang.String) that offers rich methods for manipulation (concatenation, substring, search, etc.) while ensuring thread safety and memory efficiency via internal caching. Both of them are examples of reference data types in Java.

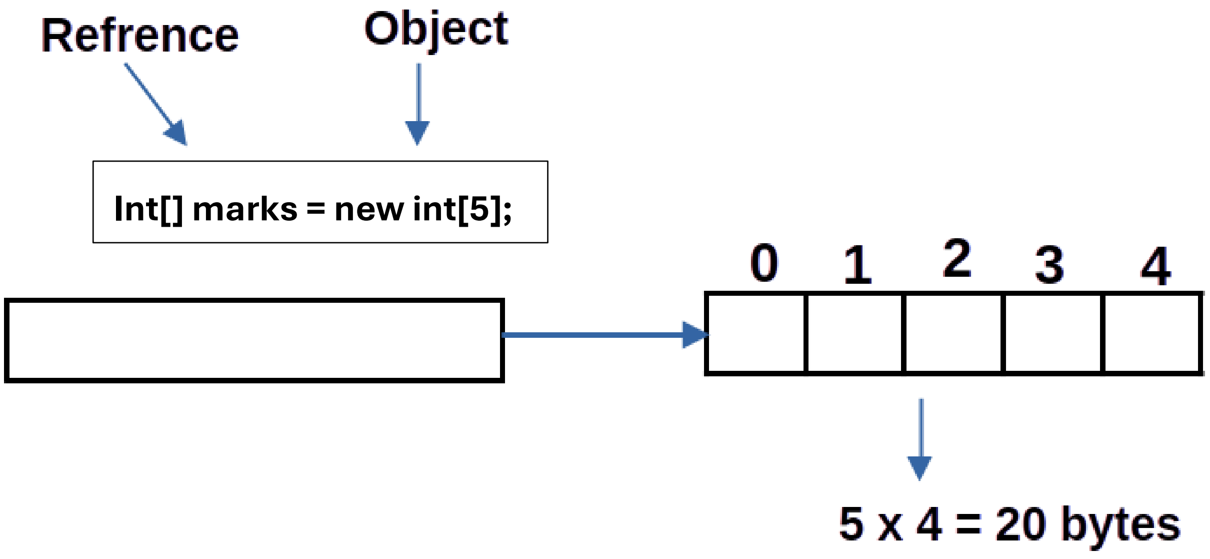
INTRODUCTION TO ARRAYS

Array is the collection of similar types of data. In Java, an array is a container object that holds a fixed number of values of a single type. The length of array is established when the array is created. After creation, its length is fixed.

USE CASE

When you need to store multiple values of a type.

int[] marks = new int[5];



WORKING WITH ARRAY

- int[] marks; // Declaration
- marks = new int[5]; // Memory allocation
- int[] marks = new int[5]; // Declaration + Memory allocation
- int[] marks = new int[]{100, 70, 80, 71, 98}; // Declare + Initialize
- int[] marks = {100, 70, 80, 71, 98}; // Declare + Initialize in short

ACCESSING ARRAY ELEMENTS

Array elements can be accessed as follows:

```
marks[0] = 100;
marks[1] = 19;
marks[2] = 78;
marks[3] = 68;
marks[4] = 99;
```

NOTE: Array indices start from 0 and go till (n-1) where n is the size of array.

DISPLAYING AN ARRAY USING LOOPS

The arrays are a continuous sequence of data. So, to access all the elements, we need to iterate through each element to print or display the elements. Generally a for loop or a for-each loop is used for iteration through arrays.

USING FOR LOOP

```
for (int i = 0; i < array.length; i++) {
        System.out.println(marks[i]);
}
// Array traversal method.
```

USING FOR-EACH LOOP

```
for (int element : arr) {
        System.out.println(element);
}
// Prints all the elements.
```

ARRAY LENGTH

Arrays have length property which gives the length of the array.

Example: marks.length   // Gives length of the array named marks.

EXAMPLE PROGRAM

```
package chap04arrayandstring;

import java.util.Scanner;

public class Eg01IntroductionToArrays {
   public static void main(String[] args) {
      int[] marks;
      marks = new int[7];
```

```
        // Take marks as input from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter marks of student in 7 subjects.");
        for (int i = 0; i < marks.length; i++) {
            marks[i] = scan.nextInt();
        }

        // Display data using for-each loop
        System.out.println("The marks of student are: ");
        for (int element : marks)
            System.out.println(element);
    }
}
```

MULTIDIMENSIONAL ARRAY

- Multidimensional arrays are the arrays of arrays.
- Each element of a multidimensional array is an array itself.

2D ARRAY

2D arrays are also known as matrices which are used to store data in tabular form.

Example: int[][] mat = new int[2][3]; // A 2D array of 2 rows and 3 columns.

We can add elements to the arrays as follows:

mat[0][0] = 15;
mat[0][1] = 17;
..... and so on...

3D ARRAY

3D arrays are commonly used to store the dimensions.

Similarly, as 2D array 3D array can be created as follows:

int[][][] dim = new int[2][3][4]; // A 3D array with length 2, breadth 3 and height 4.

NOTE: Java has no limitations in the dimensions of array. The 4D, 5D... arrays can be created in a similar way. But these are rarely used.

EXAMPLE PROGRAM

```
package chap04arrayandstring;

public class Eg02MultidimensionalArrays {
  public static void main(String[] args) {
    // Declare array
    int[][] matrix;
    // Memory allocation for array
    matrix = new int[2][3];

    // Initialize the array elements
    matrix[0][0] = 71;
    matrix[0][1] = 12;
    matrix[0][2] = 10;
    matrix[1][0] = 87;
    matrix[1][1] = 33;
    matrix[1][2] = 45;

    // Printing using for loop.
    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + "   ");
      }
      System.out.println();
    }
  }
}
```

INTRODUCTION TO STRINGS

- A string is a sequence of characters.
- A string is in-established in Java as follows:

  String name;

  name = new String ("Kushal");
- String is a class but can be used as a datatype.

  String name = "Kushal"; // Here 'name' is reference and 'Kushal' is object.

NOTE: Strings in Java are immutable and cannot be changed.

```
package chap04arrayandstring;

public class Eg03IntroductionToStrings {
  public static void main(String[] args) {
    String name = new String("Kushal");
    String fullName = "Kushal Prasad Joshi";

    System.out.println(name);
    System.out.println(fullName);
  }
}
```

STRING METHODS IN JAVA

String methods operate on Java strings to do various works such as finding length of string, converting to lowercase, converting to uppercase, etc.

SOME COMMONLY USE STRING METHODS

We are assuming a String “Kushal”, where the index of K = 0, u = 1, s = 2, h = 3, a = 4, l = 5; to understand String methods more clearly as follows:

String name = “$K_0u_1s_2h_3a_4l_5$”

1. **name.length() →** Returns length of String `name`. *(6 in this case)*
2. **name.toLowerCase() →** Returns a new String which has all lowercase letters from the String `name`. *(“kushal” in this case)*
3. **name.toUpperCase() →** Returns a new String which has all the uppercase letters from the String `name`. *(“KUSHAL” in this case)*
4. **name.trim() →** Returns a new string after removing all the leading and trading spaces from the original string. *(“Kushal” in this case since there are no leading and trading spaces)*
5. **name.substring(int start) →** Returns a string from start to end. *Note that the index starts from 0*. For example: `*name.substring(3);*` returns *"hal"*.
6. **name.substring(int start, int end) →** Returns a substring from start index to the end index. *Note that the start index is included, and the end index is excluded.*
7. **name.replace(‘a’, ‘e’) →** Returns a new string after replacing 'a' with 'e'. *(“Kushel” is returned in this case)*. Note that *strings can also be used* in place of characters.
8. **name.startsWith(“Ku”) →** Returns true if name starts with string “Ku” else returns false *(`true` is returned in this case)*. Note that *characters can also be used* in place of string.
9. **name.endsWith(“al”) →** Returns true if name ends with string “al” else returns false *(`true` is returned in this case)*. Note that *characters can also be used* in place of string.
10. **name.charAt(2) →** Returns character at a given index position *(‘s’ in this case)*.
11. **name.indexOf(“sha”) →** Returns the index of given string. *(Returns 2 in this case which is first occurrence of “sha” in the string “Kushal”; returns -1 otherwise when no matches are found.)*
12. **name.indexOf(‘s’, 3) →** Returns the index of the given character starting from the index 3 (int). *(-1 is returned in this case.)*
13. **name.lastIndexOf(‘h’) →** Returns the last index of the given character. *(3 in this case)*

14. **name.lastIndexOf('a', 2) →** Returns the last string of given character before index 2. *(-1 is returned in this case.)*
15. **name.equals("kushal") →** Returns true if the given string is equal to "kushal"; false otherwise. *[Case Sensitive] (` false` is returned in this case)*
16. **name.equalsIgnoreCase("kushal") →** Returns true if two strings are equal ignoring the case of characters. *[Case Insensitive] (true is returned in this case]*

```
package chap04arrayandstring;

public class Eg04StringMethods {
  public static void main(String[] args) {
    String name = "Kushal Prasad Joshi";

    // Finding length of String.
    System.out.println(name.length());

    // Converting String to Lower case.
    System.out.println(name.toLowerCase());

    // Converting String to Upper case.
    System.out.println(name.toUpperCase());

    // Removing all the leading and trading spaces from String.
    System.out.println(name.trim());

    // Returning a substring.
    // 1. Returning a string from the index to end.
    System.out.println(name.substring(5));
    // 2. Returning a string from index to index.
    System.out.println(name.substring(5,15));
    /*
    NOTE:
    1. The index starts from 0.
    2. The start index is included and end index is excluded.
     */

    // Replacing characters and strings in given string.
    System.out.println(name.replace("Kushal", "Kushal bro"));
    /*
    NOTE: Characters can also be used in the place of strings.
     */

    // Checking whether the String starts with the string or character.
    System.out.println(name.startsWith("Kush"));

    // Checking whether the String ends with the string or character.
```

```
        System.out.println(name.endsWith("Joshi"));

        //  Finding character at given index position.
        System.out.println(name.charAt(5));

        // Finding index of given String.
        System.out.println(name.indexOf("al"));

        // Finding the index of given string after the given index.
        System.out.println(name.indexOf('s', 6));

        // Finding the last index of given string or character.
        System.out.println(name.lastIndexOf('a'));

        // Finding the last index of given string or character before the index given.
        System.out.println(name.lastIndexOf('a', 10));

        // Checking equality of Strings.
        System.out.println(name.equals("kushal prasad joshi"));

        // Checking equality of string ignoring the case of characters.
        System.out.println(name.equalsIgnoreCase("kushal prasad joshi"));

    }
}
```

EXERCISE

1. **Write a program to store the names of 5 friends in an array and print the array in reverse order. Take names as input from user.**

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn01PrintArrayInReverseOrder {
    public static void main(String[] args) {
        // Declaration and memory allocation of an array
        String[] friends = new String[5];

        // Take name of friends as input from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter names of five friends.");
        for (int i = 0; i < friends.length; i++) {
            friends[i] = scan.nextLine();
        }

        // Display name of friends in reverse order
        for (int i = friends.length - 1; i >= 0; i--)
            System.out.println(friends[i]);
    }
}
```

2. **Create an array to store 5 integers. Also calculate their sum and average. Take the integers as input from the user.**

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn02SumAndAverageOfArrayElements {
    public static void main(String[] args) {
        // Declare and memory allocation of array
        int[] integer = new int[5];

        // Take input
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter any 5 integers.");
        for (int i = 0; i < integer.length; i++) {
            integer[i] = scan.nextInt();
        }

        // Calculate sum and average
        int sum = 0;
        float average;
        for (int i = 0; i < integer.length; i++) {
            sum += integer[i];
        }

        average = (float)sum / integer.length;

        // Display sum and average
        System.out.println("The sum of numbers is " + sum);
        System.out.println("The average of numbers is " + average);
    }
}
```

3. **Write a program to check whether the given element is present in an array or not. If found display the position in array.**

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn03FindElementInArray {
    public static void main(String[] args) {
        // Declaration and memory allocation of array
        int[] array = new int[10];

        // Taking input for elements of array
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter 10 integers.");
        for (int i = 0; i < array.length; i++) {
            array[i] = scan.nextInt();
        }

        // Taking input for the integer to search
        System.out.println("Enter element to search in the array.");
        int elementToSearch = scan.nextInt();

        // Searching the element
        for (int i = 0; i < array.length; i++) {
            if (array[i] == elementToSearch) {
                System.out.println("The integer is found in index " + i);
                break;
            }
            if ( i == array.length - 1)
                System.out.println("The integer is not found in array.");
        }
    }
}
```

4. WAP to add two matrices. Take elements of matrices as input from the user.

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn04AdditionOfMatrices {
    public static void main(String[] args) {
        // Declaration and memory allocation of matrices
        int[][] matrix1 = new int[3][3];
        int[][] matrix2 = new int[3][3];

        // Taking elements of matrix from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the elements of first matrix.");
        for (int i = 0; i < matrix1.length; i++) {
            for (int j = 0; j < matrix1[i].length; j++) {
                matrix1[i][j] = scan.nextInt();
            }
        }
        System.out.println("Enter the elements of second matrix.");
        for (int i = 0; i < matrix2.length; i++) {
            for (int j = 0; j < matrix2[i].length; j++) {
```

```
            matrix2[i][j] = scan.nextInt();
         }
      }

      // Calculate sum of matrix
      int[][] sum = new int[3][3];
      for (int i = 0; i < sum.length; i++) {
         for (int j = 0; j < sum[i].length; j++) {
            sum[i][j] = matrix1[i][j] + matrix2[i][j];
         }
      }

      // Display sum
      System.out.println("The sum of given matrix is: ");
      for (int i = 0; i < sum.length; i++) {
         for (int j = 0; j < sum[i].length; j++) {
            System.out.printf("%6d", sum[i][j]);
         }
         System.out.println();
      }
   }
}
```

5. WAP to add two matrices. Take elements of matrices as input from the user.

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn04AdditionOfMatrices {
   public static void main(String[] args) {
      // Declaration and memory allocation of matrices
      int[][] matrix1 = new int[3][3];
      int[][] matrix2 = new int[3][3];

      // Taking elements of matrix from user
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter the elements of first matrix.");
      for (int i = 0; i < matrix1.length; i++) {
         for (int j = 0; j < matrix1[i].length; j++) {
            matrix1[i][j] = scan.nextInt();
         }
      }
      System.out.println("Enter the elements of second matrix.");
      for (int i = 0; i < matrix2.length; i++) {
         for (int j = 0; j < matrix2[i].length; j++) {
            matrix2[i][j] = scan.nextInt();
         }
```

```
        }

        // Calculate sum of matrix
        int[][] sum = new int[3][3];
        for (int i = 0; i < sum.length; i++) {
            for (int j = 0; j < sum[i].length; j++) {
                sum[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }

        // Display sum
        System.out.println("The sum of given matrix is: ");
        for (int i = 0; i < sum.length; i++) {
            for (int j = 0; j < sum[i].length; j++) {
                System.out.printf("%6d", sum[i][j]);
            }
            System.out.println();
        }
    }
}
```

6. WAP to find the largest and smallest element in an array of integers. Take elements of array as input from the user.

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn06LargestAndSmallestElementInArray {
    public static void main(String[] args) {
        // Declaration and memory allocation of an array
        int[] array = new int[10];

        // Take array elements as input
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter 10 integers.");
        for (int i = 0; i < array.length; i++) {
            array[i] = scan.nextInt();
        }

        // Find largest element in array
        int largest = Integer.MIN_VALUE;
        for (int element : array)
            if (element > largest)
                largest = element;

        // Find the smallest element in array
        int smallest = Integer.MAX_VALUE;
        for (int element : array)
            if (element < smallest)
                smallest = element;

        // Print smallest and largest values
        System.out.println("Largest value in array is " + largest);
        System.out.println("Smallest value in array is " + smallest);
    }
}
```

7. **Write a java program to reverse an array. Take array elements as input from the user.**

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn07ReverseAnArray {
    public static void main(String[] args) {
        // Declaration and memory allocation of an array
        int[] array = new int[10];

        // Take array elements as input
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter 10 integers.");
        for (int i = 0; i < array.length; i++) {
            array[i] = scan.nextInt();
        }

        // Print the original array
        System.out.println("Your array is: ");
        for (int element : array)
            System.out.print(element + "\t");
        System.out.println();

        // Reverse the array
        for (int i = 0; i < Math.floorDiv(array.length, 2); i++) {
            // Swap array[i] and array[array.length -1 - i]
            array[i] = array[i] + array[array.length - 1 - i];
            array[array.length - 1 - i] = array[i] - array[array.length - 1 - i];
            array[i] = array[i] - array[array.length - 1 - i];
        }

        // Print array after reversing
        System.out.println("The array after reversing is: ");
        for (int element : array)
            System.out.print(element + "\t");
    }
}
```

8. **Write a java program to check whether the array is sorted or not. If not sorted sort array before display. Take elements of array as input from the user.**

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn08CheckAndSortArray {
    public static void main(String[] args) {
        // Declaration and memory allocation of an array
        int[] array = new int[10];

        // Take array elements as input
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter 10 integers.");
        for (int i = 0; i < array.length; i++) {
            array[i] = scan.nextInt();
        }

        // Print the original array
        System.out.println("Your array is: ");
        for (int element : array)
            System.out.print(element + "\t");
        System.out.println();

        // Check if the array is sorted or not
        boolean isSorted = true;
        for (int i = 0; i < array.length - 1; i++) {
            if (array[i] > array[i + 1]) {
                isSorted = false;
                break;
            }
        }

        // Sort the array if not sorted
        if (isSorted) {
            System.out.println("Your array is sorted.");
        }
        else {
            System.out.println("Your array is not sorted.");
            for (int i = 0; i < array.length - 1; i++) {
                for (int j = 0; j < array.length - 1; j++) {
                    if (array[j] > array[j + 1]) {
                        array[j] = array[j] + array[j + 1];
                        array[j + 1] = array[j] - array[j + 1];
                        array[j] = array[j] - array[j + 1];
                    }
                }
            }
            System.out.println("The array after sorting is: ");
            for (int element : array)
                System.out.print(element + "\t");
        }
    }
}
```

9. Write a program to convert a String to lower case and upper case.

```
package chap04arrayandstring;

public class Qn09StringCaseConversions {
    public static void main(String[] args) {
        String name = "Kushal Prasad Joshi";

        // Convert to lower case
        name = name.toLowerCase();
        System.out.println(name);

        // Convert to upper case
        name = name.toUpperCase();
        System.out.println(name);
    }
}
```

10. Write a program to replace spaces in String with underscores.

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn10ReplaceSpacesWithUnderscores {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        // Read String from user
        System.out.println("Enter a string with white spaces:");
        String string = scan.nextLine();

        // Replace white spaces with underscores.
        string = string.replace(' ', '_');
        System.out.println(string);
    }
}
```

11. Write a program to prepare a letter template using escape sequence characters and replace the name of sender and receiver with the needed one.

