Object Oriented Programming with JAVA

Semester 3

Course Code : BCS306A

Module-1:

An Overview of Java: Object-Oriented Programming (Two Paradigms, Abstraction, The Three OOP Principles), Using Blocks of Code, Lexical Issues (Whitespace, Identifiers, Literals, Comments, Separators, The Java Keywords).

Data Types, Variables, and Arrays: The Primitive Types (Integers, Floating-Point Types, Characters, Booleans), Variables, Type Conversion and Casting, Automatic Type Promotion in Expressions, Arrays, Introducing Type Inference with Local Variables.

Operators: Arithmetic Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The ? Operator, Operator Precedence, Using Parentheses.

Control Statements: Java's Selection Statements (if, The Traditional switch), Iteration Statements (while, do-while, for, The For-Each Version of the for Loop, Local Variable Type Inference in a for Loop, Nested Loops), Jump Statements (Using break, Using continue, return).

OOPs Principles: Encapsulation, Inheritance and Polymorphism are the basic principles of any object oriented programming language.

Encapsulation:

- It is a mechanism to bind the data and code working on that data into a single entity.
- In Java, encapsulation is achieved using classes. A class is a collection of data and code. An object is an instance of a class.
- It provides the security for the data by avoiding outside manipulations.

Inheritance:

- It allows us to have code re-usability. It is a process by which one object can acquire the properties of another object.

Polymorphism:

- It can be thought of as one interface, multiple methods. It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature

Lexical Issues: Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. We will discuss the significance of each of these here.

Whitespace :

- In Java, whitespace is a space, tab or newline. Usually, a space is used to separate tokens; tab and newline are used for indentation.

Identifiers:

- Identifiers are used for class names, method names, and variable names
- An identifier may be any sequence of uppercase and lowercase letters, numbers, or the underscore and dollar- sign characters. They must not begin with a number..
- As Java is case-sensitive, Avg is a different identifier than avg.

Literals :

- A constant value in Java is created by using a literal representation of it. For example, 25 (an integer literal), 4.5 (a floating point value), ‘p’ (a character constant, “Hello World” (a string value).

Comments :

- There are three types of comments defined by Java.
- Two of these are well-know viz. single-line comment (starting with //), multiline comment (enclosed within /* and */). The third type of comment viz. documentation comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends
- The documentation comment begins with a /** and ends with a */.

Separators :

- In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon which is used to terminate statements.

Primitive datatypes:

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and Boolean.

- The primitive types are also commonly referred to as simple types.
- These can be put in 4 groups:
  - **Integers**: This group includes byte, short, int, and long, which are for whole-valued signed numbers.
  - **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
  - **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
  - **Boolean:** This group includes Boolean, which is a special type for representing true/false values.

Integers:

Java defines four integer types: byte, short, int, and long.

- The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type.
- The **Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.**

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Byte::

The smallest integer type is **byte**.

- This is a signed 8-bit type that has a range **from –128 to 127.**
- Variables of type **byte** are especially **useful when you're working with a stream of data from a network or file.**
- Byte variables are declared by use of the **byte** keyword.

For example : The following declares two byte variables called b and c:

byte b, c ;

Short:

short is a signed 16-bit type.

- It has a range from –32,768 to 32,767.
- It is probably the least used Java type.
- Here are some examples of short variable declarations:

short s;

short t;

Int:

The most commonly used integer type is **int**.

- It is a signed 32-bit type that has a range from **–2,147,483,648 to 2,147,483,647.**
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
- Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case.
- The reason is that when byte and short values are used in an expression, they are promoted to int when the expression is evaluated.

Therefore, int is often the best choice when an integer is needed

long:

It is a signed **64-bit type** and is useful for those occasions where an int type is not large enough to hold the desired value.

- The range of a long is quite **large**.This makes it **useful when big, whole numbers are needed**.
- **For example,** here is **a program that computes the number of miles that light will travel in a specified number of days:**

```
// Compute distance light travels using long variables.
class Light {
  public static void main(String[] args) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

Output:

This program generates the following output:

- In 1000 days light will travel about 16070400000000 miles.

```
In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an int variable.

