JAVA PROGRAMMING

THE BOOK FOR BEGINNERS!

WITH HANDS-ON EXCERISES

ARCHIES GURAV

JAVA

PROGRAMMING

THE BOOK FOR BEGINNERS!

ARCHIES GURAV

"It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!"

~ Robert C Martin

CONTENTS

CONTENTS

INTRODUCTION

"Java Programming: The Book for Beginners" is the perfect guide for those who are just starting out on their programming journey.

This comprehensive e-book covers the fundamentals of Java programming, helping you to gain a solid understanding of the concepts and techniques you need to know to be successful.

With clear explanations, practical examples, and hands-on exercises, this e-book is the ideal resource for anyone looking to learn Java programming.

Whether you're a complete newcomer to programming or looking to build on your existing KNOWLEDGE, "Java Programming: The Book for Beginners" is the perfect place to start."

HISTORY OF JAVA

The history of java starts from Green Team, java team members (also known as **Green Team**), had a goal of creating a language for digital devices such as television and set-up boxes, etc.

James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the language project in June 1991. The small team of sun engineers called **Green Team**.

Firstly, it was called "**Green Talk**" by James Gosling and file extension was `.gt`.

After that, it was called "**Oak**" and was developed as a part of the Green project.

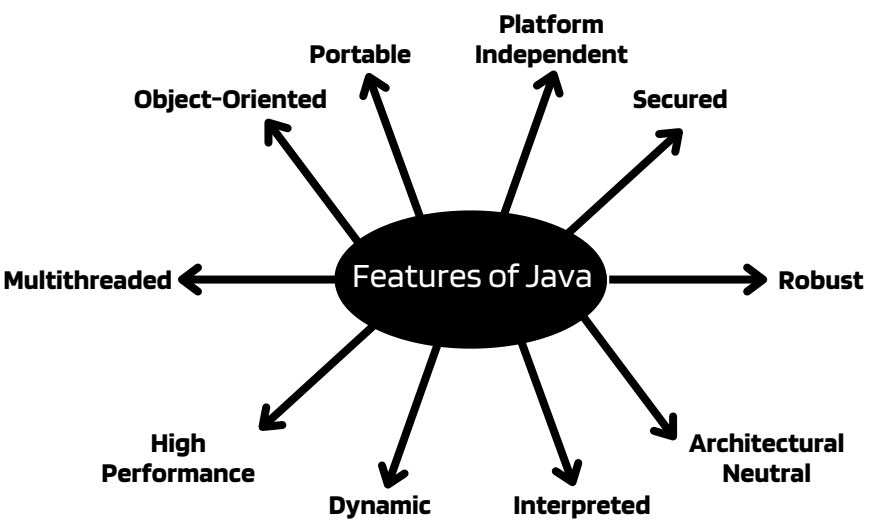
It was officially released in 1995, and quickly gained popularity due to its "**write once, run anywhere**" capability and its robust library of pre-written code.

Currently, Java is used in Internet Programming, mobile devices, games, e-business solutions, etc.

FEATURES OF JAVA

There are many features of Java. They are also known as **java buzzwords**. The Features of Java are simple and easy to understand.

1. Object-Oriented
2. Portable
3. Platform Independent
4. Secured
5. Robust
6. Architecture Neutral
7. Interpreted
8. Dynamic
9. High Performance
10. Multithreaded



SETTING UP THE DEVELOPMENT ENVIRONMENT

- Installation Instructions for Windows

- **Download the JDK Installer**

  - Access Java SE and Click the link that corresponds to **`.exe`** for your version of Windows.

  - Download the **`jdk-17_windows-x64_bin.exe`** file.

- **Run the JDK Installer**

  - Start the JDK 17 installer by double-clicking the installer's icon or file name in the download location.

  - Follow the instructions provided by the Installation wizard.

- **Setting the PATH Environment Variable**

  - Steps to set the PATH variable are:

  1. To set the PATH variable permanently, add the full path of the **`jdk-17/bin`** directory to the PATH variable. The Full Path is **`C:\Program Files\Java\jdk-17\bin`**.

  2. To set the PATH on Windows:

  2.1. Select Control Panel and then System.

  2.2. Click Advanced and then Environment Variables.

  2.3. Add the location of the `bin` folder of the JDK to the PATH variable in the System Variables.

- Installation Instructions for Mac OS

- **Download the JDK Installer**

  - Download the JDK `**.dmg**` file, `**jdk-17_macos-x64_bin.dmg**` from the Oracle Website.

- **Run the JDK Installer**

  - From either the browser Downloads window or from the file browser, double-click the `**.dmg**` file to start it.

  - A Finder window appears that contains an icon of an open box and the name of the `**.pkg**` file.

  - Double-click the JDK 17 `**.pkg**` icon to start the installation application.

  - The installation application displays the Introduction window.

  - A window appears that displays the message: `**Installer is trying to install new software.**` Enter your password to allow this.

  - Enter the Administrator username and password and click Install Software.

  - The software is installed, and a confirmation window is displayed.

- Installation Instructions for Linux

In this, there are two types of Instructions for installing the JDK on Linux System.

1. Installing the 64-Bit JDK 17 on Linux Platforms

- **Download the 64-Bit JDK**

- Download the file **`jdk-17_linux-x64_bin.tar.gz`** from the Oracle Website.

- Anyone (not only by root users) can install the archive binary in any location having write access.

- **installing the JDK Installer**

- Change the directory to the location where you want to install the JDK, then move the **`.tar.gz`** archive binary to the current directory.

- Unpack the tarball and install the JDK

`tar zxvf jdk-17_linux-x64_bin.tar.gz`

- The Java Development Kit files are installed in a directory called **`jdk-17`**.

- Delete the **`.tar.gz`** file if you want to save disk space.

2. Installing the 64-Bit JDK on RPM-Based Linux Platforms

- **Download the 64-Bit JDK on an RPM based Linux Platform**

  - Download the file, `jdk-17_linux-x64_bin.rpm`.

  - In RPM based platform ensure that you have a root user access by running the command `**su**` and entering the superuser password.

- **installing the JDK Installer**

  - Install the package using the following command:

    `**$ rpm -ivh jdk-17_linux-x64_bin.rpm**`

  - Upgrade the package using the following command:

    `**$ rpm -Uvh jdk-17_linux-x64_bin.rpm**`

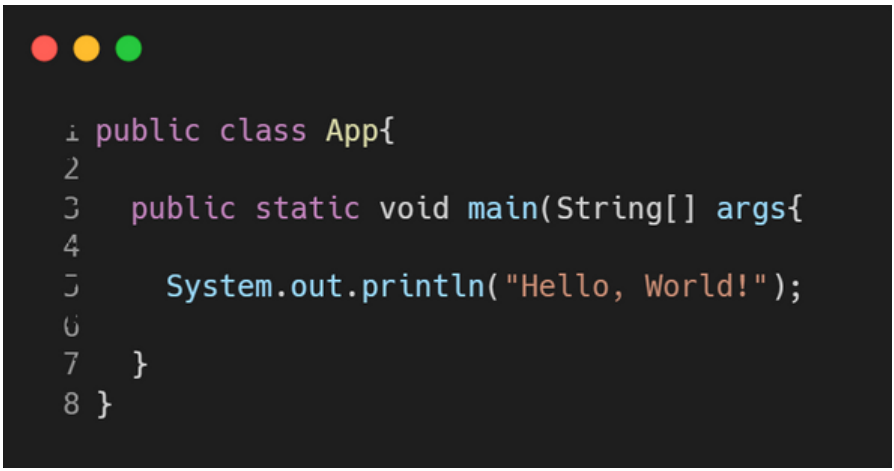
  - Delete the .rpm file if you want to save disk space.

  - Exit the root shell. Reboot is not required.

After completing installations of JDK. To verify the installation of the JDK open the terminal and run the following command `**java --version**`. This should display the version of the JDK you have installed in the system.

FIRST JAVA PROGRAM

Here is our First Java Program



```
1 public class App{
2
3   public static void main(String[] args{
4
5     System.out.println("Hello, World!");
6
7   }
8 }
```

- Let's deep dive into our First Program.

- `**public class App**`: This is a class named as App which is a blueprint for creating objects. The `public` keyword makes the class accessible from anywhere in the code.

- `**public static void main(String[] args)**`: This is the main method of the Java Program .

- The `**public**` and `**static**` keywords are used to specify that the method can be called without using the instance of the class.

- The `**void**` keyword means that the method doesn't returns a value.

- The `**main**` method is where the program starts execution.

- The `**args**` parameter is an array of `**String**` objects that can be passed to program when executed.

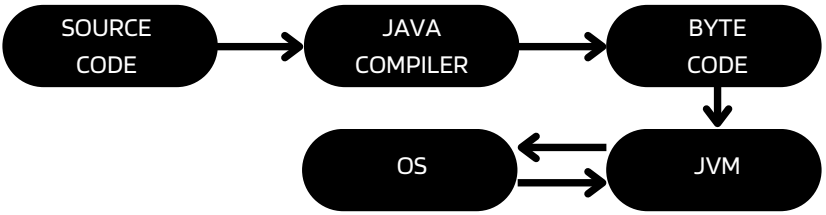
- '**System.out.println("Hello,World!")**;`: This line outputs the text "Hello, World!" to the console.

- The `**System.out.println**` method is used to print the text to the console.

- The `**println**` method automatically adds a newline character at the end of the text.

- After this explanation of first program in Java. In the next chapter we will learn about "Behind the Scenes of this Java Code".

BEHIND THE SCENES OF JAVA CODE



- The Java source code is compiled into bytecode using the Java compiler (javac). The bytecode is a platform-independent format that can be executed by the Java Virtual Machine (JVM) on any device.
- The bytecode is loaded into the JVM, which verifies it to ensure it follows the rules of the Java language.
- The JVM executes the bytecode by translating it into machine-specific instructions and executing them.
- The program produces output and terminates when it has completed execution or when it encounters an error.
- The JVM handles any errors and exceptions that occur during the execution of the program.
- In short, the Java source code is compiled into bytecode, which is executed by the JVM to produce output. The JVM provides a runtime environment for executing the program and handling errors and exceptions.

COMMENTS, DATA TYPES, VARIABLES AND CONSTANTS

- **Java Comments**

- The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

- **Types of Java Comments**

- There are 3 types of comments in Java.

1. Java Single Line Comment

- The Single Line Comment is used to comment only one line.

- Syntax:

```
//This is a Single Line Comment
```

App

```
1 public class App{
2
3   public static void main(String[] args{
4
5     int i = 10; //Here, i is a variable
6     System.out.println(i);
7
8   }
9 }
```

2. Java Multi-Line Comment

- The Multi-Line Comment is used to comment multiple lines of code.

- Syntax:

```
/*
  This is a multi-line comment.
*/
```

```java
public class App{

  public static void main(String[] args{
    /*
    Let's declare a int variable named 'i'
    and print it on the console
    */
    int i = 10;
    System.out.println(i);

  }
}
```

3. Java Documentation Comment

- The Documentation Comment is used to create documentation API. To create documentation API, you need to use **Javadoc** tool.

- Syntax:

```
/**
  This is a documentation comment.
*/
```

```java
public class App{

  public static void main(String[] args{
    /**
    This program is used to print an
    integer variable on the console
    */
    int i = 10;
    System.out.println(i);

  }
}
```

- Compile the code using `**javac**` tool:

Command: `javac App.java`

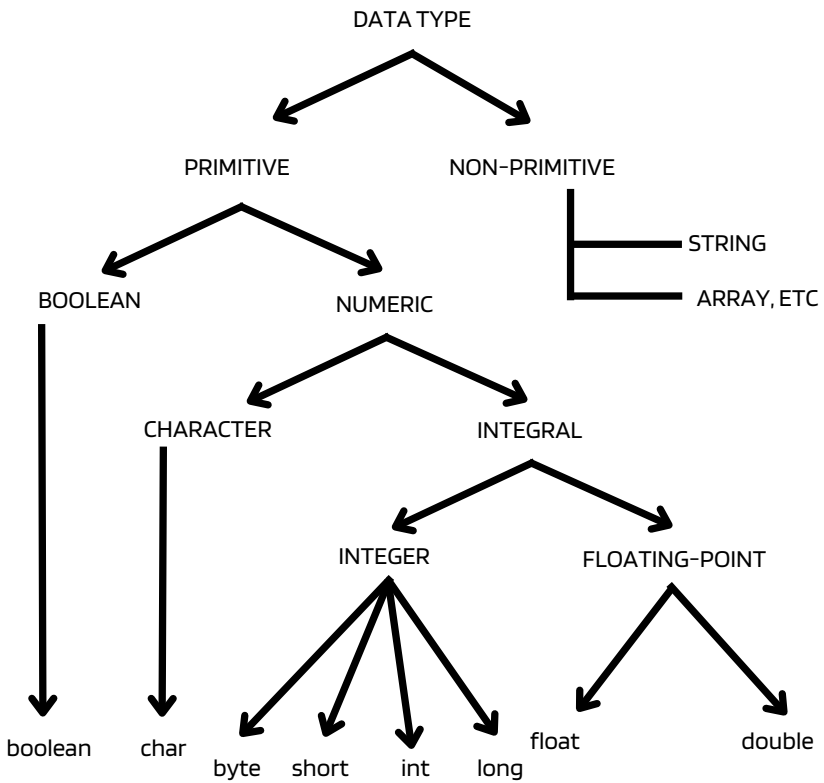
- Create Documentation API by '**javadoc**` tool

Command: `javadoc App.java`

- Now, there will be HTML files created for your App class in the current directory.
- Opening the files, you will see the explanation of App class provided through documentation comment.