```
package chap04arrayandstring;

public class Qn11PrepareLetterTemplateAndEdit {
   public static void main(String[] args) {
      // Design a letter format
      String letter = "Dear <|receiver|>.\n" +
            "\tI got the letter written by you. It was nice." +
            "This letter is written as a reply to your letter.\n" +
            "Thanks for remembering!\n" +
            "Your friend\n<|sender|>";

      // Replace receiver with your friend's name
      letter = letter.replace("<|receiver|>", "Shailendra");

      // Replace sender with your name
      letter = letter.replace("<|sender|>", "Kushal");

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

12. Write a program to find the type of websites from the given URL.

.com → commercial websites

.org → organizational websites

.edu → educational website

.np → Nepali website

```
package chap04arrayandstring;

import java.util.Scanner;

public class Qn12RecognizeTheTypeOfWebsites {
   public static void main(String[] args) {
      System.out.print("Enter a website name: ");
      Scanner scan = new Scanner(System.in);
      String website = scan.next();

      if (website.endsWith(".com"))
         System.out.println("This is a commercial website.");
      else if (website.endsWith(".org"))
         System.out.println("This is a organizational website.");
      else if (website.endsWith(".edu"))
         System.out.println("This is a educational website.");
      else if (website.endsWith(".np"))
         System.out.println("This is a nepali website.");
```

```
        else
            System.out.println("I haven't heard about such website before.");
    }
}
```

CHAP 05 – METHODS

Methods are the functions defined inside a class. The methods are building blocks that provide modularity, code reusability and efficient program design. It is a set of instructions that perform a specific task. It encapsulates a sequence of actions into a single unit. It flows DRY (Don't Repeat Yourself) principle.

ADVANTAGES

- **Modularity** → Methods allow you to break down a large program into smaller, manageable parts.
- **Reusability** → Once defined, methods can be used across different parts of your program.
- **Readability** → Well-named methods enhance code readability.
- **Maintenance** → Changes to a method affect only that method, simplifying maintenance.

WORKING WITH METHODS

Methods in Java can be created in two ways:

1. **Instance Method:**
   - Access the instance data using the object name.
   - Declared inside a class.
2. **Static Method:**
   - Access the static data using class name.
   - Declared inside a class with static keyword.

To use a method, you must declare it and then call it from another part of your program. Since Java is an Object-Oriented Programming language, we need to write methods inside same class.

METHOD DECLARATION

The method declaration contains the actual statements that execute when a method is called.

Syntax:

```
return_type method_name(list_of_parameters) {
        code_to_be_executed
}
```

Example: A method that returns sum of two numbers.

```
int sum(int number1, int number2) {
        int sum = number1 + number2;        // Method body
        return sum;    // Return value
}
```

Some terms used in method declaration are:

1. **Return Type**
   - The return type specifies the type of value returned after executing the method.
   - If no value needs to be returned, the 'void' data type is used.
2. **Method name**
   - Method name is an identity given to a method so that you can call it later.
   - Method name is case sensitive and follows all rules of defining a variable in Java.
3. **Parameter list**
   - Parameters act as variable inside methods that are specified after method name inside parentheses, as many as you need separated by comma.
   - It allows you to pass data to a method while calling.
4. **Method Body**
   - It is a block of code which gets executed when a method is called.
   - Method body may contain other methods as well.
5. **Return Value**
   - Return value is a value returned by a method while its execution finishes.
   - The return value can be any supported data type.

METHOD CALL

The method can be called inside the same class as well as outside the class. Static methods belong to the class itself and can be invoked without creating an instance, either from within the same class (simply by name or ClassName.methodName()) or from another class (using ClassName.methodName() or via a static import). In contrast, instance methods require you to first create an object of that class (e.g. MyClass obj = new MyClass();) before you can call obj.instanceMethod().

CALLING METHOD WITHOUT CREATING OBJECT (STATIC METHOD CALL)

To call a method without creating an object make sure that the method is static. If the method is static, then you can simply write method name followed by two parentheses () inside which you need to write arguments. If the method is not static, you need to create an object to call method.

Syntax:

static method_name(list_of_arguments);

NOTE: Don't forget to add static at the beginning of method declaration if you are trying to call the method without creating object.

STATIC KEYWORD

'static' keyword is used to associate a method of a given class with the class rather than the object. Static methods in the class are shared by all the objects so you don't need to create an object to call static methods.

Example: Finding sum of two numbers using method.

```java
package chap05methods;

public class Eg01IntroductionToMethods {
   // Declare a method
   static int sum(int number1, int number2) {
      int sum;
      sum = number1 + number2;
      return sum;
   }

    public static void main(String[] args) {
      System.out.println("Introduction to methods.");

      int num1 = 5, num2 = 6;

      // Call function inside same class
       int mySum = sum(num1, num2);

       System.out.println("The sum of numbers is " + mySum);
   }
}
```

Some terms used in method call are:

1. **Arguments**
   - When you call a method, the values you provide are called arguments.
   - Arguments are passed on to method parameters.
   - The arguments are copied to the parameters of the methods. Thus, even if we modify the values of parameters, the values of arguments will not change.
   - But in the case of arrays the reference is passed. Same is the case for object passing to methods. Therefore, when passed through a method (function) the main variable is not changed but the main array or object is changed.

CALLING METHOD BY CREATING AN OBJECT (INSTANCE METHOD CALL)

If the method is not static, we need to create the object of the class containing a method to call it.

Syntax:

class_name object_name = new class_name(); → Object creation

object_name.method_name(list_of_arguments); → Method call upon an object.

Example:

Calculation object = new Calculation();

object.sum(number1, number2);

```
package chap05methods;

public class Eg02MethodCallByObject {
  // Declare a method
   int sum(int number1, int number2) {
     int sum;
     sum = number1 + number2;
     return sum;
  }

  public static void main(String[] args) {
     int num1 = 5, num2 = 6;

     // Process of method call
     Eg02MethodCallByObject object = new Eg02MethodCallByObject(); // Object
creation
     int mySum = object.sum(num1, num2); // Method call upon an object

     System.out.println("The sum of numbers is " + mySum);
  }
}
```

Some other terms used in methods are:

1. **Modifier** → It defines the access type of method. In Java there are four types of access modifiers: public, protected, private and default.
2. **Exception lists** → The exceptions you except by method to throw, you can specify these exceptions.

METHOD OVERLOADING

Two or more methods can have the same name but different parameters. Such methods are called overloaded methods.

Example:

```
void function() {
        // code
}

void function(int a) {
        // code
}

void function(int a, int b) {
        // code
}
```

```
package chap05methods;

public class Eg03MethodsOverloading {
  static int sum(int a, int b) {
    return a + b;
  }

  static int sum(int a, int b, int c) {
    return a + b + c;
  }

  static int sum(int a, int b, int c, int d) {
    return a + b + c + d;
  }

  public static void main(String[] args) {
    System.out.println("The sum of 4 and 5 is " + sum(4, 5));
    System.out.println("The sum of 4, 5 and 6 is " + sum(4, 5, 6));
    System.out.println("The sum of 4, 5, 6 and 7 is " + sum(4, 5, 6, 7));

  }
}
```

NOTE: Method overloading cannot be performed by changing the return type of methods i.e. return type must be same in the functions for function overloading.

TYPES OF METHODS

Methods in java can be categorized into two types:

1. Built-in methods: already defined in Java library

2. User-defined methods: defined by user according to need

1. BUILT-IN METHODS

These are methods that are already defined in the Java library. Developers do not need to create these methods; they can use them directly.

Example: The Math.max() method is a built-in method.

```
package chap05methods;

public class Eg04BuiltInMethods {
   public static void main(String[] args) {
      int maxNumber = Math.max(3, 5); // Built-in method
      System.out.println(maxNumber);
   }
}
```

2. USER-DEFINED METHODS

These are the methods that developers create to meet specific requirements. These methods enhance code modularity, readability, and reusability.

Example: A method to calculate area of circle.

```
package chap05methods;

public class Eg05AreaOfCircle {
   // Method to find the area of circle
   static double areaOfCircle(int radius) {
      return Math.PI * Math.pow(radius, 2);
   }

   public static void main(String[] args) {
      System.out.println("The are of circle with radius 3 is " +
            areaOfCircle(3));
   }
}
```

User-defined methods can be categorized in various factors like:

1. **Access Specifiers (Modifiers):**
   i. **Public:**
      - A method declared as public is accessible by all the classes within your application.
      - It has the broadest visibility.

   ii. **Private:**
      - A private method is accessible only within the class where it is defined.

- Other classes cannot directly access it.

iii. **Protected:**
- A protected method is accessible within the same package or by subclasses in different packages.
- It has the second broadest visibility.

iv. **Default (Package-Private):**
- When no access specifier is explicitly used, Java assigns the default access level. Such methods are visible only within the same package.

2. Arguments and Return Type

i. **Methods without arguments and without return value:**
- These methods do not take any arguments and do not return any value.
- They perform actions or tasks without producing a result.

ii. **Methods with arguments and without a return value**
- These methods accept arguments but do not return any value.
- They perform actions based on the provided arguments.

iii. **Methods without arguments and with return value**
- These functions do not take any arguments but return a value.
- They compute and provide a result.

iv. **Methods with arguments and returns value**
- These functions both accept the arguments and return a value.
- They compute on given arguments and provide a result.

3. Method of Calling

i. **Instance Methods:**
- These methods require an object of their class to be created before they can be called.
- They access instance data through object reference.

ii. **Static Methods:**
- Static methods can be created directly without creating an object of the class.
- They can only access static data (class-level data).

SPECIAL TYPES OF METHODS

Java have some special methods like recursion, constructor, final method, varargs, default methods and abstract methods. These methods will be discussed below.

RECURSION

- A recursion function is a special type of function which calls itself directly or indirectly.
- It solves a problem by breaking it down into smaller sub problems.
- Recursion is commonly used to solve problems that can be divided into smaller similar sub problems.

WORKING OF RECURSION

- Recursion has two cases:
  i. **Base case:** The condition where method stops calling itself and returns a value.
  ii. **Recursive case:** The condition where method call itself with a modified argument.
- The method keeps calling itself until it reaches the base condition.

PROS AND CONS

Pros:

- Provides a clean and elegant way to write code.
- Some problems are inherently recursive, such as tree traversals or the tower of Hanoi.

Cons:

- Recursion can lead to stack overflow if not handled properly.
- It may have higher resource and time usage compared to iterative solutions.

EXAMPLE

Factorial of a number → factorial(n) = n * factorial(n – 1), for all n >= 0.

```java
package chap05methods;

import java.util.Scanner;

public class Eg06FactorialUsingRecursion {
   static long findFactorial(int n) {
      if (n == 0)
         return 1;
      else
         return n * findFactorial(n - 1);
   }

   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter a number to find factorial.");
      int number = scan.nextInt();
```

```
        System.out.println("The factorial of given number is: " + findFactorial(number));
    }
}
```

VARIABLE ARGUMENTS (VARARGS)

Variable arguments (varargs) is a method to accept a variable number of arguments. This is particularly useful when you don't know how many arguments will be passed on to a method.

Syntax:

```
return_type method_name(list_of_arguments, data_type ...array_name) {
        code_to_be_executed
}
```

NOTE: ...array_name creates an array in which variable number of arguments can be passed i.e. the function can be called with variable arguments.

EXAMPLE

A function with varargs can be created in Java using the following syntax:

```
public static void sum(int ...array) {
        // array is available here as int[] array.
}
```

This function sum() can be called with zero or more arguments like this:

```
sum();
sum(1);
sum(1, 2);
sum(1, 2, 3);
and so on...
```

We can also create a function sum like this:

```
public static void sum(int a, int ...array) {
        // code
}
```

This function can be called as:

```
sum(1);
 sum(1, 2);
sum(1, 2, 3)
and so on...
```

```java
package chap05methods;

public class Eg07VariableArguments {
   static int sum(int ...arr) {
      // Available as int[] arr.
      int result = 0;
      for (int integer :
            arr) {
         result += integer;
      }
      return result;
   }

   public static void main(String[] args) {
      System.out.println("The sum of 4 and 5 is " + sum(4, 5));
      System.out.println("The sum of 4, 5 and 6 is " + sum(4, 5, 6));
      System.out.println("The sum of 4, 5, 6 and 7 is " + sum(4, 5, 6, 7));
      System.out.println("The sum of 4, 5, 6, 7 and 8 is " + sum(4, 5, 6, 7, 8));
   }
}
/*
If you want to make 1 argument compulsory, you can write:
   static int sum(int a, int ...arr) {
      // code
   }
*/
```

ABSTRACT METHODS

In Java, an abstract method is a method that is declared but does not have an implementation. Abstract methods are declared using 'abstract' keyword and are always part of an abstract class.

Some key points about abstract methods are:

- An abstract method is a method that has declaration but does not have an implementation.
- If a class contains at least one abstract method, then the class must be declared as abstract.
- Any concrete (i.e. non-abstract) class that extends an abstract class must override all the abstract methods of the class.
- If a non-abstract class extends an abstract class and does not provide implementations for all the abstract methods, then the non-abstract class must also be declared abstract.

We will discuss these methods further with abstract classes.

FINAL METHODS

In Java, a method can be declared final using 'final' keyword. These methods cannot be overridden by sub classes.

Some key points about final methods are:

- A final method cannot be overridden by sub classes.
- This is useful for the methods that are parts of the class's public API and should not be modified by subclasses.
- The use of final can sometimes improve the performance, as the compiler can optimize the code more effectively when it knows that the method cannot be changed.

We will learn more about final methods during OOPs.

DEFAULT METHODS

Default methods are introduced in Java 8, as a means to enhance the interfaces in Java without breaking the classes that implement them.

Some key points of default methods are:

- Default methods are methods within interface that have an implementation. They can be overridden by classes that implement interface.
- The primary purpose of default methods is to allow developers to add new methods to an interface without affecting the classes that implement the interface. This feature provides backward compatibility so that existing interfaces can use lambda expressions without implementing the methods in the implementation classes.
- Default methods are defined with the help of 'default' keyword.

We will learn more about default methods in interfaces.

CONSTRUCTORS

In Java, constructor is a special method that is used to initialize objects while creating them. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. Every time an object is created using 'new' operator, at least one constructor is called.

It's important to note that if you don't write a constructor, the Java compiler creates a default constructor (constructor with no arguments) for you. However, constructors can also take parameters which are used to initialize attributes but cannot return a value, so it is a method without return value.

We will learn more about constructors while creating classes.