Floating Point Types:

Floating-point numbers, also known as **real numbers**, are used when evaluating expressions that require fractional precision. For example, calculations such as **square root**, or transcendental such as **sine** and **cosine**, **result in a value whose precision requires a floating-point type.**

- There are two kinds of floating-point types, **float** and **double**, which represent **single and double- precision numbers**, **respectively**.

float : The type float specifies a single-precision value that **uses 32 bits of storage.**

For example, float can be **useful** when representing **dollars** and **cents**. Here are some example for float variable declarations:

- **float hightemp, lowtemp;**

double: Double precision, as denoted by the **double** keyword, uses **64 bits to store a value.**

Double precision is actually **faster than single precision** on some modern processors that have been **optimized for high-speed mathematical calculations**.

```
// Compute the area of a circle.
class Area {
  public static void main(String[] args) {
    double pi, r, a;

    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

    System.out.println("Area of circle is " + a);
  }
}
```

In Java, the data type used to store characters is **char**. **char** in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java**.**

- **Java uses Unicode to represent characters. (**Unicode defines a fully international character set that can represent all of the characters found in all human languages.)
- • In Java char is a **16-bit type.** The range of a **char is 0 to 65,536**. There are **no negative chars**.

Here is a program that demonstrates char variables:

ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence**.**

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String[] args) {
    char ch1, ch2;

    ch1 = 88; // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

- Java has a primitive type, called **boolean**, for logical values.
- It can have only one of two possible values, **true or false.**
- This is the type returned by all relational operators, as in the case of **a < b**.
- **boolean** is also the type required by the **conditional expressions** that govern the control statements such as if and for.
- **Here is a program that demonstrates the boolean type:**

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String[] args) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

     // outcome of a relational operator is a boolean value
     System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

First, when a **boolean** value is output by **println( )**, "true" or "false" is displayed.

- Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement.

  There is no need to write an if statement like this: if(b == true)

- Third, the outcome of a relational operator, such as 9 displays the value "true." Further, the extra set of parentheses around **10>9** is necessary because the + operator has a higher precedence than the >.

Variables:

- Basic unit of storage in a Java program.
- A variable is **defined** by the **combination of an identifier, a type, and an optional initializer.**

It is a **quantity whose value can be changed during program execution.**

- A variable name may consists of **alphabets**, **digits**

  or **underscore**.

Declaring a Variable:

The basic form of a variable declaration is shown here:

type identifier [= *value*][, *identifier* [= *value*] ...];

Example of variable declaration

```
int a, b, c;            // declares three ints, a, b, and c.
int d = 3, e, f = 5;    // declares three more ints, initializing
                        // d and f.
byte z = 22;            // initializes z.
double pi = 3.14159;    // declares an approximation of pi.
char x = 'x';           // the variable x has the value 'x'.
```

Dynamic Initialization:

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

- Example program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
  public static void main(String[] args) {
    double a = 3.0, b = 4.0;

    // c is dynamically initialized
    double c = Math.sqrt(a * a + b * b);

    System.out.println("Hypotenuse is " + c);
  }
}
```

Type Conversion and Casting:

In java, when one type of data is assigned to another type of variable, an automatic type conversion takes place if the following two conditions are satisfied

- The destination memory is larger than the source memory. They are compatible

Casting:

A cast is an explicit type conversion. The general form is given below.

Variable = (target-type) value;

Example:int x;

byte y;

y=(byte)x; // At this point only x is converted to byte.

Types of Casting:

1. Widening Type Casting: Java automatically converts one data type to another data type.
2. Narrowing Type Casting: manually convert one data type into another using the parenthesis.

```
class Main {
    public static void main(String[] args) {
      // create int type variable
      int num = 10;
       System.out.println("The integer value: " + num);
      // convert into double type
      double data = num;
      System.out.println("The double value: " + data);
    }
}
```

```
// Java program to demonstrate Widening TypeCasting
import java.io.*;
class GFG {
public static void main(String[] args)
{
int i = 10;
// Wideing TypeCasting (Automatic Casting)
// from int to long
long l = i;
// Wideing TypeCasting (Automatic Casting)
// from int to double
double d = i;
System.out.println("Integer: " + i);
System.out.println("Long: " + l);
System.out.println("Double: " + d);
}}
```

Output:

Integer: 10

Long: 10

Double: 10.0

Narrowing Type Casting:

```
class Main {
    public static void main(String[] args) {
      // create double type variable
      double num = 10.99;
      System.out.println("The double value: " + num);
      // convert into int type
      int data = (int)num;
      System.out.println("The integer value: " + data);
    }
}
```

Java Automatic Type Conversions:

When these two conditions are met, a **widening conversion** takes place.

- For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

- For widening conversions, the numeric types, including **integer** and **floating-point** types, are compatible with each other.
- However, there are **no automatic conversions** from the numeric types **to char** or **boolean**.
- Also, **char** and **boolean** are not compatible with each other.
- As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Automatic Type Promotion in Expressions:

In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.

- For example.

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands.

- To handle this kind of problem, Java automatically promotes **each byte, short, or char** operand to **int** when evaluating an expression.
- This means that the subexpression **a*b** is performed using **integers—not bytes.** Thus, **2,000**, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type **byte**.

Array:

- An **array** is a **group of like-typed variables** that are referred to by a common name.

  One-Dimensional Arrays: A one-dimensional array is, essentially, **a list of like-typed variables**. To create an array, you first must create an array variable of the desired type.

- The general form of a one-dimensional array declaration is:

type[ ] *var-name*;