- **Data Types:**

- Data Types represent the different values to be stored in the variable.
- In Java, there are two types of data types:
    - Primitive Data Types
    - Non-Primitive Data Types



Data Type	Size(Bits)	Range
`byte`	8	-128 to 127
`short`	16	-32768 to 32767
`int`	32	-2147483648 to 2147483647
`long`	64	-9223372036854775808 to 9223372036854775807
`float`	32	Approx. ±3.40282347 x 10^38
`double`	64	Approx. ±1.7976931348623157 x 10^308
`char`	16	0 to 65535
`boolean`	Not Specifed	`true` or `false`

- **Variables in Java:**

- A variable in Java is a named storage location in memory that holds a value.

- Here, there are 3 types of variables in java:

  - Local Variable:

  - A variable which is declared inside the method is called local variable.

  - Instance Variable:

  - A variable which is declared inside the class but outside a method is called as Instance variable.

  - Static Variable:

  - A variable that is declared as static using **`static`** keyword is called a static variable.

- A Small example to understand the types of variables

```java
public class App {

  int a; //instance variable

 static int b; //static variable

  public void method1(){
    int c; //local variable
  }
}
```

- **<u>Constants in Java</u>**

- A constant in Java is a variable whose value cannot be changed once it has been assigned. It uses **`final`** keyword.

- A Small example to understand the constants in Java:

```java
public class App {

  public static void main(String[] args) {

    //PI is now a constant variable its value cannot be changed
    final double PI = 3.14159;

    System.out.println(PI);

  }
}
```

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program in Java that declares and initializes a constant of type float and prints its value to the console.

2. Write a program in Java that declares and initializes three variables of different data types (e.g. int, double, boolean), and prints their values to the console.

3. Write a program in Java that declares and initializes two variables of type char, concatenates them and prints the result to the console.

4. Write a program in Java that declares and initializes all types of data types, and print their values to the console.

5. Write a program in Java to write a documentation comment in your code and using the Javadoc tool examine the output that comes out of it.

SCOPE AND LIFETIME OF VARIABLES

- In Java, the scope of a variable refers to the part of the program where the variable can be accessed or used, and lifetime of a variable refers to the duration of time during which a variable exists in memory.

- Local Variables:
  - These are declared within a method and have a scope limited to that method. The lifetime of local variables is limited to the duration of the method call.

- Instance Variables:
  - These are declared outside of any method and are part of an object. They have a scope throughout the entire program. The lifetime of instance variables is as long as the object that they belong to exists.

- Static Variables:
  - These are declared with the static keyword and are associated with the class rather than with individual objects. They have a scope throughout the entire program. The lifetime of class/static variables is as long as the program runs.

OPERATORS AND EXPRESSIONS

- **Operators:**

- Operator is a symbol that is used to perform operations.

- There are many types of operators in Java:

  - **Unary Operator**

- The operator that can be used to perform operations on a single operand is called Unary Operator.

1. Unary Minus (`-`): Negates the value of the operand.

```
1 int x = 10;
2 int y = -x;
3 System.out.println(y); //output: -10
```

2. Unary Plus (`+`): Does not changes the value of the operand.

```
1 int x = 10;
2 int y = +x;
3 System.out.println(y); //output: 10
```

3. Logical Negation (`!`): Reverses the value of a boolean operand.

```
1 boolean x = true;
2 boolean y = !x;
3 System.out.println(y); //output: false
```

4. Increment (`++`): Increments the value of the operand by 1.

```java
int x = 10;
int y = x++; //x = x + 1
System.out.println(x); //output: 11
System.out.println(y); //output: 10
```

5. Decrement (`--`): Decrements the value of the operand by 1.

```java
int x = 10;
int y = x--; //x = x - 1
System.out.println(x); //output: 9
System.out.println(y); //output: 10
```

- Post/Pre Increment/Decrements.

- Pre-Increment/Decrement: The operator is placed before the operand. The value of the operand is incremented (or decremented) before its value is used in the expression. Example: `++x`, `--x`

- Post-Increment/Decrement: The operator is placed after the operand. The value of the operand is used in the expression, and then incremented (or decremented) afterwards. Example: `x++`, `x--`

- **Arithmetic Operator**

- The operator that can be used to perform mathematical operations is called Arithmetic Operator.

1. Addition (`+`): Adds two operands

```
1 int x = 10;
2 int y = 20;
3 int z = x + y;
4 System.out.println(z); //output: 30
```

2. Subtraction (`-`): Subtracts the second operand from the first

```
1 int x = 20;
2 int y = 10;
3 int z = x - y;
4 System.out.println(z); //output: 10
```

3. Multiplication (`*`): Multiplies two operands.

```
1 int x = 5;
2 int y = 4;
3 int z = x * y;
4 System.out.println(z); //output: 20
```

4. Division (`/`): Divides the first operand by the second.

```
1 int x = 10;
2 int y = 5;
3 int z = x / y;
4 System.out.println(z); //output: 2
```

5. Modulus (`%`): Returns the remainder of dividing the first operand by the second operand.

```java
int x = 11;
int y = 5;
int z = x % y;
System.out.println(z); //output: 1
```

- **Shift Operator**

- The shift operators are used to shift the bits of a binary representation of an integer value to the left or right..

1. Left Shift (`<<`): Shifts the bits of the first operand to the left by the number of positions specified by the second operand.

```java
int x = 10; // Binary: 00001010
int y = x << 2; //shifting last 2 bits to left
System.out.println(y); // output: 40 (00101000)
```

2. Right Shift (`>>`): Shifts the bits of the first operand to the right by the number of positions specified by the second operand.

```java
int x = 40; // Binary: 00101000
int y = x >> 2; //shifting last 2 bits to right
System.out.println(y); // output: 10 (00001010)
```

3. Unsigned Right Shift(`>>>`): Shifts the bits of the first operand to the right by the number of positions specified by the second operand, while shifting in zeros on the left.

```java
int x = -40;  //11111111111111111111111111101000
int y = x >>> 2;
System.out.println(y); //output: 1073741819
//00111111111111111111111111101000
```

- **Relational Operator**

- Relational operators are used to compare two values and determine the relationship between them.

1. Equal to (`==`): Returns `true` if the values of the operands are equal, and `false` otherwise.

```java
int x = 10;
int y = 10;
boolean res = (x == y);
System.out.println(res); //Output: true
```

2. Not Equal To (`!=`): Returns `true` if the values of the operands are not equal, and `false` otherwise.

```java
int x = 20;
int y = 10;
boolean res = (x != y);
System.out.println(res); //Output: true
```

3. Less Than(`<`): Returns `true` if the value of the first operand is less than the value of the second operand, and `false` otherwise.

```java
int x = 20;
int y = 30;
boolean res = (x < y);
System.out.println(res); //Output: true
```

4. Less than or equal to (`>=`): Returns `true` if the value of the first operand is less than or equal to the value of the second operand, and `false` otherwise.

```java
int x = 20;
int y = 30;
boolean res = (x <= y);
System.out.println(res); //Output: true
```

5. Greater than (`>`): Returns `true` if the value of the first operand is greater than the value of the second operand, and `false` otherwise

```java
int x = 30;
int y = 20;
boolean res = (x > y);
System.out.println(res); //Output: true
```

6. Greater than Equal To (`>=`): Returns true if the value of the first operand is greater than or equal to the value of the second operand, and false otherwise.

```java
int x = 40;
int y = 20;
boolean res = (x >= y);
System.out.println(res); //Output: true
```

- **Logical Operators**

- Logical operators are used to perform logical operations on boolean values.

1. Logical AND (`&&`): Returns `true` if both operands are `true`, and `false` otherwise.

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

```java
boolean x = true;
boolean y = false;
boolean res = (x && y);
System.out.println(res); //Output: false
```

2. Logical OR (`||`): Returns `true` if either of the operands is `true`, and `false` otherwise.

A	B	A || B
true	true	true
true	false	true
false	true	false
false	false	false

```java
boolean x = true;
boolean y = false;
boolean res = (x || y);
System.out.println(res); //Output: true
```

3. Logical NOT (`!`): Returns the opposite of the operand.

A	!A
true	false
false	true

```java
boolean x = true;
boolean res = !x;
System.out.println(res); //Output: false
```

- **Bitwise Operator**

- Bitwise operators are used to perform bit-level operations on integer values.

1. Bitwise AND(`&`): Performs a bit-level AND operation on two operands. The result is 1 if both corresponding bits are 1, and 0 otherwise.

```java
int x = 10; //00001010
int y = 20; //00010100
int res = x & y;
System.out.println(res); //Output: 0
```

2. Bitwise OR (`|`): Performs a bit-level OR operation on two operands. The result is 1 if either of the corresponding bits is 1, and 0 otherwise.

```java
int x = 10; //00001010
int y = 20; //00010100
int res = x | y;
System.out.println(res); //Output: 30 (00011110)
```

3. Biwise NOT(`~`): Performs a bit-level NOT operation on the operand. It inverts all the bits of the operand, so that 0 becomes 1 and 1 becomes 0.

```java
int x = 10; //00001010
int res = (~x);
System.out.println(res); //Output: -11
```

4. Bitwise XOR (`^`): Performs a bit-level XOR operation on two operands. The result is `1` if exactly one of the corresponding bits is `1`.

```java
int x = 10; //00001010
int y = 20; //00010100
int res = x ^ y;
System.out.println(res); //Output: 30
```

- **Ternary Operator(Conditional)**

- The ternary operator is a shorthand for an if-else statement.

Syntax:

condition ? expression1 : expression2

Example:

```
1 int x = 10;
2 int y = (x > 5) ? 100: 200;
3 System.out.println(y); //Output: 100
```

- **Assignment Operator**

- The assignment operator `=` is used to assign a value to a variable.

- For Example:

int x = 10;

- Java also supports compound assignment operators that allow you to perform an operation and assign the result in one step.

```
1 x += 5; //x = x + 5
2 x -= 5; //x = x - 5
3 x *= 5; //x = x * 5;
4 x /= 5; //x = x / 5;
5 x %= 5; //x = x % 5;
```

- These compound assignment operators are shorthand for their equivalent operations and can make the code more concise.

- **Expressions in Java**

- In Java, an expression is a combination of values, variables, and operators that can be evaluated to produce a single value. Expressions are used to perform operations, assign values to variables, and control the flow of execution in a program.

- Here are some examples of expressions in Java:

```
10 + 20
x + y
x < y
```

- Expressions can be combined to create more complex expressions, and they can be used in a variety of contexts, such as in conditional statements, loop conditions, and function calls. The order in which operations are performed in an expression is determined by the rules of operator precedence, and parentheses can be used to explicitly control the order of evaluation.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Create a Console based Calculator in java which adds, subtracts, multiplies, divides, and finds remainder and print the result on the console.

2. Write a Java program to determine if a number is even or odd using ternary operator.

.

3. Write a Java program to determine if a number is positive, negative, or zero using conditional operator.

4. Write a Java program to find the average of 3 numbers and print the average on the console.

5. Write a Java program to swap the values of two variables.

6. Write a Java program to find the maximum of three numbers.

7. Write a Java program to find the bitwise AND,OR and NOT of two numbers.

TYPE CONVERSION, TYPE CASTING, COMMAND LINE ARGUMENTS AND ENUMS

- **TYPE CONVERSION AND TYPE CASTING**
    - Widening or Automatic Type Conversion

- Widening Conversion takes place when two data types are automatically converted. This happens when:
    - The 2 data types are compatible
    - When we assign value of smaller data type to a bigger data type.

Byte -> Short -> Int -> Long -> Float -> Double

- Narrowing or Explicit Conversion

- If we want to assign value of a larger data type to a smaller data type we perform explicit type casting or narrowing.
    - This is useful for incompatible data types where automatic conversion cannot be done.
    - Here, target-type specifies the desired type to convert the specified value to.

Double -> Float -> Long -> Int -> Short -> Byte

- **COMMAND LINE ARGUMENTS(CLA)**

- The concept of passing arguments to the main method in Java is called "command line arguments." Command line arguments are a way to pass additional information to a Java program when it is executed. These arguments are passed to the main method as an array of strings, and they can be used to control the behavior of the program or to specify data for the program to operate on.

- Here's an example of a main method that takes command line arguments:

```java
public class App {

    public static void main(String[] args) {
      //using command line arguments.
      String name = args[0];
      System.out.println("Hello, "+ name);

}

}
```

- And here's an example of how you could call the program with command line arguments:

```
java App archies
```

- In this example, the `args` array in the main method would contain the value `"archies"`.

- **ENUMS IN JAVA**

- Enums in Java are a special type of class that represent a fixed set of constant values. Enums are used to define a set of named constants, and they provide a convenient way to represent and work with sets of related values.

- It is available from JDK 1.5.

- The Java Enums are static and final implicitly.

- Here is an example of an Enum definition:

```
1     public enum Season {
2         SUMMER,
3         MONSOON,
4         WINTER,
5         SPRING
6   }
7
```

- In this example, the enum `Season` defines a set of constant values for the season in a year. The values of an enum are instances of the enum type, and they are created automatically when the enum is defined.

- Here's an example of how an enum might be used in a for-each loop:

```
1 public static void main(String[] args){
2    for(Season s : Season.values()){
3      System.out.println(s);
4    }
5 }
6
```

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program that takes a temperature in Fahrenheit as a command line argument, converts it to Celsius, and prints the result.

2. Write a program that takes two integer command line arguments, converts them to double values, and performs division, printing the result.

.

3. Write a Java program to determine if a number is positive, negative, or zero using conditional operator.

4. Write an enum type to represent the days of the week and write a program that takes a day of the week as a command line argument and prints the next day of the week.