EXERCISE

1. Write a method to calculate the sum of first n natural numbers using recursion.

```java
package chap05methods;

public class Qn01CalculateSumOfFirstNNaturalNumbers {
   static int sumOfNaturalNumbers(int n) {
      if (n == 1) {
         return 1;
      }
      return n + sumOfNaturalNumbers(n - 1);
   }

   public static void main(String[] args) {
      int sum = sumOfNaturalNumbers(100);
      System.out.println("The sum of first 100 natural numbers is " + sum);
   }
}
```

2. Write a program to print the nth term of Fibonacci series using recursion.

```java
package chap05methods;

public class Qn02NthTermOfFibonacciSeries {
   static int fibonacciNumber(int n) {
      if (n == 1){
         return 0;
      }
      else if (n == 2) {
         return 1;
      }
      /*
      if (n == 1 || n == 2) {
         return n - 1;
      }
       */
      else {
         return fibonacciNumber(n-1) + fibonacciNumber(n-2);
      }
   }

   public static void main(String[] args) {
      int num = fibonacciNumber(7);
      System.out.println("The fibonacci number at 7th position is " + num);
   }
}
```

3. Write a method to find the roots of quadratic equations.

```java
package chap05methods;

import java.util.Scanner;

import static java.lang.Math.sqrt;
import  static java.lang.Math.abs;

public class Qn03RootsOfQuadraticEquations {
   // Method to calculate roots of quadratic equations
   static void rootsOfQuardatic(float a, float b, float c) {
      float discriminant = b * b - 4 * a * c;

      // Check nature of roots
      if (discriminant > 0) {
         float root1 = (float) ((-b + sqrt(discriminant)) / (2 * a));
         float root2 = (float) ((-b - sqrt(discriminant)) / (2 * a));
         System.out.println("Roots are real and distinct.");
         System.out.println("Root1 = " + root1);
         System.out.println("Root2 = " + root2);
      } else if (discriminant == 0) {
         float root = (float) (-b / (2 * a));
         System.out.println("Roots are real and equal.");
         System.out.println("Roots = " + root);
      }
      else {
         float realPart = (float) (-b / (2 * a));
         float imaginaryPart = (float) (sqrt(abs(discriminant))/ (2 * a));
         System.out.println("Roots are complex and conjugate of each other.");
         System.out.printf("Root1 = %f + %f i\n", realPart , imaginaryPart);
         System.out.printf("Root2 = %f - %f i\n", realPart , imaginaryPart);
      }
   }

   public static void main(String[] args) {
      // Input coefficients from the user
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter coefficients of quardatic equation (a, b, c).");
      float a = scan.nextInt();
      float b = scan.nextInt();
      float c = scan.nextInt();

      // Call the method to find roots
      rootsOfQuardatic(a, b, c);
   }
}
```

4. Write a method to find out if the number is prime or not.

```
package chap05methods;

import java.util.Scanner;

public class Qn04FindPrimeOrNot {
  // Method to check if a number is prime or not
  static boolean isPrime(int num) {
    if (num <= 1)
      return false; // 1 and numbers less than 1 are not prime

    for (int i = 2; i * i <+ num; i++)
      if (num % i == 0)
        return false; // If num is divisible by any number from r to sqrt(num), it is not
prime

    return true; // If no divisors found, the number is prime
  }

  public static void main(String[] args) {
    // Input a number from user
    System.out.println("Enter a number.");
    Scanner scan = new Scanner(System.in);
    int number = scan.nextInt();

    // Check if the number is prime using isPrime method
    if (isPrime(number))
      System.out.println(number + " is a prime number.");
    else
      System.out.println(number + " is not a prime number.");
  }
}
```

5. Write a program to find the average of a set of variable numbers passed as arguments.

```
package chap05methods;

public class Qn05AverageOfNumbers {
  static float averageNumber(int ...arr) {
    int result = 0;
    for (int a :
        arr) {
      result += a;
    }
    return (float)result / arr.length;
  }
```

```
    public static void main(String[] args) {
        float a = averageNumber(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        float b = averageNumber(1, 2, 3, 4, 5, 6, 7);
        float c = averageNumber(1, 2, 3, 4, 5);
        float d = averageNumber(1, 2, 3);

        System.out.println("The average of first 10 natural numbers is " + a);
        System.out.println("The average of first 7 natural numbers is " + b);
        System.out.println("The average of first 5 natural numbers is " + c);
        System.out.println("The average of first 3 natural numbers is " + d);
    }
}
```

6. Write a method to print triangular patterns.

```
/*
This program will print the following pattern.
  *
  * *
  * * *
  * * * *
  * * * * *
 */

package chap05methods;

public class Qn06PrintTriangularStarPattern {
    static void starPattern(int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i + 1; j++) {
                System.out.print("*  ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        starPattern(5);
    }
}
```

7. Print triangular pattern using recursion.

```
/*
This program will print the following pattern using recursion.
  *
  * *
  * * *
  * * * *
  * * * * *
```

```
*/

package chap05methods;

public class Qn07TriangularStarPatternUsingRecursion {
    static void printStar(int n) {
        if (n > 0) {
            printStar(n - 1);
            for (int i = 0; i < n; i++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        printStar(5);
    }
}

/*
---> printingStars(3)
---> printingStars(2) + 3 times * and new line
---> printingStars(1) + 2 times * and new line + 3 times * and new line
---> printingStars(0) + 1 time * and new line + 2 times * and new line + 3 times * and
new line
 */
```

8. Write a method to find if a number is palindrome or not.

```
package chap05methods;

import java.util.Scanner;

public class Qn08FindPalindromeOrNot {
    // Method to check if a number is palindrome or not
    static boolean isPalindrome(int num) {
        // No negative number is palindrome
        if (num < 0) return false;

        int originalNum = num, reversedNum = 0, remainder;

        while (num != 0) {
            remainder = num % 10;
            reversedNum = reversedNum * 10 + remainder;
            num /= 10;
        }
        return originalNum == reversedNum;
    }

    public static void main(String[] args) {
        // Input from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a number.");
        int number = scan.nextInt();

        // Check if the number is palindrome or not
        if (isPalindrome(number)) System.out.println(number + " is palindrome.");
        else System.out.println(number + " is not palindrome.");
    }
}
```

9. Write methods to find LCM and HCF of two numbers.

```
package chap05methods;

import java.util.Scanner;

public class Qn09FindHCFAndLCM {
    // Method to calculate HCF
    static int calculateHCF(int num1, int num2) {
        while (num1 != num2) {
            if (num1 > num2) num1 -= num2;
            else num2 -= num1;
        }
        return num1;
    }

    // Method to calculate LCM
    static int calculateLCM(int num1, int num2) {
        return (num1 * num2) / calculateHCF(num1, num2);
    }

    public static void main(String[] args) {
        // Take input from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter any two numbers.");
        int number1 = scan.nextInt();
        int number2 = scan.nextInt();

        // Calculate and display the HCF and LCM
        System.out.println("The HCF of given numbers is " + calculateHCF(number1,
number2));
        System.out.println("The LCM of given numbers is " + calculateLCM(number1,
number2));
```

```
  }
}
```

10. Write a method to convert decimal numbers to binary.

```
package chap05methods;

import java.util.Scanner;

public class Qn10ConvertDecimalNumberToBinary {
  // Method to convert decimal to binary
  static void decimalToBinary(int decimal) {
    if (decimal == 0) {
      System.out.println("Binary Equivalent: 0");
      return;
    }

    byte[] binary = new byte[32]; // Assuming 32-bit integer
    int i = 0;

    while (decimal > 0) {
      binary[i] = (byte) (decimal % 2);
      decimal /= 2;
      i++;
    }

    System.out.print("Binary equivalent: ");
    for (int j = i - 1; j >= 0; j--)
      System.out.print(binary[j]);
  }

  public static void main(String[] args) {
    // Input from user
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter a decimal number.");
    int decimalNumber = scan.nextInt();

    // Convert and display binary equivalent
    decimalToBinary(decimalNumber);
  }
}
```

11. Write a method to find factorial using recursion and find combination.

```
package chap05methods;

import java.util.Scanner;

public class Qn11FindCombination {
  // Method to calculate factorial using recursion
```

```
    static long factorial (int n) {
        // Base case: factorial of 0 is 1
        if (n == 0) return 1;
        else return n * factorial(n - 1); // Recursive case
    }

    // Method to calculate combination
    static int nCr(int n, int r) {
        // nCr = n! /(r! * (n-r)!)
        return (int) (factorial(n) / (factorial (r) * factorial(n-r)));
    }

    public static void main(String[] args) {
        // Input from user
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter values for n and r in nCr.");
        int n = scan.nextInt();
        int r = scan.nextInt();

        // Check n is greater than or equal to r and r is greater than or equal to 0
        if (n >= r && r >= 0) System.out.println("The combination is " + nCr(n, r));
        else System.out.println("Invalid input. Ensure n >= r and r >= 0");
    }
}
```

12. Print reverse triangular star pattern using recursion.

```
/*
This program will print the following pattern using recursion.
    * * * * *
    * * * *
    * * *
    * *
    *
 */

package chap05methods;

public class Qn12ReverseTriangularStarPatternUsingRecursion {
    static void printStar(int n) {
        if (n > 0) {
            for (int i = 0; i < n; i++) {
                System.out.print("*  ");
            }
            System.out.println();
            printStar(n - 1);
        }
    }
    public static void main(String[] args) {
        printStar(5);
    }
}
```

13. Write a method to print reverse triangular star pattern.

```
/*
This program will print the following pattern.
* * * * *
* * * *
* * *
* *
*
*/

package chap05methods;

public class Qn13PrintReverseTriangularStarPattern {
    static void starPattern(int n) {
        for (int i = n; i != 0; i--) {
            for (int j = i; j != 0; j--) {
                System.out.print("*  ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        starPattern(5);
    }
}
```

CHAP 06 – OBJECT ORIENTED PROGRAMMING

Solving a problem by creating objects is one of the most popular approaches in programming. This is called Object Oriented Programming. Object Oriented Programming tries to map the code instructions with real world, making the code short and easier to understand. It simplifies software development and maintenance. It follows the DRY concept.

DRY: DRY stands for Don't Repeat Yourself i.e. it focuses on code reusability.

NOTE:

- Extra step → Time increases
- Extra variable → Needs memory
- Large no. of calculations → TIME COMPLEXITY (More processing time)
- More intermediate results → SPACE COMPLEXITY (More memory consumption)
- Functions → follows camelCaseConvention
- Class → follows PascalConvention

Why OOPs? → To achieve modular programming.

Why modular programming? → To write organized programs with help from classes and objects.

OOPS TERMINOLOGIES

The core concepts of OOPs (Object Oriented Programming System) are:

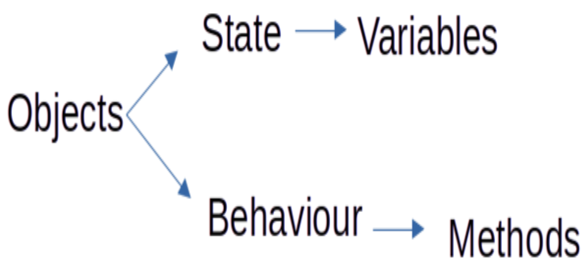
1. Object
2. Classes

Other terms used in OOPs are:

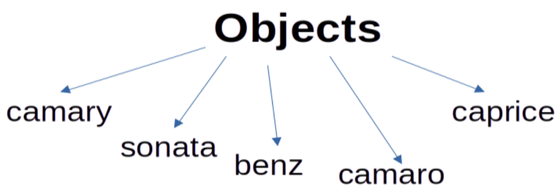
1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

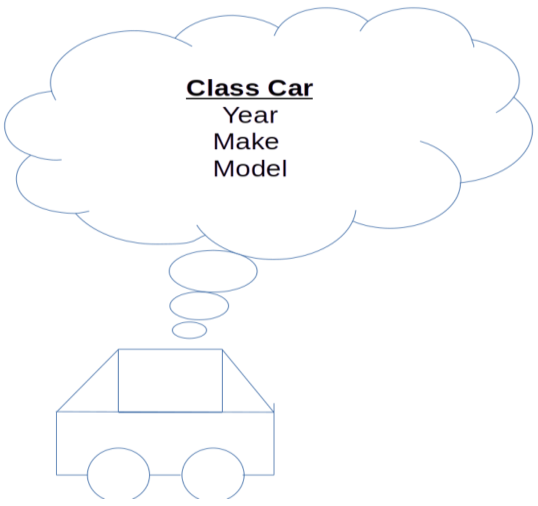
OBJECT AND CLASSES

- Object is a real-world entity.
- Objects have states and behaviors. e.g. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail).



- An object stores its state in fields (variables) and exposes its behaviors through methods (functions).
- Generally, class is a blueprint or a template, or a factory producing objects. A class contains info to create a void object.
- A class is a blueprint of an object. When you write a class, you are describing how JVM (Java Virtual Machine) should make an object of that type.





Dog	CLASS
breed size age color	DATA MEMBERS (STATES)

eat()
sleep()
sit()
run()
	METHODS (BEHAVIOURS)

NOTE: An object is an instantiation of a class. When a class is defined, a template (info) is defined. Memory is located only after object instantiation.

HOW TO MODEL A PROBLEM IN OOPS?

Noun ➔ Class → Employee

Adjective ➔ Attributes → name, age, salary, etc.

Verb ➔ Methods → getSalary(); increment();, etc.

ABSTRACTION

- Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car and applying brakes will stop the car, but he doesn't know how on pressing accelerator the speed is increasing, he doesn't know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.

❖ By restricting direct access to sensitive data and behaviors, it enhances security, prevents unintended misuse, and makes APIs simpler and more intuitive, ultimately improving the end-user and developer experience.

ENCAPSULATION

❖ Encapsulation is defined as wrapping up data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is, it is protective shield that prevents the data from being accessed by the code outside the shield.

❖ Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.

❖ In Java, encapsulation simply means that the sensitive data can be hidden from users.

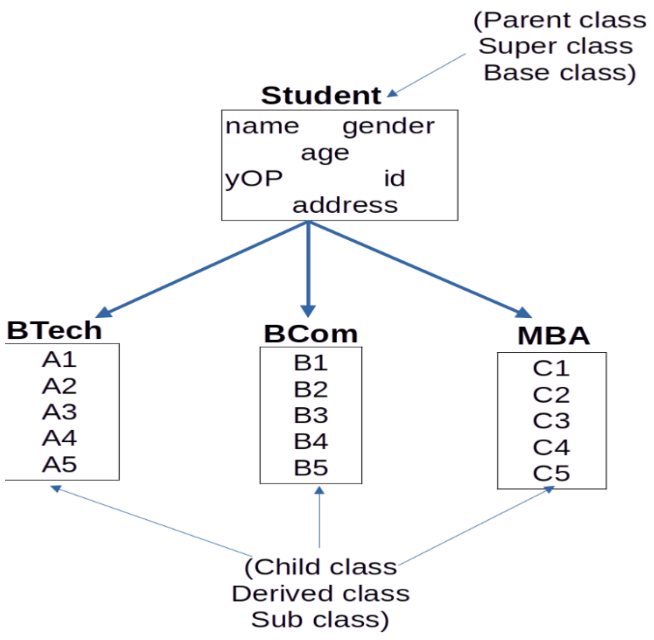
INHERITANCE

❖ It is the act of deriving new things from existing things.

❖ Classes and interfaces in Java can perform inheritance.

❖ There are a total of 5 types of inheritance:

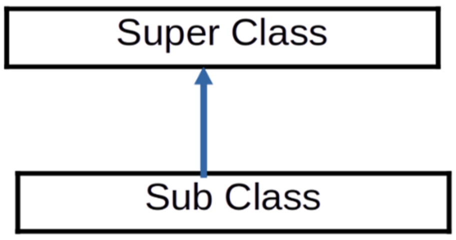
- Single inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance
- Multiple inheritance



However, Java doesn't support multiple and hybrid inheritances. In other languages they can be allowed but not recommended.

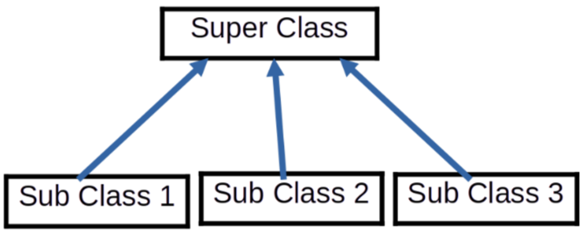
SINGLE INHERITANCE

- In single inheritance, a sub class is derived from only one super class.
- It inherits the properties and behavior of a single parent class.



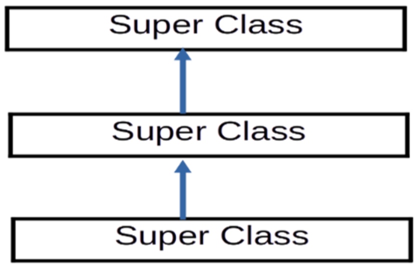
HIERARCHICAL INHERITANCE

- In hierarchical inheritance, multiple classes share a single super class.
- Each of these classes extends to the same base class.