```
int[] month_days;
```

To allocate memory for arrays: **new** is a special operator that allocates memory.

array-var = new *type* [*size*];

```
month_days = new int[12];
```

month_days will refer to an array of 12 integers

Multidimensional Arrays:

- Are implemented as arrays of arrays.

```
int[][] twoD = new int[4][5];
```

- //This allocates a 4 by 5 array and assigns it to twoD.

Java Is a Strongly Typed Language:

It is important to state at the outset that **Java is a strongly typed language.**

- **Every variable has a type, every expression has a type, and every type is strictly defined.**
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are **no automatic coercions or conversions** of conflicting types as in some languages.
- The **Java compiler checks all expressions and parameters** to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Operators:

Java provides a rich operator environment. Most of its operators can be divided into the following four groups:

- – arithmetic,
- – bitwise,
- – relational, and
- – logical.

- Java also defines some additional operators that handle certain special situations.

Arithmetic Operators:

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

Basic Arithmetic Operators:

- Addition,
- Subtraction,
- Multiplication,
- and Division

Combining multiple statements:

```
class BasicMath {

public static void main(String[] args) {

    int a = 1 + 1, b = a * 3, c = b / 4, d = c - a, e = -d;

    System.out.println("a=" + a + "\nb=" + b + "\nc=" + c + "\nd=" + d + "\ne=" + e);

}

}
```

Output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
```

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String[] args) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

- Modulus Operator: % this returns the remainder of a division operation.
- Arithmetic Compound Assignment Operators (+=)
- Increment is ++
- Decrement is –-
- Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise LogicalOperators:

- Bitwise NOT (~)
- Bitwise AND (&)
- Bitwise OR(|)
- Bitwise XOR(^)
- Left Shift(<<)
- Right Shift(>>)

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Relational Operators: It deterimine the relationship that one operand has to the other.

NOTE: Outcome of these operations is a **boolean** value.

Boolean Logical Operators : operate only on boolean operands:.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment

Operator	Result
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String[] args) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("          a = " + a);
    System.out.println("          b = " + b);
    System.out.println("        a|b = " + c);
    System.out.println("        a&b = " + d);
    System.out.println("        a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("         !a = " + g);
  }
}
```

Output:

```
          a = true
          b = false
        a|b = true
        a&b = false
        a^b = true
!a&b|a&!b = true
         !a = false
```

The Assignment Operator:

Assignment operator is the single equal sign "=".

- This fragment sets the variables x, y, and z to 100 using a single statement.

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

The value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x.

The ? Operator:

example:

expression1 ? expression2 : expression3

If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.

```
// Demonstrate ?.
class Ternary {
  public static void main(String[] args) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence:

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

Using Parentheses:

- Operators in the same row are equal in precedence.
- Examples of uses of Parentheses

a>>(b+3) (a>>b)+3

In the following statements , which of the following expressions is easier to read?

```
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))
```

Parentheses (redundant or not) do not degrade the performance of your program. Therefore, **adding parentheses to reduce ambiguity does not negatively affect your program.**

Control statements:

- **Selection statements**: if, if-else, if-else-if, switch
- **Iteration statements** :for, while, do-while
- **Jump statements** :break, continue, return.

If:

syntax:

```
if (condition) {
}
```

Example:

```
int a = 10, b = 20;
if (a < b) {
 System.out.println("a is smaller than b")
;}
```

if-else:

Syntax:

```
if (condition)
 {
}
else {
}
```

Example:

```
int a = 10, b = 20;
if (a < b)
{
  System.out.println("a is smaller");
}
 else
{
 System.out.println("b is smaller");
}
```

Nested if:

We can place one if inside another. This is called nested if.

Example:

```
if (i == 10)
 {
 if (j < 20) {
      System.out.println("j is less than 20");
   }
 else
 {
System.out.println("j is greater or equal to 20");
 }
}
```

if-else-if Ladder:

Used when we have multiple conditions to test.