5. Write a program that takes two float command line arguments, casts them to integers, adds them together, converts the result back to a float, and prints the result.

CONDITIONAL STATEMENTS

- Conditional statements in Java allow you to execute certain parts of your code based on the result of a boolean expression. The most commonly used conditional statements in Java are the `**if**` statement, `**if-else**` statement, and the `**switch**` statement.

- <u>If Statement</u>:

- The **if** statement in Java is used to execute a block of code if a specific condition is met.

- Syntax:

```
if(condition){
  //code to be executed
}
```

- Here is an example of an if statement in Java:

```java
public class App {

  public static void main(String[] args){
   int x = 10;
    if(x > 5){
      System.out.println("x is greater than 5");
    }
}
}
```

- In this example, the condition x > 5 is evaluated, and if it is true, the message "x is greater than 5" will be printed.

- **If-else Statement:**

- The **if-else** statement in Java is a type of conditional statement that allows you to execute different sections of code based on the result of a boolean expression.

- Syntax:

```
if(condition){
 //code to be executed if condition is true
}else {
// code to be executed if condition is false
}
```

- Here is an example of an if-else statement in Java:

```java
public class App {

  public static void main(String[] args){
   int x = 10;
    if(x > 5){
      System.out.println("x is greater than 5");
    }else {
      System.out.println("x is smaller than 5");
    }
}
}
```

- In this example, the condition x > 5 is evaluated, and if it is true, the message "x is greater than 5" will be printed, otherwise if the condition x > 5 is false, the message "x is not smaller than 5" will be printed.

- **Switch Statement:**

- A **switch** statement in Java is a type of conditional statement that allows you to execute different blocks of code based on the value of an expression. The switch statement is useful when you have a limited number of possible values for a variable, and you want to perform different actions based on the value of that variable.

- Here is an example of a switch statement in Java:

```java
public class App {

  public static void main(String[] args){
   int x = 3;

   switch(x) {
     case 1:
       System.out.println("The value of x is 1");
       break;
     case 2:
       System.out.println("The value of x is 2");
       break;
     case 3:
       System.out.println("The value of x is 3");
       break;
     default:
       System.out.println("Incorrect Value");
       break;
   }
 }
}
```

- In this example, the value of x is 3, so the message "The value of x is 3" is printed. The break statement is used to exit the switch statement after a match is found, and the default case is used to handle values that do not match any of the case labels.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program to find the grade of a student based on their marks using a switch statement.

2. Write a program to check if a character is a vowel or not.

.

3. Write a program to find the largest of three numbers using an if-else statement.

4. Write a program to check if a year is a leap year or not.

5. Write a program that inputs a number and prints the number of digits in the number.

6. Write a program that inputs a day of the week and prints the corresponding day of the week.

LOOPS

- In Java, loops are used to repeat a block of code for a specified number of times or until a certain condition is met.

- For Loop:

- A `**for**` loop is used when you know the number of times you want to repeat the loop, and it consists of a loop header, loop body, and an optional update clause. The loop header specifies the initialization, condition, and update. The loop body contains the statements that are executed repeatedly until the condition is met

- Syntax:

```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

- Example:

```
1 public class App {
2
3     public static void main(String[] args) {
4
5         for(int i = 1; i <= 5; i++){
6             System.out.println(i):
7         }
8
9     }
10 }
11
```

- In this example, for loop is printing the numbers from 1 to 5 on the console. The loop terminates when the condition `i <= 5` is false.

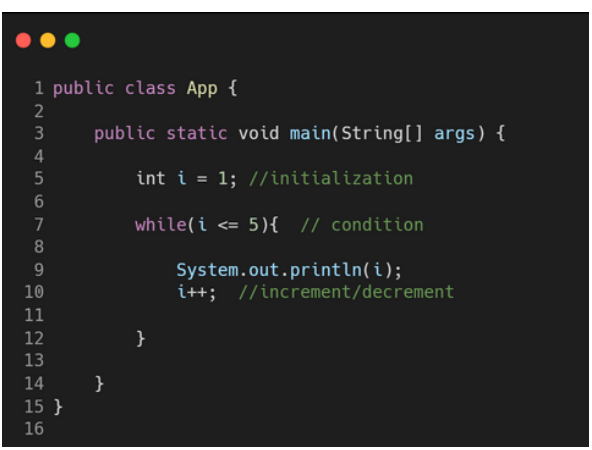
- While Loop:

- A `**while**` loop is used when you don't know the number of times you want to repeat the loop and you want to keep looping as long as a certain condition is true. The loop header specifies the condition, and the loop body contains the statements that are executed repeatedly until the condition is false.

- Syntax:

```
while(condition) {
    // code to be executed
}
```

- Example:



```java
public class App {

    public static void main(String[] args) {

        int i = 1; //initialization

        while(i <= 5){  // condition

            System.out.println(i);
            i++;  //increment/decrement

        }

    }
}
```

- In this example, while loop is printing the numbers from 1 to 5 on the console. The loop terminates when the condition `i <= 5` is false.

- Do-While Loop:

- The `**do-while**` loop in Java is similar to a while loop, but the loop body is always executed at least once, before the condition is evaluated.

- Syntax:

```
do{
 //loop body
 //code to be executed
}while(condition);
```

- Example:

```java
public class App {

    public static void main(String[] args) {

        int i = 1;

        do{

        System.out.println(i);
        i++;

        }while(i <= 5);

    }
}

```

- In this example, do-while loop is printing the numbers from 1 to 5 on the console. The loop terminates when the condition `i <= 5` is false.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program that inputs an integer n and prints the first n positive numbers.

2. Write a program that inputs an integer n and calculates the sum of the first n positive numbers.

3. Write a program that inputs a number and prints its multiplication table.

4. Write a program that inputs an integer n and prints the factorial of n.

5. Write a program that inputs a string and prints the reverse of the string.

6. Write a program that inputs two numbers a and b and calculates their greatest common divisor (GCD).

7. Write a program that inputs an integer n and calculates the sum of the squares of the first n positive numbers.

CONTINUE AND BREAK STATEMENTS

- The `**continue**` and `**break**` statements are control flow statements in Java that allow you to change the flow of execution in a loop.

- Continue:

- The continue statement is used to skip the current iteration of a loop and move on to the next iteration.

- Example:

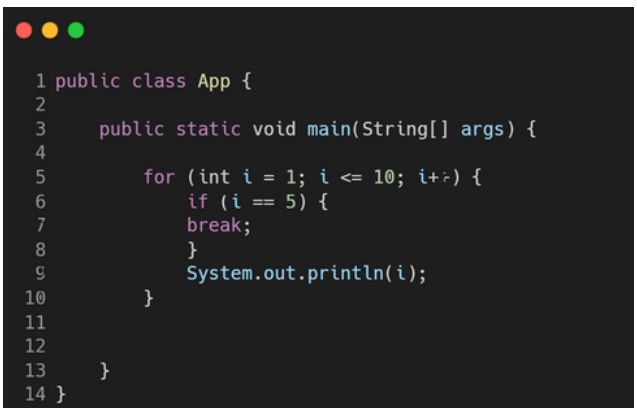
```java
public class App {

    public static void main(String[] args) {

      for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
        continue;
        }
      System.out.println(i);
}


    }
}
```

- In this code, the continue statement is used to skip the iteration of the loop if the value of i is even. As a result, only the odd numbers from 1 to 10 will be printed.

- Break:

- The break statement is used to exit a loop prematurely. When a break statement is executed, the loop terminates and the control flow moves to the statement immediately following the loop.

- Example:



```java
public class App {

    public static void main(String[] args) {

        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
            break;
            }
            System.out.println(i);
        }


    }
}
```

- In this code, the break statement is used to exit the loop when the value of `i` is 5. As a result, only the numbers 1 to 4 will be printed.

- Both the `continue` and `break` statements can be used with all types of loops in Java, including for, while, and do-while loops.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program that inputs a sentence and prints the first word that has more than 5 letters.

2. Write a program that inputs a sentence and counts the number of words in the sentence.

3. Write a program that inputs two numbers a and b and calculates their least common multiple (LCM).

4. Write a program that inputs a string and counts the number of vowels in the string.

5. Write a program that inputs a number n and calculates the sum of the first n positive numbers that are divisible by 3.

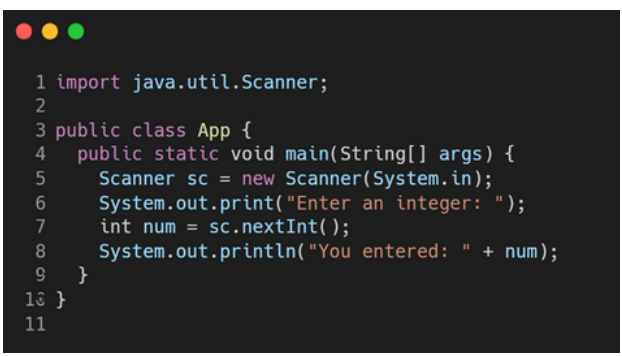
6. Write a program that inputs an integer n and calculates the sum of the cubes of the first n positive numbers.

USING SCANNER CLASS

- The Scanner class in Java is part of the `java.util` package and is used to read data from the keyboard or other input sources, such as a file or a network socket.

- The Scanner class provides several methods for reading different data types, including `nextInt()` for reading integers, `nextDouble()` for reading floating-point numbers, and `nextLine()` for reading strings.

- Here's an example of how to use the Scanner class to read an integer from the keyboard:



```java
import java.util.Scanner;

public class App {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int num = sc.nextInt();
    System.out.println("You entered: " + num);
  }
}

```

- In this example, the Scanner class is imported and then an instance of it is created using System.in as the input source. The `nextInt()` method is then used to read an integer from the keyboard, and the value is stored in the num variable. Finally, the value of num is printed to the console using `println()`.

TYPES OF FUNCTIONS

- Function:

- Functions, also known as methods, are blocks of code that perform a specific task and can be called from other parts of the program. Functions in Java help to modularize the code, make it easier to understand, and reuse code in multiple places.

- Here's the syntax for a function in Java:

- **modifiers**: These specify the access level of the method, such as public, private, protected, or default.
- **return type**: This is the data type of the value returned by the function. If the function doesn't return a value, the return type is void.
- **functionName**: This is the name of the function. The name should follow the same naming conventions as variables, and should be descriptive of what the function does.
- **parameters**: These are the values passed to the function when it's called. A function may have zero or more parameters. Each parameter has a type and a name.

- There are two types of functions in Java.
    - Instance Methods:
        - These are methods that are associated with instances of a class and can access the instance variables of the class.
        - Here's an example of an instance method in Java

Rectangle.java

```java
class Rectangle {
  int length;
  int width;

  int area() {
    return length * width;
  }
}
```

- In this example, the **area** method calculates the area of a **Rectangle** object. To call this method, you need to create an instance of the **Rectangle** class and use the **.** operator to access the method:

App.java

```java
class App {

  public static void main(String [] args){
    Rectangle rect = new Rectangle();
    rect.length = 10;
    rect.width = 5;
    int a = rect.area();
    System.out.println("Area of Rectangle is: "+ a);

  }
}

```

- Static Methods:

- These are methods that are associated with the class itself, and don't require an instance of the class to be created.

- Here's an example of a static method in Java

Sqaure.java

```java
public class Square {

    static int area(int x){
      return x * x;
    }
}

```

- In this example, the **area** method calculates the area of a square. To call this method, you can use the class name and the `.` operator:

Sqaure.java

```java
public class Square {

    public static void main(String[] args) {

      int area = Square.area(5);
      System.out.println("Area of Sqaure: "+area);

    }
}

```

- Static methods are often used to provide utility functions that can be called from anywhere in the code, without having to create an instance of the class.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program that inputs a sentence and prints the first word that has more than 5 letters.

2. Write a program that inputs a sentence and counts the number of words in the sentence.

3. Write a program that inputs two numbers a and b and calculates their least common multiple (LCM).

4. Write a program that inputs a string and counts the number of vowels in the string.

5. Write a program that inputs a number n and calculates the sum of the first n positive numbers that are divisible by 3.

6. Write a program that inputs an integer n and calculates the sum of the cubes of the first n positive numbers.

GARBAGE COLLECTION

- Garbage collection is an important feature of Java that automatically manages the memory used by the program. When a Java program creates an object, it is stored in memory, but at some point, in the program's execution, the object is no longer needed. Garbage collection frees up the memory occupied by these objects, allowing the memory to be reused by other parts of the program.

```java
// create an object
MyObject obj = new MyObject();

// use the object
obj.doSomething();

// discard the reference to the object
obj = null;

// at this point, the object is no longer being used and is eligible for garbage collection

```

- In this example, we create an object of the class **MyObject** and assign it to a reference variable called **obj**. We then use the object by calling its **doSomething()** method. Once we're done using the object, we discard the reference by setting **obj** to null. This means that there are no longer any references to the object, and it is now eligible for garbage collection.

STRINGS IN JAVA

- In Java, a string is a sequence of characters, represented as an object of the String class. Strings in Java are immutable, which means that once a string is created, its value cannot be changed.
- There are several ways to create a string in Java:

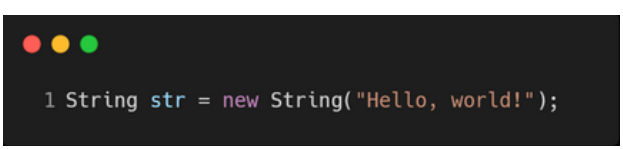
1. Using string literals: A string literal is a sequence of characters enclosed in double quotes.

   For Example:

```
1 String str = "Hello, world!";
```

2. Using the **String** constructor: The **String** class has several constructors that can be used to create a new string object.

   For Example:



```
1 String str = new String("Hello, world!");
```

- Here are some common string functions in Java:

1. `length()`: Returns the length of the string.

```java
String str = "Hello, world!";
int length = str.length(); //length will be 13
```

2. `charAt(index)`: Returns the character at the specified index in the string.

```java
String str = "Hello, World!";
char ch = str.charAt(1); //ch will be 'e'
```

3. `substring(startIndex, endIndex)`: Returns a substring of the original string, starting at the **startIndex** and ending at the **endIndex** (exclusive).

```java
String str = "Hello, World!";
String sub = str.substring(0, 5); // sub will be "Hello"
```

4. `toUpperCase()`: Returns a new string with all characters in uppercase.

```java
String str = "Hello, World!";
String upper = str.toUpperCase(); // upper will be "HELLO, WORLD!"
```

5. `toLowerCase()`: Returns a new string with all characters in lowercase.

```java
String str = "Hello, World!";
String lower = str.toLowerCase(); // lower will be "hello, world!"
```

6. `equals(Object obj)`: Returns true if the string is equal to the specified object, false otherwise.

```
1 String str = "Hello, World!";
2 boolean equal = str.equals("Hello, World!"); // equal will return "true"
```

7. `equalsIgnoreCase(String anotherString)`:Returns true if the string is equal to the specified string, ignoring case differences.

```
1 String str = "Hello, World!";
2 boolean equal = str.equalsIgnoreCase("Hello, world!"); // equal will return "true"
```

8. `indexOf(int ch)`: Returns the index of the first occurrence of the specified character in the string.

```
1 String str = "Hello, world!";
2 int index = str.indexOf('w'); // index will be 7
```

9. `indexOf(String str): Returns the index of the first occurrence of the specified substring in the string.

```
1 String str = "Hello, world!";
2 int index = str.indexOf("world"); // index will be 7
```

10. `lastIndexOf(int ch)`: Returns the index of the last occurrence of the specified character in the string.

```
1 String str = "Hello, world!";
2 int index = str.lastIndexOf('o'); // index will be 7
```

11. `lastIndexOf(String str)`: Returns the index of the last occurrence of the specified substring in the string.

```java
String str = "Hello, world!";
int index = str.lastIndexOf("Hello"); // index will be 7
```

12. `startsWith(String prefix)`:Returns true if the string starts with the specified prefix, false otherwise.

```java
String str = "Hello, world!";
boolean startsWithHello = str.startsWith("Hello"); // startsWithHello will be true
```

13. `endsWith(suffix)`: Returns true if the string ends with the specified suffix, false otherwise.

```java
String str = "Hello, world!";
boolean endsWithWorld = str.endsWith("world!"); // endsWithWorld will be true
```

14. `contains(CharSequence s): Returns true if the string contains the specified character sequence, false otherwise.

```java
String str = "Hello, world!";
boolean contain = str.contains("World!"); // endsWithWorld will be true
```

15. `replace(char oldChar, char newChar)`: Returns a new string with all occurrences of oldChar replaced by newChar.

```java
String str = "Hello, world!";
String replace = str.replace('e', 'o'); // replace will be Hollo, World!
```

16. `replaceAll(String regex, String replacement)`: Returns a new string with all occurrences of the regular expression regex replaced by replacement.

```
String str = "Hello, world!";
String replaceAll = str.replaceAll("World", "Java"); // replaceAll will be Hollo, Java!
```

17. `split(String regex)`:Splits the string into an array of substrings, using the specified regular expression regex as the delimiter.

```
String str = "Hello, world!";
String split[] = str.split(","); // split array will be ["Hello" , "World"]
```

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to find the length of a string without using the `length()` method.

2. Write a Java program to count the number of vowels in a string.

3. Write a Java program to check if a string is a palindrome.

4. Write a Java program to remove duplicate characters from a string.

5. Write a Java program to find the first non-repeated character in a string.

6. Write a Java program to find the most frequent character in a string.

7. Write a Java program to check if two strings are anagrams of each other.

8. Write a Java program to check if a string is a valid shuffle of two other strings.

STRINGBUILDER AND STRINGBUFFER

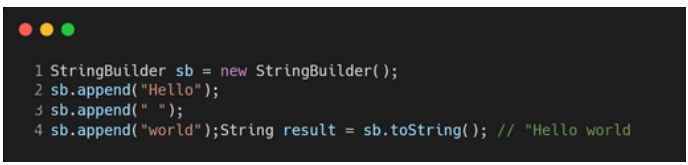
- `**StringBuilder**` and `**StringBuffer**` are both classes in Java that provide mutable (modifiable) strings, which means that you can modify the content of the string without creating a new string object. They are useful when you need to perform a large number of string manipulations, such as concatenating multiple strings, or removing or inserting characters in a string.

- `**StringBuilder**` was introduced in Java 5, and it is preferred over `**StringBuffer**` in most cases, because it is faster and more efficient.
- `**StringBuffer**` is a legacy class that was introduced in Java 1.0, and it is still available for backward compatibi--lity.

- The main difference between `**StringBuilder**` and `**StringBuffer**` is that,
- `**StringBuffer**` is thread-safe, which means that multiple threads can safely access and modify the same instance of the class at the same time, without interfering with each other.

- `**StringBuilder**` is not thread-safe, which means that it is faster and more efficient, but it can cause problems if multiple threads access and modify the same instance of the class at the same time.

- Here is an example of how to use `**StringBuilder**` to concatenate multiple strings.



```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("world");String result = sb.toString(); // "Hello world
```

- In this example, we create a new `**StringBuilder**` instance, and then we use the `append()` method to add multiple strings to the string builder. Finally, we use the `toString()` method to convert the string builder back to a regular string.

- Here is an example of how to use **StringBuffer** to concatenate multiple strings:

```
StringBuffer sb = new StringBuffer();
sb.append("Hello");
sb.append(" ");
sb.append("world");
String result = sb.toString(); // "Hello world"
```

- This example is similar to the previous one, but we are using `**StringBuffer**` instead of StringBuilder. The code is almost identical, except for the class name.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to reverse a string using StringBuilder.

2. Write a Java program to count the number of occurrences of a substring in a string using StringBuffer.

3. Write a Java program to remove all occurrences of a substring from a string using StringBuilder.

4. Write a Java program to compare two strings lexicographically using StringBuilder.

5. Write a Java program to check if a given StringBuffer contains only digits

6.Write a Java program to check if a given string is a palindrome using StringBuffer.

7. Write a Java program to check if a string is a rotation of another string using StringBuilder.

ARRAYS AND 2D ARRAYS

- In Java, an array is a collection of variables of the same data type, referred to by a common name. It can be thought of as a container that holds a fixed number of elements of a specific data type. Arrays are used to store multiple values in a single variable, making it easy to access and manipulate the elements.

- For example, suppose you want to store the ages of five people in a program. You can use an array to store these values as follows:

```
int[] ages = {23, 35, 42, 28, 19};
```

- This creates an integer array named ages that holds five values. Each value is accessed by its index, which starts at 0 and goes up to `length - 1`.

- For example, to access the first element of the array, you can use:

```
int firstAge = ages[0];
```

- This assigns the value 23 (the first element of the array) to the variable `**firstAge**`.

- Arrays are a powerful and essential data structure in Java understanding how to work with them is crucial for developing applications.

- A 2D array is an array of arrays in which each element is itself an array. This means that a 2D array can be thought of as a table or a grid of elements arranged in rows and columns. In Java, a 2D array is declared by specifying two dimensions: the number of rows and the number of columns.
- Here is an example of declaring and initializing a 2D array in Java:

```
int[][] matrix = {{1, 2, 3}, {4, 5  6}, {7, 8, 9}};
```

- This creates a 3x3 matrix with the values 1 through 9 arranged in rows and columns. You can access the elements of the array using the row and column index, like this:

```
int value = matrix[1][2]; // gets the value in the second row, third column (6)
```

- In this example, the value variable would be assigned the value 6, which is the value in the second row, third column of the matrix.
- 2D arrays are useful for representing data that is naturally arranged in a table or grid format, such as images, game boards, and matrices. They are also commonly used in algorithms and data structures, such as dynamic programming and graph algorithms to store intermediate results and states.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to find the maximum and minimum element in an array.

2. Write a Java program to find the sum of all elements in an array.

3. Write a Java program to find the second largest element in an array.

4. Write a Java program to sort an array in ascending and descending order.

5. Write a Java program to find the sum of all elements of a 2D array.

6. Write a Java program to find the transpose of a 2D array.

7. Write a Java program to find the diagonal sum of a 2D array.

8. Write a Java program to check if a given 2D array is a magic square or not.

ACCESS SPECIFIERS

- Access specifiers in Java are used to define the visibility and accessibility of classes, methods, and variables in a program.

- There are four access specifiers in Java:

1. Public: Public is the most commonly used access specifier in Java. A public member can be accessed from anywhere, both within and outside the package.

- Example:

```java
public class MyClass {
    public int myVariable;
    public void myMethod() {
        // Method code here
    }
}
```

2. Private: A private access specifier means that the class, method, or variable can only be accessed from within the same class. For example:

```java
public class MyClass {
    private int myVariable;
    private void myMethod() {
        // Method code here
    }
}
```

3. Protected: A protected access specifier means that the class, method, or variable can only be accessed from within the same class, any subclass of the class, and any class in the same package. For example:

```java
public class MyClass {
    protected int myVariable;
    protected void myMethod() {
        // Method code here
    }
}
```

1. Default (No Keyword): A default or package-private access specifier means that the class, method, or variable can only be accessed from within the same package. For example:

```java
class MyClass {
    int myVariable;
    void myMethod() {
        // Method code here
    }
}
```

- Access specifiers are important in object-oriented programming because they allow you to control the visibility and accessibility of the various components of your program. This helps to keep your code organized, secure, and easy to maintain.

- Here's an example that demonstrates the use of access specifiers in Java:

```java
package com.example;

public class Person {
    private String name;
    protected int age;
    public String gender;

    public Persen(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    private String getName() {
        return name;
    }

    protected void setAge(int age) {
        this.age = age;
    }

    public String getGender() {
        return gender;
    }
}
```

OBJECT ORIENTED PROGRAMMING

- Object-oriented programming (OOP) is a programming paradigm that is used to organize complex code into more manageable and reusable components. In Java, OOP is a fundamental concept that forms the basis of much of the language's functionality.

- OOP is based on the concept of objects, which are instances of classes.

- A **class** is a blueprint or template that defines the attributes and behaviors of an object.

- An **object** is an instance of a class that has its own unique set of attribute values.

- Here is an example of classes and objects:

```java
public class Car {
    // instance variables
    private String make;
    private String model;
    private int year;

    // constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // instance method
    public void drive() {
        System.out.println("The " + make + " " + model + " is driving.");
    }


  public static void main(String[] args){
    Car myCar = new Car("Toyota", "Camry", 2022);
    System.out.println(myCar.make); // prints "Toyota"
    System.out.println(myCar.year); // prints 2022
    myCar.drive(); // prints "The Toyota Camry is driving."
  }

}
```

- In this example, the Car class has three instance variables (make, model, and year) and a constructor that initializes these variables. It also has an instance method, drive(), which prints a message to the console.
- Created a new object of the Car class in main function and assign it to the variable `myCar`. The constructor is called with the arguments "Toyota", "Camry", and 2022, which initializes the instance variables make, model, and year.
- After creating the object of the class and assigning it to the variable `myCar`, you can access the instance methods and variables. using the dot notation and calling the function and the variables you want.

- Let's also understand about the "this" keyword
- `**this**` keyword: In Java, the this keyword is a reference to the current object. It is used to refer to the instance variables and methods of the current object, and can also be used to invoke one constructor from another constructor in the same class.
- In this example, `this.make`, `this.model`and `this.year` refers to the instance variable name of the current object instance. This is useful when the parameter of the constructor method has the same name as the instance variable.

- `**super**` keyword: In Java, the super keyword is used to refer to the parent class or superclass of a subclass. It can be used to access the members (fields and methods) of the parent class, to invoke the parent class constructor, and to differentiate between a member of the subclass and a member of the parent class with the same name.

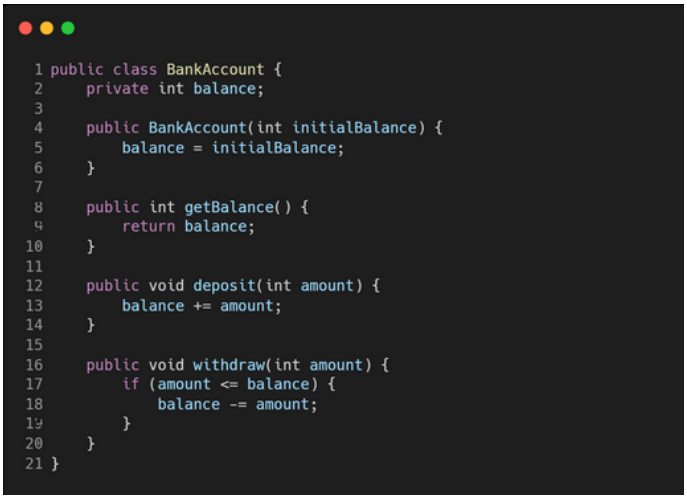
```java
public class Animal {
    protected String name;

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

public class Cat extends Animal {
    private int age;

    public Cat(String name, int age) {
        super(name); // call the constructor of the parent class with the parameter "name"
        this.age = age; // set the instance variable "age" to the parameter "age"
    }
}
```

Let's turn towards the OOP Concepts

- The four main concepts of OOP in Java are:

1. **Encapsulation**: Encapsulation is the practice of hiding the internal details of an object from the outside world and only exposing a public interface. This allows the object to be used in a modular and reusable way without exposing its internal workings. Encapsulation is an important concept in object-oriented programming because it helps to ensure that code is more modular, maintainable, and secure.

- Here is an example:



```java
public class BankAccount {
    private int balance;

    public BankAccount(int initialBalance) {
        balance = initialBalance;
    }

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        if (amount <= balance) {
            balance -= amount;
        }
    }
}
```

- In this example, we have a BankAccount class that encapsulates the balance of a bank account. The balance is stored as a private variable, which means that it can only be accessed from within the BankAccount class itself.

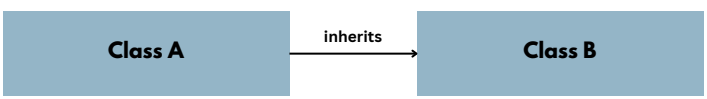
- To interact with the balance, the BankAccount class provides three public methods: getBalance, withdraw and deposit. The getBalance method allows the balance to be read, while the deposit method allows money to be added to the account and withdraw method allows money to be subtracted from the account.

- This makes the BankAccount class more secure and easier to maintain, since any changes to the balance can be made in a centralized and controlled way.

2. **Inheritance**: Inheritance is a fundamental concept in object-oriented programming that allows classes to inherit properties and behaviors from their parent classes. In Java, inheritance is implemented using the "extends" keyword to specify a parent class.

- There are several types of inheritance, including:

- Single Inheritance: Single inheritance is the simplest form of inheritance and involves a child class inheriting properties and behaviors from a single parent class.



- Here is an example:

```java
class Animal {
  public void eat() {
    System.out.println("Eating...");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Barking...");
  }
}
```

- In this example, the Dog class extends the Animal class, which means that it inherits the eat method. The Dog class also adds its own method, bark.

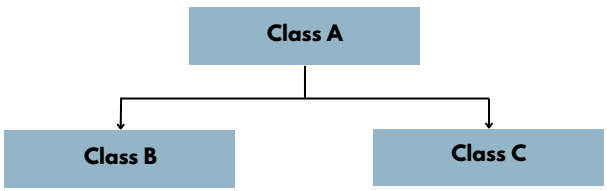
- Multilevel Inheritance: Multilevel inheritance involves a child class inheriting properties and behaviors from a parent class, which in turn inherits from another parent class.

- Here is an example:

```java
class Animal {
  public void eat() {
    System.out.println("Eating...");
  }
}

class Mammal extends Animal {
  public void move() {
    System.out.println("Moving...");
  }
}

class Dog extends Mammal {
  public void bark() {
    System.out.println("Barking...");
  }
}
```

- In this example, the Dog class extends the Mammal class, which in turn extends the Animal class. The Dog class inherits the eat method from the Animal class and the move method from the Mammal class.

- Hierarchical Inheritance: Hierarchical inheritance involves a child class inheriting properties and behaviors from a single parent class, but multiple child classes can inherit from the same parent class.



- Here is an example:

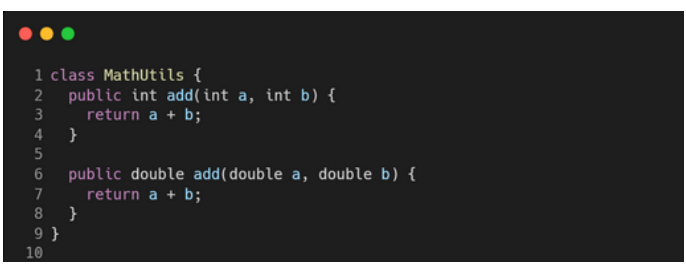
```java
class Animal {
  public void eat() {
    System.out.println("Eating...");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Barking...");
  }
}

class Cat extends Animal {
  public void meow() {
    System.out.println("Meowing...");
  }
}
```

- In this example, both the Dog class and the Cat class extend the Animal class. They both inherit the eat method from the Animal class, but each class adds its own unique method (bark for Dog and meow for Cat).

- Overall, inheritance is a powerful concept in Java that allows developers to create more specialized classes that share common properties and behaviors with their parent classes.

3. **Polymorphism**: Polymorphism is another fundamental concept in object-oriented programming and refers to the ability of an object to take on multiple forms. In Java, polymorphism can be achieved through method overloading and method overriding.

- Method Overloading: Method overloading is a technique that allows a class to have multiple methods with the same name but different parameters. The appropriate method is called based on the arguments passed to the method. Here is an example:



```
class MathUtils {
  public int add(int a, int b) {
    return a + b;
  }

  public double add(double a, double b) {
    return a + b;
  }
}
```

- In this example, the MathUtils class has two methods with the same name, add, but they take different parameter types (int and double). This allows the same method name to be used for different data types, making the code more flexible and easier to read.

- Method Overriding: Method overriding is a technique that allows a subclass to provide a specific implementation of a method that is already provided by its parent class. The overridden method has the same name, return type, and parameters as the original method. Here is an example:

```java
class Animal {
  public void makeSound() {
    System.out.println("The animal makes a sound");
  }
}

class Dog extends Animal {
  public void makeSound() {
    System.out.println("The dog barks");
  }
}
```

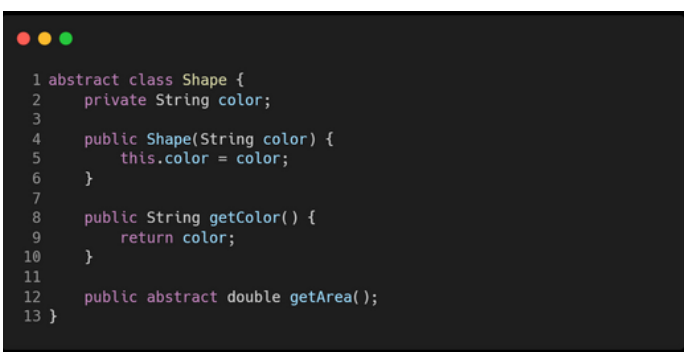
- In this example, the Animal class has a method called makeSound, and the Dog class overrides it with its own implementation. When the makeSound method is called on a Dog object, the overridden method in the Dog class is called instead of the method in the Animal class.

- Polymorphism allows us to write more flexible and reusable code by creating classes and methods that can be used in a variety of contexts.

4. **Abstraction**: Abstraction is a fundamental concept in object-oriented programming that allows you to represent the essential features of an object without showing the implementation details. It is achieved through abstract classes and interfaces.

- Abstract Classes: An abstract class is a class that is declared with the abstract keyword, and it cannot be instantiated. It can have abstract and non-abstract methods, and it can also have instance variables.

  Here is an example:



```
abstract class Shape {
    private String color;

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

    public String getColor() {
        return color;
    }

    public abstract double getArea();
}
```

- In this example, Shape is an abstract class that has a constructor, a private instance variable, a non-abstract method, and an abstract method. The abstract method, getArea(), does not have an implementation in the Shape class, and it is meant to be implemented in the subclasses that extend the Shape class.

- Interfaces: An interface is a collection of abstract methods, and it is used to define a contract that specifies what a class must do without providing any implementation details.
  Here is an example:

```java
interface Animal {
    public void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("The dog barks");
    }
}
```

- In this example, the Animal interface has only one method, makeSound(), which is implemented in the Dog class. By implementing the Animal interface, the Dog class is required to provide an implementation for the makeSound() method.

- You may have noticed that I didn't mention anything about `Multiple Inheritance` in the previous Inheritance section. This is because,
- Java supports single inheritance, which means that a class can inherit from only one superclass.
- However, Java provides a way to achieve multiple inheritance of interfaces through the use of interfaces.
- Let's understand it with help of an example.

- An interface only defines the methods that a class implementing it must have. A class can implement multiple interfaces, which enables it to inherit the abstract methods defined by each interface.

- For example, let's say we have two interfaces: Drawable and Scalable. The Drawable interface defines an abstract method draw(), and the Scalable interface defines an abstract method scale(). Now, we can create a class Circle that implements both of these interfaces, like this:

```java
interface Drawable {
    void draw();
}

interface Scalable {
    void scale();
}

class Circle implements Drawable, Scalable {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        // implementation of the draw method for a circle
    }

    @Override
    public void scale() {
        // implementation of the scale method for a circle
    }
}
```

- In this example, Circle implements both Drawable and Scalable interfaces, and provides its own implementation for the draw() and scale() methods.

- By using interfaces, we can achieve multiple inheritance in Java without running into the problems that arise with traditional multiple inheritance.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to create a class hierarchy with an abstract class as the base class and two subclasses that inherit from it. The abstract class should have at least one abstract method that is implemented in the subclasses.

2. Write a Java program to create an interface with one or more methods, and implement it in a class.

3. Write a Java program to demonstrate method overloading, with multiple methods that have the same name but different parameters.

4. Write a Java program to create a Shape class that has two subclasses, Circle and Rectangle. The Shape class should have an abstract method, getArea(), that is implemented in the subclasses.

5. Write a Java program to create an interface that defines a set of methods for a Vehicle, and implement it in two classes, Car and Truck.

JAVA 8 FEATURES

- Java 8 introduced a range of new features that improved the language in several areas, including performance, functionality, and ease of use. In this chapter, we will explore some of the key features of Java 8 with an example.

1. Lambda Expressions: Lambda expressions are a new language feature introduced in Java 8 that provide a concise way to express functionality in code. They are used extensively in functional programming and enable the creation of functional interfaces.

- Example: Suppose you want to sort a list of names alphabetically using lambda expressions. You can use the Comparator interface and lambda expressions as follows:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
Collections.sort(names, (a, b) -> a.compareTo(b));
System.out.println(names);
```

- Here, we create a list of names and use the `Collections.sort()` method to sort the names alphabetically.
- We use a lambda expression (a, b) -> a.compareTo(b) to define the comparison logic.

- This lambda expression implements the Comparator interface's `compare()` method, which compares two strings and returns a negative integer, zero, or a positive integer depending on whether the first string is less than, equal to, or greater than the second string.

2. Stream API: The Stream API is another significant addition to Java 8. It provides a powerful and flexible way to process collections of data. The Stream API allows you to perform operations such as filtering, mapping, and reducing on collections in a concise and readable way.

- Example: Suppose you want to filter a list of numbers to find the even numbers and calculate their sum using the Stream API. You can use the filter() and mapToInt() methods as follows:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = numbers.stream()
                 .filter(n -> n % 2 == 0)
                 .mapToInt(Integer::intValue)
                 .sum();
System.out.println("Sum of even numbers: " + sum);
```

- Here, we create a list of numbers and use the stream() method to convert it into a Stream. We then use the filter() method to select only the even numbers, the mapToInt() method to convert the stream of integers to an IntStream, and the sum() method to calculate the sum of the even numbers.

3. Date and Time API: Java 8 also introduced a new Date and Time API that makes it easier to work with dates and times. The new API provides classes such as LocalDate, LocalTime, and LocalDateTime to represent dates and times and provides a range of methods for parsing, formatting, and calculating durations between dates.

- Example: Suppose you want to format the current date and time in a specific format using the Date and Time API. You can use the DateTimeFormatter class as follows:

```
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
String formattedDateTime = now.format(formatter);
System.out.println("Formatted date and time: " + formattedDateTime);
```

- Here, we create a LocalDateTime object representing the current date and time and use the ofPattern() method to create a DateTimeFormatter object with the desired format. We then use the format() method to format the date and time into a string using the DateTimeFormatter.

4. Default Methods: Default methods are a new feature in Java 8 that allow interface methods to have a default implementation. This feature allows interfaces to evolve without breaking the existing code that implements them.

- Example: Suppose you have an interface MyInterface with a method myMethod(). You can add a default implementation for this method in Java 8 as follows:

```
public interface MyInterface {
    void myMethod();
    default void myDefaultMethod() {
        System.out.println("This is a default method.");
    }
}
```

- Here, we have added a default implementation for the myDefaultMethod() method in the MyInterface interface. If a class implements this interface and does not provide its implementation for myDefaultMethod(), the default implementation will be used.

5. Functional Interfaces: Functional interfaces are a new addition to Java 8 that allow the use of lambda expressions to create objects of an interface type. A functional interface is an interface that has only one abstract method, called the functional method. It can have any number of default methods and static methods.

- Example: Suppose you have a functional interface Calculator with a method calculate(int a, int b) that takes two integers as input and returns an integer result. You can use a lambda expression to create an object of this interface as follows:

```java
@FunctionalInterface
public interface Calculator {
    int calculate(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        Calculator subtract = (a, b) -> a - b;
        Calculator multiply = (a, b) -> a * b;

        int result1 = add.calculate(10, 5);
        int result2 = subtract.calculate(10, 5);
        int result3 = multiply.calculate(10, 5);

        System.out.println("Addition result: " + result1);
        System.out.println("Subtraction result: " + result2);
        System.out.println("Multiplication result: " + result3);
    }
}
```

- Here, we define the Calculator functional interface with a single abstract method calculate(int a, int b). We then use lambda expressions to create three objects of this interface - add, subtract, and multiply. We call the calculate() method on each object with the input parameters 10 and 5 and print the result.

- The @FunctionalInterface annotation is optional but recommended to indicate that the interface is intended to be a functional interface.

6. Optional: The Optional class is a new addition to Java 8 that provides a way to handle null values. It can be used to avoid NullPointerExceptions and make the code more readable and maintainable.

- Example: Suppose you have a method getUserName() that returns a user's name or null if the name is not available. You can use the Optional class to handle the null value as follows:

```
public Optional<String> getUserName() {
    // Returns the user's name or null if the name is not available
}

Optional<String> userName = getUserName();
if (userName.isPresent()) {
    System.out.println("User name: " + userName.get());
} else {
    System.out.println("User name not available.");
}
```

- Here, we use the getUserName() method to get the user's name as an Optional<String> object. We then check if the object is present using the isPresent() method and print the user's name if it is available using the get() method. If the object is not present, we print a message indicating that the user's name is not available.

- Functional interfaces and lambda expressions have made Java 8 a more expressive and powerful language. They allow us to write concise and readable code that is easy to understand and maintain.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program that uses the Date and Time API to calculate the duration between two dates in years, months, and days.

2. Write a program that uses the Stream API to convert a list of strings to uppercase and sort them in alphabetical order.

3. Write a program that uses lambda expressions to sort a list of integers in ascending order.

4. Write a program that uses the Date and Time API to display the current date and time.

5. Write a program that uses the Optional class to handle null values when accessing a map.

6. Write a program that uses default methods in an interface to calculate the area of a rectangle.

7. Create a functional interface that takes a string as input and returns the length of the string as output.

INNER CLASSES

- Java provides a feature called inner classes that allows us to define a class inside another class. Inner classes can be used to logically group classes that are only used in one place, increase encapsulation, and improve code organization and readability.

1. Non-static Inner Classes: A non-static inner class, also known as an inner class, is a class that is defined inside another class without the static keyword. Non-static inner classes have access to the variables and methods of the outer class.

- Example:

```java
public class OuterClass {
    private int x = 10;

    class InnerClass {
        public void display() {
            System.out.println("Value of x is: " + x);
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

- Here, we define an outer class OuterClass with a private integer variable x and an inner class InnerClass that has a method display() that prints the value of x. We create an object of the outer class and use it to create an object of the inner class, then call the display() method on the inner object.

2. Static Inner Classes: A static inner class is a class that is defined inside another class with the static keyword. Static inner classes can access only the static members of the outer class.

- Example:

```java
public class OuterClass {
    private static int x = 10;

    static class InnerClass {
        public void display() {
            System.out.println("Value of x is: " + x);
        }
    }

    public static void main(String[] args) {
        OuterClass.InnerClass inner = new OuterClass.InnerClass();
        inner.display();
    }
}
```

- Here, we define an outer class OuterClass with a private static integer variable x and a static inner class InnerClass that has a method display() that prints the value of x. We create an object of the inner class and call the display() method on it.

3. Local Inner Classes: A local inner class is a class that is defined inside a method or block of code. Local inner classes can access the variables and parameters of the method or block of code, but only if they are declared final or effectively final.

- Example:

```java
public class OuterClass {
    public void display() {
        final int x = 10;

        class InnerClass {
            public void display() {
                System.out.println("Value of x is: " + x);
            }
        }

        InnerClass inner = new InnerClass();
        inner.display();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.display();
    }
}
```

- Here, we define an outer class OuterClass with a method display() that declares a final integer variable x and a local inner class InnerClass that has a method display() that prints the value of x. We create an object of the inner class and call the display() method on it inside the display() method of the outer class.

- Inner classes are a powerful feature of Java that can be used to improve code organization and encapsulation. In this chapter, we explored the three types of inner classes - non-static, static, and local - and demonstrated how they can be used in practice.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to create a static inner class that displays a message.

2. Write a Java program to create a nested class that displays a message.

3. Write a Java program to access the variables of the inner class from the outer class.

4. Write a Java program to create an anonymous inner class that displays a message.

5. Write a Java program to implement an interface in a non-static inner class.

6. Write a Java program to create a non-static inner class and access it from another class.

7. Define a non-static inner class named InnerClass inside a class named OuterClass that has a private integer variable x. Add a method getX() to the inner class that returns the value of x.

PACKAGES

- Java packages are a mechanism for organizing related classes and interfaces into a single unit of reuse. Packages provide a way to group related classes and interfaces together, making it easier to manage and maintain large Java applications. In this chapter, we will discuss the basics of packages in Java and how to create, import and use packages in your Java programs.

- Creating a package: To create a package in Java, you need to include the package statement at the beginning of your Java source file. The package statement is the first non-comment statement in the source file and specifies the name of the package in which the class belongs. For example, the following code snippet creates a package named "com.example.mypackage" and includes a class named "MyClass" in that package:

```java
package com.example.mypackage;

public class MyClass {
    // class implementation goes here
}
```

- In this example, the package statement indicates that the "MyClass" class belongs to the "com.example.mypackage" package.

- Package Naming Conventions: Java package names should follow a naming convention that helps to avoid naming conflicts between different packages. The recommended convention for package names is to use the reverse domain name of the organization or company that owns the package, followed by one or more additional identifiers that describe the package. For example, if your company's domain name is "example.com" and you are creating a package for a product named "myproduct", you might use the following package name:

```
1 package com.example.myproduct;
```

- Importing Packages: To use a class or interface from another package, you need to import that package into your Java program. The import statement should be placed after the package statement and before the class declaration. For example, the following code snippet imports the "java.util" package and uses the "ArrayList" class from that package:

```
1  package com.example.mypackage;
2
3  import java.util.ArrayList;
4
5  public class MyClass {
6      public static void main(String[] args) {
7          ArrayList<String> myList = new ArrayList<String>();
8          myList.add("item 1");
9          myList.add("item 2");
10         System.out.println(myList);
11     }
12 }
```

- In this example, the "java.util" package is imported using the import statement, and the "ArrayList" class is used to create a list of strings.

- Using Packages: Once you have created or imported a package, you can use the classes and interfaces defined in that package in your Java program. To use a class or interface from another package, you need to specify the fully qualified name of the class or interface, which includes the package name and the class or interface name. For example, the following code snippet uses the "ArrayList" class from the "java.util" package:

```
java.util.ArrayList<String> myList = new java.util.ArrayList<String>();
myList.add("item 1");
myList.add("item 2");
System.out.println(myList);
```

- In this example, the fully qualified name of the "ArrayList" class is used to create a list of strings.

- Packages are an essential feature of the Java programming language, providing a way to organize and reuse code. By following the naming conventions and best practices for packages, you can create maintainable, scalable, and robust Java applications

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Create a package named "com.example.utils" and add a class "StringUtils" to it with a static method that reverses a given string.

2. Create a package named "com.example.shapes" and add a class "Rectangle" to it that represents a rectangle with width and height properties.

3. Create a package named "com.example.math" and add a class "Calculator" to it that performs basic arithmetic operations.

4. Create a package named "com.example.search" and add a class "BinarySearch" to it that implements a binary search algorithm.

5. Create a package named "com.example.utilities" and add a class "RandomUtils" to it that generates random numbers and strings.

6. Create a package named "com.example.data" and add a class "Person" to it that represents a person with a name, age, and address.

MULTI-THREADING IN JAVA

- Java is a multi-threaded programming language, which means it can handle multiple threads of execution at the same time. Multi-threading is an important concept in Java, as it allows programmers to take advantage of multi-core processors and maximize the performance of their applications. In this chapter, we'll explore the basics of multi-threading in Java and look at some examples.

What is a Thread?

- A thread is a lightweight process that can execute independently and concurrently with other threads in a Java program. Threads share the same memory space, which allows them to communicate and coordinate their activities.

Creating Threads in Java:

- Java provides two ways to create threads:

1. Extending the Thread Class: To create a thread by extending the Thread class, you need to define a new class that extends the Thread class and override the run() method. The run() method contains the code that will be executed when the thread is started.

- Example:

```java
class MyThread extends Thread {
    public void run() {
        // code to be executed
    }
}

// Create a new instance of MyThread and start the thread
MyThread t = new MyThread();
t.start();
```

2. Implementing the Runnable interface: To create a thread by implementing the Runnable interface, you need to define a new class that implements the Runnable interface and override the run() method. The run() method contains the code that will be executed when the thread is started.

- Example:

```java
class MyRunnable implements Runnable {
    public void run() {
        // code to be executed
    }
}

// Create a new instance of MyRunnable and start the thread
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Thread Synchronization: Thread synchronization is the process of controlling the access to shared resources in a multi-threaded environment. In Java, synchronization is achieved using the synchronized keyword. When a method or a block of code is marked as synchronized, only one thread can execute that code at a time.

- Example:

```java
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

- In this example, the increment() and getCount() methods are marked as synchronized, which ensures that only one thread can access the count variable at a time.

Thread Pooling: Thread pooling is a technique that allows a group of threads to be created and managed together. Thread pooling is useful in situations where creating and destroying threads frequently can be costly.

```java
ExecutorService executor = Executors.newFixedThreadPool(10);

for (int i = 0; i < 100; i++) {
    Runnable worker = new MyRunnable();
    executor.execute(worker);
}

executor.shutdown();
while (!executor.isTerminated()) {
}

System.out.println("All threads have completed their tasks");
```

- In this example, we create a fixed thread pool of size 10 and submit 100 tasks to be executed by the threads in the pool. Once all the tasks have completed, we shutdown the thread pool.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program to create and start a new thread by extending the Thread class.

2. Write a program to create and start a new thread by implementing the Runnable interface.

3. Write a program to demonstrate thread synchronization using the synchronized keyword.

4. Write a program to demonstrate the join(), sleep(), and interrupt() method of the Thread class.

5. Write a program to demonstrate the wait(), notify() and notifyAll() methods of the Object class.

6. Write a program to avoid deadlock by using the tryLock() method of the ReentrantLock class.

7. Write a program to create a thread-safe singleton class.

8. Write a program to create a deadlock between two threads.

COLLECTIONS FRAMEWORK

- The Collection Framework in Java is a group of interfaces, classes, and algorithms that provide support for storing, retrieving, and manipulating a group of objects in Java. The collection framework has been designed to simplify the task of working with collections of objects, and it is a key feature of the Java programming language.

- The Collection Framework in Java includes the following components:

1. Interfaces: The Collection Framework provides a number of interfaces that define common behaviors for collections of objects. These interfaces include:

- List: A collection that maintains the order of its elements and allows duplicates.
- Set: A collection that does not allow duplicates.
- Queue: A collection used to hold elements prior to processing. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.
- Dequeue: A collection that allows insertion and removal of elements at both ends.
- Map: A collection that stores key-value pairs, with no duplicate keys allowed.

2. Classes: The Collection Framework provides a set of concrete classes that implement the interfaces. These classes include:

- ArrayList: A resizable array implementation of the List interface.
- LinkedList: A doubly linked list implementation of the List interface.
- HashSet: A set implementation that uses a hash table to store elements.
- TreeSet: A set implementation that uses a binary tree to store elements in sorted order.
- HashMap: A set implementation that uses a binary tree to store elements in sorted order.
- TreeMap: A map implementation that uses a binary tree to store key-value pairs in sorted order.

3. Algorithms: The Collection Framework also provides a set of algorithms, such as sorting and searching, that can be used with any collection that implements the appropriate interface.

- Example: Let's consider an example of how the Collection Framework can be used to sort a list of names in alphabetical order.

```java
import java.util.*;

public class NameSorter {
   public static void main(String[] args) {
      // Create a list of names
      List<String> names = new ArrayList<String>();
      names.add("John");
      names.add("Sarah");
      names.add("Alex");

      // Sort the list in alphabetical order
      Collections.sort(names);

      // Print the sorted list
      for (String name : names) {
         System.out.println(name);
      }
   }
}
```

- In the above example, we create an ArrayList to store a list of names. We then use the Collections.sort() method to sort the list in alphabetical order. Finally, we loop through the sorted list and print out each name.

- The Collection Framework provides a powerful and flexible set of tools for working with collections of objects. Whether you're working with lists, sets, maps, or other types of collections, the Collection Framework provides a consistent and intuitive interface for accessing and manipulating the data.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to sort an ArrayList of integers using the Collections Framework.

2. Write a Java program to remove duplicates from an ArrayList of strings using the Collections Framework.

3. Write a Java program to create a priority queue of integers and add some elements to it.

4. Write a Java program to find the keys and values of a HashMap of strings and integers using the Collections Framework.

5. Write a Java program to copy one ArrayList of integers to another using the Collections Framework.

6. Write a Java program to merge two ArrayLists of strings using the Collections Framework.

7. Write a Java program to create a TreeMap of integers and strings and add some key-value pairs to it.

JAVA GENERICS

- Generics are used to make Java code more type-safe and to avoid runtime errors caused by type casting. Java generics were introduced in Java 5 and have since become an integral part of Java programming.

- Generics provide a way to create a single method, class, or interface that can work with different types of objects, without requiring the programmer to explicitly cast the objects to the correct type. By using generics, we can write code that is more concise, readable, and maintainable.

Syntax of Generics: The syntax of generics involves the use of angle brackets (<>) and a type parameter. The type parameter specifies the type of object that the class, method, or interface can work with.