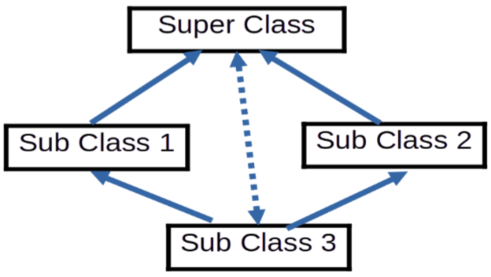
MULTILEVEL INHERITANCE

- In multilevel inheritance a class is derived from a class which is also derived from another class.
- It is like a chain of inheritance.



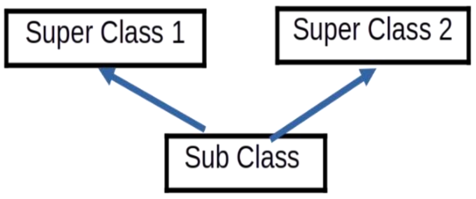
HYBRID INHERITANCE

- Hybrid Inheritance is a combination of more than one type of inheritance.
- It may be the mix of single, multilevel, hierarchical inheritance.



MULTIPLE INHERITANCE

- It is a type of inheritance in which a class can inherit the properties of two or more parent classes.
- It is supported in only a few Object-Oriented Programming languages.



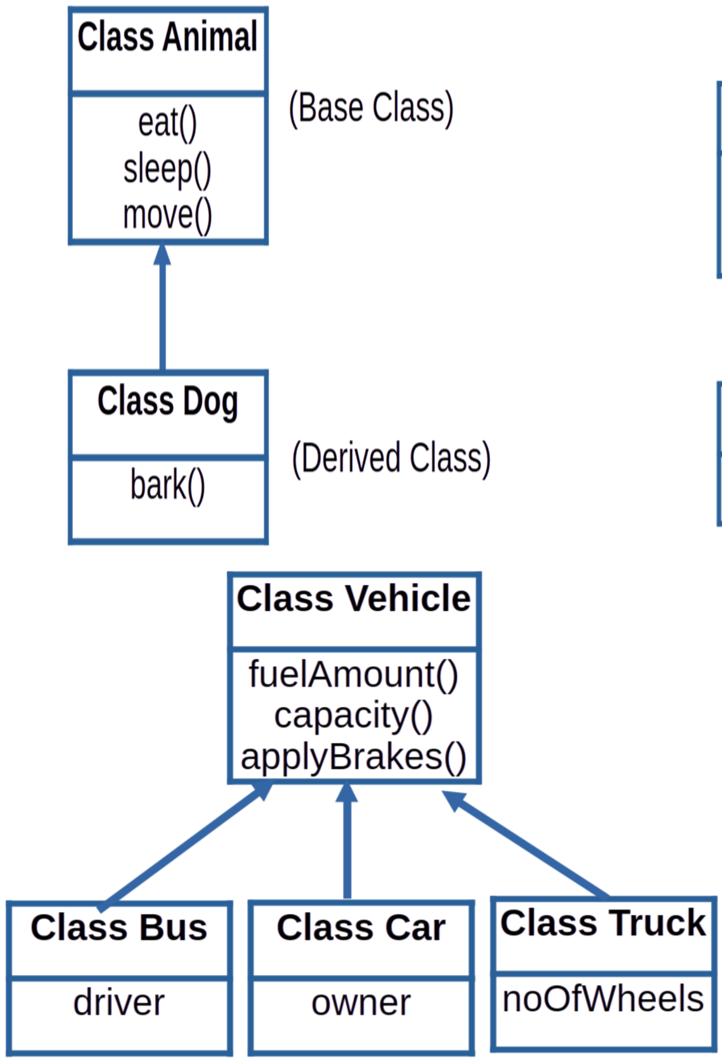
NOTE FOR JAVA DEVELOPERS

Hybrid and multiple inheritance are not supported in Java. It is because of "Diamond Problem" which occurs when there exits method with the same signature in both the super classes and sub class. On calling the method, the compiler can't determine which class method to be called and even on calling which class method gets the priority.

However, Java offers a way to achieve multiple inheritance through interfaces, allowing a class to implement multiple interfaces.

NOTE: A class in Java can extend from only one class but can implement any number of interfaces.

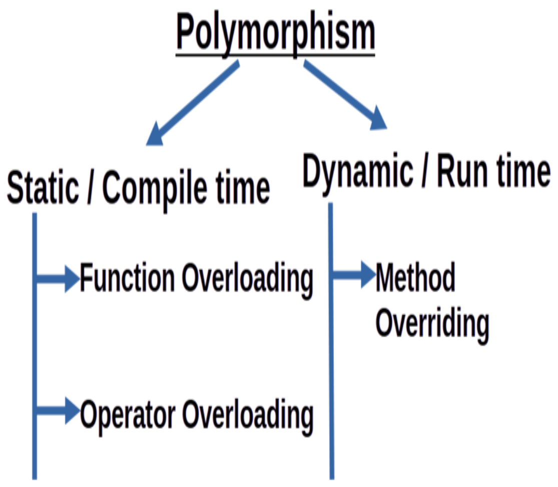
SOME REAL-LIFE EXAMPLES OF INHERITANCE



POLYMORPHISM

❖ The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Example:
Animal a = new Dog(); // Dog extends Animal
a.speak(); // invokes Dog's overridden speak() at runtime



METHOD OVERLOADING

Method overloading occurs in Java when there are the methods having the same name but having different number of parameters passed to it, which can be different in data like int, double, float and used to return different values are computed inside the respected overloaded method. Two or more methods can have the same name inside the same class if they accept different arguments. This feature is known as method overloading.

NOTE: Method overloading is achieved either by changing the number of arguments or changing the data type of arguments not by changing the return type of method.

Example:

```
sum(int a, int b);
sum(int a, int b, int c);
sum(int a, int b, int c, int d);
sum(int a, int b, int c, int d, int e);
```

```java
package chap06oop;

public class Eg01MethodOverloading {
   static double sum(int a, int b) { return a + b; }
   static double sum(int a, int b, int c) { return a + b + c; }
   static double sum(float a, float b) { return a + b; }
   static double sum(float a, float b, float c) { return a + b + c; }

   public static void main(String[] args) {
      System.out.println("2 + 3 = " + sum(2, 3));
      System.out.println("2 + 3 + 4 = " + sum(2, 3, 4));
      System.out.println("2.5 + 3.6 = " + sum(2.5f, 3.6f));
      System.out.println("2.5 + 3.6 + 4.7 = " + sum(2.5f, 3.4f + 4.7f));
   }
}
```

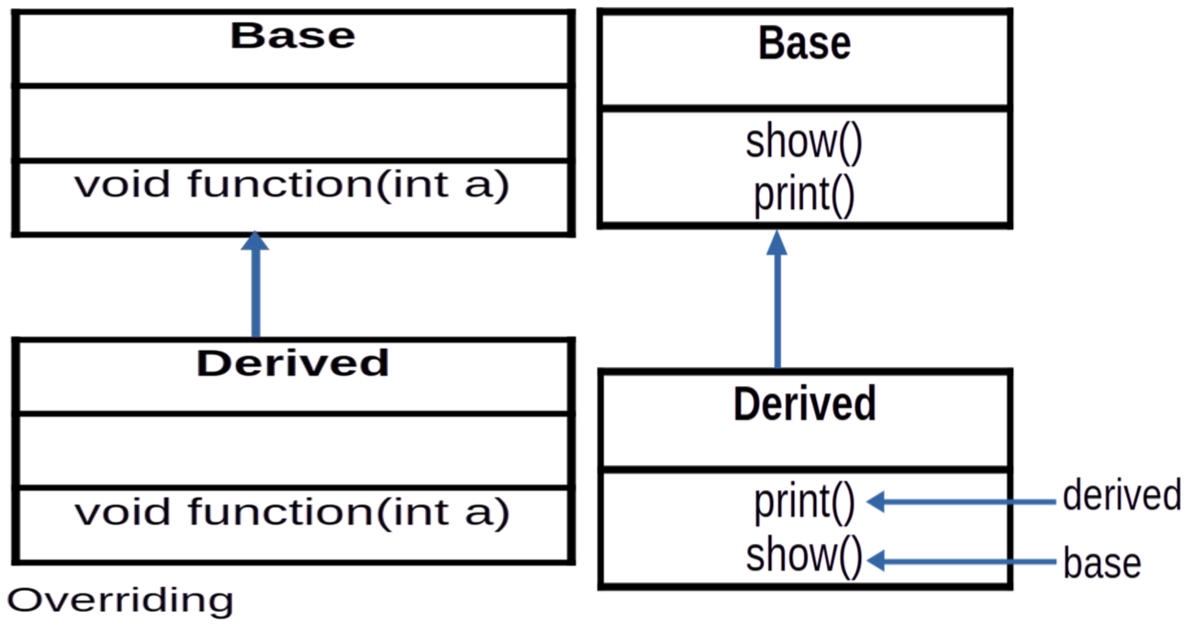
OPERATOR OVERLOADING

Operator overloading is a programming method where the operators are implemented in user-defined types with specific logic dependent on the types of given arguments. However, Java doesn't support operator overloading. The only aspect of Java which comes close to operator overloading is the handling of '+' for strings. A side from that, user defined operator overloading is not supported in Java. The handling of '+' for concatenating strings is only component of Java that is close to the operator overloading.

NOTE: Java doesn't support operator overloading. But `+` operator in Java can both do arithmetic addition and concatenate Strings.

METHOD OVERRIDING

If a class has function same as its parent class, then method overriding takes place. It replaces parent class function with child class function.
When a method in a subclass has the same name, the same parameters or signature, and the same return type as mentioned in the super class. Method overriding is one of the ways by which Java achieves Runtime Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If the object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of child class is used to invoke the method, then the version in the child class will be executed.



CREATING OUR OWN CLASS

You can create your own classes in Java, which can be further used to create objects. Here is a basic overview of how to create a class in Java.

1. **Class Definition:** A class is declared by using class keyword. The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called the members of the class.
2. **Creating an Object:** Once you have defined the class, you can create objects from the class blueprint with the 'new' operator.
3. **Accessing Class Members:** After you have created an object, you can use the dot operator (.) to access the object's variables and methods.
4. **Constructors:** A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of the class is created.

5. **Encapsulation:** Encapsulation in Java is a mechanism of wrapping the data (variables) together as a single unit. Declare the variables of a class as private and provide public setter and getter methods to modify and view the variable values.

NOTE:

- A java file can have only one public class.
- Comparing object in OOPs with real world object:
  Any real-world object = Properties + Behaviors
  Object in OOPs = Attributes + Methods

Syntax of Creating a Class:

```
modifier class class_name {
        attributes
        methods
}
```

Example: A class Student with attributes roll, name, address, and method printDetails().

```
class Student {
        public int roll;
        public String name, address;
        public void printDetails() {
                // Code
        }
}
```

```
package chap06oop;

import java.util.Scanner;

class Student {
   byte roll;
   String name, address;

   public void printDetails() {
      System.out.println("The roll of student is " + roll);
      System.out.println("The name of student is " + name);
      System.out.println("The address of student is " + address);
   }
}
```

```java
public class Eg02CreatingOwnClass {
  public static void main(String[] args) {
    Student student = new Student(); // Instantiating a new Student object

    // Setting attributes for student
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter roll, name and address of student.");
    student.roll = scan.nextByte();
    scan.nextLine(); // To consume the left over new line character
    student.name = scan.nextLine();
    student.address = scan.nextLine();

    // Printing details of student
    System.out.println("\nThe details of students are: ");
    student.printDetails();
  }
}
```

ACCESS MODIFIERS

Access modifiers in Java determine the scope of a class, constructor, variable, method, or data member. They provide security and accessibility depending on the access modifier used with the element.

There are four types of access Modifiers: default, private, protected and public.

DEFAULT

- When no access modifier is specified for a class, method, or data member, it is said to have default access modifier.
- The data members, classes, or methods that are not declared using access modifiers, i.e. having default access modifiers, are accessible only within the package.
- It is also called a package private access modifier.

PRIVATE

- The private access modifier is specified using the keyword ‘private’.
- The methods and data members declared as private are accessible only within the class in which they are declared.
- Any other class of the same package will not be able to access these members.

PROTECTED

- The protected access modifier is specified using keyword ‘protected’.
- The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

PUBLIC

- The public access modifier is specified using the keyword 'public'.
- The methods or data members declared as public are accessible from everywhere.
- They can be accessed from the class within the class, outside class, within package and outside the package.
- Note that a java file can contain a single public class.

SUMMARY

Access modifiers determine whether classes can use a particular field or invoke a particular method.

Modifiers	Class	Package	Subclass	World
public	YES	YES	YES	YES
protected	YES	YES	YES	NO
default	YES	YES	NO	NO
private	YES	NO	NO	NO

GETTERS AND SETTERS

In Java, getter and setter are two conventional methods that are used for retrieving and updating the value of a variable. They are a part of Java's encapsulation feature which helps in data hiding.

GETTER

- This method is used to retrieve the value of a variable.
- It starts with the word "get" followed by the variable name, with the first letter of the variable is uppercase.

Example:

```
public String getName() {
        return name;
}
```

SETTER

- This method is used to set or update the value of a variable.
- It starts with the word "set" followed by the variable name, with the first letter of the variable in uppercase.

Example:

```
public void setName(String newName) {
```

```
            this.name = newName;
        }
```

NOTE

- Getter and setter are written inside class to access the private variables from outside the class.
- Getter → returns the value (accessor)
- Setter → set/update value (mutator)

```
package chap06oop;

import java.util.Scanner;

class Employee {
   private int id;
   private float salary;
   private String name, address;

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public float getSalary() {
      return salary;
   }

   public void setSalary(float salary) {
      this.salary = salary;
   }

   public String getName() {
      return name;
   }

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

   public String getAddress() {
      return address;
   }
```

```java
    public void setAddress(String address) {
        this.address = address;
    }
}

/*
'this' keyword is optional if the variable names are different but compulsory if variable
names are same.
For example:
1. this is compulsory in name = name i.e. this.name = name;
2. this is not compulsory in name = n i.e. name = n also works.
*/

public class Eg03GettersSetters {
    public static void main(String[] args) {
        Employee employee = new Employee();

        /*
        employee.id = 56;
        employee.name = "Kushal Prasad Joshi";
        employee.address = "Nepal";
        employee.salary = 100000;
        // This will throw error due to private access modifiers.
        // Also, you cannot access their values.
        System.out.println(employee.id); // also throws an error
         */

        // So we use getters and setters to access private variables.
        employee.setId(45);
        employee.setName("Kushal Prasad Joshi");
        employee.setAddress("Nepal");
        // Lets take salary as an input
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter salary of " + employee.getName());
        employee.setSalary(scan.nextFloat());

        // Print the details of employee
        System.out.println("Employee id: " + employee.getId());
        System.out.println("Employee name: " + employee.getName());
        System.out.println("Employee address: " + employee.getAddress());
        System.out.println("Employee salary: " + employee.getSalary());
    }
}
```

THIS KEYWORD

‘this’ keyword is a way for us to reference an object of the class which is being created/referred. ‘this’ is a reference variable that refers to current object.

Some common uses of ‘this’ keyword are as follows:

1. **Refer to the current class instance variable:** If there is ambiguity between instance variables and parameters, `this` keyword resolves the problem of ambiguity.
   Example:
   ```
   public class Main {
           int x;    // Instance variable

           public setX( int x) {     // A method with a parameter
                   this.x = x;       // ‘this’ refers to the instance variable
           }
   }
   ```
   In the above example, ‘this.x’ refers to the instance variable x and ‘x’ refers to the parameter of method.
2. **Invoke the current class method:** You may invoke the method of the current class using ‘this’ keyword.
3. **Invoke the current class constructor:** ‘this()’ can be used to invoke the current class constructor.
4. **Pass an argument in the method call:** ‘this’ can be passed as an argument in the method call.
5. **Pass an argument in the constructor call:** ‘this’ can be passed as an argument in the constructor call.
6. **Return the current class instance from the method:** ‘this’ can be used to return the current class instance from the method.

NOTE:

‘this’ keyword is optional if the variable names are different but compulsory if variable names are same to recognize the instance variable.

For example:

1. ‘this’ is compulsory in name = name i.e. this.name = name;
2. ‘this’ is not compulsory in name = n i.e. name = n; also works.

```
package chap06oop;

class MyClass {
  private int number; // Instance variable

  public int getNumber() {
```

```
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
        // this.number refers to instance variable and number refers to parameter of setter
    }
}

public class Eg04ThisKeyword {
    public static void main(String[] args) {
        MyClass myObject = new MyClass();
        myObject.setNumber(45);
        System.out.println(myObject.getNumber());
    }
}
```

CONSTRUCTORS

In Java, constructor is a special method that is used to initialize objects while creating them. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. Every time an object is created using 'new' operator, at least one constructor is called.

It's important to note that if you don't write a constructor, the Java compiler creates a default constructor (constructor with no arguments) for you. However, constructors can also take parameters which are used to initialize attributes but cannot return a value, so it is a method without return value.