Syntax:

```
if (condition1) {
   // code
}
```

```
else if (condition2)
 {    // code
}
 else if (condition3)
 {    // code
}
else {
   // default block}
```

Example:

```
int marks = 85;
if (marks >= 90)
{
    System.out.println("Grade A");
}
 else if (marks >= 75)
 {
   System.out.println("Grade B");
}
 else if (marks >= 50)
 {
   System.out.println("Grade C");
}
 else
 {
   System.out.println("Fail");
```

switch Statement:The switch statement is used when we have many choices based on one variable.

Syntax:

```
switch (expression)
{    case value1:
       // statements
 break;
case value2:
 // statements
 break;    ...
  default:        // statements
```

Example:int day = 3;

```
switch(day)
 {
case 1:
 System.out.println("Monday");
break;
    case 2:
 System.out.println("Tuesday");
 break;
case 3:
 System.out.println("Wednesday");
 break;
 default:
 System.out.println("Invalid day");}
```

.

2. Iteration Statements (Loops): Loops are used to repeat a block of code multiple times until a condition becomes false. **while Loop**:Executes the block while the condition is true.

Syntax:

```
while (condition) {
    // body of loop}
```

Example:

```
int i = 1;
while (i <= 5)
 {
 System.out.println(i);
    i++;
```

for Loop:

Used when the number of iterations is known.

Syntax:

```
for (initialization; condition; update) {
    // body of loop}
```

Example:

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

do-while Loop:

Executes the loop at least once, even if the condition is false.

Syntax:

```
do {
 // body of loop
}
 while (condition);
```

Example:

```
int i = 1;
do {
    System.out.println(i);
 i++;
}
 while (i <= 5);
```

for-Each Loop :The for-each loop is a special type of for loop that is used to go through (iterate) all elements of an array or a collection easily — from start to end — without using an index.

Syntax:

```
for (type variable : arrayName) {
   // use variable here}
```

type → type of elements in the array
variable → name of a temporary variable that holds each element
arrayName → the array or collection you want to loop through.

Example:

```
class ForEachExample {
    public static void main(String[] args) {
       int nums[] = {1, 2, 3, 4, 5};
  for (int x : nums) {
     System.out.println("Value: " + x);
  }   }
}
```

Local Variable Type Inference in a for Loop:

Example (Normal for loop without inference):for (int i = 0; i < 5; i++) {
System.out.println(i);
}

use type inference in nested loops:

```
public class NestedVar {
    public static void main(String[] args) {
        for (var i = 1; i <= 3; i++) {
         // outer loop           for (var j = 1; j <= 2; j++) {

     // inner loop              System.out.println("i = " i  ", j = " j);
            }
        }
    }
}
```

3.Jump Statements:

Break Statement: To stop a loop immediately before its condition becomes false.
Example : Break in a for loop

```
class BreakExample {
   public static void main(String[] args)
 {
   for (int i = 0; i < 100; i++) {

         if (i == 10) break; // loop stops when i = 10
  System.out.println("i: " + i);
 }
       System.out.println("Loop complete.");
 }
}
```

Continue:

To skip the current iteration of the loop and go to the next one.

Example:

```
class ContinueExample {
public static void main(String[] args) {
for (int i = 0; i < 10; i++) {
          if (i % 2 == 0) continue; //
  System.out.println(i);        }    }
```

Return Statement:

To exit from a method and go back to where it was called.

Example:

```
class ReturnExample {
 public static void main(String[] args) {
        System.out.println("Before return.");
        if (true) return; // exit main()
  System.out.println("This will not execute.");
   }
}
```

Output:Before return.

Explanation: When return executes, the program ends immediately because main() returns to the JV

Thank you

Module-2

Introducing Classes: Class Fundamentals, Declaring Objects, Assigning Object Reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection.

Methods and Classes: Overloading Methods, Objects as Parameters, Argument Passing, Returning Objects, Recursion, Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes.

Introducing classes:

Class:class is a template or blueprint which can be used to create an object

- Object is an instance of a class.

Example:class Student {
  String name;
int age;
void study()
 {
 System.out.println(name + " is studying.");

  }}
class Main {
   public static void main(String[] args) {
    Student s1 = new Student();
  // Object 1
s1.name = "Divya";
 s1.age = 20;

```
s1.study();
 Student s2 = new Student();   // Object 2
 s2.name = "Rahul";
s2.age = 21;        s2.study();    }}
```

Output:

Student → class
s1, s2 → objects

Declaring objects:

Declaring Objects :

- A class is just a blueprint; to use it, we must create an object.
- An object is a real instance of a class created in memory.

syntax:

ClassName objectName;
Example: Box b1;

- The above line declares an object reference but does not create the object (it contains null)
- Memory is allocated for the object using the **new** keyword:b1 = new Box()

We can also declare and create the object in a single statement

Box b1 = new Box();

- The new keyword calls the constructor and allocates memory dynamically at runtime.