- For Example, consider the following generic class:

```
public class Box<T> {
    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

- In the above example, we have created a generic class called Box, which can work with any type of object. The type parameter T specifies the type of object that the class can work with.

- Let's see how we can use this class:

```
Box<Integer> integerBox = new Box<Integer>();
Box<String> stringBox = new Box<String>();

integerBox.set(10);
stringBox.set("Hello");

int intValue = integerBox.get();
String stringValue = stringBox.get();
```

- In the above example, we have created two instances of the Box class - one that works with integers and another that works with strings. We have used the set() method to set the value of the object and the get() method to retrieve the value.

- Here are some more concepts related to Java Generics:

- Wildcards in Generics: The wildcard '?' in Java Generics represents an unknown type. It is used to write generic code that can work with any type. The '?' can be used as an upper bound or a lower bound to restrict the type of objects that can be used.

- Bounded Type Parameters: Java Generics allow us to define bounded type parameters that restrict the type of objects that can be used with a generic class or method. Bounded type parameters can be used as upper bounds or lower bounds.

- Type Erasure: Java Generics use type erasure to ensure backward compatibility with pre-existing code that does not use generics. Type erasure replaces the type parameter with its upper bound, which means that the type information is lost at runtime.

- Generic Methods: Java Generics also allow us to define generic methods that work with any data type. Generic methods can be defined in generic and non-generic classes.

- Example: Consider the following code snippet that demonstrates the use of wildcards and bounded type parameters in Java Generics:

```java
import java.util.*;

public class MyGenericsClass<T extends Number> {

    private List<T> list;

    public MyGenericsClass(List<T> list) {
        this.list = list;
    }

    public double getAverage() {
        double sum = 0.0;
        for (T n : list) {
            sum += n.doubleValue();
        }
        return sum / list.size();
    }

    public void printList(List<?> list) {
        for (Object obj : list) {
            System.out.println(obj);
        }
    }

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
        MyGenericsClass<Integer> intObj = new MyGenericsClass<Integer>(intList);
        System.out.println("Average of integers: " + intObj.getAverage());

        List<Double> doubleList = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
        MyGenericsClass<Double> doubleObj = new MyGenericsClass<Double>(doubleList);
        System.out.println("Average of doubles: " + doubleObj.getAverage());

        List<String> stringList = Arrays.asList("one", "two", "three", "four", "five");
        intObj.printList(stringList);
    }
}
```

- In the above example, we have defined a generic class 'MyGenericsClass' with a bounded type parameter 'T extends Number'. The class has a constructor that accepts a list of type 'T'. The 'getAverage' method calculates the average of the numbers in the list, and the 'printList' method prints the elements of a list of any type using the wildcard '?'. We have created instances of the 'MyGenericsClass' class with Integer and Double data types and passed them to the 'getAverage' method. Finally, we have called the 'printList' method with a list of strings, which will not compile due to type safety.

- Java Generics provide a powerful way to write generic code that is type-safe, reusable, and maintainable. By using Generics, we can write code that works with any data type and can be reused in different scenarios. The use of wildcards, bounded type parameters, and generic methods adds flexibility to the generic code and makes it more powerful.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program to create a generic method that can swap two elements of an array.

2. Write a program to create a generic method that can print the elements of a list of any type.

3. Write a program to create a generic interface that can be implemented by any class.

4. Write a program to create a generic class that can implement a binary search tree of any type.

5. Write a program to create a generic method that can concatenate two lists of any type.

6. Write a program to create a generic method that can return the median of an array of any type.

7. Write a program to create a generic method that can remove duplicates from a list of any type.

8. Write a program to create a generic method that can sort a list of any type.

EXCEPTION HANDLING

- Exception handling is the process of handling the runtime errors that may occur while executing a program. These errors can be caused by various reasons, such as user input, network issues, or bugs in the code. To handle these errors effectively, Java provides a set of keywords and constructs that help us to identify and handle exceptions.

Types of Exceptions:

- In Java, there are two types of exceptions: checked and unchecked exceptions.

1. Checked Exception: These are exceptions that the compiler checks during compilation. The programmer must handle these exceptions either by using try-catch blocks or by declaring them in the method signature using the throws keyword. Examples of checked exceptions include IOException, ClassNotFoundException, and SQLException.

2. Unchecked Exception: These are exceptions that the compiler does not check during compilation. They are thrown at runtime and can be handled by using try-catch blocks. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBound--Exception and ClassCastException.

Try-Catch Blocks: The try-catch block is used to handle exceptions in Java. The try block contains the code that may throw an exception, and the catch block contains the code that handles the exception. If an exception is thrown in the try block, the catch block is executed, and the exception is handled.

- Syntax:

```
try {
   // code that may throw an exception
}
catch(Exception e) {
   // code to handle the exception
}
```

- Example:

```
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      try {
         int a = 5 / 0;
      }
      catch(ArithmeticException e) {
         System.out.println("Exception occurred: " + e);
      }
   }
}
```

- In this example, we have a try block that contains the code that may throw an ArithmeticException. If an exception occurs, the catch block is executed, and the exception is handled. In this case, we are printing the exception message to the console.

Throwing Exceptions: In Java, we can also throw our own exceptions using the throw keyword. This is useful when we want to indicate that an error has occurred in our program.

- Syntax:

```
throw new ExceptionType("Exception message");
```

- Example:

```
public class CustomExceptionExample {
   public static void main(String[] args) {
      try {
         throw new CustomException("Custom exception occurred");
      }
      catch(CustomException e) {
         System.out.println("Exception occurred: " + e.getMessage());
      }
   }
}

class CustomException extends Exception {
   public CustomException(String message) {
      super(message);
   }
}
```

- In this example, we have defined our own custom exception class called CustomException. We are throwing this exception in the main method and catching it in the catch block. When we catch the exception, we print the exception message to the console.

- Exception handling is an essential feature in Java that allows us to handle and manage errors effectively. By using try-catch blocks and throwing exceptions, we can write code that is more robust and reliable.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program to handle the ArithmeticExceptio--n that can be thrown when dividing two numbers.

2. Write a program that reads input from the user and handles the InputMismatchException that can be thrown if the input is of the wrong type.

3. Write a program that handles the ClassNotFound--Exception that can be thrown when a class is not found.

4. Write a program that handles the NoSuchElementException that can be thrown when trying to access an element of an empty collection.

5. Write a program that handles the ClassCastExcep--tion that can be thrown when trying to cast an object to a class that it is not an instance of.

6. Write a program that handles the IllegalArgumentException that can be thrown when an illegal argument is passed to a method.

I/O & FILE HANDLING

- In Java, input/output (I/O) operations and file handling are essential features that allow us to interact with the external world. The java.io package provides classes and interfaces for performing I/O operations in Java.
- This chapter will cover the following topics:
  1. Overview of I/O and File Handling in Java
  2. Reading and Writing to Files
  3. Reading and Writing to Text Files
  4. Reading and Writing to Binary Files
  5. Buffered I/O
  6. Serialization
  7. Compression and Decompression

1. Overview of I/O and File Handling in Java:

- In Java, the I/O operations can be categorized into two types: character-based I/O and byte-based I/O. The character-based I/O is used to read and write text data, whereas byte-based I/O is used to read and write binary data.
- The java.io package contains a variety of classes that can be used for I/O operations. The most commonly used classes include InputStream, OutputStream, Reader, and Writer.

- These classes are the abstract base classes for all input and output streams in Java.

2. Reading and Writing to Files:

- In Java, files can be read and written using the FileReader and FileWriter classes, respectively. The FileReader and FileWriter classes are used to read and write character-based data to a file.

- Example:

```java
import java.io.*;

public class FileReadWrite {
   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

- In the above example, we are using FileReader to read data from the file "input.txt" and FileWriter to write data to the file "output.txt".

3. Reading and Writing to Text Files:

- The Java BufferedReader and BufferedWriter classes are used to read and write text files.

- The BufferedReader class is used to read character-based data from a file, and the BufferedWriter class is used to write character-based data to a file.

- Example:

```java
import java.io.*;

public class TextFileReadWrite {
    public static void main(String args[]) throws IOException {
        BufferedReader in = null;
        BufferedWriter out = null;

        try {
            in = new BufferedReader(new FileReader("input.txt"));
            out = new BufferedWriter(new FileWriter("output.txt"));

            String line;
            while ((line = in.readLine()) != null) {
                out.write(line);
                out.newLine();
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

- In the above example, we are using BufferedReader to read data from the file "input.txt" and BufferedWriter to write data to the file "output.txt". The readLine() method of BufferedReader reads a line of text from the file, and the newLine() method of BufferedWriter inserts a new line character after each line.

4. Reading and Writing to Binary Files:

- Reading and writing binary files in Java involves reading and writing data in binary format. This is different from reading and writing text files, which are stored in human-readable format.

- Here's an example that demonstrates how to read and write binary data using Java's I/O classes:

```java
import java.io.*;

public class BinaryFileExample {
    public static void main(String[] args) throws IOException {
        // Writing to a binary file using FileOutputStream and DataOutputStream
        FileOutputStream fileOutputStream = new FileOutputStream("output.bin");
        DataOutputStream dataOutputStream = new DataOutputStream(fileOutputStream);
        int intValue = 42;
        double doubleValue = 3.14159;
        String stringValue = "Hello, world!";
        dataOutputStream.writeInt(intValue);
        dataOutputStream.writeDouble(doubleValue);
        dataOutputStream.writeUTF(stringValue);
        dataOutputStream.close();

        // Reading from a binary file using FileInputStream and DataInputStream
        FileInputStream fileInputStream = new FileInputStream("output.bin");
        DataInputStream dataInputStream = new DataInputStream(fileInputStream);
        int receivedIntValue = dataInputStream.readInt();
        double receivedDoubleValue = dataInputStream.readDouble();
        String receivedStringValue = dataInputStream.readUTF();
        dataInputStream.close();

        // Print the values read from the binary file
        System.out.println("Received int value: " + receivedIntValue);
        System.out.println("Received double value: " + receivedDoubleValue);
        System.out.println("Received string value: " + receivedStringValue);
    }
}
```

- In this example, we create a binary file using FileOutputStream and DataOutputStream, and write three values to it: an int, a double, and a String. We then close the DataOutputStream, and open the file again using FileInputStream and DataInputStream. We use the DataInputStream to read the values back from the file in the same order as they were written, and store them in variables. Finally, we print out the values that were read from the file.

5. Buffered I/O:

- Buffered streams in Java allow for efficient reading and writing of data. They use an internal buffer to store data, so that multiple I/O operations can be combined into a single operation.

- Here's an example that demonstrates how to read and write data using buffered streams:

```java
import java.io.*;

public class BufferedStreamExample {
    public static void main(String[] args) throws IOException {
        // Writing to a file using buffered output stream
        FileOutputStream fileOutputStream = new FileOutputStream("output.txt");
        BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(fileOutputStream);
        String message = "Hello, world!";
        bufferedOutputStream.write(message.getBytes());
        bufferedOutputStream.close();

        // Reading from a file using buffered input stream
        FileInputStream fileInputStream = new FileInputStream("output.txt");
        BufferedInputStream bufferedInputStream = new
BufferedInputStream(fileInputStream);
        byte[] buffer = new byte[1024];
        int bytesRead = bufferedInputStream.read(buffer);
        String receivedMessage = new String(buffer, 0, bytesRead);
        System.out.println("Received message: " + receivedMessage);
        bufferedInputStream.close();
    }
}
```

- In this example, we create a BufferedOutputStream to write data to a file, and a BufferedInputStream to read data from the same file. We write a message to the output stream, and then read the message back from the input stream. Finally, we print the received message to the console.

6. Serialization:

- Serialization is a process in which an object in Java is converted into a stream of bytes so that it can be saved to a file or transmitted over a network. It is used to store and retrieve the state of an object. The process of serialization involves converting the object into a byte stream, and then deserialization is the reverse process of converting the byte stream back into the object.

- Serialization can be used in various scenarios, such as caching and sharing objects across the network. However, it is important to note that not all objects can be serialized, such as objects that have references to non-serializable objects. Additionally, the serialized data is platform-independent, meaning that it can be deserialized on any platform that supports Java.

- In Java, serialization is supported by the Serializable interface, which is a marker interface that indicates that the class can be serialized. To serialize an object, we need to implement the Serializable interface and override the default serialization behavior using the readObject() and writeObject() methods.

- Here is an example:

```java
import java.io.*;

public class SerializationDemo {
    public static void main(String[] args) {
        // create an object of Student class
        Student student = new Student("John Doe", "Java");

        // serialize the object
        try {
            FileOutputStream fileOut = new FileOutputStream("student.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(student);
            out.close();
            fileOut.close();
            System.out.println("Object serialized successfully");
        } catch (IOException i) {
            i.printStackTrace();
        }

        // deserialize the object
        Student deserializedStudent = null;
        try {
            FileInputStream fileIn = new FileInputStream("student.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            deserializedStudent = (Student) in.readObject();
            in.close();
            fileIn.close();
            System.out.println("Object deserialized successfully");
        } catch (IOException i) {
            i.printStackTrace();
        } catch (ClassNotFoundException c) {
            System.out.println("Student class not found");
            c.printStackTrace();
        }

        // print the deserialized object
        System.out.println("Deserialized Student:");
        System.out.println("Name: " + deserializedStudent.getName());
        System.out.println("Course: " + deserializedStudent.getCourse());
    }
}

class Student implements Serializable {
    private String name;
    private String course;

    public Student(String name, String course) {
        this.name = name;
        this.course = course;
    }

    public String getName() {
        return name;
    }

    public String getCourse() {
        return course;
    }
}
```

- In the above example, we have a Student class that implements the Serializable interface. We create an object of the Student class and serialize it to a file named "student.ser" using the ObjectOutputStream class. We then deserialize the object using the ObjectInputStream class and print the deserialized object's properties.

7. Compression and Decompression:

- In Java, the I/O and File Handling API provides a way to compress and decompress data. This can be particularly useful when dealing with large amounts of data, as compression can significantly reduce the amount of disk space or network bandwidth required to store or transmit the data.

- Java provides two main classes for compression and decompression: java.util.zip.ZipOutputStream and java.util.zip.ZipInputStream. These classes provide a simple way to compress and decompress data using the popular ZIP file format.

To compress data, you create an instance of ZipOutputStream and use it to write data to a new ZIP file. Here's an example that compresses a file:

```java
import java.io.*;
import java.util.zip.*;

public class ZipExample {
    public static void main(String[] args) {
        String inputFileName = "input.txt";
        String outputFileName = "output.zip";

        try {
            FileInputStream inputStream = new FileInputStream(inputFileName);
            FileOutputStream outputStream = new FileOutputStream(outputFileName);
            ZipOutputStream zipOutputStream = new ZipOutputStream(outputStream);
            zipOutputStream.putNextEntry(new ZipEntry(inputFileName));

            byte[] buffer = new byte[1024];
            int length;
            while ((length = inputStream.read(buffer)) > 0) {
                zipOutputStream.write(buffer, 0, length);
            }

            zipOutputStream.closeEntry();
            zipOutputStream.close();
            inputStream.close();

            System.out.println("File compressed successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- In this example, we create a new ZipOutputStream object and add a new entry to it using the putNextEntry method. We then read the contents of the input file using a FileInputStream and write the data to the ZIP file using the write method of the ZipOutputStream. Finally, we close the entry using the closeEntry method and close the ZipOutputStream and FileInputStream.