WRITING OUR OWN CONSTRUCTOR

To write our own constructor, we define a method with the same name as class name.

Example:

```
class Student {
        int roll;
        String name;

        public Students (int roll, String name) {
                this.roll = roll;
                this.name = name;
        }
}
```

// This constructor in class has two parameters roll and name. So, while creating a new object, we can call this constructor to initialize roll and name like:

Student student = new Student(45, “Kushal”);

// While creating a new object it will set roll 45 and name Kushal.

```
package chap06oop;

class Students {
   int roll;
   String name;

   // Constructor with two parameters: roll and name
   public Students(int roll, String name) {
      this.roll = roll;
      this.name = name;
   }
}

public class Eg05ConstructorInJava {
   public static void main(String[] args) {
      Students student = new Students(45, "Kushal");

      System.out.println("Roll of student is " + student.roll);
      System.out.println("Name of student is " + student.name);
   }
}
```

CONSTRUCTORS OVERLOADING IN JAVA

Constructors overloading in Java is a technique where a class can have multiple constructors, each with a different parameter list. Constructors in Java can be overloaded just like other methods in Java. This allows you to create objects using different initialization values or different ways of creating an object.

NOTE:

- Constructors can take parameters without being overloaded.
- There can be more than two overloaded constructors.

Example:

```
class Student {
        int roll;
        String name;

        public Student(int roll) {        // Constructor initializing roll
```

```
                this.roll = roll;
            }

            public Student(int roll, String name) {  // Constructor initializing roll and
name
                this.roll = roll;
                this.name = name
            }
        }
```

```
package chap06oop;

class MyStudent {
   int roll;
   String name;

   // Constructor with one parameter
   public MyStudent(int roll) {
      this.roll = roll;
   }

   // Constructor with two parameters
   public MyStudent(int roll, String name) {
      this.roll = roll;
      this.name = name;
   }
}

public class Eg06ConstructorOverloading {
   public static void main(String[] args) {
      MyStudent student1 = new MyStudent(40); // Initialize roll of student to 45
      MyStudent student2 = new MyStudent(45, "Kushal"); // Initialize roll of student to
45 and name to Kushal

      // Print details initialized
      System.out.println("Roll of student1: " + student1.roll);
      System.out.println("Roll of student2: " + student2.roll);
      System.out.println("Name of student2: " + student2.name);
   }
}
```

INHERITANCE IN JAVA

Inheritance in Java is used to borrow properties and methods from an existing class. It allows one class (sub class) to inherit the features (fields and methods) of another class (super class). This mechanism promotes code reusability and enables polymorphism.

Example:

```
// Inheriting class Cuboid from class Rectangle
class Cuboid extends Rectangle {
        // methods and fields
}
```

```
package _09_OOPs;

class Rectangle {
   int length, breadth;

   int getArea() {
      return length * breadth;
   }
}

class Cuboid extends Rectangle {
   int height;

   int getVolume() {
      return length * breadth * height;
   }
}

public class Eg07IntroductionToInheritance {
   public static void main(String[] args) {
      // Finding area of a Rectangle
      Rectangle rectangle = new Rectangle();
      rectangle.length = 45;
      rectangle.breadth = 34;
      System.out.println("The area of rectangle is " + rectangle.getArea());

      // Finding volume of Cuboid
      Cuboid cuboid = new Cuboid();
      cuboid.length = 23;     // Inherited from class Rectangle
      cuboid.breadth = 12;    // Inherited from class Rectangle
      cuboid.height = 32;
      System.out.println("The volume of cuboid is " + cuboid.getVolume());
   }
}
```

CONSTRUCTOR IN INHERITANCE

In Java, when a Derived class is extended from the Base class, the constructor of Base class is also inherited.

Here's how constructor works in inheritance:

- When an object of a subclass is created, the default constructor of the parent class is called automatically.
- If parameterized constructor is defined in the base class, it can be called using 'super ()'.
- The base class constructor must be called in the first line of derived class constructor.

NOTE: When a Derived class is extended from the Base class, the constructor of Base class is executed first followed by the constructor of Derived class. For the following inheritance hierarchy, the constructors are executed in the order: C1 → C2 → C3.

C1 (parent) → C2 (child) → C3 (grandchild)

Example:

```java
package chap06oop;

class Parent {
   Parent() {
      System.out.println("I'm constructor of Parent class.");
   }
}

class Child extends Parent {
   Child() {
      System.out.println("I am constructor of Child class.");
   }
}

class GrandChild extends Child {
   GrandChild() {
      System.out.println("I'm constructor of GrandChild class.");
   }
}

public class Eg08ConstructorInInheritance {
   public static void main(String[] args) {
      GrandChild grandChild = new GrandChild();
   }
}
```

CONSTRUCTOR OVERLOADING DURING INHERITANCE

While there are multiple constructors in the parent class, the constructor without any parameter is called from the child class. If we want to call the constructor with parameters from the parent class, we can use 'super' keyword.

Syntax: super(list_of_arguments); // This will call the constructor from the parent class which has equivalent parameters.

Example: super(x, y);

```java
package chap06oop;

class Base {
   Base(int y, int z) {
      System.out.println("The sum of x and y is " + (y + z));
   }
}

class Derived extends Base {
   Derived(int x, int y, int z) {
      super(y, z);
      System.out.println("The sum of x, y and z is " + (x + y + z));
   }
}

class DerivedFromDerived extends Derived {
   DerivedFromDerived(int w, int x, int y, int z) {
      super(x, y, z);
      System.out.println("The sum of w, x, y and z is " + (w + x + y + z));
   }
}

public class Eg09ConstructorOverloadingDuringInheritance {
   public static void main(String[] args) {
      // Instantiation of object of DerivedFromDerived class
      DerivedFromDerived object = new DerivedFromDerived(1, 2, 3, 4);
   }
}
```

SUPER KEYWORD

The 'super' keyword in Java is a reference variable that is used to refer to the immediate parent class object. It is particularly useful in the context of inheritance where a sub class inherits properties and behaviors from super class.

Its uses are:

1. **Refer to parent class instance variable:** If the derived class and base class have the same data members, 'super' can be used to refer to the parent class's variable.
2. **Invoke parent class method:** If the subclass contains the same method as the parent class, 'super' can be used to call the parent class's method. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.
3. **Invoke parent class constructor:** 'super ()' can be used to invoke the parent class constructor. *Remember, when calling a super class constructor, the super() must be in the first statement in the constructor of subclass.*

NOTE: 'super' keyword cannot be used in static context, such as in a static method or static variable initializer. *'super' keyword works only inside extended classes.*

```
package chap06oop;

class Vehicle {
   int maxSpeed = 120;
}

class Car extends Vehicle {
   int maxSpeed = 180;
   int maxSpeedOfVehicle = super.maxSpeed;
}

public class Eg10SuperKeyword {
   public static void main(String[] args) {
      // Instantiation of the object of class Car
      Car car = new Car();

      System.out.println("The max speed of car is " + car.maxSpeed);
      System.out.println("The max speed of vehicle is " + car.maxSpeedOfVehicle);
   }
}
```

DYNAMIC MEMORY DISPATCH

Dynamic memory dispatch, also known as runtime polymorphism, is a mechanism in Java that allows the resolution of overridden methods at runtime.

METHOD OVERRIDING IN JAVA

Method overriding is a process of redefining the method of super class in sub class. If a child class implements the same method present in parent class again, it is known as a method overriding. When an object of subclass is created and an overridden method is called, the method which has been implemented in the subclass is called and its code is executed.

```
package chap06oop;

class LivingBeing{
   void eatFood() {
      System.out.println("Eat food to survive.");
   }
}

class Carnivorous {
   void eatFood() {
      System.out.println("Eat flesh to survive.");
   }
}

public class Eg11MethodOverridingInJava {
   public static void main(String[] args) {
      Carnivorous animal = new Carnivorous();
      animal.eatFood();
      // Display --> Eat flesh to survive.
   }
}
```

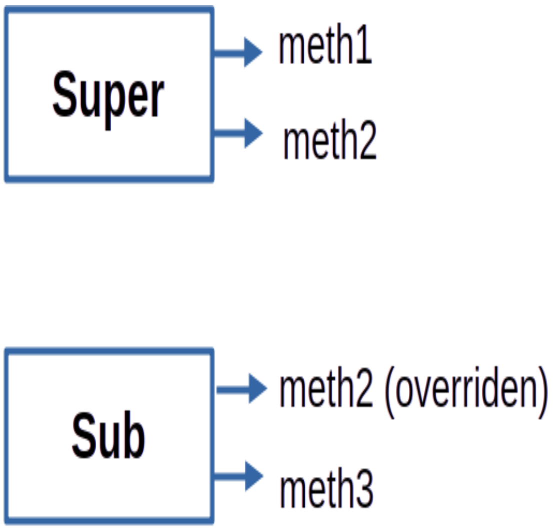
Note that static methods cannot be overridden because they belong to the class, not the instance of the class. If you declare the static method in the subclass, it hides the super class's method rather than overriding it. This is known as method hiding.

Method overriding is one of the ways to achieve Runtime Polymorphism. The JVM determines the proper method to call at the program's runtime, not at the compile time. That's why it is also known as dynamic memory dispatch.

HOW JAVA SUPPORTS METHOD OVERRIDING?

Method overriding is central to runtime polymorphism. When an overridden method is called through a super class reference, Java determines which version (superclass or subclass) of that method is to be executed based on type of the object being referred to at the time the call occurs. This determination is made at runtime.

Consider the following inheritance hierarchy:



Scenario 1:

Super obj = new Sub(); → Allowed

obj.meth1() → Allowed

obj.meth2(); → meth2 of Sub class is called (the method of object not reference)

obj.meth3(); → Not allowed

Scenario 2:

Sub obj = new Super(); → Not allowed

Example of Dynamic Memory Dispatch in Java:

```
package chap06oop;

class Phone {
   public void on() {
      System.out.println("Turning on phone...");
   }
   public void showTime() {
      System.out.println("It's 8'o clock.");
   }
}

class SmartPhone extends Phone {
   @Override
   public void on() {
      System.out.println("Turning on smartphone...");
   }
   public void playMusic() {
      System.out.println("Playing music...");
   }
}


public class Eg12DynamicMethodDispatch {
   public static void main(String[] args) {
      Phone obj = new Phone();   // Allowed
      SmartPhone obj1 = new SmartPhone();   // Allowed
      obj.showTime(); // Allowed
      obj1.playMusic(); // Allowed
      Phone obj2 = new SmartPhone();  // Yes!! It is Allowed
//       SmartPhone obj3 = new Phone();  // Not Allowed
      obj2.showTime(); // Allowed
      obj2.on(); // Allowed and method of Smartphone (subclass) will be executed.
//       obj2.playMusic(); // Not allowed because object reference is Phone doesn't
contain this method.
   }
}
```

ABSTRACT CLASSES IN JAVA

What does Abstract (class) means? Abstract means existing in thoughts. It is an idea without concrete existence.

ABSTRACT METHOD

A method that is declared without implementation.

```
abstract void move(double x, double y);
```

// Details are in methods chapter.

ABSTRACT CLASS

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class PhoneModel {
        abstract void switchOff();
        // more code
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all the methods in parent class. If it doesn't, it must be declared abstract.

Some key points about abstract class in Java are:

- An abstract class cannot be instantiated by itself; it needs to be subclassed by another class to use its properties.
- If you try to instantiate the abstract class directly, you need to provide implementations for all the abstract methods in it.
- If a class contains at least one abstract method, then it must be declared as an abstract class.
- An abstract class can have constructors and static methods.
- It can also have final methods which will force the subclass not to change the body of the method.

NOTE:

- It is possible to create a reference of an abstract class.
- It is not possible to create an object of abstract class.
- We can also assign reference of an abstract class to the object of a concrete subclass.

```
package chap06oop;

abstract class Shapes {
   int dimension;

   abstract int getArea();
}

class Square extends Shapes {
   @Override
   int getArea() {
      return dimension * dimension;
   }
}

public class Eg13AbstractClass {
   public static void main(String[] args) {
      Square square = new Square();
      square.dimension = 5;
      System.out.println("The area of square is " + square.getArea());

//       Shapes shapes = new Shape(); // Not allowed

      Shapes shape = new Square(); // Allowed
      shape.dimension = 23; // Allowed
      System.out.println("The area of shape is " + shape.getArea());  // Allowed
   }
}
```

INTERFACES IN JAVA

Interface in English is a point where to systems meet and interact.

TV <---------------> Human

Buttons

In Java, interface is a group of related methods with empty bodies. It is a blueprint of a class that can have static constants and abstract methods.

Some key points of interfaces in Java are:

- An interface is used to achieve abstraction and multiple inheritance.

- An interface can only have abstract methods and variables. It cannot have a method body.
- Since Java 8, interfaces can have default and static methods.
- Since Java 9, interfaces can have private methods.
- A class that implements an interface must implement all the methods declared in the interface.
- Interfaces are used to achieve loose coupling.
- The properties in interfaces are final i.e. you cannot modify them.

Example:

```
interface Shape {
        float perimeter();
        float area();
}

class Circle implements Shape {
        int radius;

        public float perimeter () {
                return 2 * 3.14f * radius;
        }

        public float area()  {
                return 3.14f * radius * radius;
        }
}
```

```
package chap06oop;

interface Shape {
   float getPerimeter();
   float getArea();
}

class Circle implements Shape{
   int radius;

   public float getPerimeter() {
      return 2 * 3.14f * radius;
   }

   public float getArea() {
      return 3.14f * radius * radius;
```

```
    }
}
public class Eg14IntroductionToInterface {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.radius = 7;
        System.out.println("The radius of circle is " + circle.radius);
        System.out.println("The perimeter of circle is " + circle.getPerimeter());
        System.out.println("The area of circle is " + circle.getArea());
    }
}
```

ABSTRACT CLASSES VS INTERFACES

We cannot extend multiple abstract classes, but we can implement multiple interfaces at a time.

Abstract classes and interfaces in Java are both used to achieve abstraction, where you can declare what a class can do, but not how it does it. However, they have some key differences:

1. **Default Methods:** Abstract classes can have default methods (methods with an implementation), while interfaces could only have method declaration until Java 7. Since Java 8, interfaces can have default and static methods.
2. **Final Variables:** Variables declared in an interface are implicitly final and static, whereas an abstract class can have non-final variables.
3. **Inheritance:** A class can extend only one abstract class, but it can implement multiple interfaces. This allows Java to get around the lack of multiple inheritance.
4. **Constructors:** Abstract class can have constructors, whereas interfaces cannot. This means that you cannot use 'new' keyword to instantiate an interface.
5. **Access Modifiers:** All methods in an interface are implicitly public, where an abstract class can have private, protected, and public methods. After Java 8, interfaces can also have static and default methods. After Java 9, interfaces can also have private methods.

IS MULTIPLE INHERITANCE ALLOWED IN JAVA?

Multiple inheritance faces problems when there exist methods with same signature in both the super classes. Due to such problems Java doesn't support multiple inheritance directly but a similar concept can be achieved by using interfaces. A class can implement multiple interfaces and extends a class at the same time.

NOTE:

- Interfaces in Java are a bit like the classes but with significant differences.
- An interface can only have method signatures, fields(constant) and default methods.
- The class implementing an interface needs to define all the methods (necessarily not the fields).
- You can create a reference of interfaces but not the object.
- Interface methods are public by default. So, do not forget to make them public while implementing them inside a class.

POLYMORPHISM USING INTERFACES

Implementing an interface forces method implementation. We can achieve polymorphism in interfaces same as dynamic method dispatch in classes.

Example:

```
/*
Once there used to be a Cellphone with its properties and methods. Now
Smartphone is inherited from Cellphone which implements GPS, Camera,
MediaPlayer interfaces.
*/

class Smartphone extends Cellphone implements GPS, Camera, MediaPlayer {
        // Fields and methods
}

// Similar to dynamic method dispatch in class.
GPS gps = new Smartphone();          // Can only use GPS methods.
Camera camera = new Smartphone();       // Can only use Camera methods.
MediaPlayer mp = new Smartphone();      // Can only use MediaPlayer methods.
Smartphone s = new Smartphone();        // Can use all Smartphone methods.
```

```java
package chap06oop;

class CellPhone {
  void call() {
    System.out.println("Calling... Call");
  }

  void message() {
    System.out.println("Messaging... Message");
  }
}

interface GPS {
   void navigation();
   void mapping();
}

interface Camera {
   void photo ();
   void video();
}

interface MediaPlayer {
   void playMusic();
   void playVideo();
}

class SmartPhones extends Phone implements GPS, Camera, MediaPlayer {
   @Override
   public void navigation() {
      System.out.println("Navigating... Location");
   }

   @Override
   public void mapping() {
      System.out.println("Mapping... Points");
   }

   @Override
   public void photo() {
      System.out.println("Clicking... Photo");
   }

   @Override
   public void video() {
      System.out.println("Recording... Video");
   }

   @Override
   public void playMusic() {
      System.out.println("Playing... Music");
   }

   @Override
   public void playVideo() {
      System.out.println("Playing... Video");
   }
}

public class Eg15PolymorphismUsingInterfaces {
   public static void main(String[] args) {
      GPS gps = new SmartPhones();    // Can only use GPS methods
      gps.navigation();
      gps.mapping();
//        gps.playMusic(); // Not possible --> throws an error

      // Similarly
      Camera camera = new SmartPhones();  // Can only use Camera methods.
      MediaPlayer mp = new SmartPhones(); // Can only use MediaPlayer methods.
      SmartPhones s = new SmartPhones();  // Can use all Smartphone methods.

   }
}
```

DEFAULT METHODS IN INTERFACES

This feature was introduced in Java 8 to ensure background compatibility while updating an interface.