. If memory allocation fails, a runtime exception will occur.

Assigning object variables::

- when one object reference variable is assigned to another, no new object is created.
- . The second reference variable will point to the same memory location as the first one.

```
Box b1 = new Box();
Box b2 = b1;
```

- Any changes made using one reference will reflect in the other because both refer to the same object.
- If one reference is set to null, the other still points to the original object.b1 = null; // b2 still points to the object
- This process copies the reference, not the object itself same object

- It is useful for sharing the same object between different parts of a program.
- Care must be taken because modifying data through one reference affects all references pointing to the

Example:

Box b1 = new Box();
Box b2 = b1; // b2 refers to the same object as b1Both b1 and b2 now point to the same object in memory.

- Any update using b1 or b2 changes the same object.

Introducing methods:

- method is a block of code that performs a specific task.

General syntax:

```
return_type methodName(parameter_list)
 {
    // method body
 return value;
}
```

return_type: Data type of the value returned by the method (use void if nothing is returned)..
methodName: Any valid name that describes the method's task.
parameter_list: Input values passed to the method (can be empty).
return: Keyword used to return a value to the calling method

Methods can be:

Without return type: Just perform a task.
With return type: Perform a task and return a result.
With parameters: Take input and process it.

Example:

```
double volume() {
   return w * h * d;
}---
```

Benefits of Using Methods:Reduces code duplication
Improves readability and structure Easy to test and debug
Promotes modular programming

Constructors:

- A constructor is a special method that is called automatically when an object is created.
- Its name must be same as the class name.
- It has no return type, not even void..
- Constructors are used to initialize objects.

Types of Constructors:

1. **Default Constructor** : No arguments, sets default values.

```
Box() {    w = h = d = 5;}
```

2. **Parameterized Constructor** : Accepts arguments to initialize variables

```
Box(double wd , double ht , double dp ) {
  w = wd;    h = ht;    d = dp;}---
```

- **Important Points about Constructors**:If no constructor is defined, Java provides a default constructor automatically.If you define any constructor, the default one is not provided.
- Constructors are called when an object is created using new . They help set up initial values and make objects ready for use.

```
Box b1 = new Box();          // Calls default constructor
Box b2 = new Box(2, 4, 3);   // Calls parameterized constructor
```

This keyword:

- this is a reference variable in Java that refers to the current object.
- Used when a method or constructor needs to refer to the object that invoked it.

Syntax: this. variableName;

Uses of this:

- To distinguish instance variables from local variables with the same name.

- To call other constructors in the same class.
- To pass the current object as a parameter to another method.
- To return the current object from a method.

Instance Variable Hiding: Occurs when a local variable or parameter name is the same as an instance variable name.

- In such cases, the local variable hides the instance variable within that method
- Its means the local variable is uses instead of instance.
- Use

  this. variableName

Example:

```
Class Box
{
Int width;
Box(int width)
{
this.width=width;
}
}
```

Garbage Collection in Java:

- In C/C++, memory must be manually released using delete.
- In Java, Garbage Collection (GC) is automatic.
- GC reclaims memory from objects that are no longer referenced.
- It helps prevent memory leaks and improve application performance.
- GC runs periodically in the background of the Java Virtual Machine (JVM).

The finalize() Method:

- Called by the garbage collector before reclaiming an object's memory.
- Used to release resources (e.g., closing file handles, database connections).

Syntax: protected void finalize() { // cleanup code
}

Important Notes:It is not guaranteed to run immediately or at all. Should not be relied upon for critical resource management.

Methods and classes

Overloading methods:when more than one method has the same name but different parameters, it is called method overloading.

Rules:

- Number of arguments must be different.
- Return type alone is not enough for overloading

Example:

```
class Overload {
   void test()
{
 System.out.println("No parameters");
 }
    void test(int a)
{
 System.out.println("Integer: " + a);
 }
    void test(int a, int b)
 {
 System.out.println("Two integers: " + a + ", " + b);
 }
```

```
 void test(double a)
 {
 System.out.println("Double: " + a);
 }
}
```

Explanation: Same method name → test(), but parameters differ.

```
Overload ob = new Overload();
ob.test();
ob.test(10);
ob.test(10, 20);
ob.test(12.5);
```

Constructor overloading: A class can have multiple constructors with different parameter lists.