- To decompress data, you create an instance of ZipInputStream and use it to read data from a ZIP file. Here's an example that decompresses a file:

```java
import java.io.*;
import java.util.zip.*;

public class UnzipExample {
    public static void main(String[] args) {
        String inputFileName = "input.zip";
        String outputFileName = "output.txt";

        try {
            FileInputStream inputStream = new FileInputStream(inputFileName);
            ZipInputStream zipInputStream = new ZipInputStream(inputStream);
            ZipEntry entry = zipInputStream.getNextEntry();

            FileOutputStream outputStream = new FileOutputStream(outputFileName);
            byte[] buffer = new byte[1024];
            int length;
            while ((length = zipInputStream.read(buffer)) > 0) {
                outputStream.write(buffer, 0, length);
            }

            outputStream.close();
            zipInputStream.closeEntry();
            zipInputStream.close();

            System.out.println("File decompressed successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- In this example, we create a new ZipInputStream object and use it to read the contents of the input ZIP file. We then create a new FileOutputStream object and write the data to a new file using the write method. Finally, we close the ZipInputStream and FileOutputStream.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a program to read a file line by line and print its content to the console.

2. Write a program to read a binary file and display its content in hexadecimal format.

3. Write a program to create a new directory and file inside it.

4. Write a program to create a zip file and add multiple files to it.

5. Write a program to extract the content of a zip file.

6. Write a program to write data to a file using FileWriter and read data from a file using FileReader.

7. Write a program to copy a directory and its content to a new location.

8. Write a program to serialize and deserialize an object to a file.

CONNECTING TO DATABASE (JDBC)

- Java Database Connectivity (JDBC) is an API for Java that enables Java programs to interact with relational databases like MySQL, Oracle, and SQL Server. It provides a standard interface for Java programs to communicate with databases and execute SQL queries.

JDBC Architecture:

- The JDBC architecture consists of two main layers:

1. JDBC API: The JDBC API provides a set of classes and interfaces that allow Java programs to interact with databases. Some of the important interfaces in JDBC API are:

i. Connection: It represents a connection to a database.

ii. Statement: It is used to execute SQL queries.

iii. PreparedStatement: It is used to execute parameterized SQL queries.

iv.. ResultSet: It represents the result set of a SQL query.

2. JDBC Driver: A JDBC driver is a software component that allows Java programs to connect to a database. There are four types of JDBC drivers:

a. JDBC-ODBC Bridge Driver: This driver acts as a bridge between the JDBC API and the ODBC API. It translates JDBC calls into ODBC calls, which are then sent to the database. This driver requires an ODBC driver to be installed on the system.

b. Native API Driver: This driver communicates directly with the database system using the vendor's native API. It requires the vendor-specific libraries to be installed on the system where the application is running.

c. Network Protocol Driver: This driver uses a middleware to communicate with the database system. It sends requests to the middleware, which then communicates with the database using the appropriate protocol. This driver is useful when the application needs to access databases on remote systems.

d. Thin Driver: This driver is similar to the network protocol driver, but it communicates with the database system using a lightweight protocol. It does not require any middleware to be installed on the system, making it easier to deploy and use.

- Choosing the right JDBC driver depends on the specific requirements of the application.

- If the application needs to access a local database, then a Native API Driver or a JDBC-ODBC Bridge Driver may be the best choice. If the database is located on a remote system, then a Network Protocol Driver or a Thin Driver may be more suitable.

- Let's understand with an example, we will establish a connection to a MySQL database and execute a simple SQL query to retrieve data from a table.

1. First, we need to load the MySQL JDBC driver:

```
Class.forName("com.mysql.jdbc.Driver");
```

2. Next, we need to establish a connection to the database:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "username", "password");
```

3. Once we have a connection, we can execute SQL queries using Statement or PreparedStatement:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
```

4. Finally, we can process the result set:

```
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println(id + ", " + name + ", " + age);
}
```

- This is a basic example of how to use JDBC to interact with a database in Java.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Write a Java program to perform CRUD operations into a MySQL database using JDBC.

2. Write a Java program to create/drop a table in/from a MySQL database using JDBC.

3. Write a Java program to execute a SQL query with parameters in a MySQL database using JDBC.

4. Write a Java program to execute a SQL query with a where clause in a MySQL database using JDBC.

5. Write a Java program to execute a SQL query with a join in a MySQL database using JDBC.

6. Write a Java program to execute a SQL query with a group by clause in a MySQL database using JDBC.

7. Write a Java program to execute a SQL query with an order by clause in a MySQL database using JDBC.

8. Write a Java program to execute a SQL query with a like clause in a MySQL database using JDBC.

GUI PROGRAMMING

- Graphical User Interface (GUI) is a type of user interface that allows users to interact with programs or software visually. Java provides a rich set of libraries and APIs for creating GUI-based applications. In this chapter, we will discuss how to create GUI applications in Java using Swing and AWT libraries.

Introduction to AWT:

- AWT stands for Abstract Window Toolkit.
- AWT was introduced with the initial release of Java, and it provides the basic set of tools for creating a graphical user interface.
- AWT uses the native platform's widgets and components to create the interface.
- Let's know what do each interface and class of AWT toolkit perform:

1. **Component**: The base class for all AWT components that can be added to a container.
2. **Container**: A component that can hold other components.
3. **LayoutManager**: An interface for objects that manage the layout of components within a container.
4. **Event**: A base class for all AWT events.

5. **EventQueue**: A queue that manages AWT events.
6. **Window**: A top-level container that represents a window on the screen.
7. **Dialog**: A window that is designed to be used as a dialog box.
8. **Frame**: A window with a title bar and other decorations.
9. **Menu**: A menu bar or pop-up menu.
10. **MenuBar**: A container for menus that can be added to a window.
11. **MenuItem**: An item in a menu.
12. **PopupMenu**: A pop-up menu that can be attached to a component.
13. **Button**: A component that represents a button.
14. **Label**: A component that displays a text label.
15. **TextField**: A component that allows the user to enter text.
16. **TextArea**: A component that displays multiple lines of text.
17. **Checkbox**: A component that represents a checkbox.
18. **List**: A component that displays a list of items.
19. **Scrollbar**: A component that provides a scrollbar for a container.
20. **Canvas**: A component that provides a drawing surface.

- Let's Create a GUI with AWT: To create a GUI with AWT, we need to create instances of various classes such as Frame, Button, Label, TextField, etc. Here is an example of how to create a simple GUI with AWT:

```java
import java.awt.*;

public class MyGUI {
    public static void main(String[] args) {
        Frame frame = new Frame("My GUI");
        Button button = new Button("Click Me");
        Label label = new Label("Welcome to My GUI");
        TextField textField = new TextField("Enter your name");

        frame.add(label);
        frame.add(textField);
        frame.add(button);

        frame.setLayout(new FlowLayout());
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

- Here, we have created a Frame object that represents the main window of the application. We have also created a Button, a Label, and a TextField. Then, we have added these components to the Frame using the add() method. Finally, we have set the layout of the Frame to FlowLayout and set its size and visibility.

Introduction to Swing:

- Swing was introduced later, and it provides a complete set of tools to create a more sophisticated and flexible interface.
- Swing is built on top of AWT, and it uses its own set of components, which makes it platform-independent.

- Let's know what do each interface and class of AWT toolkit perform:

1. **JFrame** - A top-level container that represents the main window of a GUI application.
2. **JPanel** - A container that holds other Swing components.
3. **JButton** - A component that represents a clickable button.
4. **JLabel** - A component that displays text or an image.
5. **JTextField** - A component that allows the user to input a single line of text.
6. **JPasswordField** - A component that allows the user to input a password.
7. **JComboBox** - A component that allows the user to select an item from a list.
8. **JList** - A component that displays a list of items.
9. **JTable** - A component that displays data in a tabular format.
10. **JScrollPane** - A container that provides scrolling capabilities for a component that is too large to fit in a container.
11. **ActionListener** - An interface that defines a method for handling action events, such as button clicks.

12. **MouseListener** - An interface that defines methods for handling mouse events, such as mouse clicks and mouse movement.

13. **KeyListener** - An interface that defines methods for handling keyboard events, such as key presses and releases.

14. **SwingUtilities** - A utility class that provides methods for working with Swing components on the event dispatch thread.

15. **UIManager** - A class that provides access to the look and feel of a Swing application.

- Let's create a GUI with Swing: To create a GUI with Swing, we need to create instances of various classes such as JFrame, JButton, JLabel, JTextField, etc. Here is an example of how to create a simple GUI with Swing:

```java
import javax.swing.*;

public class MyGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My GUI");
        JButton button = new JButton("Click Me");
        JLabel label = new JLabel("Welcome to My GUI");
        JTextField textField = new JTextField("Enter your name");

        frame.add(label);
        frame.add(textField);
        frame.add(button);

        frame.setLayout(new FlowLayout());
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

- Here, we have created a JFrame object that represents the main window of the application. We have also created a JButton, a JLabel, and a JTextField.

- Then, we have added these components to the JFrame using the add() method. Finally, we have set the layout of the JFrame to FlowLayout and set its size and visibility.

Difference between AWT and Swing:

- AWT

  - AWT is the oldest GUI library introduced by Java.

  - AWT is based on native components, which means that it uses the platform's native components to create a GUI. This makes it more efficient in terms of performance, but it also means that the look and feel of the GUI can vary between platforms.

  - AWT components are heavyweight.

  - AWT is more efficient in terms of performance.

- Swing

  - Swing is a more advanced and newer GUI library.

  - Swing is based on a set of Java classes and provides a consistent look and feel across all platforms.

  - This is because the components are created using pure Java code, rather than relying on the platform's native components

  - Swing components are lightweight and offers a more consistent look and feel and greater flexibility in creating a GUI.

Event Handling in Swing and AWT

- Event handling is an important part of GUI programming in Java. Events are user actions such as button clicks, mouse movements, and key presses. In Swing and AWT, events are handled using event listeners. An event listener is an object that is notified when an event occurs.

- Here's an example of how to add an event listener to a button in Swing:

```
JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // code to handle button click event
    }
});
```

- In this example, an anonymous inner class is used to define an ActionListener object and pass it to the addActionListener() method of the button. The actionPerformed() method of the ActionListener object will be called when the button is clicked.

Layout Managers in Swing and AWT

- Layout managers are used to arrange the components in a GUI. Swing and AWT provide several layout managers that can be used to arrange components in different ways. Some of the commonly used layout managers are:

- BorderLayout: Arranges the components in the north, south, east, west, and center regions of the container.
- FlowLayout: Arranges the components in a row, wrapping them to the next row when the container's width is exceeded.
- GridLayout: Arranges the components in a grid of rows and columns.
- BoxLayout: Arranges the components in a row or column.

- Here's an example of how to use the BorderLayout manager to arrange components in a frame:

```
JFrame frame = new JFrame("BorderLayout Example");
frame.setLayout(new BorderLayout());

JButton button1 = new JButton("North");
JButton button2 = new JButton("South");
JButton button3 = new JButton("East");
JButton button4 = new JButton("West");
JButton button5 = new JButton("Center");

frame.add(button1, BorderLayout.NORTH);
frame.add(button2, BorderLayout.SOUTH);
frame.add(button3, BorderLayout.EAST);
frame.add(button4, BorderLayout.WEST);
frame.add(button5, BorderLayout.CENTER);

frame.setSize(300, 300);
frame.setVisible(true);
```

- In this example, the buttons are added to different regions of the frame using the add() method and specifying the region using the constants provided by BorderLayout.

- **HANDS-ON ASSIGNMENTS ON THIS TOPIC:**

1. Create a swing application to display a list of customers with their details such as name, address, and phone number. Allow the user to add, edit, and delete customers.

2. Create a swing application to display username textfield and password field with a Login Button.

3. Create a swing application that allows the user to enter a time in hours, minutes, and seconds, and then display the time in 24-hour format.

4. Create a swing application that allows the user to enter a sentence and then display the sentence with each word reversed.

5. Create a swing application that allows the user to enter a number and then display the binary, octal, and hexadecimal representations of that number.

6. Create a swing application that allows the user to enter a number and then display the factorial of that number.

JAVA PROJECTS 

1. Employee Management System: Create a system that allows the user to add, edit, and delete employees, and search for employees by name, department, or position using Collection Framework

2. Currency Convertor: Create a program that can convert between different currencies using exchange rates

3. Contact Management System: Create a system that allows the user to add, edit, and delete contacts, and search for contacts by name using JDBC.

4. Quiz Game: Create a multiple-choice quiz game where the user has to select the correct answer from a set of options.


scanner.use
ale(LOCALE);
catch (IOException ioe) {
err.println("Could not