An interface can have static and default methods. Default methods enable us to add new functionality to existing interfaces. Classes implementing the interface need not implement the default methods. Interface can also include private methods for default methods to use. These private methods cannot be accessed directly by the Classes implementing the interface.

```
package chap06oop;

interface TestInterface {
    // Abstract method
    public void square(int a);

    // Default method
    default void show() {
        System.out.println("Default Method Executed");
    }
}

class TestClass implements TestInterface {
    public void square(int a) {
        System.out.println("The square of a is " + (a * a));
    }
}

public class Eg16DefaultMethodsInInterfaces {
    public static void main(String[] args) {
        TestClass test = new TestClass();
        test.square(5);
    test.show();
  }
}
```

INHERITANCE IN INTERFACES

Interfaces can extend another interface like follows:

```
interface Interface1 {
        void meth1();
}

interface Interface2 extends interface1 {
        void meth2();
}
```

Remember that the interface cannot implement another interface, only classes can do that! Unlike classes, a Java interface can extend multiple other interfaces. You simply list them after the '*extends*' keyword, separated by commas.

```
package chap06oop;

interface SampleInterface {
  void meth1();
  void meth2();
}

interface ChildSampleInterface extends SampleInterface {
  void meth3();
  void meth4();
}

class SampleClass implements ChildSampleInterface {

  @Override
  public void meth1() {
    System.out.println("Running Method1...");
  }

  @Override
  public void meth2() {
    System.out.println("Running Method2...");
  }

  @Override
  public void meth3() {
    System.out.println("Running Method3...");
  }
  @Override
  public void meth4() {
    System.out.println("Running Method4...");
  }
}

public class Eg17InheritanceInInterfaces {
  public static void main(String[] args) {
    SampleClass obj = new SampleClass();
    obj.meth1();
    obj.meth2();
    obj.meth3();
    obj.meth4();
  }

}
```

ANONYMOUS CLASSES AND LAMBDA EXPRESSIONS

Introduced in Java 8, lambda expressions provide a concise way to implement single-method interfaces, letting you replace bulky anonymous classes with clear, inline functions. Anonymous classes still offer flexibility, allowing you to override multiple methods or add state, while lambdas shine when you need a quick, readable implementation of a functional interface.

ANONYMOUS CLASSES

A one-off, unnamed class declared and instantiated in a single expression, typically to implement an interface or extend a class for a specific task. Before Java 8, they were the primary way to provide inline implementations of single-method interfaces.

Characteristics: Can override multiple methods, capture final or effectively final variables from the enclosing scope, but tend to be verbose.

Example:

```
        Greeting greeting = new Greeting() {
                @Override
                public void greet(String name) {
                        System.out.println(“Hello, ” + name);
                }
        };
```

```
package chap06oop;
```

```java
interface Greeting {
   void greet(String name);
}

public class Eg18AnonymousClasses {
   public static void main(String[] args) {
      // Create an anonymous class that implements the Greeting interface
      Greeting greeting = new Greeting() {
         @Override
         public void greet(String name) {
            System.out.println("Hello, " + name);
         }
      };

      // Use the anonymous class
      greeting.greet("John");
   }
}
```

LAMBDA EXPRESSIONS

Introduced in Java 8 to provide a concise way to represent a function (implementation of a single abstract method) inline. It is ideal for functional interfaces (interfaces with exactly one abstract method), such as Runnable, Comparator<T>, or custom SAMs.

Characteristics:

- Much more concise than anonymous classes.
- Can capture effectively final variables.
- Support target typing and can be used in streams and functional-style APIs.
- Cannot define additional methods or state beyond the single method's implementation.

Example:

```java
Greet greet = (name) -> {
        System.out.println("Hello, " + name);
};

// Same lambda expression can be written in single line as:
Greet greet1 = (name) -> System.out.println("Hello, " + name);
```

```java
package chap06oop;

interface Greet {
   void greet(String name);
}
```

```
public class Eg19LambdaExpressions {
   public static void main(String[] args) {
      Greet greet = (name) -> {
         System.out.println("Hello, " + name);
      };

      // Use lambda expression
      greet.greet("Kushal");

      // Same lambda expression can be written in single line as:
      Greet greet1 = (name) -> System.out.println("Hello, " + name);
      greet1.greet("Kushal Prasad Joshi");
   }
}
```

EXERCISE

1. Create a class MyEmployee with required attributes and methods.

```
package chap06oop;

import java.util.Scanner;

class MyEmployee {
   int id;
   float salary;
   String name, address;

   public void printDetails() {
      System.out.println("The id of employee is " + id);
      System.out.println("The name of employee is " + name);
      System.out.println("The address of employee is " + address);
      System.out.println("The salary of employee is " + salary);
   }
}
public class Qn01CreateOwnClassEmployee {
   public static void main(String[] args) {
      MyEmployee myEmployee = new MyEmployee(); // Instantiating a new Employee
object

      // Set attributes for employee
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter id, name , address and salary of employee.");
      myEmployee.id = scan.nextInt();
      scan.nextLine(); // To consume the left over new line character
      myEmployee.name = scan.nextLine();
      myEmployee.address = scan.nextLine();
```

```
        myEmployee.salary = scan.nextFloat();

        // Print details of employee
        System.out.println("The details of employee is as follows: ");
        myEmployee.printDetails();
    }
}
```

2. **Create a command line guess the number game by creating a class Game with a constructor which generates a random number. Create an object of the class Game for a player to play game infinitely.**

```
package chap06oop;

import java.util.Random;
import java.util.Scanner;

class Game {
    int generatedNumber, inputNumber;

    // Constructor that generate random number
    public Game() {
        Random random = new Random();
        generatedNumber = random.nextInt(0, 100);
    }

    // Method for wining condition
    boolean win() {
        if (inputNumber == generatedNumber) {
            System.out.println("You guessed it right.");
            return true;
        } else if (inputNumber < generatedNumber) {
            System.out.println("You guessed lower. Try again.");
        } else {
            System.out.println("You guessed higher. Try again.");
        }
        return false;
    }
}

public class Qn02GuessTheNumberGame {
    public static void main(String[] args) {
        // Instantiation of a new Game object
        Game player = new Game();

        System.out.println("Guess the number between 0 and 100.");
        Scanner scan = new Scanner(System.in);
        player.inputNumber = scan.nextInt();
```

```
      while (! player.win()) {
        player.inputNumber = scan.nextInt();
      }
    }
}
```

3. **Create a geometric measurement program to calculate geometric measures of all the geometric shapes. Use interfaces and classes according to need to make the program more optimized.**

```
package chap06oop;

import java.util.Scanner;

interface MyShape {
    double area();
    double perimeter();
}

class MyCircle implements MyShape {
    double r;
    MyCircle(double r) { this.r = r; }
    public double area()      { return Math.PI * r * r; }
    public double perimeter() { return 2 * Math.PI * r; }
}

class MyRectangle implements MyShape {
    double w, h;
    MyRectangle(double w, double h) { this.w = w; this.h = h; }
    public double area()      { return w * h; }
    public double perimeter() { return 2 * (w + h); }
}

class MyTriangle implements MyShape {
    double a, b, c;
    MyTriangle(double a, double b, double c) { this.a = a; this.b = b; this.c = c; }
    public double perimeter() {
        return a + b + c;
    }
    public double area() {
        double s = perimeter() / 2;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }
}

public class Qn03GeometricCalculator {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("1) Circle  2) Rectangle  3) Triangle\nChoose shape: ");
        int choice = in.nextInt();

        MyShape shape;
        switch (choice) {
            case 1 -> {
                System.out.print("Radius: ");
                shape = new MyCircle(in.nextDouble());
            }
            case 2 -> {
                System.out.print("Width and height: ");
                shape = new MyRectangle(in.nextDouble(), in.nextDouble());
            }
            case 3 -> {
                System.out.print("Sides a, b, c: ");
                shape = new MyTriangle(in.nextDouble(), in.nextDouble(), in.nextDouble());
            }
            default -> {
                System.out.println("Invalid choice");
                return;
            }
        }

        System.out.printf("Area: %.2f  Perimeter: %.2f%n",
                shape.area(), shape.perimeter());
    }
}
```

CHAP 07 – ADVANCED CONCEPTS

We will discuss some very important advanced concepts required for efficient Java programming.

ERRORS IN JAVA

No matter how smart you are, errors are our constant companions. With practice, we keep getting better at finding and correcting them.

There are three types of errors in Java:

1. Syntax errors
2. Logical errors
3. Runtime errors → Also called exceptions!

SYNTAX ERRORS

When the compiler finds something wrong with our program, it throws a syntax error.

Example:

- int a = 9 ---> no semi-colon, syntax error!
- a = a + 3; ---> variable not declared, syntax error!
- d = 4; ---> variable not declared, syntax error!

```
package chap07advancedconcepts;

public class Eg01SyntaxError {
   public static void main(String[] args) {
//       int a = 5 ---> Syntax error
       int a = 5;
//       d = 4; ---> Syntax error
//       System.out.println(a + b); ---> Syntax error
       System.out.println(a);

   }
}
```

LOGICAL ERRORS

A logical error or a bug occurs when a program compiles and runs but does the wrong thing like:

- message delivered wrongly.
- wrong time of chats being displayed.
- incorrect redirects!

```java
package chap07advancedconcepts;

public class Eg02LogicalError {
   public static void main(String[] args) {
      // Write a program to print all prime numbers less than 10.
      System.out.println(2);
      for (int i = 1; i < 5; i++) {
         System.out.println(2 * i + 1);
      }
   }
}
// Output: 2 3 5 7 9 ---> Logical Error because 9 is not prime
// This is a logic of printing odd numbers.
```

RUNTIME ERRORS

Java may sometime encounter an error while the program is running. These are also called exceptions! These are encountered due to circumstances like bad input and(or) resource constraints.

Example: User supplies 's' + 8 to a program which adds 2 numbers or user supplies 0 as a divisor.

```java
package chap07advancedconcepts;

import java.util.Scanner;

public class Eg03RuntimeError {
   public static void main(String[] args) {
      int k;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter a divisor.");
      k = sc.nextInt();
      System.out.println("Integer part of 1000 divided by k is " + 1000 / k);
   }
}
// When user input 0 the program throws error ---> Runtime error
```

NOTE: Syntax errors and logical errors are encountered by the programmer whereas runtime errors are encountered by the user.

EXCEPTIONS IN JAVA

An exception is an event that occurs when a program is executed disrupting the normal flow of instructions.

There are mainly two types of exceptions in Java:

1. Checked exception → Compile the exceptions (Handled by compiler)
2. Unchecked exception → Runtime exceptions

COMMONLY OCCURRING EXCEPTIONS

The following are the commonly occurring exceptions in Java:

1. Null Pointer Exception
2. Arithmetic Exception
3. Array Index Out of Bound Exception
4. Illegal Argument Exception
5. Number Format Exception

TRY CATCH BLOCK IN JAVA

In Java, exceptions are managed by using try-catch blocks.

Syntax:

```
try {
        // Code to try
}
catch (Exception e) {
        // Code if exception
}
```

Example: Any number divided by zero is an arithmetic exception.

```java
package chap07advancedconcepts;

public class Eg04TryCatchBlock {
   public static void main(String[] args) {
      int a = 78;
      int b = 0;
      // 78/0 is an Arithmetic Exception which gives error
      try {
         int c = a / b;
         System.out.println("The result is " + c);
      }
      catch (Exception e) {
         System.out.println("We failed to divide. Reason : ");
         System.out.println(e);
      }
      System.out.println("End of the program.");
   }
}
```

HANDLING SPECIFIC EXCEPTION

In Java we can handle specific exceptions by adding multiple catch blocks.

Syntax:

```
try {
        // code
}
catch(IoException e) {          → Handle all exceptions of type IoException
        // code
}
catch(ArithmeticException e) {          → Handle all exceptions of type ArithmeticException
        // code
}
catch(Exception e) { → Handle all other Exceptions
        // code
}
```

Example:

```java
package chap07advancedconcepts;

import java.util.Scanner;

public class Eg05HandleSpecificException {
  public static void main(String[] args) {
    int[] array = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};

    // Take input from user
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the array index for value: ");
    int index = sc.nextInt();
    System.out.print("Enter the divisor: ");
    int number = sc.nextInt();

    // Handle Specific Exception that can occur
    try {
      System.out.println("The value at array index entered is : " + array[index]);
      System.out.println("The value after dividing is : " + (float) (array[index] /
number));
    }
    catch (ArithmeticException e) {
      System.out.println("ArithmeticException occurred!");
      System.out.println(e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("ArrayIndexOutOfBoundsException occurred!");
```

```
        System.out.println(e);
      }
      catch (Exception e) {
        System.out.println("Some exception occurred!");
        System.out.println(e);
      }
    }
}
```

NESTED TRY CATCH

When a try catch block is written inside another, it is called nested try catch.

Syntax:

```
        try {
                // code
                try {
                        // code
                }
                catch(Exception e) {
                        // code
                }
        }
        catch(Exception e) {
                        // code
        }
```

Example:

```
package chap07advancedconcepts;

import java.util.Scanner;

public class Eg06NestedTryCatch {
  public static void main(String[] args) {
    int[] array = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};

    Scanner sc = new Scanner(System.in);

    try {
      System.out.print("Enter the value of array index: ");
      int index = sc.nextInt();
      System.out.println("The value at index is " + array[index]);

      try {
        System.out.print("Enter a divisor: ");
```

```
            int divisor = sc.nextInt();
            System.out.println("The number after dividing is " + (float) (array[index] /
divisor));
         } catch (ArithmeticException e) {
            System.out.print("Cannot divide the number. Reason: ");
            System.out.println(e);
         }
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.print("Cannot find the array index. Reason: ");
         System.out.println(e);
      }
   }
}
```

EXCEPTION CLASS IN JAVA

In Java, the Exception Class is a form of Throwable that indicates conditions that a reasonable application might want to catch.

We can write our custom Exceptions using Exception class in Java as follows:

```
public class MyException extends Exception {
        // overridden methods
}
```

The exception class has the following important methods:

1. String toString() → executed when` System.out.println(e)` is run
2. void printStackTrace() → prints stack trace
3. String getMessage() → prints the Exception message

Example:

```
package chap07advancedconcepts;

class MyExceptions extends Exception {
   @Override
   public String toString() {
      return "This is a exception from toString() method.";
   }

   @Override
   public String getMessage() {
      return "This is an exception from getMessage() method.";
   }
}

public class Eg07ExceptionClassInJava {
   public static void main(String[] args) {
      try {
         throw new MyExceptions();
      } catch (Exception e) {
         System.out.println(e);  // This is an exception from toString() method
         System.out.println(e.toString()); // This is an exception from toString() method
         System.out.println(e.getMessage()); // This is an exception from getMessage()
method
      }
   }
}
```

THE THROW KEYWORD

The throw keyword is used to throw an exception explicitly by the programmer.

Example:

```
if (b == 0) {
        throw new ArithmeticException("Div by 0");
}
else {
        return a / b;
}
```

In a similar manner, we can throw user defined exceptions:

```
throw new MyException("Exception thrown");
```

NOTE: We need to handle exceptions with try-catch block.

```
package chap07advancedconcepts;

public class Eg08ThrowKeyword {
   public static void main(String[] args) {
      try {
         throw new Exception("This is an exception");
      } catch (Exception e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }
}
```

THE THROWS KEYWORD

In Java throws keyword is used to declare an exception. This gives information to the programmer that there might be an exception so, it's better to be propended with a try catch block!

Example:

```
public void calculate(int a, int b) throws ArithmeticException {
        // code
    }
```

```
package chap07advancedconcepts;

public class Eg09ThrowsKeyword {

    public static void main(String[] args) {
        try {
            methodThatThrowsException();
        } catch (Exception e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }

    static void methodThatThrowsException() throws Exception {
        throw new Exception("This is an exception");
    }
}
```

THE FINALLY BLOCK

Finally block contains the code which is always executed whether the exception is handled or not. It is used to execute code containing instructions to release the system resources, close a connection, etc. The finally block in Java executes even if a return statement is encountered in the try or catch block.