Example:

```
class OverloadConstruct {
 OverloadConstruct()
 {
 System.out.println("No arguments");
 }
   OverloadConstruct(int x)
{
```

```
System.out.println("One argument: " + x);
}
    OverloadConstruct(int x, int y)

{
 System.out.println("Two arguments: " + x + ", " + y);
}
OverloadConstruct obj1 = new OverloadConstruct();
OverloadConstruct obj2 = new OverloadConstruct(10);
OverloadConstruct obj3 = new OverloadConstruct(5, 12);
```

Object as a parameters:

Objects can also be passed as arguments to methods.

Example:

```
class Test {
  int a, b;
    Test(int i, int j)
 {
```

```
 a = i;
      b = j;
}
   boolean equals(Test ob) {
      return (ob.a == a && ob.b == b);
   }
}
```

In main():
Test t1 = new Test(10, 20);
Test t2 = new Test(10, 20);
System.out.println(t1.equals(t2));

Using One Object to Initialize Another:

✓ You can use an existing object to initialize a new one.

Example:

```
class Box {
double h, w, d;
    Box(double ht, double wd, double dp)
```

```
{
 h = ht; w = wd; d = dp;
}
    Box(Box bx)
 {
 h = bx.; w = bx; d = bx;
 }

}
main():
Box b1 = new Box(2, 3, 4);
Box b2 = new Box(b1);
```

Method Overloading: Same method name, different parameters
Constructor Overloading: Same constructor name, different arguments.
Objects as Parameters: Objects can be passed to and compared in methods.
Object Initialization: One object can copy another using a constructor.

Argument Passing:

Two Ways to Pass Arguments:

Call by Value:

- Copies the value of an argument into the method parameter.Changes made inside the method don't affect the original variable.

Call by Reference:

- Passes a reference to the actual object.Changes made inside the method affect the original object.

Example:class Test

```
 {
 int a, b;
 Test(int i, int j)
 {
 a = i;
 b = j;
}
 void meth(Test ob)
 {
```

```
  ob. *= 2;
   ob./= 2;
   }
   }

class CallByRef
 {
 public static void main(String args[])
 {
Test ob = new Test(15, 20);
 System.out.println("before call: " + ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("after call: " + ob.a + " " + ob.b);
 }
 }
```

Returning objects:

- In Java, a method can return an object of user defined class.

Example:

```
 class Test {
int a;
 Test(int i)
 {
 a = i;
 }
 Test incrByTen()
 {
Test temp = new Test(a+10);
 return temp;
}
}
 class RetOb {
 public static void main(String args[])
test ob1 = new Test(2);
 Test ob2;
```

```
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: " + ob2.a);
```

output:ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22

Recursion: Recursion is when a method calls itself directly or indirectly.

Important Rules:

- Must have a base condition (non-recursive terminating condition).
- Each recursive call must bring the problem closer to the solution (smaller sub-problem).

Example:

```
class Factorial {
    int fact(int n)
{
     if (n == 0)
```

```
return 1;  // Base case

return n * fact(n - 1);  // Recursive call
  }
}
class FactDemo {
   public static void main(String args[]) {
      Factorial f = new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
      System.out.println("Factorial of 8 is " + f.fact(8));
    }
}
```

Access Control: Access control in Java helps in encapsulation — restricting access to data.

Access Specifiers:
Specifier Scope
Public: Accessible every where
Private : Accessible only within the same class
Protected: Accessible within same package and subclasses
default (no keyword):Accessible only within the same package

Example:

```
class Test {
  int a;   // default
  public int b;     // public
private int c;    // private
   void  setc(int i)
{
    c = i;
    }
```

```
 int getc()
 {
      return c;
    }
}
class AccessTest {
  public static void main(String args)

{
        Test ob = new Test();
       ob.a = 10;
    ob.b = 20;
 ob.c = 30; //
Error:
  ob.setc(30);
     System.out.println("a, b, and c: " + ob.a + " " + ob.b + " "
+ ob.getc());
    }
}
```

Output:a, b, and c: 10 20 30
Explanation:c is private, so can't be accessed directly.Must use setc() and getc() methods to modify or read

Understanding static:

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object

Methods declared as static have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way.

```
class UseStatic {
static int a = 3;
 static int b;
 static void meth(int x) //static method
{
 System.out.println("x = " + x);
System.out.println("a = " + a);
 System.out.println("b = " + b);
 }
 static //static block {
 System.out.println("Static block initialized.");
 b = a * 4;
 }
 public static void main(String args[])
 {
meth(42);
}
}
```

Output: Static block initialized. x = ? a = ? b = ?

- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator. The general form is –

classname.method();

Example:

```
class StaticDemo {
static int a = 42;
 static int b = 99;
 static void callme()
 {
 System.out.println("Inside static method, a = " + a);
 }
 }
 class StaticByName  {
 public static void main(String args[])
 {
StaticDemo.callme();
 System.out.println("Inside main, b = " + StaticDemo.b);
```

```
}
}
```

Output:

Inside static method, a = 42
Inside main, b = 99

Thank you

Module -3

Inheritance: Inheritance Basics, Using super, Creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

Interfaces: Interfaces, Default Interface Methods, Use static Methods in an Interface, Private Interface methods

Inheritance:

- Inheritance is one of the building blocks of object oriented programming languages. It allows creation of classes with hierarchical relationship among them.
- Using inheritance, one can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- A class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass**.

In Java, inheritance is achieved using the keyword "**extends"**.

```
class A            //super class
{
 //members of class A
}
 class B extends A       //sub class
 {
//members of B
```

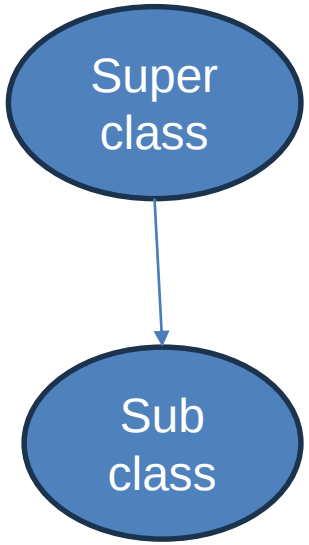
Example:

```
class A {
int i, j;
 void showij()
 {
 System.out.println("i and j: " + i + " " + j);
 }
 }
 class B extends A {
 int k;
void showk()
 {
 System.out.println("k: " + k);
 }
 void sum()
 {
 System.out.println("i+j+k: " + (i+j+k));
}
 }
```

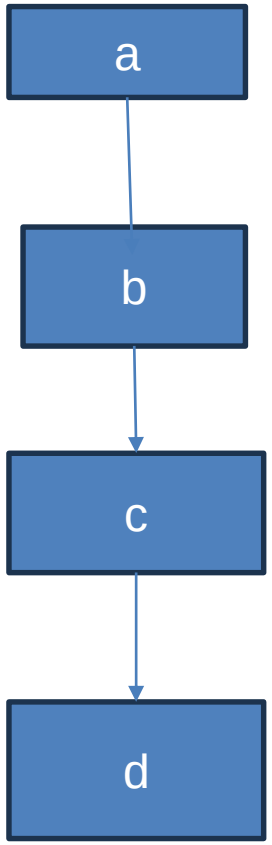
```
class SimpleInheritance
 {
public static void main(String args[])
 {
 A superOb = new A();
 B subOb = new B();
superOb.i = 10;
 superOb.j = 20;
System.out.println("Contents of superOb: ");
 superOb.showij();
 subOb.i = 7;
 subOb.j = 8;
 subOb.k = 9;
 System.out.println("Contents of subOb: ");
 subOb.showij();
 subOb.showk();
 System.out.println("Sum of i, j and k in subOb:");
 subOb.sum();
 }
 }
```

Type of Inheritance :

Single Inheritance: If a class is inherited from one parent class, then it is known as single inheritance. This will be of the form as shown below



Multilevel Inheritance: If several classes are inherited one after the other in a hierarchical manner, it is known as multilevel inheritance, as shown below



A Superclass variable can reference a subclass object:

```
class Base {
 void dispB()
 {
 System.out.println("Super class " );
  }
}
 class Derived extends Base
{
 void dispD()
 {
 System.out.println("Sub class ");
 }
 }
 class Demo {
 public static void main(String args[])
 {
 Base b = new Base();
```

```
Derived d=new Derived();
 b=d;
 b.dispB(); //superclass reference is holding subclass object //
b.dispD();
}
}
```

Note:

- type of reference variable decides the members that can be accessed, but not the type of the actual object.

Using super:

the keyword super can be used in following situations:

- To invoke superclass constructor within the subclass constructor
- To access superclass member (variable or method) when there is a duplicate member name in the subclass

To invoke superclass constructor within the subclass constructor Sometimes:

- we may need to initialize the members of super class while creating subclass object.
- Writing such a code in subclass constructor may lead to redundancy in code.

example :

```
class Box
{
 double w, h, b;
 Box(double wd, double ht, double br)
 {
 w=wd; h=ht; b=br;
 }
 }
 class ColourBox extends Box
{
int colour; ColourBox(double wd, double ht, double br, int c)
 {
 w=wd; h=ht; b=br; colour=c;              //code  redundacy
 }
 }
```

- if the data members of super class are private, then we can't even write such a code in subclass constructor.
- If we use super() to call superclass constructor, then it must be the first statement executed inside a subclass constructor.

```
class Box {
 double w, h, b;
 Box(double wd, double ht, double br)
{
w=wd; h=ht; b=br;
}
 }
 class ColourBox extends Box
{
 int colour;
 ColourBox(double wd, double ht, double br, int c)
 {
 super(wd, ht, br);
 colour=c;
```

```
}
 }
 //calls superclass constructor
 class Demo
{
public static void main(String args[])
 {
 ColourBox b=new ColourBox(2,3,4, 5);
 }
 }
```

Creating Multilevel Hierarchy Java supports multi-level inheritance. A sub class can access all the non-private members of all of its super classes.