Example:

```
package chap07advancedconcepts;

public class Eg10FinallyBlock {
    public static void main(String[] args) {
        try {
            int divideByZero = 5 / 0;
            System.out.println("Rest of try block");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException => " + e.getMessage());
        } finally {
            System.out.println("This is the finally block");
        }
    }
}
```

FILE HANDLING

Reading from and writing to files is an important aspect of any programming language. We can use File class in Java to create a File object:

- createNewFile() method → Creates a file
- For reading files we can use the same Scanner class and supply it a file object.
- To delete a file in Java we can use File object's delete() method.

```
package chap07advancedconcepts;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Eg11FileHandling {
   public static void main(String[] args) {

      // Code to create a new file
      File myFile = new File(".\\kushal.txt");
      try {
         myFile.createNewFile();
      }
      catch (IOException e) {
         System.out.println("Unable to create your file.");
         throw new RuntimeException(e);
      }

      // Code to write to a file.
      try {
         FileWriter myFileWriter = new FileWriter(".\\kushal.txt");
         myFileWriter.write("This is our first file from this java course.\n Ok now bye!");
         myFileWriter.close();
      }
      catch (IOException e) {
         System.out.println("Unable to write to your file.");
         throw new RuntimeException(e);
      }

      // Reading a file
      try {
         Scanner scanner = new Scanner(myFile);
         while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            System.out.println(line);
         }
         scanner.close();
      }
      catch (FileNotFoundException e) {
         System.out.println("Unable to read from the file.");
```

```
        throw new RuntimeException(e);
      }

      // Deleting a file
      if (myFile.delete()) {
        System.out.println("I have deleted " + myFile.getName());
      }
      else {
        System.out.println("Unable to delete the file.");
      }
    }
}
```

ANNOTATIONS

Annotations are used to provide extra information about a program. Annotations provide metadata to class/methods.

The following are some annotations built into Java:

1. @Override →Used to mark overridden elements in the child class.
2. @SuppressWarnings → Used to suppress the generated warning by the compiler.
3. @Deprecated →Used to make deprecated methods.
4. @FunctionalInterface → Used to ensure an interface is a functional interface i.e. interface having a single method.

```
package chap07advancedconcepts;

@FunctionalInterface
interface MyFunctionalInterface {
    void method();
//    void secondMethod();
}

class Parent {
    public void greet() {
        System.out.println("Good Morning!");
    }
}

class Child extends Parent {
    @Override
    public void greet() {
        System.out.println("Good Afternoon!");
    }

    @Deprecated
    public int sum(int a, int b) {
        return a + b;
    }
}

public class Eg12AnnotationsInJava {
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        Child child = new Child();
        child.greet();
        System.out.println(child.sum(6, 7));
    }
}
```

PACKAGES

In Java, a package is a mechanism to encapsulate a group of classes, sub packages, and interfaces. Package helps in avoiding conflicts.

There are two types of packages:

- Built in packages → Java API
- User defined packages → Custom Packages

Some key points about packages in Java are:

1. **Purpose:** Packages are used for preventing naming conflicts, making searching/locating and uses of classes, interfaces, enumerations, and annotations easier, and providing controlled access.
2. **Structure:** Package names and directory structures are closely related. For example, if package name is com.company.project, then there are three directories, com, company, and project such that project is present in company and company is present in com.
3. **Naming Conventions:** Packages are named in reverse order of domain names. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.tech.history, etc.
4. **Subpackages:** Packages that are inside another package are the subpackages. They are not imported by default; they have to be imported explicitly.
5. **Accessing Classes:** To access classes inside a package, you can use 'import package_name.*; to import all the classes from the package, import package_name.class_name; to import a specific class, or use the fully qualified name.

USING A JAVA PACKAGE

import java.lang.*;   → Import everything from java.lang
import java.lang.String;   → Import String from java.lang

s = new java.lang.String("Hi"); → use without importing

```
package chap07advancedconcepts;

import java.lang.String;   // Import String from java.lang

public class Eg13UsingJavaPackage {
   public static void main(String[] args) {
      java.util.Scanner scan = new java.util.Scanner(System.in); // Use without importing

      System.out.println("This is about how to use Java packages");
   }
}
```

CREATING OWN PACKAGES IN JAVA

Creating our own package in Java helps us in organizing and managing our code. The procedure to create a Java package is as follows:

1. **Choose a Package Name:** Choose a name for your package. This should be unique to avoid conflicts with other packages.
2. **Declare the Package:** Write the package name at the top of every source file (classes, interfaces, enumerations, and annotations). There must be only one package statement in each source file.
3. **Create Classes in the Package:** You can now create classes, interfaces, etc. as part of package.
4. **Compile the Java File:** Compile the java file using 'javac' command in your terminal. This will create a .class file in the same directory.
5. **Access the Package:** To access the package in your java program, you need to import it at the top of your program using the 'import' keyword.

NOTE:

- **javac File.java** → Creates File.class
- **javac -d . File.java** → Creates a package folder (We can keep adding classes to a package like this). Here, '.' means the folder should be created with same as package name.

We can also create inner packages by adding "package.subpackage" as package name where 'package' is folder and 'subpackage' is subfolder. These packages once created can be used by other classes.

CREATING DOCUMENTATION

Java documentation is great! It helps us to get info about which class/method/entity to use when. We can create our own package's documentation in Java.

JAVADOC TOOL

javadoc command allows us to create documentation in HTML format for our own package. Java provides tags for class or package to assist with the java doc generation. Note that the Javadoc comment is always written just above the class or method.

TAG FOR CLASS OR A PACKAGE

1.	@author	→ adds the author's name
2.	@version	→ adds the version
3.	@since	→ to add when the version was written
4.	@see	→ adds a see also heading with a link

Description can be added at start of javadoc comment.

TAGS FOR METHODS

1.	@parum	→ for declaring parameters of a method
2.	@return	→ for declaring about the return value
3.	@throws	→ for declaring exception thrown
4.	@depricated	→ for declaring depreciated methods

Description can be added at start of javadoc comment.

EXAMPLE

```
/**
 * I hope this package may be helpful to you.
 * @author Kushal Prasad Joshi
 * @version 0.1.1.1
 * @since 2024
 */

package _15_CreatingDocumetation;

/**
 * This class is to demonstrate what javadoc is and how it is used in java industry
 * You can use html inside this like <i>italic</i><b>bold</b><p>New paragraph</p>
 * @author Kushal Prasad Joshi
 * @version 0.1.1.1
 * @since 2024
 * @see <a href="https://docs.oracle.com/en/java/javase/14/docs/ap1/index.html"
target=_blank>Java Docs</a>
 */
public class Eg14CreateOwnDocumentation {
    /**
     * This method can add two numbers.
     * @param a This is argument1 supplied.
     * @param b This is argument2 supplied.
     * @deprecated This is deprecated. Use <b>+</b> operator instead.
     */
    public void add(int a, int b) {
        System.out.println("The sum is " + (a + b));
    }

    /**
     * This is a method to divide two numbers.
     * @param a This is first argument supplied to function.
     * @param b This is second argument supplied to function.
     * @return The quotient is returned.
     * @throws ArithmeticException if b = 0.
     */
    public int divide(int a, int b) {
        try {
            return a / b;
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
        return 0;
    }

    /**
     * This is my main method in class
     */
    public static void main(String[] args) {
        System.out.println("This is my main method");
    }
}
```

COLLECTIONS FRAMEWORK

The collection framework in Java is a unified architecture for representing and manipulating collections, which are groups of objects. It provides a standard set of interfaces and classes that allow collections to be manipulated independently of their implementation details. A collection represents a group of objects. Java collections provide Classes and Interfaces for us to be able to write code quickly and efficiently.

WHY DO WE NEED COLLECTIONS?

We need Collections for efficient storage and better manipulation of data in Java.

For Example: We use array to store integers but what if we want to:

1. Resize the array?
2. Insert an element in between?
3. Delete an element in array?
4. Apply certain operations to change this array?

HOW ARE COLLECTIONS AVAILABLE?

Collections in Java are available as Classes and Interfaces.

The following are few commonly used Collections in Java:

1. ArrayList → for variable size collection
2. Set → for distinct collection
3. Stack → a LIFO (Last In First Out) data structure
4. HashMap → for storing key-values pairs

Collection class is available in java.util package. Collection class also provides static methods for sorting, searching, etc.

```
package chap07advancedconcepts;

import java.util.ArrayList; // Import a variable size collection
import java.util.HashSet;
import java.util.Set;   // Import distinct collection
import java.util.TreeSet;   // Import TreeSet

public class Eg15AvailabilityOfCollections {
  public static void main(String[] args) {
    ArrayList<String> arrayList = new ArrayList<>();
    Set<Integer> set = new HashSet<>();
    TreeSet<Float> treeSet = new TreeSet<>();

    System.out.println("Collections are available as Classes and Interfaces.");
  }
}
```

COLLECTION HIERARCHY

[detailed picture at → search: java collection framework + Wikipedia]

ARRAYLIST CLASS

The ArrayList class in Java is a part of the Java Collection Framework and it's a class of java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

Some key features of ArrayList class are:

- **Resizable-array implementation of the List interface:** Implements all optional list operations, and permits all elements, including null.
- **Dynamic size:** The main advantage of ArrayList in Java is that if we declare an array then we need to mention a size, but in ArrayList, it is not needed to mention the size of ArrayList.
- **Manipulate the size of array:** This class provides methods to manipulate the size of an array that is used internally to store the list.
- **Capacity:** Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size.
- **Not synchronized:** Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

NOTE: ArrayList is typically a better choice when you need efficient access to elements via their indices.

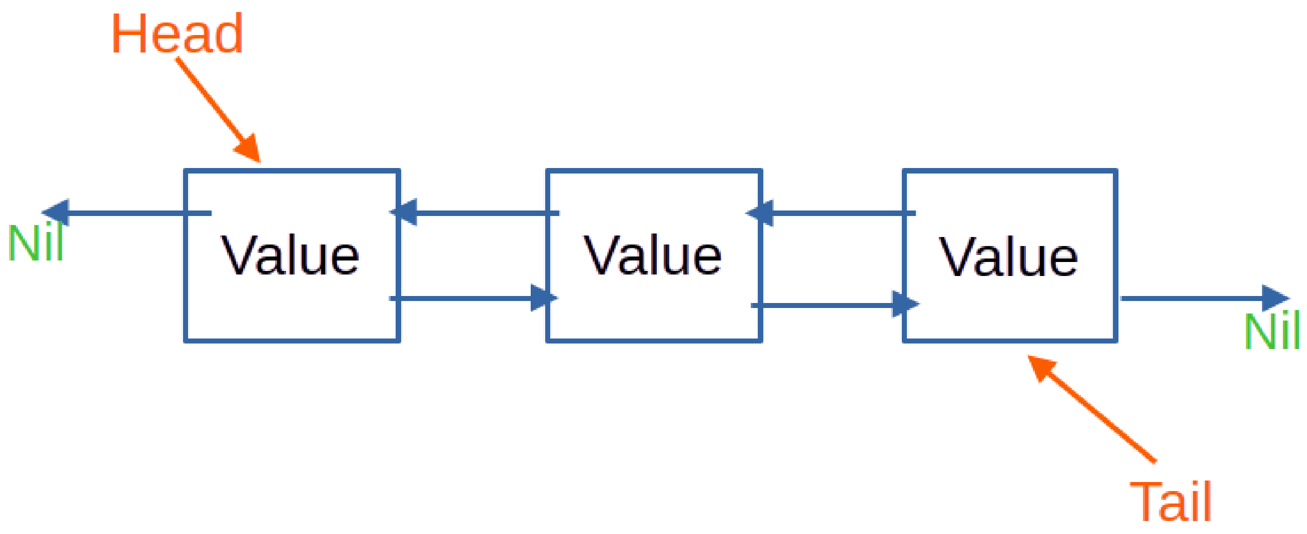
```
package chap07advancedconcepts;

import java.util.*;

public class Eg16ArrayList {
   public static void main(String[] args) {
         // Create an ArrayList
         ArrayList<String> fruits = new ArrayList<>();

         // Add elements
         fruits.add("Apple");
         fruits.add("Banana");
         fruits.add("Cherry");

         // Print the ArrayList
         System.out.println("ArrayList: " + fruits);
   }
}
```

LINKEDLIST CLASS

The linked list in Java is a part of the Java Collection Framework and provides a linked list implementation of the List and Deque Interfaces. It allows for the storage and retrieval of elements in a doubly linked list data structure, where each element is linked to its predecessor and successor elements.

Some key features of LinkedList are:

- **Doubly linked list implementation:** Implements all optional list operations, and permits all elements (including null).
- **Dynamic size:** The size of the list automatically increases when we dynamically add or remove items.
- **Not stored in contiguous locations:** The elements are not stored in a contiguous fashion. Therefore, there is no need to increase the size.
- **Ease of insertion and deletion:** Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.
- **Not synchronized:** If multiple thread accesses a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.



NOTE: Linked list uses independent objects.

```
package chap07advancedconcepts;

import java.util.*;

public class Eg17LinkedList {
   public static void main(String[] args) {
      // Create a LinkedList
      LinkedList<String> fruits = new LinkedList<>();

      // Add elements
      fruits.add("Apple");
      fruits.add("Banana");
      fruits.add("Cherry");

      // Print the LinkedList
      System.out.println("LinkedList: " + fruits);
```

```
    }
}
```

ARRAYDEQUE

The ArrayDeque class in Java is a part of the Java Collections Framework and provides a linked list implementation of the Deque and Queue interfaces. It allows for the storage and retrieval of the elements in a doubly linked list data structure, where each element is linked to its predecessor and successor elements.

Here are some key features of the ArrayDeque class:

- **Resizable-array implementation of the Deque interface:** Implements all optional list operations and permits all elements (including null).
- **Dynamic size:** The size of the list automatically when we dynamically add or remove items.
- **Not stored in contiguous locations:** The elements are not stored in contiguous fashion. Therefore, there is no need to increase the size.
- **Ease of insertion and deletion:** Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.

```
package chap07advancedconcepts;

import java.util.*;

public class Eg18ArrayDeque {
    public static void main(String[] args) {
        // Create a ArrayDeque
        ArrayDeque<String> fruits = new ArrayDeque<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // Print the LinkedList
        System.out.println("ArrayDeque: " + fruits);
    }
}
```

HASHSET CLASS

The HashSet class in Java is a part of the Java Collection Framework and it's a class of the java.util package. It provides us with a collection that uses a hash table for storage. Hashing is a technique to convert a range of key-value pairs into a range of indices.

Classes that use hashing techniques are: HashSet, HashMap, LinkedHashMap, HashTable.

Some key features of HashSet class are:

- **Initial Default Capacity:** The initial store capacity of HashSet is 16 and the load factor is 0.75.
- **Hashing Mechanism:** HashSet stores the elements by using a mechanism called hashing.
- **Unique Elements:** HashSet contains unique elements only.
- **Null Value:** HashSet allows null values.
- **Non-Synchronized:** HashSet class is non-synchronized.
- **No insertion order:** HashSet class does not maintain the insertion order. Here elements are inserted based on their hash code.
- **Search operations:** HashSet is the best approach for search operations.

```
package chap07advancedconcepts;

import java.util.*;

public class Eg19HashSet {
   public static void main(String args[]) {
      //Creating HashSet and adding elements
      HashSet<String> set = new HashSet<>();
      set.add("One");
      set.add("Two");
      set.add("Three");
      set.add("Four");
      set.add("Five");

      // Print HashSet
      System.out.println("HashSet: " + set);
   }
}
```

JAVA GENERICS

Java generics are introduced from JDK 5.0 and onwards. They are very similar to C++ templates (but not the same).

If we write:

```
ArrayList a = new ArrayList();          // Without using generics
a.add(75);
// int anum = a.get(0);                 → We can't do this
int anum = (int) a.get(0);      // We have to do this
```

Hence generics aim to reduce bugs and enhance type safety.

```
ArrayList<Integer> a = new ArrayList();       // Using generics
a.add(75);
int anum = a.get(0);   // We can do this
// This is the best practice of programming.
```

NOTE: <type_parameter> in java generics cannot be a primitive datatype.

```
package chap07advancedconcepts;

import java.util.ArrayList;

public class Eg20JavaGenerics {
   public static void main(String[] args) {

//       ArrayList<int> arrayList = new ArrayList();   // Type Parameter in java generics
cannot be a primitive datatype.

      ArrayList<Integer> arrayList = new ArrayList();

      //       arrayList.add("String 1");
      arrayList.add(54);
      arrayList.add(643);
//       arrayList.add(new Scanner(System.in));

//       int a = (int) arrayList.get(2);

      int a = arrayList.get(0);
      System.out.println(a);
   }
}
```

CREATING OUR OWN GENERICS

```
package chap07advancedconcepts;

class MyGeneric<T1, T2> {
   int val;
   private T1 t1;
   private T2 t2;

   public T2 getT2() {
      return t2;
   }

   public void setT2(T2 t2) {
      this.t2 = t2;
   }
```

```java
    public MyGeneric(int val, T1 t1, T2 t2) {
        this.val = val;
        this.t1 = t1;
        this.t2 = t2;
    }

    public int getVal() {
        return val;
    }

    public void setVal(int val) {
        this.val = val;
    }

    public T1 getT1() {
        return t1;
    }

    public void setT1(T1 t1) {
        this.t1 = t1;
    }
}

public class Eg21CreateOwnGenerics {
    public static void main(String[] args) {
        MyGeneric<String, Integer> g1 = new MyGeneric<>(23, "My String", 45);
        String str = g1.getT1();
        System.out.println(str);

        Integer int1 = g1.getT2();
        System.out.println(int1);
    }
}
```

DATE AND TIME

java.time → package for Date and Time in Java (from Java 8 onwards)

Before Java 8, java.util package used to hold the Date and Time classes. Now these classes are depreciated.

HOW JAVA STORES A DATE?

Date in Java is stored in the form of a long number. This long number holds the number of milliseconds passed since 1 Jan 1970. Java assumes that 1900 is the start year which means it calculates years passed since 1900 whenever we ask it for years passed.

HOW TO USE DATE IN JAVA?

System.currentTimeMillis() returns no of milliseconds passed. Once no of milliseconds are calculated, we can calculate minutes, seconds and years passed.

```
package chap07advancedconcepts;

public class Eg22DateAndTime {
   public static void main(String[] args) {
      System.out.println(System.currentTimeMillis());
   }
}
```

THE DATE CLASS IN JAVA

Date date = new Date();

System.out.println(date);

We can also use constructors provided by the Date class.

Java Date class has few methods which can be used. For example: getDate(), getDay(), etc.

NOTE: All these methods are depreciated.

```
package chap07advancedconcepts;

import java.util.Date;  // This is depreciated

public class Eg23DateClass {
   public static void main(String[] args) {
      Date d = new Date();

      System.out.println(d);
      System.out.println(d.getTime());
      System.out.println(d.getDate());
      System.out.println(d.getSeconds());
      System.out.println(d.getYear());
   }
}
```

CALENDAR CLASS IN JAVA

Calendar class in Java is an abstract class that provides calendar related methods.

Calendar.getInstance → returns a Calendar instance based on current time.

Syntax:

Calender calendar = Calendar.getInstance();

```
calendar.getTime();   // returns time
```

CALENDAR CLASS METHODS

get method is used to get year, date, min, second, etc.

```
calendar.get(Calender.SECOND)
calendar.get(Calender.MINUTE)
calendar.get(Calender.DATE)
calendar.get(Calender.YEAR)
```

getTime() method returns a Date object.

Other methods can be looked up from the Java docs!

```
package chap07advancedconcepts;

import java.util.Calendar;
import java.util.TimeZone;

public class Eg24CalendarClass {
  public static void main(String[] args) {
    Calendar c = Calendar.getInstance();

    System.out.println(c.getCalendarType());
    System.out.println(c.getTimeZone());
    System.out.println(c.getTime());
    System.out.println(c.get(Calendar.DATE));
    System.out.println(c.get(Calendar.SECOND));
    System.out.println(c.get(Calendar.HOUR));

    System.out.println();
    System.out.println(c.get(Calendar.HOUR_OF_DAY) + ":" + c.get(Calendar.MINUTE) +
":" + c.get(Calendar.SECOND));

    System.out.println();
    Calendar cal = Calendar.getInstance(TimeZone.getTimeZone("Asia/Singapore"));
    System.out.println(cal.getCalendarType());
    System.out.println(cal.getTimeZone().getID());
  }
}
```

GREGORIANCALENDAR CLASS

This class is used to create an instance of Gregorian calendar. We can change the year, month and date using set......() methods.

```
package chap07advancedconcepts;

import java.util.GregorianCalendar;

public class Eg25GregorianCalendar {
   public static void main(String[] args) {
      GregorianCalendar cal = new GregorianCalendar();
      System.out.println(cal.isLeapYear(2024));
   }
}
```

TIMEZONE CLASS

TimeZone Class is used to create time zones in Java.

Some of the important methods of TimeZone class are:

1. getAvailableIDs() → get all the available IDs supported
2. getDefault() → get the default time zone
3. getID() → get the ID of a time zone

```
package chap07advancedconcepts;

import java.util.TimeZone;

public class Eg26TimeZoneClass {
   public static void main(String[] args) {
      System.out.println(TimeZone.getAvailableIDs()[8]);
      System.out.println(TimeZone.getAvailableIDs()[1]);
      System.out.println(TimeZone.getAvailableIDs()[2]);
   }
}
```

JAVA.TIME PACKAGE

- Available from Java 8 onwards.
- Capable of storing even nanoseconds.

The following are the most commonly used classes from java.time package:

1. LocalDate → Represents a Date
2. LocalTime → Represents a Time
3. LocalDateTime → Represents a Date + Time.
4. DateTimeFormatter → Formatter for displaying and passing date-time objects.

```
package chap07advancedconcepts;

import java.time.*;
```

```
public class Eg27JavaTimePackage {
   public static void main(String[] args) {
      LocalDate date = LocalDate.now();
      System.out.println(date);

      LocalTime t = LocalTime.now();
      System.out.println(t);

      LocalDateTime dt = LocalDateTime.now();
      System.out.println(dt);
   }
}
```

DATETIMEFORMATTER CLASS

DateTimeFormatter class in Java is used for formatting the date and time.

```
package chap07advancedconcepts;

import java.time.format.DateTimeFormatter;
import java.time.LocalDateTime;

public class Eg28DateTimeFormatterClass {
   public static void main(String[] args) {
      LocalDateTime dt = LocalDateTime.now(); // Actual date
      System.out.println(dt);

      DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd|MM|yyyy|E|h|m|a");
// Format
      String myDate = dtf.format(dt); // Creating date using Date and Format
      System.out.println(myDate);

      DateTimeFormatter dtf2 = DateTimeFormatter.ISO_LOCAL_DATE;  // Format
      String myDate2 = dtf2.format(dt); // Creating date using Date and Format
   }

}
```

MULTITHREADING

Multithreading in Java is a feature that allows a program to operate more effectively by doing multiple things at the same time. A thread is a lightweighted sub-process and shares the same memory space and resources. Threads can improve the performance and responsiveness of a program by allowing parallel execution of multiple tasks.

Multiprocessing and multi-threading are both used to achieve multi-tasking.

```
OS -------------> Process1
        |--------> Process2
        |--------> Process3
        |--------> Process4

Process -------->Thread1
        |---------> Thread2
        |--------> Thread3
        |--------> Thread4
```

In a nutshell…

1. Threads use shared memory area.
2. Threads → Faster content switching.
3. A thread is lightweight whereas a process is heavyweight.

For Example: A word processor can have one thread running in foreground as an editor and another on background auto saving document.

FLOW OF CONTROL IN JAVA

We can have one or more threads on Java as Java is a multithreading programming language. In a multithreaded Java program, "flow control" shifts from simple sequential constructs to mechanisms that coordinate when and how threads run, wait, signal each other, or terminate.

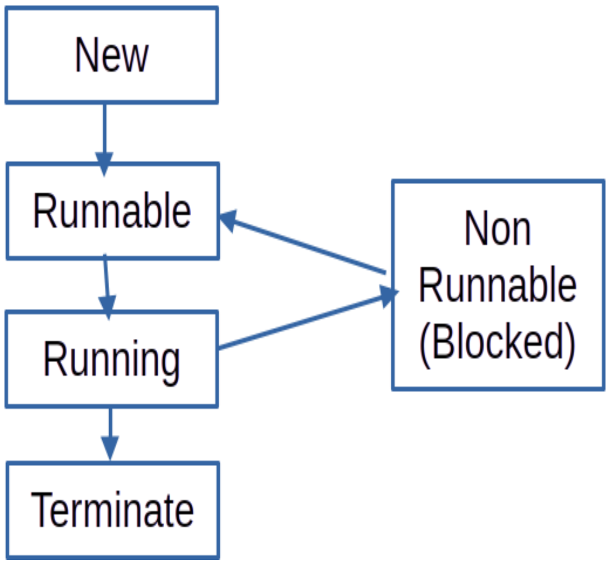
WITHOUT THREADING

```
main() -----> func1()  -----> func2 -----> END
```

WITH THREADING

```
main() ------------------------|

        func1() --------------|---------> END

                func2() -----|
```

LIFE CYCLE OF A THREAD



```
New Thread
       |
       V
Runnable State <----> Blocked State
       |
       V
Waiting State <----> Timed Waiting State
       |
       V
Terminated State
```

The life cycle of a thread in Java involves several states:

1. New
2. Runnable
3. Blocked
   - Waiting
   - Timed Waiting
4. Terminated

NEW

When a new thread is created, it is in a new state. The thread has not yet started to run when the thread is in this state. This is a stage where instance of thread is created which is not yet started by invoking start().

RUNNABLE

A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running, or it may be ready to run at any instance of time. This is the stage after invocation of start() and before it is selected to run by the scheduler.

RUNNING

This is a time when the thread is executing the task. This is the stage after the thread schedular has selected it.

NON-RUNNABLE

The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by other thread. The thread will move from the blocked state to runnable state when it acquires the lock. This is a stage when the thread is alive, but not eligible to work.

WAITING

The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread notifies or that thread will be terminated.

TIMED WAITING

A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the time-out is completed or until a notification is received.

TERMINATED

In this method the run() method has been exited. A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous events, like a segmentation fault or an unhandled exception.

CREATING A THREAD

There are two ways to create a thread in Java:

1. By extending Thread class.
2. By implementing Runnable interface.

The major difference between extending the Thread class and implementing the Runnable interface is that when a class extends the Thread class, you cannot extend any other class, but by implementing the runnable interface, it is possible to extend from other class as well.

However, because thread runs at the same time as other parts of the program, there is no way to know in which order the code will run. When the main program and threads are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems. To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible is to use the isAlive() method of thread to check whether the thread has finished running before using attributes that a thread can change.

BY EXTENDING THREAD CLASS

Thread class provides constructors and methods to create and perform operations on a thread. It extends the Object class and implements the Runnable interface.

Example:

```
class Multi extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

```java
package chap07advancedconcepts;

class Thread1 extends Thread {
   @Override
   public void run() {
      int i = 0;
      while (i < 1000) {
         System.out.println("Thread1 is running...");
         i++;
      }
   }
}

class Thread2 extends Thread {
   @Override
   public void run() {
      int i = 0;
      while (i < 1000) {
         System.out.println("Thread2 is running...");
         i++;
      }
   }
}

public class Eg29ExtendThreadClass {
   public static void main(String[] args) {
      Thread1 thread1 = new Thread1();
      Thread2 thread2 = new Thread2();

      thread1.start();
      thread2.start();
   }
}
```

BY IMPLEMENTING RUNNABLE INTERFACE

The runnable interface should be implemented by any class whose instances are intended to be executed by a Thread. The runnable interface has only one method named run(). You can create a thread by implementing the Runnable interface and overriding the run() method. Then you can create a Thread class object passing the object of class implementing runnable and call the start() method on object of Thread class.

NOTE: Remember that if you implement Runnable interface to a class for multithreading, you cannot call start() method directly on that class object. You need to create an object of Thread class passing that object to constructor and then start() method can be called on object of thread class.

```
package chap07advancedconcepts;

class ThreadImplementingRunnable1 implements Runnable {
   @Override
   public void run() {
      int i = 0;
      while (i < 1000) {
         System.out.println("Bullet firing from gun1");
         i++;
      }
   }
}

class ThreadImplementingRunnable2 implements Runnable {
   @Override
   public void run() {
      int i = 0;
      while (i < 1000) {
         System.out.println("Bullet firing from gun2");
         i++;
      }
   }
}

public class Eg30ImplementRunnableInterface {
   public static void main(String[] args) {
      ThreadImplementingRunnable1 bullet1 = new ThreadImplementingRunnable1();
      Thread gun1 = new Thread(bullet1); // Start method cannot be called directly if
Runnable interface is implemented
      gun1.start(); // Create an object of Thread class passing object of class
implementing Runnable then call start() method on object

      ThreadImplementingRunnable2 bullet2 = new ThreadImplementingRunnable2();
      Thread gun2 = new Thread(bullet2);
      gun2.start();
   }
}
```

THE START() METHOD

The start() method of the Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts (with a new call stack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

CONSTRUCTORS OF THREAD CLASS

Some commonly used constructors of Thread Class are:

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r, String name)

```java
package chap07advancedconcepts;

class NewThread extends Thread {
   public NewThread(String name) {
      super(name);
   }

   @Override
   public void run() {
      int i = 0;

      while (i < 10) {
         System.out.println("I am a new thread.");
         i++;
      }
   }
}

public class Eg31ConstructorsOfThreadClass {
   public static void main(String[] args) {
      NewThread thread1 = new NewThread("Kushal");
      NewThread thread2 = new NewThread("Kushal Prasad Joshi");

      thread1.start();
      thread2.start();

      System.out.println("The id of the thread thread1 is " + thread1.getId());
      System.out.println("The name of the thread thread1 is " + thread1.getName());

      System.out.println("The id of the thread thread2 is " + thread2.getId());
      System.out.println("The name of the thread thread2 is " + thread2.getName());
```

```
  }
}
```

THREAD CLASS METHODS

In Java, Thread class provides several methods to create and perform operations on a thread. Here are some most important methods:

1. **start():** This method starts the execution of the new thread and calls the run() method. The start() method returns immediately, and the new thread normally continues until the run() method returns.
2. **run():** If this thread was constructed using a separate Runnable run object, then the run() method is invoked on that Runnable object.
3. **getName():** This method returns the name of the thread.
4. **getPriority():** This method returns the priority of the thread.
5. **isAlive():** This tests if the method is alive.
6. **join():** This method waits for the thread to die.
7. **sleep(long milliseconds):** This method causes the currently executing thread to sleep for the specified number of milliseconds.
8. **getState():** This method returns the state of the thread.

Remember, the methods wait(), notify(), and notifyAll() are not actually part of the Thread class, they are part of the Object class in Java. They are used in threads to support interthread communication.

```
package chap07advancedconcepts;

class ThreadExample extends Thread {
  public void run() {
    System.out.println("Thread is running...");
  }
}

public class Eg32ThreadClassMethods {
  public static void main(String args[]) {
    ThreadExample thread1 = new ThreadExample();
    ThreadExample thread2 = new ThreadExample();

    System.out.println("Id of thread1: " + thread1.getId());
    System.out.println("Name of thread1: " + thread1.getName());

    System.out.println("Id of thread2: " + thread2.getId());
    System.out.println("Name of thread2: " + thread2.getName());

    thread1.setName("Thread A");
```

```
        thread2.setName("Thread B");

        System.out.println("After changing name of thread1: " + thread1.getName());
        System.out.println("After changing name of thread2: " + thread2.getName());
    }
}
```

THREAD PRIORITIES

In Java, each thread has a priority that is represented by an integer between 1 and 10. The thread scheduler uses these priorities to determine which thread should be allowed to execute.

Some key points about thread priorities in Java are:

1. **Default Priority:** The default priority of a thread is 5 (NORM_PRIORITY).
2. **Range:** The range of priority of thread is between 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY).
3. **Setting and Getting Priority:** You can set priority of a thread by using the setPriority(int newPriority) and get the current priority by using the getPriority() methods.
4. **Exceptions:** The setPriority(int newPriority) method throws an IllegalArgumentException if the value of newPriority is out of valid range.

```
package chap07advancedconcepts;

class ThreadPriorityExample extends Thread {
    public void run() {
        System.out.println("Inside the run() method");
    }
}

public class Eg33ThreadPriorities {
    public static void main(String argvs[]) {
        ThreadPriorityExample thread1 = new ThreadPriorityExample();
        ThreadPriorityExample thread2 = new ThreadPriorityExample();
        ThreadPriorityExample thread3 = new ThreadPriorityExample();

        System.out.println("Before setting priority of thread manually.");
        System.out.println("Priority of the thread thread1 is: " + thread1.getPriority());
        System.out.println("Priority of the thread thread2 is: " + thread2.getPriority());
        System.out.println("Priority of the thread thread3 is: " + thread3.getPriority());

        thread1.setPriority(6);
        thread2.setPriority(3);
        thread3.setPriority(9);
        System.out.println("After setting priority of thread manually.");
        System.out.println("Priority of the thread thread1 is: " + thread1.getPriority());
        System.out.println("Priority of the thread thread2 is: " + thread2.getPriority());
        System.out.println("Priority of the thread thread3 is: " + thread3.getPriority());
    }
}
```

EXERCISE

1. **Create your own package with individual classes for all the geometric shapes like Rectangle, Square, Circle, Cylinder, Sphere, etc. You can use interfaces as well as inheritance to make your code organized and properly managed code. Include methods for the geometric measures of shapes as well as getters and setter for dimensions.**

[Try it yourself]