Example:

```
class A {
 int a;
{
```

```
}
 class B extends A
{
 int b;
 }
 class C extends B
{
Int c;
C(int x, int y, int z)
 {
 a=x; b=y; c=z;
 }
 void disp()
 {
 System.out.println("a= "+a+ " b= "+b+" c="+c);
 }
 }
}
 }
```

```
class MultiLevel {
 public static void main(String args[])
 {
 C ob=new C(2,3,4);
 ob.disp();
}
}
```

When Constructors are called:

- When class hierarchy is created (multilevel inheritance), the constructors are called in the order of their derivation.
- That is, the top most super class constructor is called first, and then its immediate sub class and so on.
- If super is not used in the sub class constructors, then the default constructor of super class will be called

```
class A
{
 A()
 {
 }
 }
 System.out.println("A's constructor.");
class B extends A { B() {
 }
System.out.println("B's constructor.");
 }
class C extends B {
C()
 {
 }
 System.out.println("C's constructor.");
 }
```

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

Output: A's constructor B's constructor C's constructor

Method Overriding:

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

```
class A
{
 int i, j;
 A(int a, int b)
 {
 i = a; j = b;
 }
 void show()
{                          //suppressed
 System.out.println("i and j: " + i + " " + j);
}
 }
class B extends A
{
 int k;
 B(int a, int b, int c)
{
 super(a, b);
 k = c;
 }
```

```
void show()
 { //Overridden method
System.out.println("k: " + k);
}
 }
 class Override
{
 public static void main(String args[])
 {
 B subOb = new B(1, 2, 3);
 subOb.show();
 }
 }
```

Output: k: 3

Dynamic Method Dispatch:

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- java implements run-time polymorphism using dynamic method dispatch. We know that, a superclass reference variable can refer to subclass object.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.

Example:

```
class A
{
void callme()
 {
 System.out.println("Inside A");
 }
 }
 class B extends A
 {
```

```
 void callme()
 {
 System.out.println("Inside B");
 }
 }
class C extends A
{
 void callme()
 {
 System.out.println("Inside C");
 }
 }
 class Dispatch {
 public static void main(String args[])
 {
 A a = new A();
 B b = new B();
 C c = new C();
 A r; r = a; //Superclass reference
```

```
//holding subclass
object r.callme();
r = b;
r.callme();
r = c;
r.callme();
}
}
```

Using Abstract class:

- A method which does not contain any definition in the superclass is termed as **abstract method**. Such a method declaration should be preceded by the keyword **abstract**.
- A class containing at least one abstract method is called as abstract class.
- Abstract classes can not be instantiated, that is one cannot create an object of abstract class. Whereas, a reference can be created for an abstract class.

```
abstract class A
 {
abstract void callme();
 void callmetoo()
{
 System.out.println("This is a concrete method.");
 }
 }
class B extends A
{
 void callme() //overriding abstract method
{
System.out.println("B's implementation of callme.");
 }
 }
 class AbstractDemo {
 public static void main(String args[])
 {
 B b = new B(); //subclass object
```

```
b.callme(); //calling abstract method
b.callmetoo(); //calling concrete method } }
```

Using final:

The keyword **final** can be used in three situations in Java:

- To create the equivalent of a named constant.
- To prevent method overriding
- To prevent Inheritance

To create the equivalent of a named constant:

A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS =
4; final int FILE_QUIT = 5;
```

To prevent method overriding:

Sometimes, we do not want a superclass method to be overridden in the subclass. Instead, the same superclass method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword final

```
class A {
final void meth()
 {
}
 System.out.println("This is a final method.");
 }
 class B extends A
{
void meth() // ERROR! Can't override.
 {
 }
 } System.out.println("Illegal!")
```

To prevent Inheritance:

- the subclass is treated as a specialized class and superclass is most generalized class.
- During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword final.

```
final class A
{
 // ...
}
 class B extends A {
 } // ..
```

Note:

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations