UNIT-1

UNIT I: Object Oriented Programming: Basic concepts, Principles, Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences Comments, Programming Style.

Data Types, Variables, and Operators :Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Static Variables and Methods, **Introduction to Operators**, Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (- -) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators.

Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, Break Statement, Continue Statement

Object Oriented Programming

- Object Oriented Programming is a programming concept that works on the principle that objects are the most important part of your program.
- It allows users create the objects that they want and then create methods to handle those objects.
- Manipulating these objects to get results is the goal of Object Oriented Programming. Object Oriented Programming popularly known as OOP, is used in a modern programming languages like Java.

Object :

Any real world entity that has state and behaviour is called as Object .(or)

Objects have state and behaviour. Example: Apple, Orange, Bat, Table, etc..

In Java, An Object is an Instance of class.

Class :

Collection of similar objects is called Class . For Example, Apple, orange, Papaya are grouped into a class called “Fruits” where as Apple, Table, Bat cannot be grouped as class because they are not similar groups. It is only an logical component not as physical entity.

Inheritance :

One object acquires all properties and behaviour of the parent object.

It’s creating a parent-child relationship between two classes. It offers robust and natural,mechanism for organizing and structure of any software.

Polymorphism :

It refers as “ one interface and many forms” (or) the ability of a variable , object or function to take on multiple forms.

Ex:- In English, the verb “run” has a different meaning if you see it with a “laptop”, and “a foot race”.

Abstraction :

Abstraction is a process of hiding the implementation details from the user.
Ex:- while driving a car, you do not have to be concerned with its internal working.
Abstraction can be achieved using Abstract Class and Abstract Method in Java.

Encapsulation :

Encapsulation is a principle of wrapping data (Variables) and code together as a single unit. In this OOPS concept, variables of a class are always hidden from other classes. It can only be accessed using the methods of their current classes.

Program Structure in Java

Elements or Tokens in Java Programs

In Java programming, elements or tokens are the smallest individual units in a program. These tokens are the building blocks of Java code and are used to construct statements and expressions. Here are the main types of tokens in Java

1. **Keywords**
2. **Identifiers**
3. **Literals**
4. **seperators**
5. **comments**
6. **operators**

1. Keywords

Keywords are reserved words in Java that have a predefined meaning and cannot be used as identifiers (names for variables, classes, methods, etc.). Examples of keywords include class, public, static, void, int, if, else, for, while, return, etc.

2. Identifiers

Identifier: A name in java program is called identifier. It may be class name, method name, variable name and label name.

Example:

```
class Test  ---1
{
public static void main(String[] args){
int x=10;
}
}
```

(1 = Test, 2 = main, 3 = String, 4 = args, 5 = x)

Rules to define java identifiers:

Rule 1: The only allowed characters in java identifiers are:

1) a to z
2) A to Z
3) 0 to 9
4) _
5) $

Rule 2: If we are using any other character we will get compile time error.

Example:

1) total_number-------valid
2) Total#------------------invalid

Rule 3: identifiers are not allowed to starts with digit.

Example:

1) ABC123---------valid
2) 123ABC---------invalid

Rule 4: java identifiers are case sensitive up course java language itself treated as case sensitive language.

Example:

```
class Test{
int number=10;
int Number=20;
int NUMBER=20;     we can differentiate with case.
int NuMbEr=30;
}
```

Rule 5: There is no length limit for java identifiers but it is not recommended to take more than 15 lengths.

Rule 6: We can't use reserved words as identifiers.

Example: int if=10; --------------invalid

Rule 7: All predefined java class names and interface names we use as identifiers.

3. Constants or Literals

- Entities that do not change their values in a program are called Constants or Literals.
- Java Literals are classified into 5 types:
    1. Integer Literals
    2. Floating Point Literals
    3. Character Literals
    4. Boolean Literals
    5. String Literals

1) Integer Literals :

- A whole number is called an integer. Eg: 25,27 etc…are integers
- Java supports 3 types of integer literals Decimal, Octal, Hexadecimal.
- 25, 27 are decimal integers
- Octal stats from 0 and followed by 0 to 7. Eg: 0.037, 0.08656 are octal integer
- Hexadecimal start with OX and followed by digits 0 to 9 , A to F. Eg: 0*29, 0*2AB9 are hexadecimal integer literals .

2. Floating Point Literals :

- Numbers with decimal point and fractional values are called floating point literals.
- They can be expressed in either standard or scientific notation.
- Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.
- A Floating point number followed by letter E (or) and a signed integer. Eg: 6.237E-35 stands for 6.237*10^-35.
- Floating point literals in java defaults to double precision.

3. Boolean literals :

- In java, Boolean literals take two values false or true.
- These two values are not related to any numeric value as in C or C++.
- The Boolean value true is not equal to 1 and false is not value is not equal to 0.

4. Character literals :

- Single characters in java are called character literals.

- In java characters belong to 16-bit character set called Unicode.
- Java characters literals are written within a pair of single quote. Eg: ‘a’, ‘z’, represent character literals.
- To represent such characters, java provides a set of character literals called escape sequence.

5. **String Literals :**
   - A sequence of characters written within a pair of double quote is called String Literal.
     Eg: “This is String”.
   - String Literals are to be started and ended in one line only.

4. Separators

Separators (or delimiters) are symbols that separate elements of the code. Common separators in Java include:

- **Parentheses:** ()
- **Braces:** {}
- **Brackets:** []
- **Semicolon:** ;
- **Comma:** ,
- **Dot:** .

5. Comments

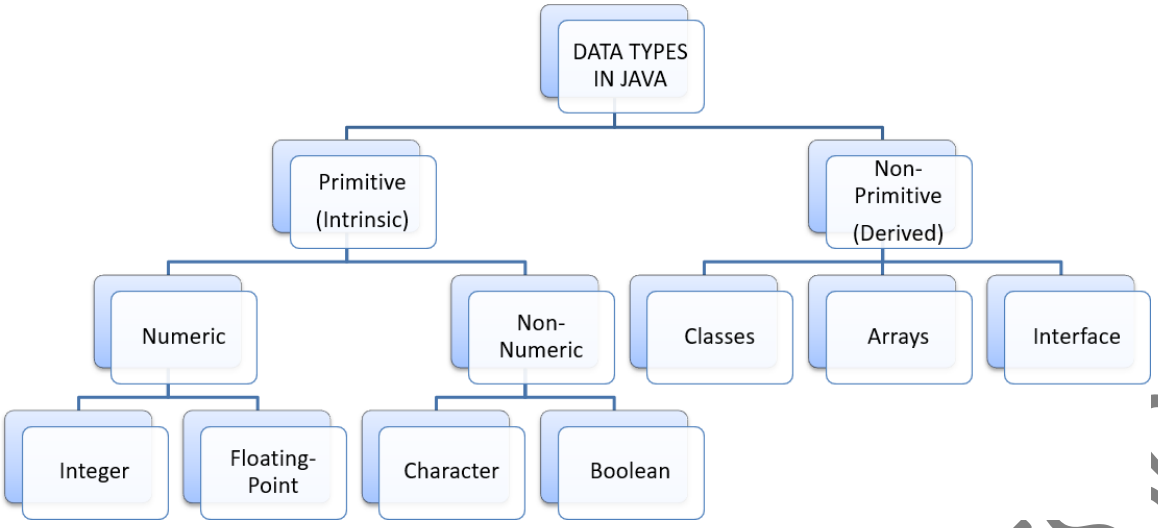
Comments are non-executable parts of the code that are used to describe or explain the code. They are ignored by the Java compiler. There are two types of comments in Java:

- **Single-line comments:** Start with //
- **Multi-line comments:** Enclosed between /* and */

Data Types, Variables, and Operators

Data Types

- Every variable in java has a data type.
- Data type specify the size and type of values that can be stored.
- Data type in java under various categories are shown as:



A. Primitive data types :

Primitive data types are whose variables allows us to store only one value but they never allow us to store multiple values of same type. This is a data type whose variables can hold maximum one value at a time.

Example:

```
int a;
a=10;//valid
a=10,20,30;//invalid
```

B. Non Primitive Data Types or Derived

Derived data types are those which are developed by programmers by making use of appropriate features of the language. User defined data types related variables allow us to store multiple values either of same type or different type or both.

Example:

```
Student s=new Student();
```

Java defines some primitive types of data. They are:

Integer types

Character types

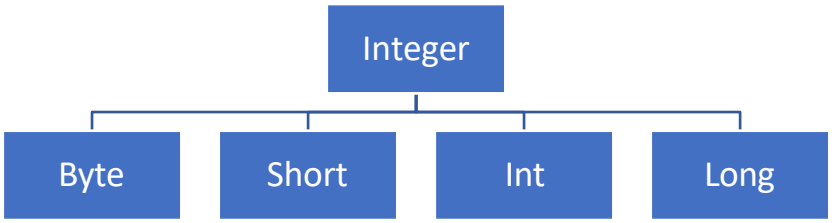
Floating type

Boolean type

Integer Types :

This type indicates byte, short, int, long which are for whole-valued signed numbers.

The width and ranges of these integer types vary widely as shown in below :



The width and ranges of these integer types vary widely as shown in below :

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 :

- Smallest integer type is byte.
- This is signed 8-bit type that has range from -128 to 127
- It is declared by byte keyword

Short :

- Short is signed as 16 bit type
- It has range from -32,768 to 32,767
- It is declared by short keyword.

Int :

- The most commonly used type is integer type as int.
- It is signed as 32 bit type and has range from -2,147,483,648 to 2,147,483,647

Long :

- long is signed as 64-bit type and is useful for those occasions where as int.
- The range of long is quite large.
- This makes it useful when big, whole numbers are needed.

Floating Point Types :

- This group includes float and double which represented numbers in fractional precision.
- They are two types of floating point types ,float and double, which represents single and double precision numbers

Name	Width in bits	Approximate range
Double	64	4.9e-324 to 1.8e+308
Float	32	1.4e-045 to 3.4e+038

3. **Characters :**
   - In Java, the data type is used to store characters is char.
   - Char in java is not same as C or C++
   - In C/C++ char is 8 bit type whereas in java char is 16-bit type
4. **Boolean Type :**
   - Java has a primitive data type called Booleans, for logical values.
   - It can have only one of two possible values true or false.

Variables

- A Variable is an identifier that denote s a storage location used to store a data value . (or) Variables are the names of storage locations.
- Variable names may consist of alphabets, digits, the underscore and dollar characters.
  - They must not begin with a digit.
  - Uppercase and Lowercase are distinct. This means that the variable Total is not same as total or TOTAL.
  - It should not be a keyword.
  - White space is not allowed.
  - Variable names can be any length.

Declaration of Variables :

- A Variable must be declared before it is used in the program. The general form of declaration of a variable is

  Type variable1, variable2, ...variable
- Variables are separated by commas. A declaration statement must end with a semicolon. Some valid declarations are:

Int count;
float x, y;

Giving values to Variables :

A Variable must be given a value after it has been declared but before it is used in an expression. This can be in two ways:

1. By using an Assignment statement

   VariableName=value

2. By using a read statement

   we may also give values to variables interactively through the keyword using the readLine().

Scope of the variable :

- the scope refers to validity across the java program.
- The scope of a variable is limited to the block defined within the braces { and }
- It means a variable cannot be accessed outside the scope (Or) The scope or a particular variable is the range within a program 's source code in which that variable is recognized by the compiler.

Type Conversion and Casting

Assigning a value of one type to a variable of another type is known as **Type Casting.**

Example:

```
int x=10;
byte y=(byte)x;
```

In Java, type casting is classified into two types.

Widening Casting (Implicit) : Process of Converting lower data type into higher data type

byte --->short --->int --->long --- >float --- >double

---------->

Widening

Narrowing Casting (Explicitly done) : Process of converting Higher Data type into LowerData Type

double --->float ---->long ----> int ---->short --- >byte

---------->

Example: Converting int to double

```
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);
  }
}
```

Output
The integer value: 10
The double value: 10.0

Example: Converting double into an int

```
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);
  }
}
```

Output
The double value: 10.99
The integer value: 10

-

Types of Variables

- Based the type of value represented by the variable all variables are divided into 2 types. They are:
  1) Primitive variables
  2) Reference variables

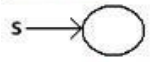
Primitive variables: Primitive variables can be used to represent primitive values.

Example: int x=10;

Reference variables: Reference variables can be used to refer objects.

Example: Student s=new Student();

Diagram:



- Based on the purpose and position of declaration all variables are divided into the following 3 types.
  1) Instance variables
  2) Static variables
  3) Local variables

Instance variables

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.
- Instance variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is exactly same as scope of objects.
- Instance variables will be stored on the heap as the part of object.
- Instance variables should be declared with in the class directly but outside of any method or block or constructor.
- Instance variables can be accessed directly from Instance area. But cannot be accessed directly from static area.
- But by using object reference we can access instance variables from static area.

Example:

```
class Test
{
        int i=10;
        public static void main(String[] args)
        {
                //System.out.println(i);//C.E:non-static variable i cannot be referenced from a
static context(invalid)
                Test t=new Test();
                System.out.println(t.i);//10(valid)
                t.methodOne();
        }
        public void methodOne()
        {
                System.out.println(i);//10(valid)
        }
}
```

- For the instance variables it is not required to perform initialization JVM will always provide default values.

Static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as instance variables. We have to declare such type of variables at class level by using static modifier.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables for entire class only one copy will be created and shared by every object of that class.
- Static variables will be crated at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the **.class** file.
- Static variables will be stored in method area. Static variables should be declared with in the class directly but outside of any method or block or constructor.
- Static variables can be accessed from both instance and static areas directly.
- We can access static variables either by class name or by object reference but usage of class name is recommended.
- But within the same class it is not required to use class name we can access directly.

jk

- For the static variables it is not required to perform initialization explicitly, JVM will always provide default values.

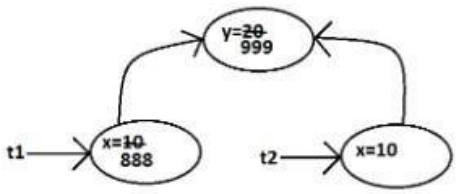
Example:

```
class Test
{
	static String s;
	public static void main(String[] args)
	{
		System.out.println(s);//null
	}
}
```

Example:

```
class Test
{
	int x=10;
	static int y=20;
	public static void main(String[] args)
	{
		Test t1=new Test();
		t1.x=888;
		t1.y=999;
		Test t2=new Test();
		System.out.println(t2.x+"----"+t2.y);//10----999
	}
}
```

Diagram:



- Static variables also known as class level variables or fields.

Local variables:

- Some time to meet temporary requirements of the programmer we can declare variables inside a method or block or constructors such type of variables are called local variables or automatic variables or temporary variables or stack variables.

- The local variables will be created as part of the block execution in which it is declared and destroyed once that block execution completes. Hence the scope of the local variables is exactly same as scope of the block in which we declared.

Example 1:

```
class Test
{
	public static void main(String[] args)
	{
		int i=0;
		for(int j=0;j<3;j++)
		{
			i=i+j;
		}
		System.out.println(i+"----"+j);
	}
}
```

C.E

```
javac Test.java
Test.java:10: cannot find symbol
symbol  : variable j
location: class Test
```

Example 2:

```
class Test
{
	public static void main(String[] args)
	{
		try
		{
			int i=Integer.parseInt("ten");
		}
		catch(NullPointerException e)
		{
			System.out.println(i);
		}
	}
}
```

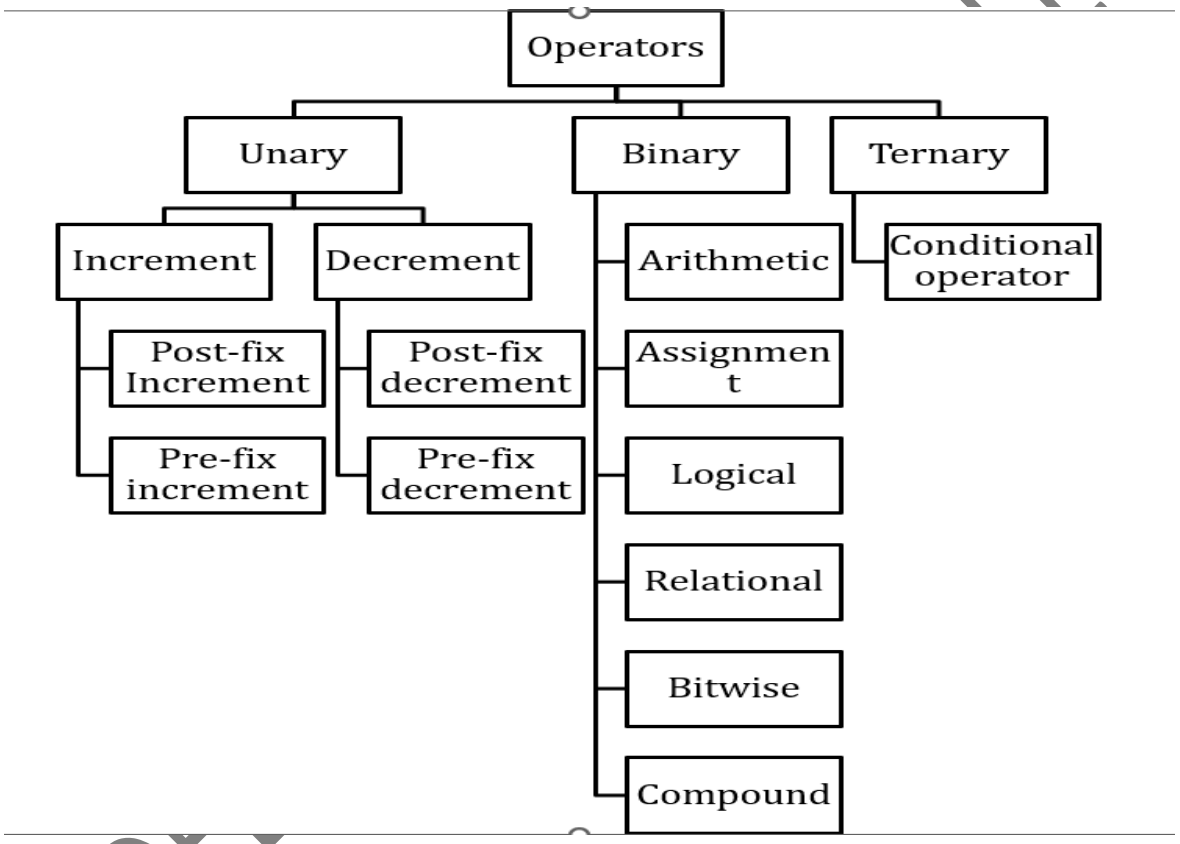
C.E

```
javac Test.java
Test.java:11: cannot find symbol
symbol  : variable i
location: class Test
```

OPERATORS AND EXPRESSIONS

- In java Operators are symbols that are used to perform some operations on the operands.
- Combination of operands and operators are known as Expressions.
- Java provides, a rich set of operators to manipulate the variables. There are three types of operators in java.

  1. Unary operators
  2. Binary operators
  3. Ternary operators



1. UNARY OPERATORS:

In which we use one operand is called unary operator. It has two types:

1.1 Increment Unary operator

1.2 Decrement Unary operator

1.1 INCREMENT UNAY OPERATOR:

This is used to increase the value one by one. It has two types:

* Post-fix Increment operator
* pre-fix Increment operator

1.1.1 POST-FIX INCREMENT OPERATOR:

"++" symbol is used to represent Post-fix Increment operator. This symbol is used after the operand.

In this operator, value is first assign to a variable and then incremented the value.

```
EX:     int a , b;
        a=10;
        b=a++;
```

In the above example first the value of "a" is assign to the variable "b", then I Increment the value, so the value of b variable is "10".

1.1.2 PRE-FIX INCREMENT OPERATOR:

"++" symbol is used to represent Pre-Fix operator, this symbol is used after the operand. In this operator value is incremented first and then assigned to a variable.

```
EX:     int a,b;
        a=10;
        b=++a;
```

In the above example first the increment is done then the value of "a" variable is assigned to the variable "b", so the value of "b" variable is "11".

1.2 DECREMENT UNARY OPERATOR:

"-" symbol is used to decrease the value by one. It has two types:

1. Post-fix decrement operator
2. pre-fix decrement operator

1.2.1 POST_FIX DECREMENT OPEATOR:

"-" symbol is used to represent post-fix decrement operator, this symbol is used after the operand. In this operator, value is first assigned to a variable and then decrement the value.

```
EX:     int a , b;
        a=10;
        b=a--;
```

In the above example first the value of “a” is assign to the variable ” b”, then decrement the value. So the value of “b” variable is “10”.

1.2.2 PRE-FIX DECREMENT OPERATOR:

“-” Symbol is used to represent the pre-fix decrement operator. This symbol is used after the operand. In this operator, value is decremented first and then decremented value is used in expression.

```
EX:        int a,b;
    a=10;
        b=--a;
```

In the above example first the value of “a” is decrement then assign to the variable “b”. So the value of b variable is “9”.

2. BINARY OPERATOR:

In which we use two operand is called Binary operator. Java supports many types of Binary operators:

* Assignment operator
* Arithmetic operator
* Logical operator
* Comparison operator

2.1 ASSIGNMENT OPERATOR:

This operator is used to assign the value. This symbol “=“ is used to assign the value .

```
EX:        int a=12;
```

2.2 ARITHMETIC OPERATOR:

This operator is used to perform mathematical operand. Arithmetic operator are:

	Operators	Description	Use
1.	Additional operator (“+”) :	Used to add the value of two operand.	a+b
2.	Subtract operator (“-“) :	Used to subtract the value of two operand.	a-b
3.	Multiply operator (“*”) :	Used to multiply the value of two operand.	a*b
4.	Division operator (“/”) :	Used to divide the value of two operand.	a/b
5.	Modulus operator(“%”) :	Used returns the remainder of a division operation.	a%b

LOGICAL OPERATOR:

The logical operator ||(conditional-OR),&&(conditional-AND),!(conditional-NOT) operates on boolean expressions, here’s how they work:

OPERATOR	DESCRIPTION	EXAMPLE
||	Conditional-OR; true if either of the boolean expression is true.	False || true is evaluated to true.
&&	Conditional-AND; true if all boolean expressions are true.	False && true is evaluated to false.
!	Conditional-NOT; true if expression is false.	! False is evaluated to true.

LOGICAL NOT OPERATOR:

- Logical NOT operator is used to reverse the logical state of its operand. If a condition is true then

  logical NOT operator will make false. If a condition is false then Logical NOT operator will make true.

- Then NOT operator is probably the easiest to understand. It is simply the opposite of what the condition says.

```
EX:             boolean a=true;
        if(!a)
                System.out.println("u r win");
                else
        System.out.println("u r not win");
```

- In above example "if not true" is asking if the variable "a" variable is not true, otherwise known as false.
- If "a" variable is false, java will displays "u r win" "a" variable is not true , so that code will not execute, then the else part is execute shown in output.

RELATIONAL OPERATOR:

- This operator is used to compare the two values, so this operator is also known as "comparison operator"
- Conditional symbols and their meanings for comparison operator are below:

OPERATOR	CONDITION	DESCRIPTION	EXAMPLE
==	is equal to	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a==b) is not true
!=	Is not equal to	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a!=b) is true
>	Is greater than	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a>b) is not true
<	Is less than	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a<b) is true

TERNARY OPERATOR:

In ternary operator use three operands. It is also called Conditional assignment statement because the value assigned to a variable depends upon a logical expression.

SYNTAX:

variable=(test expression)?Expression 1: Expression 2

EX:

c=(a>b)?a:b:

c= (a>b) ? a:b;

Test condition ? Expression1 : Expression2;

BITWISE OPERATORS:

Java provides 4 bitwise and 3 bit shift operators to perform bit operations.

* **|** Bitwise OR
* **&** Bitwise AND
* **-** Bitwise Complement
* **^** Bitwise XOR
* **<<** Left shift
* **>>** Right shift
* **>>>** Unsigned Right Shift

Bitwise and bit shift operators are used on integral types (byte, short, int and long) to perform bit-level operations.

OPERATOR	DESCRIPTION
|	Bitwise OR
&	Bitwise AND
~	Bitwise Complement
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Unsigned Right shift

BITWISE OR:

Bitwise OR is a binary operator(operates on two operands). It's denoted by |. The | operator compares corresponding bits of two operands. If their of the bits is 1. If not, it gives 0.

EX:

12= 00001100

25= 00011001

Bitwise OR Operation of 12 and 25

00001100

00011001

00011101 =29(In decimal)

BITWISE AND :

Bitwise AND is a binary operator (operates on two operands). It's denoted by &. The & operator compares corresponding bits of two operands. If both bits are 1. If either of the bits is not 1, it gives 0.

EX: 12= 00001100

25=00011001

Bit operation of 12 and 25

00001100

00011001

00001000 = 8(in decimal)

BITWISE COMPLIMENT :

Bitwise compliment is an unary operator(works on only one operand). It is denoted by ~. The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

EX: 35= 00100011(in binary)

bitwise complement of 35

~ 00100011

11011100 = 220(in decimal)

BITWISE XOR :

Bitwise XOR is a binary operator(operates on two operands). It's denoted by "^" . The operator compares corresponding bits of two operands. If corresponding bits are different, It gives 1. If corresponding bits are same, it gives 0.

EX: 12=00001100

25=00011001

Bitwise XOR operation of 12 and 25 is:

00001100

1 00011001

00010101 =21(in decimal)

Control Statements

Causes the flow of execution to advance and branch based on changes to the state of program.

In Java, control statements can be divided into the following three catego

1) Selection Statements
2) Iteration Statements
3) Jump Statements

1) Selection Statements

Selection statements allow you to control the f
outcome of an expression or state of a va
can be divided into the following ca

a) The if and if-else statemen
b) The if-else statemen
c) The if-else-if statement
d) The switch

The if statements :

The first contained statement (that can be a block) of an if statement only executes when the specified condition is true. If the condition is false and there is not else keyword then the first contained statement will be skipped and execution continues with the rest of the program. The condition is an expression that returns a boolean value.

General form of simple if statement is

```
if<expression>
{
        Statement-block;
}
```

The statement-block may be single statement or a group of statements .

if the expression is true, the statement block will be executed, otherwise the statement block will be skipped to the statement-x.

if else statement:-

if else statement is an extension of the simple if statement. The general form is

```
if(expression)
{
        True-block statements
}
Else
{
        False-block statements
}
```

- if the test expression is true, then the true-block statements immediately following the if statement are executed. Otherwise, the false-block statementsare executed .
- In either case, Either true-block or false-block will be executed, not both.
- In both the cases, the control is transferred subsequently to the statement-x Diagram

Nested if else statement :-

- A nested if is an if statement that is the target of another if or else.
- Nested ifs are very common in programming
- General form of Nested if looks like
- Nested if else statement is made by placing one if else in another if else statement.
- Nested if else statement helps to select one out of many chooses.
- General form of Nested if else is

```
if<cond1>
    {
        if<cond2>
        {
            if<cond3>
                stmt 4
            else
                stmt3
            }
            else
                stmt2
            }
            else
                stmt 1
```

- In the nested if else statement, the outermost if is evaluated first.
- If the condition1 is false, the statement is the outermost else is evaluated and if else ends.
- If the conditon1 is true, the control goes to execute the next inner if statement.
- If conditon2 is false, statement2 is executed otherwise conditon3 is evaluated
- If condition3 is false statement3 is executed. Otherwise statement is executed.

else if ladder:-

- A common programming construct that is based a sequence of nested is based upon a sequence of nested ifs is the if else if ladder.
- • General form of if else ladder

```
if<condition>
        stmt
else if<condition>
        stmt;
else if<condition>
        stmt;
else
        stmt;
```

- The if statements are Executed from the top down. As soon as one of the conditions controlling the if is true, the stmt associated with that if is Executed, and the rest of the ladder is bypassed.
- If none of the condition is true, then the final else stmt will be executed.
- The final else acts as a default condition; i.e if all other conditional tests fail, then the last else stmt is performed.
- If there is no final else and all other condition are false.

Switch statement:-

- The switch statement helps to select one out of many chooses.
- It often provides a better alternative than a large d=series of if else if statements
- General form of switch statement is

```
Switch(expression)
{
        Case value 1:stmt1;
                Break;
        Case value 2:stmt2;
                Break;
                .
                .
        Case value N: stmt N;
                Break;
        Default:    stmt;
}
```

- The expression must be of type byte, short, int or char.
- Each of the values specified in the case stmts must be of a type compatible with the expression.
- Each case value must be unique literal.
- Duplicate case value are not allowed.
- The switch stmt works like this

while:-

The while loop is java's most fundamental loop stmt

• It repeats a stmt or block while its controlling expression is true.

• The general form of while stmt is

```
While <condition>
{
Body of the loop
}
```

The condition can be any Boolean expression.

The body of the loop will be executed as long as the conditional expression is true

When condition becomes false, control passes to the next line of code immediately following the loop.

The curly braces are unnecessary if only a single stmt is being repeated.

Do-while:-

- If the conditional expression controlling a while loop is initially false, then the body of the loop will not executed at all
- However, it is desirable to execute the of a loop at least once even if condition expression is false to begin with
- Fortunately, java supplies a loop that does just that : the do while
- The do while loop always execute its body at least once, because its conditional expression is at bottom of loop
- The general form of do while is

```
do
{
    Body of the loop
}
While<condition>
```

For statement

- General form of traditional for statement

```
isfor(initialization; condition;
iteration)
        {
         Body of the loop
        }
```

- It is important to understand that initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression .i.e the

loop controlvariable against a target value.

- If this expression is true, then the body of the loop is executed.
- If it is false, the loop terminates.
- Next, the iteration portion of the loop executed
- This is usually an expression that increments or decrements the loop controlsvariable.
- This loop then ITERATES
- First evaluating the conditional expression , then executing the body of the loop ,and then executing the iteration expression with each pass.
- This process repeats until the controlling expression is false.

Nested loop:-

Like all other programming languages, java allows loops to be nested.
i.e one loop may be inside
anotherEg:-

```
For(i=0 ; i<10 ; i++)
{
        For(j=I ; j<10 ; j++)
        {
                statement block
        }
}
```

Jump stmts:_

Java supports 3 jump stmts

1. break
2. continue
3. return.

Break stmt:-

It has 3 uses.

1. It terminates a stmts sequence in a switch stmt.
2. If can be used to exit a loop.
3. If can be used as a "civilized" form of goto.

When a break stmt is encountered inside a loop. The loop is terminated and program control resumes at the next stmt following the loop.

i.e by using break, we can force immediate termination of a loop, by passing the conditional expression (eg: i<=10) and any Remaining code in the body of the loop.

continue:-

sometimes, you might want to continue running the loop but stop continue running the remainder of the code in its body for this particular iteration

the continue stmt performs such as an action

Return:-

Return stmt is used to explicitly return from a method

i.e it causes program control to transfer back to the caller of the method

return stmt can be used to cause execution to branch back to the caller at the method.

M.DEEPIKA ASSISTANT PROFESSOR

UNIT-2

Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this**.**

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

CLASSES AND OBJECTS:

Class Declaration And Modifiers

Defining a Class

- A class is a user-defined data type with a template that serves to define its properties.
- Once the class type has been defined, we can create "variables" of that type using declarations that are similar to the basic type declarations.
- In Java, these variables are termed as instances of classes, which are the actual objects.
- Class Defines Data and Methods that manipulate the Data.

The basic form of a class definition is

```
class ClassName [extends SuperClassName]
        {
                [fields declaration]
                [methods declaration]
        }
```

Modifiers

Modifiers are keywords that you can use to change the behavior or visibility of classes, methods, and variables. They can be divided into two categories: **Access Modifiers** and **Non-Access Modifiers**.

Access Modifiers

Access modifiers determine the visibility of the class to other classes. Java provides four access levels:

1. **public**: The class is accessible from any other class.

2. **protected**: The class is accessible within its package and by subclasses.
3. **default (no modifier)**: The class is accessible only within its own package.
4. **private**: The class is accessible only within the class it is defined. Note that private is not applicable to top-level classes.

Non-Access Modifiers

Non-access modifiers provide functionality other than visibility control:

1. **final**: The class cannot be subclassed.
2. **abstract**: The class cannot be instantiated and may contain abstract methods.
3. **static**: The modifier indicates that the nested class is a static member of the outer class

Class Members

Class members include fields (variables), methods, constructors, and nested classes/interfaces.

Fields Declaration

- Data is encapsulated in a class by placing data fields inside the body of the class definition.
- These variables are called instance variables because they are created whenever an object
- of the class is instantiated.
- We can declare the instance variables exactly the same way as we declare local variables

Class Rectangle

```
{
        int length;
        int width;
}
```

- The class Rectangle contains two integer type instance variables.
- It is allowed them in one line as
- **int length,width;**

Methods Declaration

The General form of a method declaration is

type methodName(parameter-list)

{

Method-body;

}

Method declarations have four basic parts

- The name of the method(method name)
- The type of the value the method returns(type)
- A list of parameters(parameter-list)
- The body of the method

Constructors

- Java supports a special type of method called a constructor, that enables an object to initialize itself when created.
- Constructors are used to initialize instance variables.

Constructor	Method
Constructor's are used to initialize instance variables	Methods are used to do general purpose calculation
Constructor Name and Class name should be same	Constructor name and Class name may or may not same
Constructor should have neither return type or void	Method should have either return type or void
Constructors are invoked at the time of object creation	Methods are invoked after object is created.

Nested Classes/Interfaces

- Classes and interfaces defined within another class.

```
public class OuterClass
{
   public class InnerClass
{
      public void display()
 {
```

```
            System.out.println("Inner Class");
        }
    }
}
```

Example:

```
class OuterClass
{
    static int x = 10;
    int y = 20;
    private static int z = 30;
    static class Innerclass
    {
        void display()
        {
            System.out.println("x = " +x);
            System.out.println("z = "+z);
            OuterClass obj = new OuterClass();
            System.out.println("y = " + obj.y);


        }
    }
}
public class Demo {
    public static void main(String args[])
    {
        // accessing a static nested class
        OuterClass.Innerclass obj1= new OuterClass.Innerclass();

        obj1.display();
    }
```

Declaration of Class Objects

Creating an instance of a class is called declaring a class object.

Person person = new Person("John", 30);

Assigning One Object to Another

Assigning one object to another makes both references point to the same object in memory.

```
public class Person
{
   public String name;
   public int age;

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

   // Method to display person's details

  public void display()
{
      System.out.println("Name: " + name + ", Age: " + age);
   }

   public static void main(String[] args)
{
      // Create a Person object

      Person person1 = new Person("Alice", 25);

      System.out.println("Details of person1:");

      person1.display();

      // Assign person1 to person2

      Person person2 = person1;
```

```
        System.out.println("Details of person2 (after assignment):");

        person2.display();

        // Modify person2's details

        person2.name = "Bob";

        person2.age = 30;

        // Display details of both person1 and person2

        System.out.println("Details of person1 (after modifying person2):");
        person1.display();
        System.out.println("Details of person2 (after modifying person2):");
        person2.display();
    }
}
```

Access Control for Class Members

Access control determines the visibility of class members.
Java provides four access levels:

1. public: Accessible from any other class.
2. protected: Accessible within the same package and subclasses.
3. default (no modifier): Accessible only within the same package.
4. private: Accessible only within the same class.

1. Public Access Modifier

The public modifier allows class members to be accessed from any other class.

```
import java.util.*;
 public class Demo5
{
        public int a = 10;

        public void display()
        {
                System.out.println("Public  method");

        }
}

class Maindemo
{
          public static void main(String[] args)
        {
                Demo obj = new Demo();
                System.out.println(obj.a); // Accessible
                obj.display(); // Accessible
```

```
        }
}
```

2. Protected Access Modifier

The protected modifier allows class members to be accessed within the same package and subclasses.

```
import java.util.*;
class Demo6
{
        protected int a = 20;

        protected void display()
        {
                System.out.println("Protected  method");
        }
}

class Demo7 extends Demo6
{
        void display1()
        {
                System.out.println(a); // Accessible
                display(); // Accessible
        }
}

class Maindemo1
{
        public static void main(String[] args)
        {
                Demo7 obj = new Demo7();
                obj.display1(); // Access protected members via subclass
         }
}
```

Default Access Modifier

The default access modifier (no modifier) allows class members to be accessed only within the same package.

```
import java.util.*;
class DemoDefault
{
   int a = 30; // Accessible only within the same package
   void display()
  {
            System.out.println("Default  method");
            }
}

class TestDefault {
   public static void main(String args[])
  {
                  DemoDefault obj = new DemoDefault();
            System.out.println("Default Field: " + obj.a);
            obj.display(); // Accessible
            }
}
```

4.Private Access Modifier

The private modifier allows class members to be accessed only within the same class.

```
  import java.util.*;

  public class Demoprivate
  {
    private int a = 40; // Accessible only within the same class
   private void display()
            {
                  System.out.println("Private  method");
            }

            public void display1()
            {
             System.out.println("Private Field: " + a);
             display(); // Accessible within the same class
            }
  }
```

```
class Testprivate
{
        public static void main(String[] args)
        {
                Demoprivate obj = new Demoprivate();
                obj.display1(); // Accesses private members through public method
        }
}
```

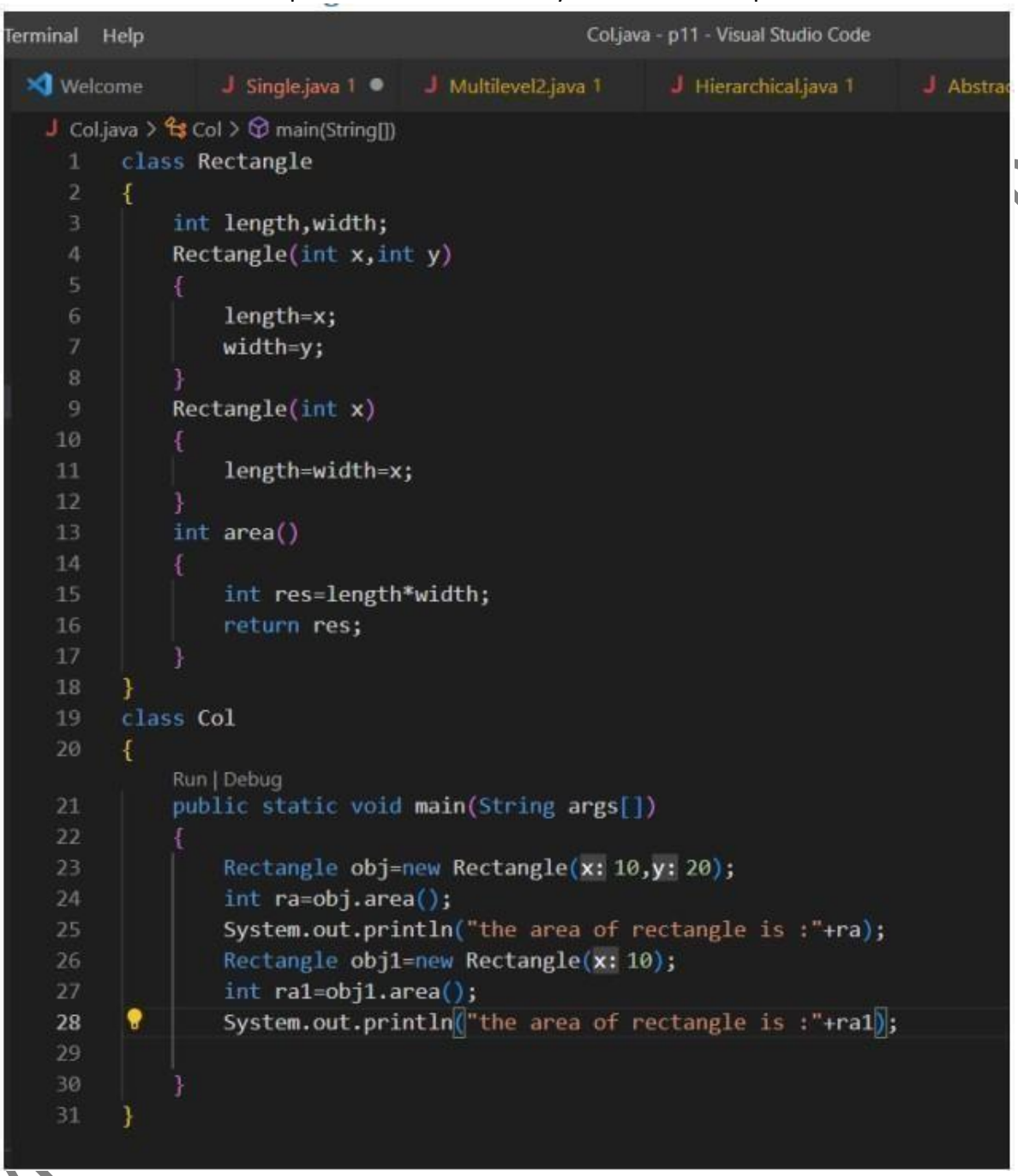
This Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor

```
class Main
{
  int age;
  Main(int age)
  {
        this.age = age;
  }
  public static void main(String[] args)
  {
    Main obj = new Main(8);
    System.out.println("obj.age = " + obj.age);
  }
}
```

Constructor Overloading

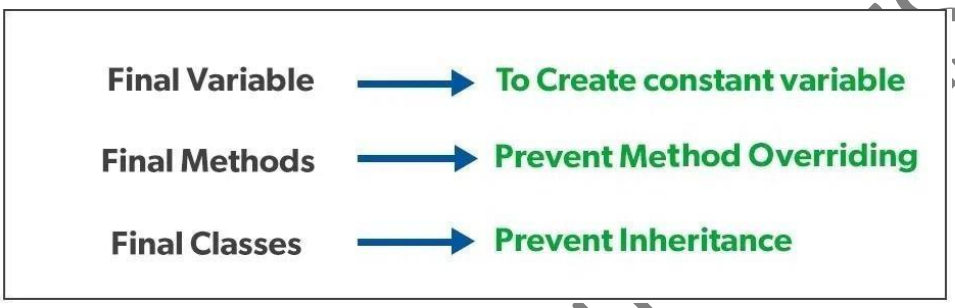
The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task



Final Class and method

The **final method** in Java is used as a **non-access modifier** applicable only to a **variable**, a **method**, or a **class.** It is used to **restrict a user** in Java**.**
The following are **different contexts** where the final is used:

1. Variable
2. Method
3. Class



Parameter Passing In Java

- There are different ways in which parameter data can be passed into and out of methods and functions.
- Let us assume that a function B() is called from another function A().
- In this case A is called the "caller function" and B is called the "called function or callee function". Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of parameters

Formal Parameter: A variable and its type as they appear in the prototype of the function or method.

Syntax:
function _ name(datatype var _ name);

Actual Parameter
The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

fun _ name(var _ name(s));

Call By Value:

- Changes made to formal parameter do not get transmitted back to the caller.
- Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.
- This method is also called as call by value

Call by reference:

- Changes made to formal parameter do get transmitted back to the caller through parameter passing.
- Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.
- This method is also called as **call by reference**. This method is efficient in both time and space.

Introduction to Methods

Methods in Java are blocks of code that perform specific tasks and are typically defined within classes. They encapsulate behavior and promote code reusability and modularity.

```
public class MethodsExample
{

   // Method to print a greeting message
   public void greet()
         {
           System.out.println("Hello, welcome to Java methods!");
         }

   // Method with parameters to calculate the sum of two numbers
   public int sum(int a, int b)
         {
           return a + b;
         }

   public static void main(String[] args)
```

```
        {
        MethodsExample example = new MethodsExample();

    // Calling the greet method
    example.greet();

    // Calling the sum method
    int result = example.sum(5, 3);
    System.out.println("Sum: " + result);
  }
}
```

Overloaded Methods

Overloaded methods are methods in the same class with the same name but different parameter lists (number or types), allowing flexibility in method invocation.

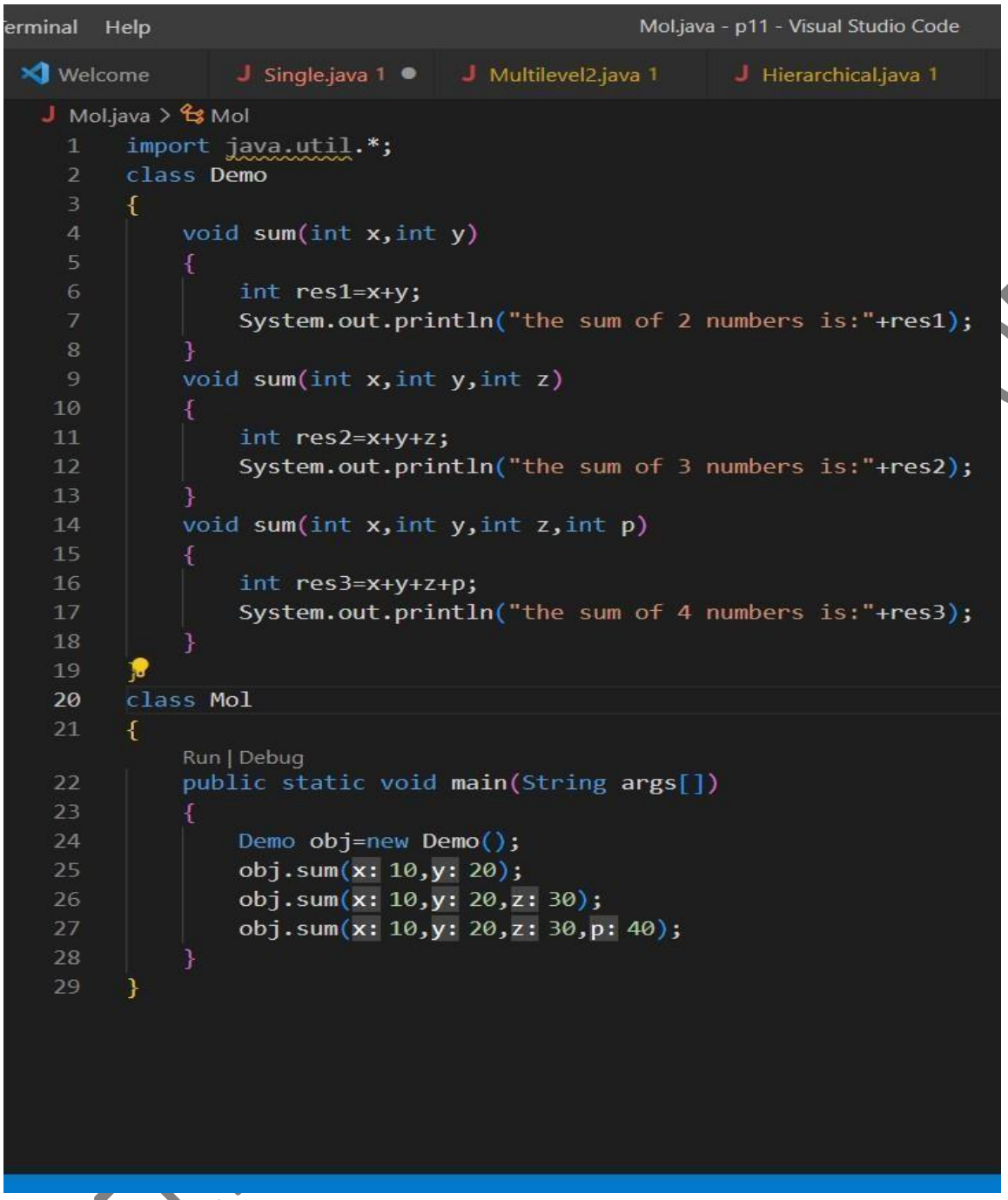
```
public class OverloadedMethodsExample
{

  // Method to add two integers
  public int add(int a, int b)
        {
              return a + b;
         }

  // Overloaded method to add three integers
  public int add(int a, int b, int c)
        {
              return a + b + c;
         }

  public static void main(String[] args)
        {
              OverloadedMethodsExample example = new OverloadedMethodsExample();

              System.out.println("Sum of two numbers: " + example.add(5, 3));
              System.out.println("Sum of three numbers: " + example.add(5, 3, 2));
        }
}
```




```
[Running] cd "f:\p11\" && javac Mol.java && java Mol
the sum of 2 numbers is:30
the sum of 3 numbers is:60
the sum of 4 numbers is:100

[Done] exited with code=0 in 1.602 seconds
```

Method overriding

Terminal Help — Mor.java - p11 - Visual Studio Code

Welcome | Single.java 1 | Multilevel2.java 1 | Hierarchical.java 1

Mor.java > ...

```java
import java.util.*;
class Rectangle
{
    double area(double l,double b)
    {
        double res=l*b;
        return res;
    }
}
class Triangle extends Rectangle
{
    double area(double l,double b)
    {
        double res1=0.5*l*b;
        return res1;
    }
}
class Mor
{
    Run | Debug
    public static void main(String args[])
    {
        Triangle obj1=new Triangle();
        double ta=obj1.area(l: 4,b: 5);
        System.out.println("the area of triangle is"+ta);
    }
}
```

```
[Running] cd "f:\p11\" && javac Mor.java && java Mor
the area of triangle is10.0

[Done] exited with code=0 in 3.822 seconds
```

Recursive Methods

Recursive methods call themselves directly or indirectly, useful for solving problems where a method repeats its behavior.

```
public class RecursiveMethodExample
{

   // Recursive method to calculate factorial
   public int factorial(int n)
         {
                  if (n == 0 || n == 1)
                  {
                           return 1;
                  }
                   else
                  {
                           return n * factorial(n - 1);
                   }
         }

   public static void main(String[] args)
         {
                  RecursiveMethodExample example = new RecursiveMethodExample();

                  // Calculate factorial of 5
                   int result = example.factorial(5);
                   System.out.println("Factorial of 5: " + result);
            }
}
```

Arrays: Introduction, Declaration and Initialization of Arrays, Storage of Array in Computer Memory, Accessing Elements of Arrays, Operations on Array Elements, Assigning Array to Another Array, Dynamic Change of Array Size, Sorting of Arrays, Search for Values in Arrays, Class Arrays, Two-dimensional Arrays, Arrays of Varying Lengths, Three-dimensional Arrays, Arrays as Vectors.

Inheritance: Introduction, Process of Inheritance, Types of Inheritances, Universal Super Class- Object Class, Inhibiting Inheritance of Class Using Final, Access Control and Inheritance, Multilevel Inheritance, Application of Keyword Super, Constructor Method and Inheritance, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Interfaces and Inheritance.

Interfaces: Introduction, Declaration of Interface, Implementation of Interface, Multiple Interfaces, Nested Interfaces, Inheritance of Interfaces, Default Methods in Interfaces, Static Methods in Interface, Functional Interfaces, Annotations.

Arrays

- An array is a group of continuous or related items that share a common name.
- For instance, we can define an array name salary to represent a set of salaries of a group of employees.
- A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

One –Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.

Declaration of Array :

Arrays in java may be declared in two forms

Form1

type arrayname[ ];

Form2

type[ ] arrayname;

Creating Arrays :

you can create an array by using the new operator by using syntax

Syntax:

arrayname=new type[array_Size];

It creates an array using new type[array_Size]

It assigns the reference of the newly created array to the variable arrayname.

Declaring , Creating and assigning an array to the variable can be combined in one statement as:

type[]=arrayname=new type[array_Size];

(or)

type[] arrayname={value0, value 1,.....value k};

Array indices are start from 0 to arrayname. length-1

Two Dimensional Array :

It is used to store two dimensional data. It is also used to store data, which contains rows and columns.

If the data is linear we can use one dimensional array to work with multi-level data we have to use Multi-Dimensional Array.

Creating Two Dimensional Array :

Data_Type[][] Array_Name=new int[Row_Size][Column_Size];

Initialization of Two Dimensional Array :

We can initialize the Two Dimensional Array in some ways

Example :

int[][] Student_Marks = new int[2][3];

int[][] Employees = { {10,20,30}, {15,25,35}, {22,44,66}, {33,55,77} };

Accessing Elements of Arrays

Accessing Elements of a One-Dimensional Array

```
Class ArrayExample
{
   public static void main(String[] args)
          {
              // Declare and initialize a one-dimensional array
              Int data [ ] = {5, 10, 15, 20, 25};

      // Access and print individual elements
      System.out.println("First element: " + data[0]);  // Output: 5
      System.out.println("Second element: " + data[1]); // Output: 10
      System.out.println("Third element: " + data[2]);  // Output: 15
      System.out.println("Fourth element: " + data[3]); // Output: 20
      System.out.println("Fifth element: " + data[4]);  // Output: 25
   }
}
```

Accessing Elements of a two-Dimensional Array

```
class Access2DArray
{
  public static void main(String[] args)
 {
     // Declare and initialize a two-dimensional array
      Int matrix[ ] [ ] = {
                                {1, 2, 3},
                                {4, 5, 6},
                                {7, 8, 9}
                            };

     // Access and print individual elements
     System.out.println("Element at row 0, column 0: " + matrix[0][0]); // Output: 1
     System.out.println("Element at row 0, column 1: " + matrix[0][1]); // Output: 2
     System.out.println("Element at row 1, column 2: " + matrix[1][2]); // Output: 6
     System.out.println("Element at row 2, column 1: " + matrix[2][1]); // Output: 8
     System.out.println("Element at row 2, column 2: " + matrix[2][2]); // Output: 9
```

Storage of Array in Computer Memory

In computer memory, arrays are stored in a contiguous block of memory. The array elements are stored sequentially in memory, meaning that each element is placed directly after the previous one. This arrangement allows for efficient access to any element in the array using an index, making arrays a popular data structure in programming languages like Java.

How Arrays Are Stored in Memory:

1. **Contiguous Memory Allocation**:
   - Arrays are stored in a **continuous block of memory**. The size of this memory block is calculated based on the data type of the array and the number of elements.
   - If the array is an integer array, for example, each element will take up 4 bytes of memory (assuming a 32-bit integer), and the total memory size will be 4 * n, where n is the number of elements.
2. **Indexing**:
   - Elements in an array are accessed using an index. The index is used to calculate the memory address of the element.
   - For example, in a one-dimensional array, the memory address of the element at index i is calculated as:

     Address_of_element(i) = Base_address + (i * size_of_element)

where Base_address is the memory address of the first element in the array, i is the index, and size_of_element is the size of each array element in bytes.

3. **Memory Layout Example**:
   - Consider the following integer array:

     ```
     int[] arr = {10, 20, 30, 40, 50};
     ```

   - If the array is stored starting at memory address 1000, the elements are laid out in memory as:

     Address Value

     1000 10 (arr[0])

     1004 20 (arr[1])

     1008 30 (arr[2])

     1012 40 (arr[3])

     1016 50 (arr[4])

   - Here, each element takes 4 bytes (since it's an int), and the elements are stored consecutively.

Multi-Dimensional Arrays:

- In the case of multi-dimensional arrays (e.g., 2D arrays), the elements are stored in **row-major order** in Java. This means that the elements of each row are stored sequentially in memory.
- Consider a 2D array:

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

In memory, this array would be laid out as:

Address Value

1000 1 (matrix[0][0])

1004 2 (matrix[0][1])

1008 3 (matrix[0][2])

1012 4 (matrix[1][0])

1016 5 (matrix[1][1])

1020 6 (matrix[1][2])

1024 7 (matrix[2][0])

1028 8 (matrix[2][1])

1032 9 (matrix[2][2])

- row are stored first, followed by the elements of the second row, and so on.

Types of Arrays and Memory Allocation:

1. **Primitive Arrays**:
   - Arrays that store primitive types like int, char, float, etc., store the actual values in contiguous memory.
   - Example:

     ```
     int[] arr = {1, 2, 3};
     ```

     Each int value (4 bytes) is stored contiguously in memory.
2. **Object Arrays**:
   - Arrays that store object references (e.g., arrays of String or user-defined objects) do not store the actual objects in contiguous memory.
   - Instead, the array stores references (memory addresses) to the objects, which may be located anywhere in memory.
   - Example:

     ```
     String[] arr = {"Apple", "Banana", "Cherry"};
     ```

     The array arr contains references to String objects, and those strings are stored at different memory locations.

Advantages of Contiguous Memory Storage:

1. **Efficient Indexing**:
   - Since arrays are stored in contiguous memory, the memory address of any element can be calculated quickly using the index. This makes accessing elements very fast (O(1) time complexity).
2. **Cache-Friendly**:
   - Contiguous memory storage takes advantage of CPU caching. When an element of an array is accessed, nearby elements are likely loaded into the cache, speeding up future access.

Disadvantages:

1. **Fixed Size**:
   - array size is too large or too small.
2. **Inefficient Insertion/Deletion**:
   - Inserting or deleting elements in the middle of an array requires shifting elements, which can be slow (O(n) time complexity).

Operations on Array Elements

In Java, you can perform various operations on array elements, such as arithmetic operations, traversals, modifications, and more. Below are some examples of common operations performed on array elements.

Sum of All Elements in an Array

```
class ArrayOperations
{
   public static void main(String[] args)
{
      int numbers[ ] = {10, 20, 30, 40, 50};
      int sum = 0;
      // Loop through the array to calculate the sum of elements
      for (int i = 0; i < numbers.length; i++)
      {
            sum = sum+ numbers[i];
      }

      System.out.println("Sum of all elements: " + sum);
   }
}
```

Finding the Maximum Element in an Array

```
class ArrayOperations
{
   public static void main(String[] args)
{
      Int    numbers[ ] = {10, 20, 30, 40, 50};
      int max = a[i];

      // Loop through the array to find the maximum element
      for (int i = 1; i < numbers.length; i++)
{
                  if (numbers[i] > max)
 {
                  max = numbers[i];
```

```
}
```

```
        System.out.println("Maximum element: " + max);
    }
}
```

Finding the Minimum Element in an Array

```
class ArrayOperations {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int min = numbers[0];

        // Loop through the array to find the minimum element
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] < min) {
                min = numbers[i];
            }
        }

        System.out.println("Minimum element: " + min);
    }
}
```

Get the First and Last Element of an Array

To get the first and last elements of an array, you need to access the elements at index 0 (for the first element) and index array.length - 1 (for the last element). Here are examples in different programming languages:

```
public class ArrayFirstLastElement {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};

        // Get the first element
        int firstElement = arr[0];

        // Get the last element
```

```
int lastElement = arr[arr.length - 1];
```

```
        System.out.println("First element: " + firstElement);
        System.out.println("Last element: " + lastElement);
    }
}
```

Output:

First element: 10

Last element: 50

To compare two arrays in Java, you need to determine if they are equal in terms of their content and order. You can use the Arrays class from the java.util package, which provides utility methods for comparing arrays.

Here's how you can compare two arrays:

Using Arrays.equals()

The Arrays.equals() method checks if two arrays are equal by comparing their length and corresponding elements.

Example Code

```
import java.util.Arrays;

public class CompareArrays {
    public static void main(String[] args) {
        int[] array1 = {1, 2, 3, 4, 5};
        int[] array2 = {1, 2, 3, 4, 5};
        int[] array3 = {1, 2, 3, 4, 6};

        // Compare array1 and array2
        boolean areEqual1 = Arrays.equals(array1, array2);
        System.out.println("array1 and array2 are equal: " + areEqual1);

        // Compare array1 and array3
        boolean areEqual2 = Arrays.equals(array1, array3);
        System.out.println("array1 and array3 are equal: " + areEqual2);
```

```
  }
}
```

Assigning One Array To Another Array

```
public class CopyArray {
   public static void main(String[] args) {
      // Initialize the original array
      int[] arr1 = new int[] {1, 2, 3, 4, 5};

      // Create another array arr2 with the same size as arr1
      int[] arr2 = new int[arr1.length];

      // Copy all elements from arr1 to arr2
      for (int i = 0; i < arr1.length; i++) {
         arr2[i] = arr1[i];
      }

      // Displaying elements of the original array
      System.out.println("Elements of the original array: ");
      for (int i = 0; i < arr1.length; i++) {
         System.out.print(arr1[i] + " ");
      }

      System.out.println();

      // Displaying elements of the new array
      System.out.println("Elements of the new array: ");
      for (int i = 0; i < arr2.length; i++) {
         System.out.print(arr2[i] + " ");
      }
   }
}
```

Dynamic change of arrays

In Java, arrays have a fixed size once they are created. If you need a dynamically sized collection, you'll want to use ArrayList from the java.util package, which provides dynamic resizing capabilities. Here's how you can use ArrayList:

Using ArrayList in Java

1. **Import ArrayList**: Make sure to import the ArrayList class:

   ```
   import java.util.ArrayList;
   ```

2. **Create an ArrayList**: You can create an ArrayList and use it similarly to an array, but with dynamic resizing:

```
public class Main {
  public static void main(String[] args) {
    // Create an ArrayList of integers
    ArrayList<Integer> myList = new ArrayList<>();

    // Add elements
    myList.add(1);
    myList.add(2);
    myList.add(3);

    // Remove an element
    myList.remove(Integer.valueOf(2)); // Removes the element with value 2

    // Print the elements
    for (int num : myList) {
      System.out.print(num + " ");  // Output: 1 3
    }
  }
}
```

Common Operations with ArrayList:

Adding Elements:

```
myList.add(4);  // Adds 4 to the end of the list
```

```
myList.add(1, 5);  // Adds 5 at index 1
```

Removing Elements:

```
myList.remove(2);  // Removes the element at index 2
myList.remove(Integer.valueOf(3));  // Removes the first occurrence of the value 3
```

Accessing Elements:

```
int element = myList.get(0);  // Gets the element at index 0
```

-

Iterating Over Elements:

```
for (int i = 0; i < myList.size(); i++) {
 System.out.println(myList.get(i));
}
```

Getting Size:

```
int size = myList.size();  // Gets the number of elements in the list
```

Clearing All Elements:

```
myList.clear();  // Removes all elements from the list
```

ArrayList is a versatile and commonly used collection in Java for managing dynamic-sized list.

Arrays Sorting

Array sorting refers to the process of arranging the elements of an array in a specific order, typically in ascending or descending order. In Java, there are several ways to sort arrays, including using built-in methods or implementing custom sorting algorithmS

```
public class SortArrayExample2
{
    public static void main(String[] args)
    {
    //creating an instance of an array
    int[] arr = new int[] {4,2,3,1};
    System.out.println("Array elements after sorting:");
    //sorting logic
    for (int i = 0; i < arr.length; i++)
    {
    for (int j = i + 1; j < arr.length; j++)
```

{

```
        int tmp = 0;
        if (arr[i] > arr[j])
        {
        tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
        }
        }
        //prints the sorted element of the array
        System.out.println(arr[i]);
        }
        }
}
```

Descending Order

```
public class SortArrayExample2
{
        public static void main(String[] args)
        {
        //creating an instance of an array
        int[] arr = new int[] {78, 34, 1, 3, 90, 34, -1, -4, 6, 55, 20, -65};
        System.out.println("Array elements after sorting:");
        //sorting logic
        for (int i = 0; i < arr.length; i++)
        {
        for (int j = i + 1; j < arr.length; j++)
        {
        int tmp = 0;
```

```
if (arr[i] <  arr[j])
{
```

```
tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp;
}
}
//prints the sorted element of the array
System.out.println(arr[i]);
}
}
}
```

Search for Values in Arrays

To search for values in arrays in Java, you can use various methods depending on the type of search you want to perform. Below are examples of two common types of searches: **linear search** and **binary search**.

Linear Search

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

```
Class LinearSearch
{
 Public static void main (String args[])
```

{

```
Int a[]={10,20,40,50,30};
Int search_ele=50;
Boolean flag=false;
For(int i=0;i<a.length;i++)
{
        If(search_ele==a[i])
        {
                System.out.println("the element is found at :+i);
                flag=true;
                break;
        }
}
If(flag==false)
{
        System.out.println("element is not found");
        }
}
```

search element 12

Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Binary search

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

```
public class BinarySearch {

   public static void main(String[] args) {

      int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  // Should be in sorted order

      boolean flag = false;

      int key = 5;

      int l = 0;
      int h = a.length - 1;

      while (l <= h)
      {
         int m = (l + h) / 2;

         if (a[m] == key) {
            System.out.println("Element Found..");
            flag = true;
            break;
```

}

```
            if (a[m] < key) {
                l = m + 1;
            }

            if (a[m] > key) {
                h = m - 1;
            }
        }

        if (flag == false) {
            System.out.println("Element NOT found..");
        }
    }
}
```

search element 12

Step 1:

search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

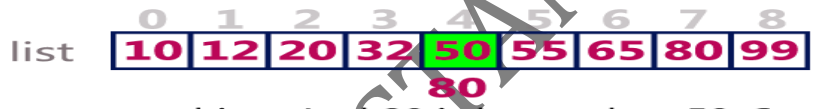
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Arrays as Vectors (Vector Class in Java)

The Vector class in Java implements a dynamic array where elements can be added or removed. It is synchronized, which means it's thread-safe for use in multi-threaded applications. However, because of the synchronization overhead, it's generally slower than ArrayList.

Key Features of Vector:

- Dynamic resizing
- Can hold any type of data
- Supports operations like insertion, deletion, and searching
- Synchronization makes it thread-safe

Declaring and Using a Vector in Java:

```
import java.util.Vector;


public class VectorExample {
   public static void main(String[] args) {
      // Create a Vector to hold integer values
      Vector<Integer> vector = new Vector<>();

      // Adding elements to the Vector
      vector.add(10);
      vector.add(20);
      vector.add(30);
      vector.add(40);
      vector.add(50);

      // Accessing elements using an index
      System.out.println("Element at index 2: " + vector.get(2));  // Output: 30

      // Removing an element at a specific index
      vector.remove(3);  // Removes the element at index 3 (40)

      // Iterating over the elements
      System.out.println("Vector elements after removal:");
```

```
for (int i = 0; i < vector.size(); i++) {
```

```
            System.out.println("Element at index " + i + ": " + vector.get(i));
        }

        // Size of the vector
        System.out.println("Size of the vector: " + vector.size());

        // Checking if the vector contains a specific element
        if (vector.contains(30)) {
            System.out.println("Vector contains 30");
        } else {
            System.out.println("Vector does not contain 30");
        }
    }
}
```

Output:

```
Element at index 2: 30
Vector elements after removal:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 50
Size of the vector: 4
Vector contains 30
```

Arrays Of Varying Lengths

In Java, you can create arrays of varying lengths, also known as **jagged arrays** or **ragged arrays**. A jagged array is an array whose elements are arrays of different lengths, unlike a regular multidimensional array where all rows have the same number of elements.

Declaring and Using Jagged Arrays

When you declare a 2D array, you don't have to specify the size of each row. Instead, you can assign arrays of varying lengths to each row.

Example Program: Arrays of Varying Lengths (Jagged Arrays)

java

Copy code

```
public class JaggedArrayExample {
   public static void main(String[] args) {
      // Declaring a 2D array with 3 rows
      int[][] jaggedArray = new int[3][];

      // Initializing each row with a different number of columns
      jaggedArray[0] = new int[3];  // First row has 3 elements
      jaggedArray[1] = new int[2];  // Second row has 2 elements
      jaggedArray[2] = new int[4];  // Third row has 4 elements

      // Populating the jagged array with values
      int value = 1;
      for (int i = 0; i < jaggedArray.length; i++)
         {
                  for (int j = 0; j < jaggedArray[i].length; j++)
      {
                   jaggedArray[i][j] = value++;
                   }
      }

      // Printing the elements of the jagged array
```

```
System.out.println("Jagged Array Elements:");
for (int i = 0; i < jaggedArray.length; i++)
```

```
        {
                for (int j = 0; j < jaggedArray[i].length; j++)
        {
                        System.out.print(jaggedArray[i][j] + " ");
                }
        System.out.println();  // Move to the next line after each row
    }
  }
}
```

Output:

Jagged Array Elements:

1 2 3

4 5

6 7 8 9

INHERITANCE

- **The mechanism of deriving a new class from an old class such that the new class acquires all the properties of the old class is called Inheritance.**
- The old class is known as Parent, base or Super class and the new class that is derived is known as child, derived or subclass.
- The Inheritance allows subclasses to inherit all the variables and methods of their parent classes.

Defining a Subclass

- A Subclass is defined as follows

Class subclassname extends superclassname

{

Variables declaration

Methods declaration

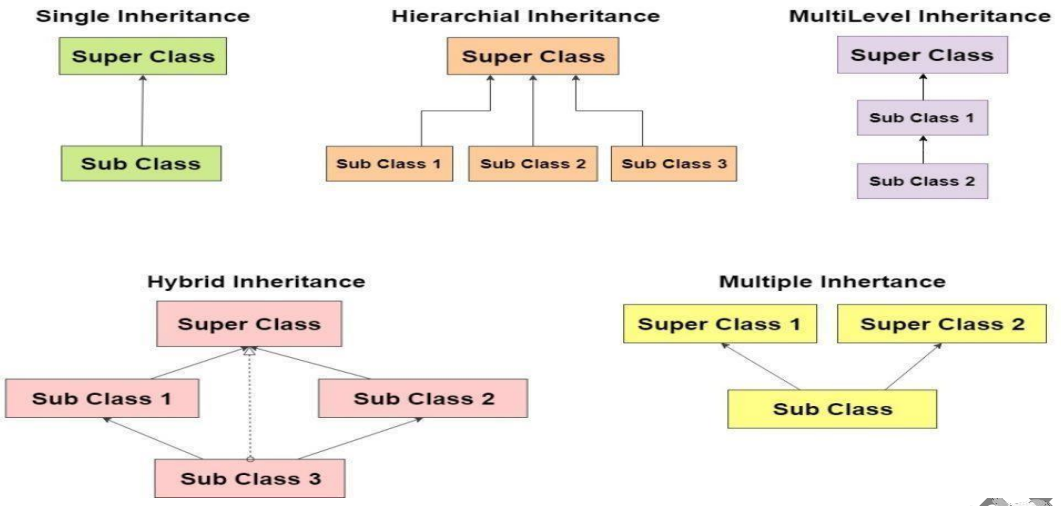
}

- The keyword extends signifies that the properties of the **superclassname** are extended **subclassname.**
- The subclass will now contain its own variables and methods as well those superclass.
- This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

Inheritance may take different types

1. Single inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance (**Does not supports in java**)

- These forms of inheritance are shown as



1. Single inheritance

The process of Creating one Child class from one Parent class is called single inheritance.

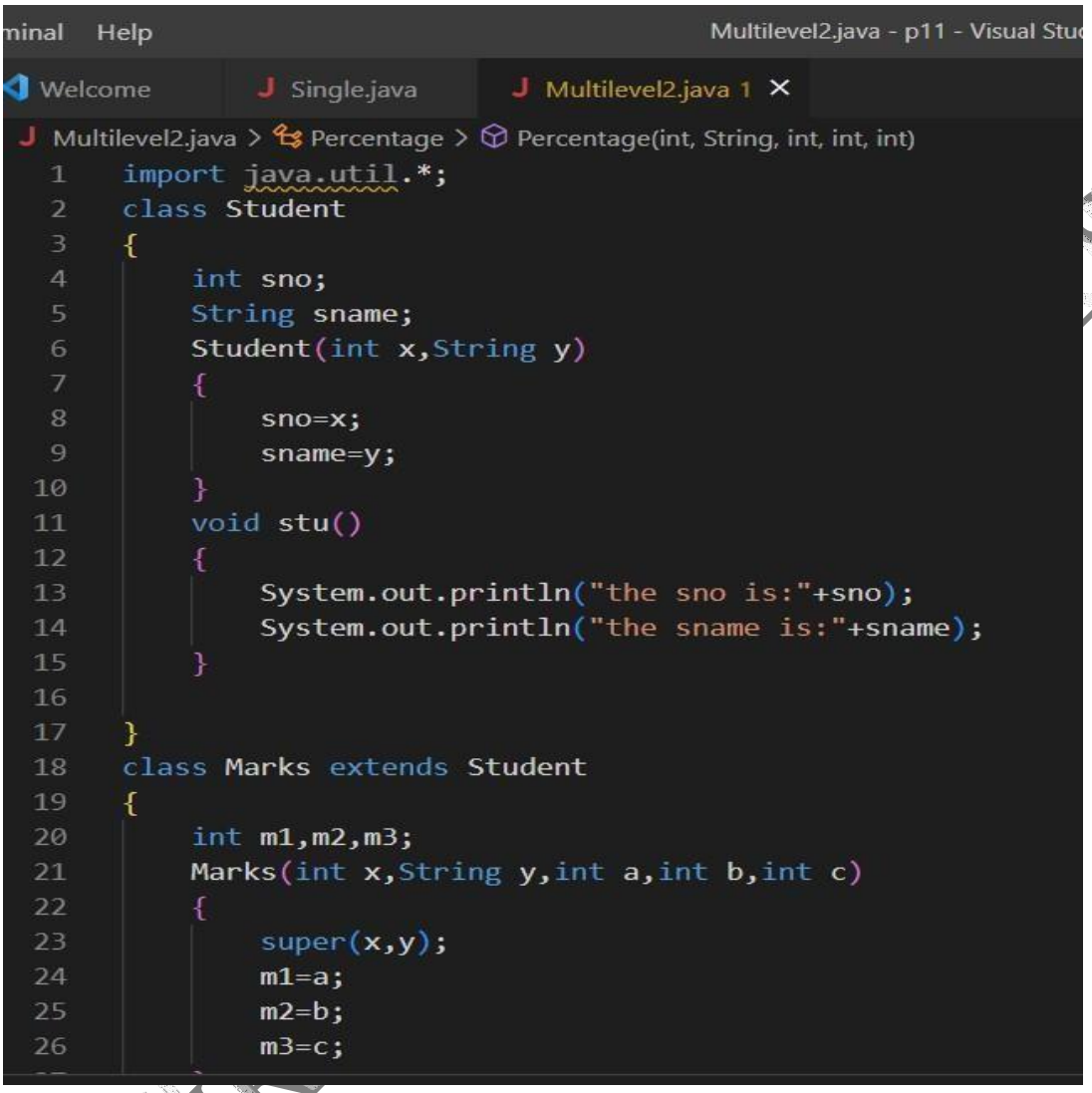
Example:

Single.java > Triangle > tarea()

```
1   class Rectangle
2   {
3       int length,width;
4       Rectangle(int x,int y)
5       {
6           length=x;
7           width=y;
8       }
9       int area()
10      {
11          int res=length*width;
12          return res;
13      }
14  }
15  class Triangle extends Rectangle
16  {
17      int height;
18      Triangle(int x,int y,int z)
19      {
20          super(x,y);
21          height=z;
22      }
23      int tarea()
24      {
25          int res1=area()*height;
26          return res1;
27      }
28  }
29  class Single
30  {
        Run | Debug
31      public static void main(String args[])
32      {
33      Triangle obj=new Triangle(x: 10,y: 5,z: 20);
34      int ra=obj.area();
35      int ta=obj.tarea();
36      System.out.println("the area of rectangle is:"+ra);
37      System.out.println("the area of triangle is:"+ta);
38      }
39  }
```

2. Multilevel Inheritance

Process of deriving a class from another derived class is called multilevel inheritance




```java
import java.util.*;
class Student
{
    int sno;
    String sname;
    Student(int x,String y)
    {
        sno=x;
        sname=y;
    }
    void stu()
    {
        System.out.println("the sno is:"+sno);
        System.out.println("the sname is:"+sname);
    }

}
class Marks extends Student
{
    int m1,m2,m3;
    Marks(int x,String y,int a,int b,int c)
    {
        super(x,y);
        m1=a;
        m2=b;
        m3=c;
```

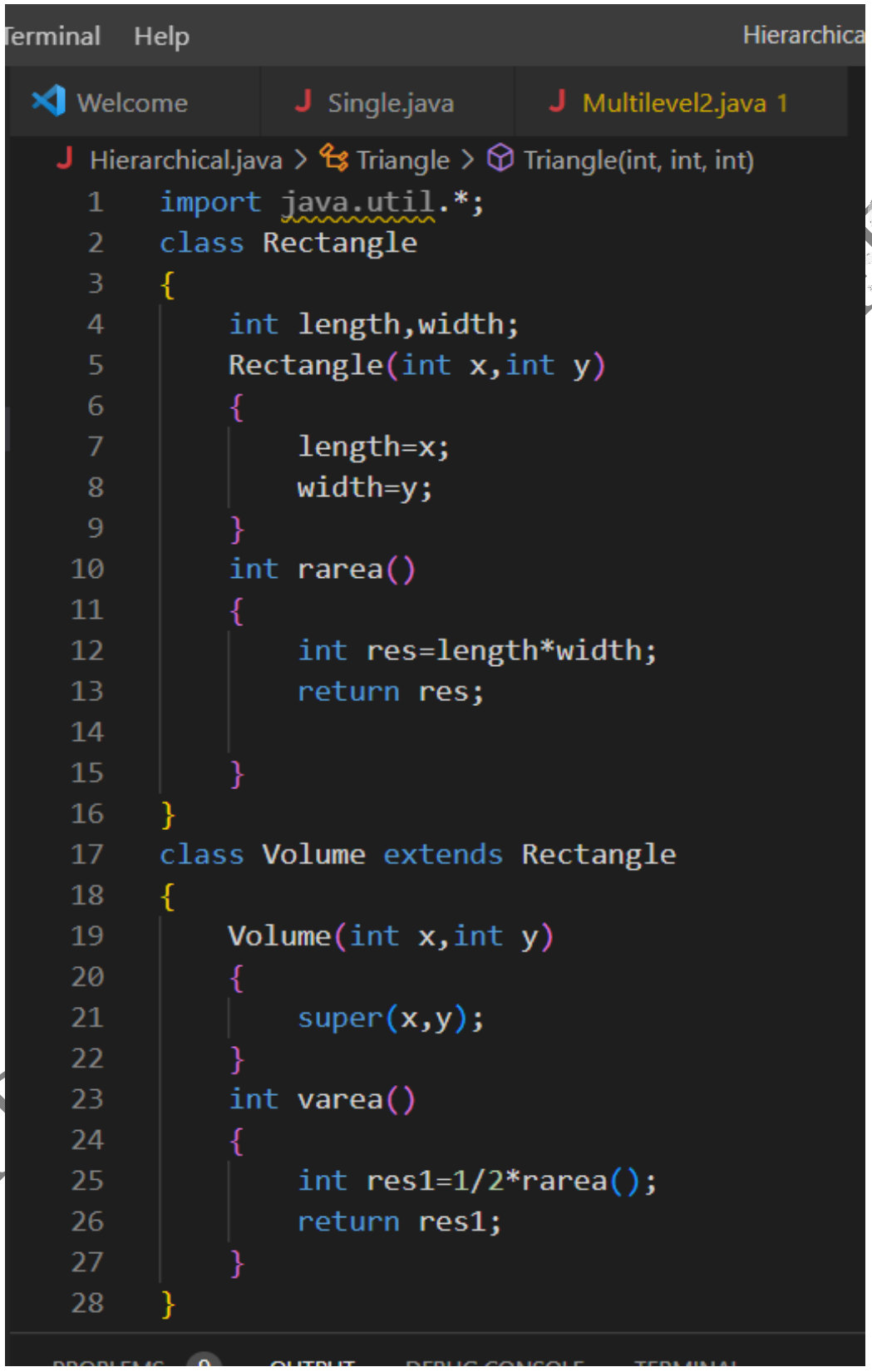
```
26          m3=c;
27      }
28      void stu_marks()
29      {
30          System.out.println("the sub1 marks is:"+m1);
31          System.out.println("the sub2 marks is:"+m2);
32          System.out.println("the sub3 marks is:"+m3);
33      }
34  }
35  class Total extends Marks
36  {
37      Total(int x,String y,int a,int b,int c)
38      {
39          super(x,y,a,b,c);
40
41      }
42      int total()
43      {
44          int tot=m1+m2+m3;
45          return tot;
46      }
47  }
48  class Percentage extends Total
49  {
50      Percentage(int x,String y,int a,int b,int c)
51      {
```

Multilevel2.java > Percentage > Percentage(int, String, int, int, int)

```
51      {
52          super(x,y,a,b,c);
53      }
54      double per()
55      {
56          double avg=total()/3;
57          return avg;
58      }
59  }
60  class Multilevel2
61  {
        Run | Debug
62      public static void main(String args[])
63      {
64          Percentage obj=new Percentage(x: 18,y: "yuvaraju",a: 90,b: 92,c: 93);
65          obj.stu();
66          obj.stu_marks();
67          int tm=obj.total();
68          double pa=obj.per();
69          System.out.println("the student total marks is"+tm);
70          System.out.println("the student total marks is"+pa);
71      }
72  }
```

3. Hierarchical Inheritance

Process of deriving one or more subclasses from one super class is called hierarchical inheritance



```
28  }
29  class Triangle extends Rectangle
30  {
31      int height;
32      Triangle(int x,int y,int z)
33      {
34          super(x,y);
35          height=z;
36      }
37      int tarea()
38      {
39          int res3=rarea()*height;
40          return res3;
41      }
42  }
43  class Hierarchical
44  {
        Run | Debug
45      public static void main(String args[])
46      {
47          Triangle obj1=new Triangle(x: 10,y: 20,z: 30);
48          int ta=obj1.tarea();
49          int ra=obj1.rarea();
50          System.out.println("the area of rectangle is:"+ra);
51          System.out.println("the area of triangle is:"+ta);
52          Volume obj2=new Volume(x: 10,y: 20);
53          int va=obj2.varea();
54          System.out.println("the area of volume is:"+va);
```

4. Hybrid Inheritance

Combination of above any inheritance is called hybrid inheritance

5. Multiple inheritance

Process of deriving a subclass from one or more superclasses is called multiple inheritance.

Java does not directly implement multiple inheritance.

however, this concept is implemented using a secondary inheritance path in the form of interfaces. Class A

{

}

Class B

{

}

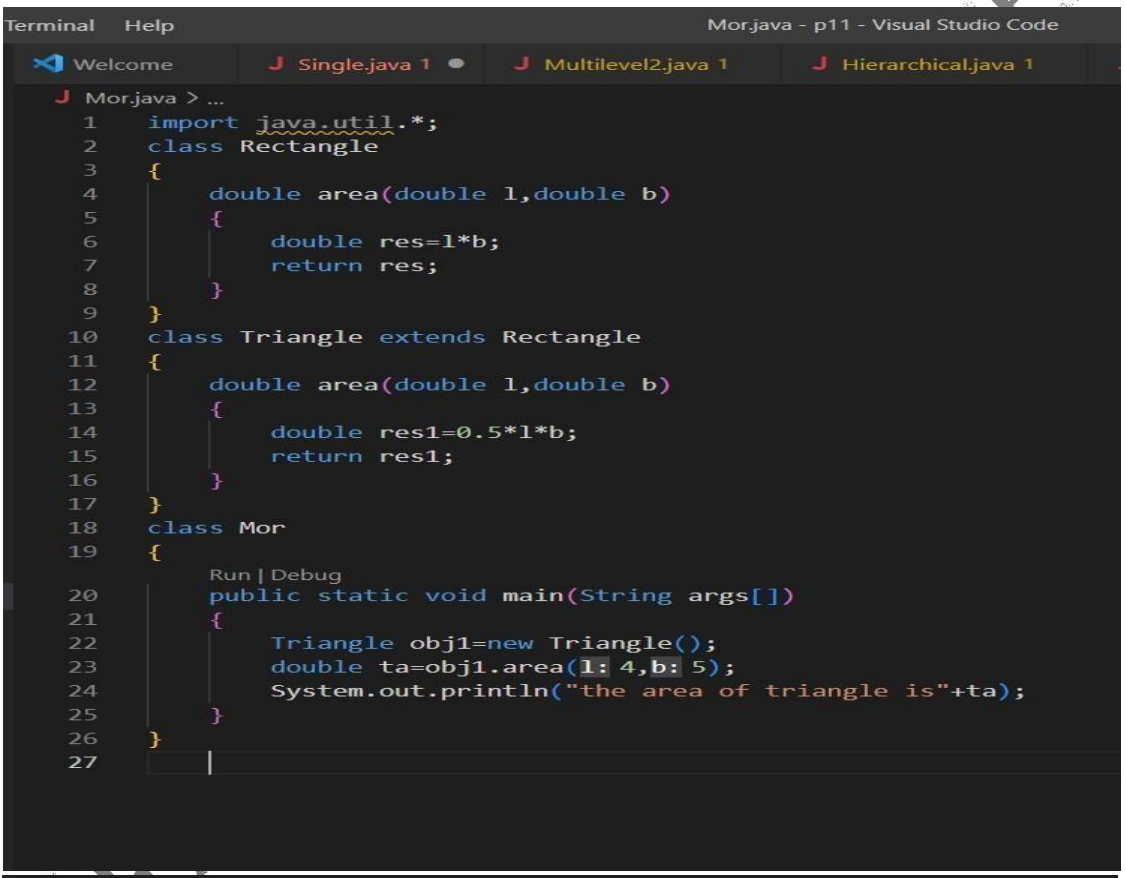
Class C extends A,B // java does not allow this { }

{

}

Method Overrididng

A method in subclass, whose name, parameter list and return type are same as that of the method in superclass is called overrided methods.




```
[Running] cd "f:\p11\" && javac Mor.java && java Mor
the area of triangle is10.0

[Done] exited with code=0 in 3.822 seconds
```

Abstract Methods and Classes

- An Abstract method is a method without method body or a method without implementation.
- An Abstract method is written when the same method has to perform different tasks depending on the object calling it.

Example:

```
class A                          // Automatically Becomes Abstract Class
{
    void m1();                   // Abstract Method
    void m2()                    // Concrete Method
{
    System.out.println("method  2");
}
}
```

- A Class that contains one or more Abstract Methods is called Abstract Class.
- An Abstract class is a class that contains 0 or more Abstract Methods.
- Abstract class can contain instance variables and concrete methods in addition to abstract methods. Since, abstract class contains incomplete methods, it is not possible to estimate the total memory required to create the object.

Example:

```
Abstract class MyClass
{
abstract void calculate(double x);
}
Class Sub1 extends MyClass
{
void calculate(double x)
```

```
{
System.out.println("Square ="+(x*x));
```

```
}
}
Class Sub3 extends MyClass
{
void calculate(double x)
{
System.out.println("Square Root ="+Math.sqrt(x));
}
}
Class Different
{
public static void main(String args[])
{
Sub1 obj1 = new Sub1();
Sub2 obj2 = new Sub2();
Sub3 obj3 = new Sub3();
obj1.calculate(3);
obj2.calculate(4);
obj3.calculate(5);
}
}
```

Example:2

Welcome | Single.java 1 | Multilevel2.java 1 | Hierarchical.java 1 | Abstraction.java

Abstraction.java > Maruthi > breaking()

```
1   import java.util.*;
2   abstract class Car
3   {
4       int regno;
5           Car(int x)
6           {
7               regno=x;
8           }
9           void fulltank()
10          {
11              System.out.println(x: "the car is full tank:");
12          }
13          abstract void steering();
14          abstract void breaking();
15  }
16  class Maruthi extends Car
17  {
18      Maruthi(int x)
19      {
20          super(x);
21
22      }
23      void steering()
24      {
25          System.out.println(x: "the maruthi car is normal steering:");
26      }
27      void breaking()
28      {
29          System.out.println(x: "the  maruthi car is ready to breaking:");
30      }
```

erminal Help Abstraction.java - p11 - Visual Studio Code

Welcome | Single.java 1 | Multilevel2.java 1 | Hierarchical.java 1 | Abstraction.java

Abstraction.java > Maruthi > breaking()

```
32  class Santro extends Car
33  {
34      Santro(int x)
35      {
36          super(x);
37
38      }
39      void steering()
40      {
41          System.out.println(x: "the santro car is power steering");
42
43      }
44      void breaking()
45      {
46          System.out.println(x: "santro car is hydralic breaking:");
47      }
48  }
49  class Abstraction
50  {
        Run | Debug
51      public static void main(String args[])
52      {
53          Santro obj=new Santro(x: 10);
54          obj.fulltank();
55          obj.steering();
56          obj.breaking();
57          Maruthi obj1=new Maruthi(x: 20);
58          obj1.fulltank();
59          obj1.steering();
60          obj1.breaking();
61
62      }
63  }
```

```
[Running] cd "f:\p11\" && javac Abstraction.java && java Abstraction
the car is full tank:
the santro car is power steering
santro car is hydralic breaking:
the car is full tank:
the maruthi car is normal steering:
the  maruthi car is ready to breaking:

[Done] exited with code=0 in 2.407 seconds
```

final class : prevents inheritance

sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclasses is called a final class. Any attempt to inherit final classes will cause an error and the compiler will not allow it.

```
final class A
{
}
class B extend A                    //error, cannot inherit a because it is a final class
{
}
```

Interfaces:

Defining an Interface

- An Interface is basically a kind of class
- Like classes, interface contain methods and variables but with a major difference.
- ***The difference is that interfaces de□ine only***
  - ***Abstract Method &***
  - ***Final and Static Variables***
- i.***e methods*** are declared without any body ***and variables*** are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized.
- All ***Methods*** and ***Variables*** in the interface are implicitly ***public.***

The syntax for defining an interface is very similar to that of defining a class

```
Interface InterfaceName
{
static and final Variables

Abstract Methods
}
```

Where ***Interface*** is the keyword and ***InterfaceName*** is any valid java variable

Example:

```
Interface Item
{
static final int code = 1001;
static final String name = "Fan";

void display();
}
```

Implementing Interface

- ✓ An Interface will have 0 or more abstract methods which are all **public and abstract by default.**
- ✓ An Interface can have variables **which are public, static and final by default, means all the variables of the interface are constants**.
- ✓ Objects cannot be created to an interface whereas reference can be created.
- ✓ ***Once interface is de□ined, any number of classes can implement an interface.***
- ✓ ***Also one class can implement any number of interfaces.***
- ✓ To Implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ To implement an interface, include the ***implements clause*** in a class definition, and then create the methods defined by the interface.
- ✓ General form of a class that includes the implements clause looks like

Class ClassName [extends SuperClass] [implements Interface1[,... Interface N]]

{

// class body

}

Example

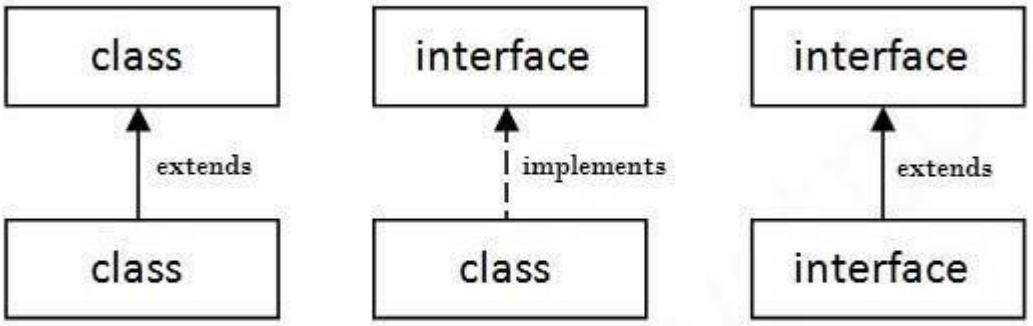
Class A Extends B Implements I1,I2

{

}

- ✓ i.e if a class implements more than one interface, the interfaces are separated with a comma.

The Relationship between classes and Interfaces are



Example:

```
Interface Bank
{
float rateOfInterest();
}
Class SBI implements Bank
{
public float reateOfInterest()
{
    return (7.8f);
}
}
class ICICI implements Bank
{
public float reateOfInterest()
{
return (9.8f);
```

```
}
}
class IB implements Bank
{
public float reateOfInterest()
{
return (8.8f);
}
}
class InterfaceDemo
{
public staticvoid main(String args[])
{
SBI obj1 = new SBI();
float sbi_roi = obj1.rateOfInterest();

ICICI obj1 = new ICICI();
float icici_roi = obj1.rateOfInterest();
IB obj1 = new IB();
float ib_roi = obj1.rateOfInterest();
System.out.println("SBI rate of Interest is "+ sbi_roi);
System.out.println("ICICI rate of Interest is "+ sbi_icici);
System.out.println("IB rate of Interest is "+ sbi_ib);
}
}
```

Interfaces can be Extended

- ✓ Like classes, interface can also be extended.
- ✓ i.e an interface can be sub interfaced from other interfaces.
- ✓ The new sub interface will inherit all the members of the super interface in the manner similar to subclasses.
- ✓ This is achieved using the keyword "extends".

- ✓ General form of extending interfaces is

Syntax:

Interface NameNew extends name1[,...nameN]

{

Body of Interface

}

Example:

```
interface A
{
void meth1();

void meth2();
}
interface B extends A
{
void meth3();
}
Class  MyClass implements B
{
public void meth1()
{
System.out.println("implementing meth1().....");
}
public void meth2()
{
System.out.println("Implementing meth2()....");
}
public void meth3()
{
System.out.println("Implementing meth3()....");
}
```

```
}
Class InterfaceDemo
{
Public static void main(string args[])
{
MyClass obj = new MyClass();
obj.meth1();
obj.meth2();
obj.meth3();
}
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

  Note : if a class that implements an interface and the class does not give implementations to all the methods of the interface, then the class becomes an abstract class and cannot be instantiated

Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto- unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java.time.Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions.

Java I/O and File: Java I/O API, standard I/O streams, types, Byte streams, Character streams, Scanner class, Files in Java

1. Introduction to Packages in Java

In Java, a **package** is a mechanism for organizing Java classes, interfaces, and sub-packages into namespaces. Packages act like containers that group related classes and interfaces, helping to avoid naming conflicts and managing large codebases efficiently.

Key Benefits of Using Packages:

1. **Namespace Management**: Packages help in organizing classes and interfaces into different namespaces, which prevents naming conflicts. For example, you can have two classes with the same name in different packages without causing any conflicts.
2. **Access Control**: Packages allow the application of access control. Classes, methods, and fields can be declared `public`, `protected`, `private`, or package-private (default), controlling how they are accessed from other packages or within the same package.
3. **Code Reusability**: Packages make it easier to reuse classes across different projects or parts of a project. You can easily import them into other programs and extend their functionality.
4. **Logical Grouping**: Grouping related classes together makes it easier to maintain and manage code. It also provides structure, making the code more readable and understandable.

package:

- Package a group of similar types of classes and interfaces and subpackages

Or

- Package is a folder that contains collection of related classes and interfaces.
- In java ,packages can be categorized into two types
- 1.Built-in packages

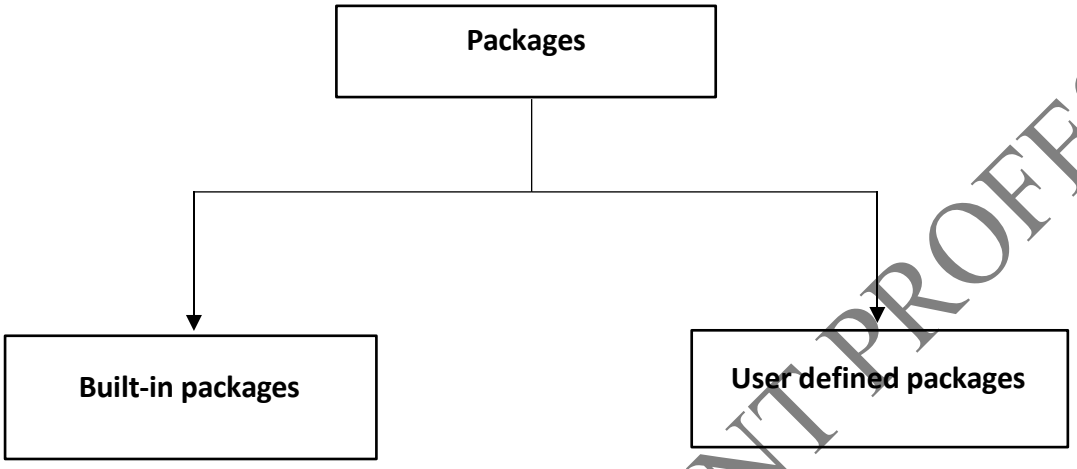
  2.user-defined packages

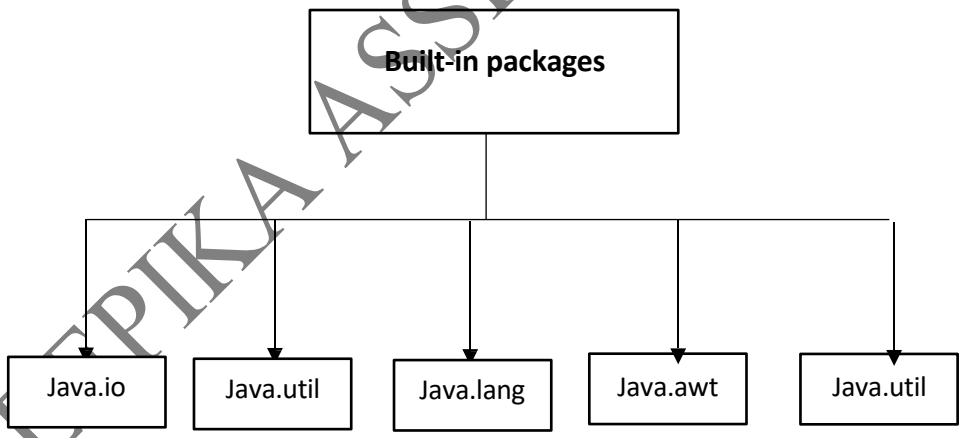
Built-in packages

In java we have various built-in packages that are already created by java people and these packages contain large number of classes and interfaces

User defined packages

As the name suggests user-defined packages are a package that is defined by theuser or programmer.





Java.io	Contains classes related to input/output operations
Java.util	Contains classes and interfaces of collection framework, scanner class
Java.lang	Contains fundamental classes like system, object etc for designing java program
Java.awt	Contains classes and interfaces for creating graphical components
Java.swing	Contain classes and interfaces for creating graphical components

Advantages

- Java package is used to categorized the classes and interfaces so that theycan be easily categorized.
- Java package provides access protection
- Java package helps to avoid name space collision.

How to create user defined packages

- To create the package should be starts with the keyword is **package**

  Syntax: package package_name;
- It should not contain main class
- Multiple programs should be written for placing multiple classes in samepackage.

Steps to create a simple user defined packageStep-1

```
package pack;
public class PackDemo
{
        public void show()
        {
                System.out.println("welcome to java");
        }
}
```

Step-2

```
import pack.PackDemo;
class Pack1
{
        public static void main(String args[])
        {
                PackDemo obj=new PackDemo();
                obj.show();
        }
}
```

Step-3

```
D:\java>javac -d . PackDemo.java

D:\java>
```

Step-4

```
D:\java>javac Pack1.java

D:\java>java Pack1
welcome to java

D:\java>
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package java.awt.image;

Example: Package demonstration

```
package  pack; public
class Addition
{
        int x,y;
        public Addition(int a, int b)
        {
                x=a;
                y=b;
        }
        public void sum()
        {
                System.out.println("Sum :"+(x+y));
```

```
        }
}
```

Step 1: Save the above file with Addition.java

```
package pack;
public class Subtraction
{
        int x,y;
        public Subtraction(int a, int b)
        {
                x=a;
                y=b;
        }
        public void diff()
        {
                System.out.println("Difference  :"+(x-y));
        }
}
```

Step 2: Save the above file with Subtraction.javaStep 3:

Compilation

To compile the java files use the following commands

javac -d directory_path name_of_the_java fileJavac –d

. name_of_the_java file

Note: -d is a switching options creates a new directory with package name. Directory path represents in which location you want to create package and . (dot)represents current working directory.

Step 4: Access package from another package

There are three ways to use package in another package:

1. **With fully qualified name.**

```
class UseofPack


{
    public static void main(String arg[])
    {
        pack.Addition a=new pack.Addition(10,15);
        a.sum();
        pack.Subtraction s=new pack.Subtraction(20,15);
        s.difference();
    }
}
```

2. **import package.classname;**

```
import pack.Addition;
import pack.Subtraction;
class UseofPack
{
    public static void main(String arg[])
    {
        Addition a=new Addition(10,15);a.sum();
       Subtraction s=new Subtraction(20,15);
        s.difference();
    }
}
```

3. **import package.*;**

```
import pack.*; class
UseofPack
    {
        public static void main(String arg[])
        {
            Addition a=new Addition(10,15);a.sum();
            Subtraction s=new Subtraction(20,15);
            s.difference();
        }
    }
```

Note: Don't place Addition.java, Subtraction.java files parallel to the pack directory. If you place JVM searches for the class files in the current working directory not in the pack directory.

Access Control

- Java provides four types of **access modifiers**: `public`, `protected`, `default` (no modifier), and `private`.
    - `public`: Accessible from any class.
    - `protected`: Accessible within the same package and subclasses.
    - `default`: Accessible only within the same package.
    - `private`: Accessible only within the class where it is declared.

Packages in Java SE

1. java.lang Package

- This package is automatically imported into every Java program, providing fundamental classes essential for the language.
- **Key classes:**
    - Object: The root class from which all classes in Java inherit.
    - String: Immutable sequences of characters.
    - Math: Provides mathematical operations such as sqrt(), pow(), abs().
    - System: Used to interact with system resources, e.g., System.out for standard output.
    - Thread: For multithreading operations.

2. java.util Package

- Contains utility classes and interfaces used for collections, date/time manipulation, and random number generation.

- Key classes:
    - ArrayList, LinkedList, HashSet, HashMap: For handling dynamic collections of data.
    - Collections: Utility class for manipulating collections (e.g., sorting, searching).
    - Date, Calendar, TimeZone: For handling date and time.
    - Random: For generating random numbers.

3. java.io Package

- Provides classes for input and output operations, such as reading and writing data to files, handling streams, and working with serializable objects.
- Key classes:
    - File: Represents file and directory pathnames.
    - BufferedReader, BufferedWriter: For efficient reading/writing of text from/to files.
    - InputStream, OutputStream: Base classes for byte stream operations.
    - Serializable: Marks classes for object serialization

Wrapper Classes in Java

Wrapper classes in Java are used to convert **primitive data types** into **objects**. Each of Java's eight primitive types (int, char, etc.) has a corresponding wrapper class in the `java.lang` package. These wrapper classes provide a way to treat primitive data types as objects, which is necessary in scenarios where only objects are allowed, such as with Java Collections (e.g., `ArrayList`, `HashMap`).

The process of converting a primitive type to its corresponding wrapper object is known as **boxing**, and converting it back to a primitive is called **unboxing**.

Primitive Types and Corresponding Wrapper Classes:

Primitive Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`

char	Character

Primitive Type	Wrapper Class
short	Short
int	Integer
long	Long
float	Float
double	Double

Why Use Wrapper Classes?

1. **Object-Oriented Collection Classes**: Java's collection classes (e.g., `ArrayList`, `HashMap`) can only store objects, not primitives. Wrapper classes allow primitive data types to be stored in collections by converting them into objects.
2. **Utility Methods**: Wrapper classes provide many useful methods for manipulating and converting primitive values.
3. **Default Values in Generics**: Java Generics work only with objects, so wrapper classes are used when you need to work with generic types.
4. **Nullability**: Wrapper classes can be `null`, whereas primitive types cannot. This can be useful for representing the absence of a value.

Boxing and Unboxing

- **Boxing** is the process of converting a primitive type into its corresponding wrapper object.
- **Unboxing** is the reverse process, where the wrapper object is converted back into a primitive type.

Example of Boxing and Unboxing:

```
public class BoxingUnboxingExample {

   public static void main(String[] args) {

      // Boxing (primitive to object)

      int num = 100;

         Integer obj = Integer.valueOf(num); // explicitly boxing



         // Unboxing (object to primitive)

         Integer obj1 = new Integer(200);
```

```
int num2 = obj1.intValue(); // explicitly unboxing
```

```
        System.out.println("Boxed Integer: " + obj);
        System.out.println("Unboxed int: " + num2);
    }
}
```

Auto Boxing and Auto Unboxing

Java automatically handles the conversion between primitives and their corresponding wrapper classes through **auto-boxing** and **auto-unboxing**.

- **Auto-boxing**: Automatic conversion of a primitive type into its wrapper class object.
- **Auto-unboxing**: Automatic conversion of a wrapper object to its corresponding primitive type.

Example of Auto Boxing and Auto Unboxing:

```
public class AutoBoxingUnboxingExample {
    public static void main(String[] args) {
        // Auto-boxing
        int num = 100;
        Integer obj = num; // no need to call Integer.valueOf(num)

        // Auto-unboxing
        Integer obj1 = new Integer(200);
        int num2 = obj1; // no need to call wrappedNum2.intValue()

        System.out.println("Auto-boxed Integer: " + obj);
        System.out.println("Auto-unboxed int: " + num2);
    }
}
```

Java util Classes and Interfaces

Formatter Class (java.util.Formatter)

The Formatter class in Java provides support for formatting data (such as strings, numbers, dates, etc.) in a way similar to printf() in C. It can format output based on a format string that specifies how data should be presented. It is often used in logging, console output, or file writing.

Key Methods:

- **format()**: This is the primary method for formatting. It supports a variety of data types, and the format string uses placeholders.

Key Concepts:

1. **Format String**: The format string specifies how data should be formatted. It contains placeholders like %d, %f, %s, which get replaced with actual values.
2. **Supported Data Types**:
    - **%d**: Integer (decimal).
    - **%f**: Floating-point number (decimal).
    - **%s**: String.
    - **%x**: Integer (hexadecimal).
    - **%o**: Integer (octal).
    - **%t**: Date/time values.

Example:

```
public class FormatterExample {
   public static void main(String[] args) {
      Formatter fmt = new Formatter();
      fmt.format("Value of Pi to 2 decimals: %.2f", 3.14159);
      System.out.println(fmt);
      fmt.close();
   }
}
```

Output:

```
Value of Pi to 2 decimals: 3.14
```

Example2:

```
import java.util.Formatter;

public class FormatterExample {
   public static void main(String[] args) {
      // Create a Formatter
      Formatter fmt = new Formatter();

      // Format an integer, a float, and a string
      fmt.format("Integer: %d\n", 123);
      fmt.format("Floating-point: %.2f\n", 3.14159);
      fmt.format("String: %s\n", "Hello, World!");
```

```
        // Print the formatted output
        System.out.println(fmt);

        // Close the Formatter to release resources
        fmt.close();
    }
}
```

Output:

Integer: 123

Floating-point: 3.14

String: Hello, World!

Formatting Dates and Times:

You can use the Formatter class to format dates and times using the %t prefix.

- **%tY**: Year (4 digits).
- **%tm**: Month (2 digits).
- **%td**: Day of the month.
- **%tH**: Hour (24-hour clock).
- **%tM**: Minute.
- **%tS**: Second.

Example3:

```
import java.util.Formatter;
import java.util.Calendar;

public class DateFormatExample {
    public static void main(String[] args) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        // Format current date and time
        fmt.format("Current Date: %tY-%tm-%td\n", cal, cal, cal);
        fmt.format("Current Time: %tH:%tM:%tS\n", cal, cal, cal);

        // Print the formatted output
```

```
System.out.println(fmt);
```

```
        // Close the Formatter
        fmt.close();
    }
}
```

Output:

Current Date: 2024-10-13

Current Time: 09:30:47

2. Random Class (java.util.Random)

The Random class in Java is used to generate pseudo-random numbers. It provides methods to generate random integers, floats, longs, and even boolean values.

Key Methods:

- **nextInt()**: Returns a random integer.
- **nextInt(int bound)**: Returns a random integer within the specified bound.
- **nextDouble()**: Returns a random double between 0.0 and 1.0.
- **nextBoolean()**: Returns a random boolean.

Example:

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        Random random = new Random();

        // Generate random integers
        int randInt = random.nextInt(100);  // Random integer between 0 and 99
        System.out.println("Random Integer: " + randInt);

        // Generate random doubles
        double randDouble = random.nextDouble();  // Random double between 0.0 and 1.0
        System.out.println("Random Double: " + randDouble);
```

```
// Generate random booleans
boolean randBoolean = random.nextBoolean();
```

```
        System.out.println("Random Boolean: " + randBoolean);
    }
}
```

Output:

Random Integer: 70

Random Double: 0.024016527282495925

Random Boolean: false

3. Time Package (java.time)

The java.time package introduced in Java 8 provides a comprehensive API for handling dates and times. It offers a much more flexible and modern way of working with time compared to the legacy java.util.Date and java.util.Calendar classes.

Key classes include:

- **LocalDate**: Represents a date without time.
- **LocalTime**: Represents a time without a date.
- **LocalDateTime**: Represents a date and time.
- **Duration**: Represents a time duration (e.g., 5 hours, 30 minutes).
- **Period**: Represents a date-based amount of time (e.g., 2 years, 3 months).
- **ZonedDateTime**: Represents a date-time with a time zone.

Example:

1. LocalDate

Represents a date without a time zone (year, month, day).

```
import java.time.LocalDate;

public class LocalDateExample {
    public static void main(String[] args) {
        // Get the current date
        LocalDate today = LocalDate.now();
        System.out.println("Today's date: " + today);
```

// Create a specific date

```
        LocalDate specificDate = LocalDate.of(2024, 10, 13);
        System.out.println("Specific date: " + specificDate);

        // Add days to a date
        LocalDate nextWeek = today.plusDays(7);
        System.out.println("Date after one week: " + nextWeek);

        // Check if a year is a leap year
        boolean isLeapYear = today.isLeapYear();
        System.out.println("Is this year a leap year? " + isLeapYear);
    }
}
```

Output:

```
Today's date: 2024-10-13
Specific date: 2024-10-13
Date after one week: 2024-10-20
Is this year a leap year? false
```

Example2:

2. LocalTime

Represents a time without a date and without a time zone.

```
import java.time.LocalTime;

public class LocalTimeExample {
    public static void main(String[] args) {
        // Get the current time
        LocalTime now = LocalTime.now();
        System.out.println("Current time: " + now);

        // Create a specific time
        LocalTime specificTime = LocalTime.of(14, 30, 45); // 2:30:45 PM
```

```
System.out.println("Specific time: " + specificTime);
```

```
        // Add hours and minutes to the current time
        LocalTime later = now.plusHours(2).plusMinutes(15);
        System.out.println("Time after 2 hours and 15 minutes: " + later);
        // Get the hour, minute, and second
        int hour = now.getHour();
        int minute = now.getMinute();
        int second = now.getSecond();

        System.out.println("Hour: " + hour + ", Minute: " + minute + ", Second: " + second);
    }
}
```

Output:

Current time: 09:30:47.123

Specific time: 14:30:45

Time after 2 hours and 15 minutes: 11:45:47.123

Hour: 9, Minute: 30, Second: 47

5. Formatting for Date/Time in Java (DateTimeFormatter)

The DateTimeFormatter class (from java.time.format) is used to format and parse date/time objects. It provides flexible and powerful formatting options.

Common Predefined Formatters:

- **ISO_LOCAL_DATE**: Formats a date as yyyy-MM-dd.
- **ISO_LOCAL_DATE_TIME**: Formats a date and time as yyyy-MM-ddTHH:mm:ss.

Custom Format Example:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

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

```
LocalDateTime now = LocalDateTime.now();
```

```
        // Custom format: "dd-MM-yyyy HH:mm:ss"
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
        String formattedDateTime = now.format(formatter);

        System.out.println("Formatted DateTime: " + formattedDateTime);
    }
}
```

Output:

Formatted DateTime: 13-10-2024 08:52:35

6. TemporalAdjusters Class (java.time.temporal.TemporalAdjusters)

The TemporalAdjusters class provides common temporal adjusters, which allow date manipulations such as finding the next day of the week, the last day of the month, etc. Adjusters are often used with classes like LocalDate.

Common Temporal Adjusters:

- **firstDayOfMonth()**: Returns the first day of the current month.
- **lastDayOfMonth()**: Returns the last day of the current month.
- **next(DayOfWeek dayOfWeek)**: Returns the next occurrence of the specified day of the week.
- **previous(DayOfWeek dayOfWeek)**: Returns the previous occurrence of the specified day of the week.

Example:

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class TemporalAdjustersExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        // Get the next Sunday
        LocalDate nextSunday = today.with(TemporalAdjusters.next(DayOfWeek.SUNDAY));
```

```
System.out.println("Next Sunday: " + nextSunday);
```

```
        // Get the last day of the current month

        LocalDate lastDayOfMonth = today.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("Last Day of Month: " + lastDayOfMonth);
    }
}
```

Output:

Next Sunday: 2024-10-20

Last Day of Month: 2024-10-31

Exception Handling

Java Errors are classified into 3 types

1) Compile -Time Errors
2) Run -Time Errors
3) Logical Errors

Compile-Time Errors

- Errors occurred at Compile Time are called Compile Time Errors
- Syntax Errors are detected at Compile Time.
- **These are Syntactical Errors found in the code, due to which a program fails to compile.**
- For Example, forgetting a semicolon at the end of a Java statement, or writing a statement without proper syntax will result in compile-time error.
- Detecting and Correcting compile-time errors is easy as the Java Compiler displays the list of errors with the line numbers along with theirdescription

Run Time Errors

- Errors occurred at **Run Time** are called Run Time Errors
- Run time errors are not detected by the **java compiler**.
- It is the JVM which detects it while the program is running.
- **Semantic Errors** like division by zero, Index out of Bound are detectedby JVM at runtime.

Logical Errors

- These errors are due to the mistakes made by the programmer.
- It will not be detected by a **compiler nor by the JVM.**
- errors may be due to wrong idea or concept used by a programmer while coding.

Introduction to Exception Handling

- An Exception is a **Run Time Error** (or) An exception is **abnormal condition** that arises in a code sequence at the **run time.**
- When the jvm encounter an **Run Time Error** such as **Division by zero**, JVM creates an object to the Corresponding Class and throws it.
- If the Programmer does not catch the thrown object and handles properly, ***the interpreter will display an error message and the program gets terminated abnormally***.
- **In order to stop abnormal termination of the program and to fix the error, exceptions should be caught and handled.**

Java exception handling is managed via five keywords

- Try
- Catch
- Throw
- Throws
- Finally

TRY

- Statements that need to be monitored for exceptions should be placed within **a try** block

CATCH

- If an exception occurs within the try block, it is thrown and Your code can catch this exception using **catch** block and handles it in some rational manner.

THROW

- System generated exception are automatically thrown by the jvm. To manually throw an exception, use the keyword **throw**.

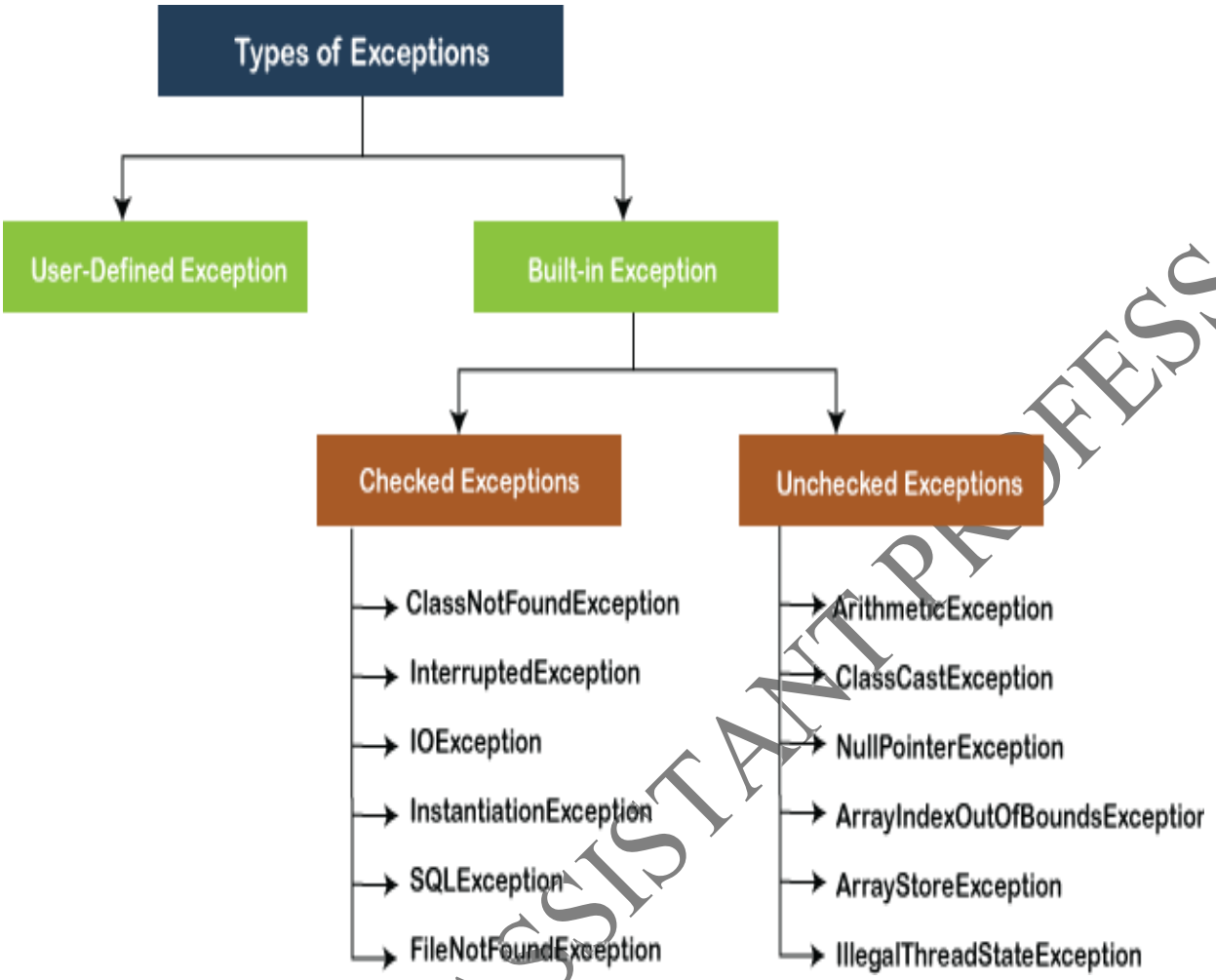
THROWS

- Any exception that is thrown out of method must be specified as such by a **throws** clause

FINALLY

- Any code that obsolutely must be executed after a try block completes isput in a **finally** block.

Hierarchy of Standard Exception Classes



1. User-Defined Exceptions

- These are custom exceptions created by users.
- In Java, users can define their own exceptions by extending the Exception class (for checked exceptions) or RuntimeException (for unchecked exceptions).

2. Built-in Exceptions

Built-in exceptions are predefined in Java and categorized as:

a. Checked Exceptions:

- Checked exceptions must be either caught or declared in the throws clause of a method.
- Examples include:
    - ClassNotFoundException
    - IOException
    - SQLException
    - FileNotFoundException

b. Unchecked Exceptions:

- These exceptions occur at runtime and don't need to be declared in a method's throws clause.
- Examples include:
    - ArithmeticException
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - IllegalArgumentException

Examples of Programs

1. **User-Defined Exception Example**: Here's how you can create and use a user-defined exception.

```
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
public class Main {
    public static void validateAge(int age) throws CustomException {
        if(age < 18) {
            throw new CustomException("Age is less than 18, not eligible.");
        } else {
            System.out.println("Eligible");
        }
    }
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (CustomException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Output:

Caught: Age is less than 18, not eligible.

b. Unchecked Exceptions:

- These exceptions occur at runtime and don't need to be declared in a method's throws clause.
- Examples include:
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - IllegalArgumentException

Built-in-Exception-Creating own Exceptions

Arithmetic exception

```
class ArithmeticException_Demo {
public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a / b; // cannot divide by zero
            System.out.println("Result = " + c);
        }


        catch (ArithmeticException e) {
```

ArrayIndexOutOfBounds Exception

```
class ArrayIndexOutOfBound_Demo {
public static void main(String args[])
  {
    try {
      int a[] = new int[5];
      a[6] = 9; // accessing 7th element in an array of
        // size 5
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array Index is Out Of Bounds");
    }
```

```
}
```

}

FileNotFoundException

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo {

public static void main(String args[])
    {
        try {

            // Following file does not exist
            File file = new File("E:// file.txt");

            FileReader fr = new FileReader(file);
        }
```

NullPointerException

```
class NullPointer_Demo {
public static void main(String args[])
    {
        try {
            String a = null; // null value
            System.out.println(a.charAt(0));
        }
        catch (NullPointerException e) {
            System.out.println("NullPointerException..");
```

Java File Handling

In Java by reading and writing text and binary files. File handling is crucial for any software developer since it allows you to store and retrieve data, create logs, and process input/output files.

java provides several classes and methods to work with files. The most common classes used for file handling are:

- **File**: Represents a file or directory and provides methods to work with them.
- **FileInputStream** and **FileOutputStream**: Used for reading and writing binary files.
- **FileReader** and **FileWriter**: Used for reading and writing text files.
- **BufferedReader** and **BufferedWriter**: Used for buffered reading and writing.

To read a text file, follow these steps:

1. Create a Fileobject representing the text file.
2. Create a FileReaderobject to read the file.
3. Create a BufferedReaderobject to read text from the file efficiently.
4. Read the file using the readLine()method.
5. Close the BufferedReaderobject.

Types of Streams

Java defines two types of streams:

- **Byte Streams**: Used to perform input and output of 8-bit bytes.
- **Character Streams**: Used to perform input and output for characters (16-bit Unicode).

Byte Streams

Byte streams in Java are used to handle raw binary data. These streams read/write data in the form of bytes. Classes for byte streams are part of the java.io package and typically extend InputStream or OutputStream.

Common Byte Stream Classes:

- **FileInputStream**: Reads bytes from a file.
- **FileOutputStream**: Writes bytes to a file.
- **BufferedInputStream**: Reads bytes from a file with buffering.
- **BufferedOutputStream**: Writes bytes to a file with buffering.

1. FileInputStream

- **Purpose**: Reads raw bytes from a file.
- It is used to read the content of a file byte by byte, making it ideal for reading binary files like images, audio, etc.
- It is part of the java.io package and extends the InputStream class.

Example:

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int data;
            while ((data = fis.read()) != -1) {  // Read byte by byte
                System.out.print((char) data);  // Convert byte to char and print
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. FileOutputStream

- **Purpose**: Writes raw bytes to a file.
- It is used to write data into a file byte by byte, useful for writing binary data.
- It is part of the java.io package and extends the OutputStream class.

Example:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String content = "Hello, World!";
            fos.write(content.getBytes());  // Convert string to bytes and write to file
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The content "Hello, World!" is converted into bytes using the getBytes() method and written to the file output.txt.
- The write() method writes bytes to the file.

3. BufferedInputStream

- **Purpose**: Reads bytes from a file with buffering to improve performance.
- It wraps a FileInputStream and provides buffering, which reduces the number of actual read operations performed on the file, improving efficiency.
- It is part of the java.io package and extends the InputStream class.

Example:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

```
public class BufferedInputStreamExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("example.txt"))) {
            int data;
            while ((data = bis.read()) != -1) {
                System.out.print((char) data);  // Convert byte to char and print
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. BufferedOutputStream

- **Purpose**: Writes bytes to a file with buffering to improve performance.
- It wraps a FileOutputStream and provides buffering, reducing the number of actual write operations performed on the file.
- It is part of the java.io package and extends the OutputStream class.

Example:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class BufferedOutputStreamExample {
    public static void main(String[] args) {
        try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("output.txt"))) {
            String content = "Hello, Buffered World!";
            bos.write(content.getBytes());  // Write bytes to buffer
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

}

Advantages of Buffered Streams:

- **Performance Improvement**: Buffered streams improve performance by reducing the number of disk I/O operations. Instead of reading/writing byte-by-byte, buffered streams work with larger blocks of data.
- **Efficiency**: Buffered streams are more efficient when reading from or writing to slow sources, such as files on a disk or network connections.

Key Differences:

1. **FileInputStream/FileOutputStream**:
    - Read/write data one byte at a time.
    - Suitable for binary data but not optimized for frequent I/O operations.
2. **BufferedInputStream/BufferedOutputStream**:
    - Read/write data in chunks, improving efficiency by reducing I/O operations.
    - Suitable for larger files or when efficiency is a concern.

character Streams

Character Streams handle 16-bit Unicode characters, making them ideal for reading and writing text data. Classes for character streams typically extend the Reader class (for reading) or the Writer class (for writing).

Common Character Stream Classes:

1. **FileReader**: Reads characters from a file.
2. **FileWriter**: Writes characters to a file.
3. **BufferedReader**: Wraps FileReader to provide efficient character buffering while reading text.
4. **BufferedWriter**: Wraps FileWriter to provide efficient character buffering while writing text.

FileReader

- **Purpose**: Reads characters from a file.
- It is a convenient class for reading text files as it reads characters rather than bytes, making it suitable for handling text data.

Example:

```
import java.io.FileReader;

import java.io.IOException;

public class FileReaderExample {

   public static void main(String[] args) {

      try (FileReader fr = new FileReader("example.txt")) {
```

```
int data;
```

```
            while ((data = fr.read()) != -1) {
                System.out.print((char) data);  // Read character-by-character.
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. FileWriter

- **Purpose**: Writes characters to a file.
- FileWriter is used for writing text data to a file, character-by-character. It's a simple way to write text files.

Example:

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            String content = "Hello, FileWriter!";
            fw.write(content);  // Write string to file.
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. BufferedReader

- **Purpose**: Wraps FileReader to provide efficient character buffering while reading text.
- It reads text from a file more efficiently by buffering character input. It also provides convenient methods like readLine() for reading entire lines of text.

Example:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {  // Read line-by-line.
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. BufferedWriter

- **Purpose**: Wraps FileWriter to provide efficient character buffering while writing text.
- It writes text to a file more efficiently by buffering character output. It also provides convenient methods like newLine() to write line separators.

Example:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
            bw.write("Hello, BufferedWriter!");  // Write text to file.
            bw.newLine();  // Insert a new line.
            bw.write("This is the second line.");
```

```
System.out.println("Data written to file successfully.");
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1. Scanner Class

The **Scanner** class in Java is part of the java.util package. It is widely used to parse primitive types (e.g., int, double, float, etc.) and strings using regular expressions. A common use case for the Scanner class is reading input from the user, reading files, or processing input from other data sources like input streams.

Common Uses:

1. **Reading from the Console** (Standard Input)
2. **Reading from Files**

---

a. Reading from the Console (Standard Input)

The Scanner class can read input from the console using the standard input stream (System.in).

Example:

```
import java.util.Scanner;
public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();  // Read a full line of text
        System.out.println("Hello, " + name + "!");

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();  // Read an integer value
        System.out.println("You are " + age + " years old.");
    }
}
```

Explanation:

- The nextLine() method is used to read a full line of text.
- The nextInt() method reads an integer value.

b. Reading from a File

The Scanner class can also be used to read data from a file by passing a File object or the file path to the Scanner constructor.

Example:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileReadingExample {
    public static void main(String[] args) {
        try {
            File file = new File("input.txt");
            Scanner scanner = new Scanner(file);

            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();  // Read line-by-line
                System.out.println(line);
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The Scanner reads lines from the file input.txt line-by-line using the nextLine() method.
- The hasNextLine() method checks if there are more lines to read.

2. Files Class

The **Files** class in Java is part of the java.nio.file package, which provides a variety of utility methods for file handling, including reading, writing, creating, copying, moving, and deleting files and directories. It supports working with **Path** objects, which represent file and directory locations in the file system.

Common Operations:

1. **Reading a File**
2. **Writing to a File**
3. **Copying Files**
4. **Deleting Files**
5. **Creating Files and Directories**

1. Introduction to String Handling

In Java, strings are objects used to store and manipulate sequences of characters. Java provides several classes, such as String, StringBuilder, and StringBuffer, for handling strings. Strings in Java are immutable, meaning once created, their values cannot be changed. This immutability allows for more efficient memory usage and easier handling of strings.

The java.lang.String class is used to create a string object.

There are two ways to create String object:

1. By string literal

2. By new keyword

1. By String Literal

Java String literal is created by using double quotes. For Example:

String s="welcome";

2. By new keyword

String s=new String("Welcome");

String methods in java

1. **length():**Returns the number of characters in the string.

```
String str = "Hello, World!";
int len = str.length(); // 13
System.out.println("Length of the string: " + len);
```

Output:

Length of the string: 13

2. **equals():**Checks if two strings have the same content (case-sensitive).

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = "hello";
boolean isEqual = str1.equals(str2); // true
boolean isEqualCaseSensitive = str1.equals(str3); // false
System.out.println("str1 equals str2: " + isEqual);
System.out.println("str1 equals str3 (case-sensitive): " + isEqualCaseSensitive);
```

Output:

str1 equals str2: true

str1 equals str3 (case-sensitive): false

3. **equalsIgnoreCase():**Compares strings, ignoring case differences.

```
String str1 = "Hello";
String str2 = "hello";
boolean isEqualIgnoreCase = str1.equalsIgnoreCase(str2); // true
System.out.println("str1 equals str2 (ignoring case): " + isEqualIgnoreCase);
```

Output:

str1 equals str2 (ignoring case): true

4. **startsWith(String prefix):**Checks if the string starts with the specified prefix.

```
String str = "Java Programming";
boolean startsWithJava = str.startsWith("Java"); // true
System.out.println("String starts with 'Java': " + startsWithJava);
```

Output:

String starts with 'Java': true

5. **endsWith(String suffix):**Checks if the string ends with the specified suffix.

```
String str = "Hello, World!";
boolean endsWithWorld = str.endsWith("World!"); // true
System.out.println("String ends with 'World!': " + endsWithWorld);
```

Output:

String ends with 'World!': true

6. **StringBuffer reverse()**

Reverses the contents of a StringBuffer.

```
StringBuffer sb = new StringBuffer("Hello");
sb.reverse(); // "olleH"
System.out.println("Reversed StringBuffer: " + sb);
```

Output:

Reversed StringBuffer: olleH

7. **replace(char oldChar, char newChar):**Replaces all occurrences of a specified character in a string.

```
String str = "balloon";
String replacedStr = str.replace('o', 'a'); // "ballaan"
System.out.println("Replaced String: " + replacedStr);
```

Output:

Replaced String: ballaan

8. **concat(String str):**Concatenates the specified string to the end of the current string.

```
String str1 = "Hello";
String str2 = str1.concat(" World");
System.out.println("Concatenated String: " + str2);
```

Output:

Concatenated String: Hello World

9. **charAt(int index):**Returns the character at the specified index.

```
String str = "Hello, World!";
char ch = str.charAt(7); // 'W'
System.out.println("Character at index 7: " + ch);
```

Output:

Character at index 7: World

10. **substring(int start, int end):**Returns a new string containing the characters from the specified start to end index.

```
String str = "Hello, World!";
String subStr = str.substring(7, 12); // "World"
System.out.println("Substring from index 7 to 12: " + subStr);
```

Output:

Substring from index 7 to 12: World

11. **toCharArray():**Converts the string to a character array.

```
String str = "Hello";
char[] charArray = str.toCharArray();
System.out.println("Character array: " + Arrays.toString(charArray));
```

Output: Character array: [H, e, l, l, o]

12. **compareTo(String anotherString):**Compares two strings lexicographically.

```
String str1 = "Apple";
String str2 = "Banana";
int comparison = str1.compareTo(str2); // returns a negative value because "Apple" < "Banana"
System.out.println("Comparison result: " + comparison);
```

Output:

Comparison result: -1

13. **concat(String str):**Concatenates the specified string to the end of the current string.

```
String str1 = "Hello";
```

String str2 = “world”

System.out.println(str1.concat(str2);

Output:

Concatenated String: Hello World

14. replaceAll(String regex, String replacement): Replaces each substring that matches the given regular expression with the specified replacement.

```
String sentence = "The rain in Spain";

String replacedSentence = sentence.replaceAll("ain", "oon"); // "The roon in Spoon"

System.out.println("Replaced Sentence: " + replacedSentence);
```

Output:

Replaced Sentence: The roon in Spoon

15. toLowerCase() and toUpperCase(): Converts all characters in the string to lowercase or uppercase.

```
String str = "Hello, World!";

String lower = str.toLowerCase(); // "hello, world!"

String upper = str.toUpperCase(); // "HELLO, WORLD!"

System.out.println("Lowercase: " + lower);

System.out.println("Uppercase: " + upper);
```

Output:

Lowercase: hello, world!

Uppercase: HELLO, WORLD!

Multithreading

Multithreaded programming is a method of concurrent execution in which multiple threads, or smaller units of a process, run simultaneously. This technique enhances the efficiency of a program, particularly on multi-core processors, by allowing multiple tasks to execute at once. Let's break down some of the essential concepts in multithreaded programming:

1. Need for Multiple Threads

Multiple threads enable concurrent execution, which improves program performance and responsiveness. For example, in a GUI application, one thread can handle the user interface while another thread performs calculations in the background.

2. Multithreaded Programming for Multi-core Processors

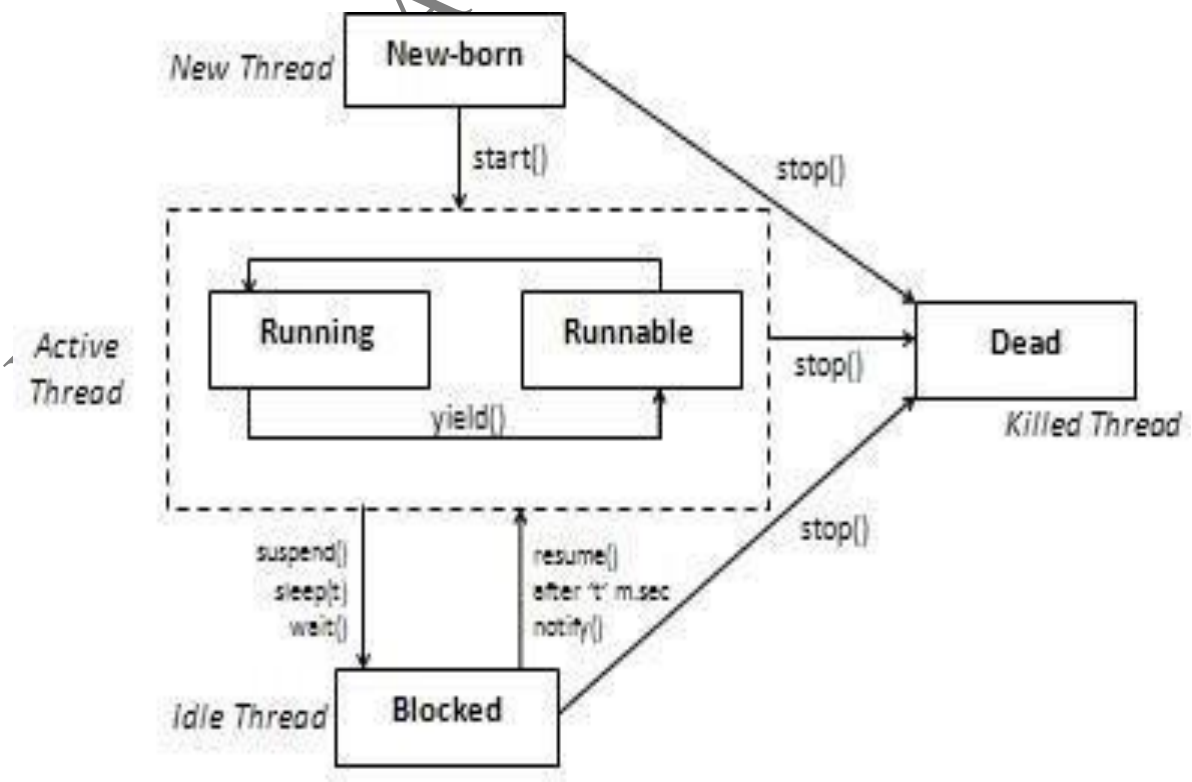
Multi-core processors can execute multiple threads in parallel, allowing programs to take full advantage of the processor's capabilities. This enables faster computation and can reduce the time required for processing tasks.

Thread Life Cycle

During the life time of a thread there are many states it can enter. They are

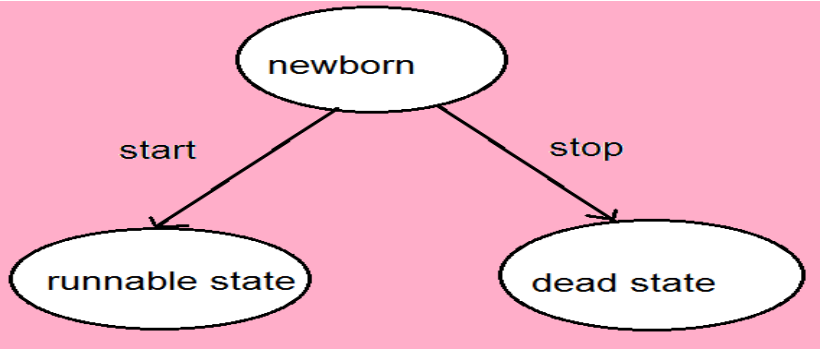
A. NewBorn state
B. Runnable state
C. Running State
D. Blocked state
E. Dead state

A thread is always in any one of these five states.It can move from one state to another via a variety of ways as shown below



New Born state

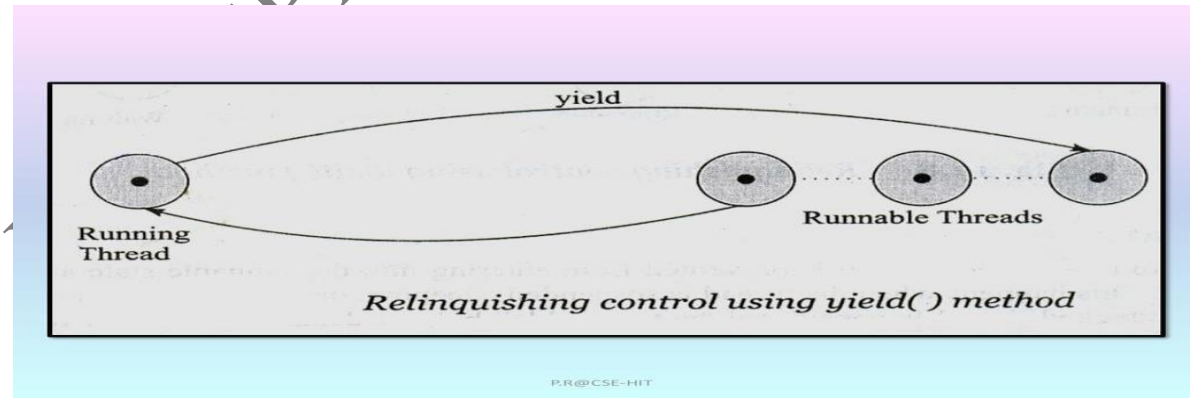
- ✓ when we create a thread object, the thread is born and is said to be in new born state.
- ✓ The thread is not yet scheduled for running .At this state, we can do only one of the following things with it.
- ❖ Schedule it for running using start() method.
- ❖ Kill it using stop() method.



- ❖ **If scheduled ,it moves to the runnable state**

Runnable State

- ➢ The runnable state means that the thread is ready for execution and is waiting for availability of the processor .i.e the thread has joined the queue of threads that are waiting for execution.
- ➢ If all threads have equal priority, then they are given time slots for execution in Round Robin fashion,i.e FCFS manner.
- ➢ The thread that relinquishes control joins the queue at the end & again waits for its turn



Running State

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it relinguishes its control in one of the following situations.

1) It has been suspended using suspend().

   a suspended thread can be received by using the resume() method.

2) It has been made wait by using wait() method

   A thread that is waiting will get resumed after notify() method

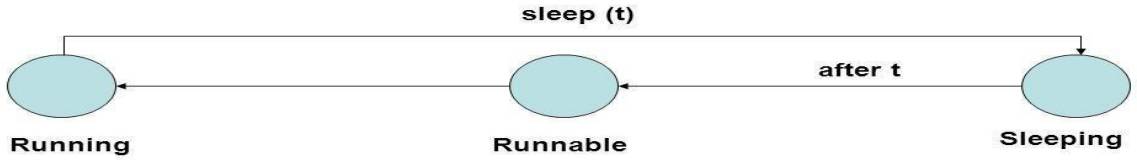
3) It has been slept for a t seconds.

A thread will get invoked after t seconds

Example

2. sleep()

It has been made to sleep, We can put a thread to sleep for a specified time period using the method sleep (time) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re – enters the runnable state as soon as this time period is elapsed.



Blocked state

- Thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods.

Sleep() **// blocked for specified time**

Suspended() **// blocked until further orders**

Wait() **// blocked until certain condition occurs.**

- These methods cause the thread to go into the blocked sate. The thread will return to runnable state when the specified time is elapsed in the case of sleep(),the resume() method is invoked in case of suspend(),and notify() method is called in case of wait().

Dead State

- Every thread has a life cycle .
- A running thread ends its life when it has completed executing its run().it is natural death.
- However, we can kill it by using stop message to it at any stage.Thus causing premature death to it.

Creating threads in java is simple. Threads in java can be created in two ways

1) By extending the thread class.

2) By implementing the runnable interface.

1) **Creating threads by extending the thread class:**

- Define a class that extends thread class and override its run()with the code required by the thread.
- Steps to create thread by extending thread class are
    a) Declaring the class
    b) Implementing the run() method.
    c) Starting New Thread.

Declaring the class:

Declare the class by extending the thread

class as: Class MyThread extends Thread

{

}

Implementing the run() method:

- the run method is the heart and soul of any thread.
- We have to override this method in order to implement the code to be executed by our thread.
- It makes up the entire body of a thread and is the only method in which the threads behavior can be implemented.
- The basic implementation of run()

  will look like public void run()

```
{
Thread code
}
```

When we start new thread ,java calls the threads run() method.

Starting New Thread:

- create a thread object and call the start() method to initiate the thread execution.
- To create and run an instance of our thread class, we must write the

following: MyThread t1=new MyThread();

T1.start();

- The first line instantiates a new object of class MyThread.
- The second line calls start() causing the thread to move into runnable state.
- Then, the java runtime will schedule the thread to run by invoking its run().Hence the thread is said to be in Running state.

```java
class A extends Thread
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread A :"+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
class B extends Thread
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread B :"+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
public class HelloWorld{

     public static void main(String []args)
     {
        A ob1 = new A();
        B ob2 = new B();
        ob1.start();
        ob2.start();
     }
}
```

```
$javac HelloWorld.java
$java -Xmx128M -Xms16M HelloWorld
From Thread A :1
From Thread B :1
From Thread A :2
From Thread B :2
From Thread A :3
From Thread B :3
From Thread A :4
From Thread B :4
From Thread A :5
From Thread B :5
```

Creating Thread using Runnable Interface

A) Create a Class that implements Runnable Interface

B) override run() method

create a thread by passing an object to the implementation class of runnable interface

```
class A implements Runnable
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread A: "+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
class B implements Runnable
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("From Thread B: "+i);
                Thread.sleep(100);
            }
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
public class HelloWorld{

     public static void main(String []args)
     {
        A ob1 = new A();
        B ob2 = new B();
        Thread t1 = new Thread(ob1);
        Thread t2 = new Thread(ob2);
        t1.start();
        t2.start();
     }
}
```

```
$javac HelloWorld.java

$java -Xmx128M -Xms16M HelloWorld
From Thread B: 1
From Thread A: 1
From Thread B: 2
From Thread A: 2
From Thread B: 3
From Thread A: 3
From Thread B: 4
From Thread A: 4
From Thread B: 5
From Thread A: 5
```

Thread Priority and Synchronization

Thread Priority is a concept in multithreaded programming that determines the relative importance of each thread when they compete for CPU time. Thread priorities help the system's scheduler decide which thread to run when multiple threads are ready for execution.

1. **Priority Levels**:
    - Threads are assigned priority levels, typically as integers. In Java, for example, thread priorities range from MIN_PRIORITY (1) to MAX_PRIORITY (10), with NORM_PRIORITY (5) as the default.
    - A higher priority thread is more likely to be selected by the CPU scheduler over a lower-priority thread, although this behavior is platform-dependent.

Synchronization in multithreaded programming is crucial for managing access to shared resources to avoid data inconsistency and ensure thread safety.

1. **The Need for Synchronization**:
    - When multiple threads access shared resources (e.g., shared variables, files, or memory), there is a risk of **race conditions**, where the final outcome depends on the timing of thread execution.
    - Synchronization prevents threads from interfering with each other and ensures that only one thread accesses a shared resource at a time.
2. **Synchronized Blocks and Methods**:
    - In many programming languages, synchronization is achieved using synchronized blocks or methods. A synchronized block allows only one thread at a time to access the code block or resource.
    - For example, in Java, the synchronized keyword locks an object, so no other thread can access the synchronized code block or method of that object until the current thread completes it.
3. **Locks (Mutexes)**:
    - A **lock** (or **mutex**) is a mechanism used to enforce synchronization by allowing only one thread to hold the lock at a time.
    - When a thread acquires a lock on a resource, other threads must wait until the lock is released before they can access the same resource.
4. **Deadlock**:
    - **Deadlock** occurs when two or more threads wait indefinitely for resources held by each other, creating a cycle of dependencies with no resolution.
    - Avoiding deadlock requires careful resource allocation and sometimes the use of timeout-based locking mechanisms.

5. **Avoiding Race Conditions**:
    - Race conditions occur when multiple threads attempt to modify shared data concurrently, leading to inconsistent results. Synchronization helps avoid race conditions by enforcing an orderly access to shared resources.

Deadlock and RaceConditions

Both deadlock and race conditions are critical concurrency issues in multithreaded programming.

Deadlock involves threads waiting indefinitely for each other, which halts progress, often requiring a restart or intervention.

Race Conditions involve unpredictable results due to concurrent access to shared data, leading to data inconsistency.

Using synchronization techniques and careful resource management can help prevent both deadlock and race conditions, resulting in safer and more predictable multithreaded programs.

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a standard Java API that enables Java applications to interact with a wide range of databases. It provides methods for querying and updating data in a database and is widely used for developing Java applications that need to communicate with databases like MySQL, PostgreSQL, Oracle, and others.

1. Introduction to JDBC

JDBC allows Java programs to:

- Connect to a database.
- Send SQL queries and update statements to the database.
- Process the results retrieved from the database.

JDBC provides a universal data access API that is independent of any particular database or platform, enabling developers to switch databases without altering their Java code significantly.

2. JDBC Architecture

The JDBC architecture consists of two main components:

1. **JDBC API**: This provides a standard interface for Java applications to connect to the database, execute SQL queries, and retrieve results. The JDBC API includes classes and interfaces such as DriverManager, Connection, Statement, PreparedStatement, and ResultSet.
2. **JDBC Driver**: JDBC drivers are database-specific implementations of the JDBC API that communicate with the database. JDBC drivers translate the API calls into database-specific calls, making the interaction between Java applications and databases possible. There are four types of JDBC drivers:
    - **Type 1**: JDBC-ODBC Bridge Driver
    - **Type 2**: Native API Driver
    - **Type 3**: Network Protocol Driver
    - **Type 4**: Thin Driver (pure Java driver; commonly used for databases like MySQL)

3. Installing MySQL and MySQL Connector/J

To use JDBC with MySQL, you need to install both MySQL and the MySQL Connector/J.

Installing MySQL

1. Download the MySQL installer from the MySQL official website.
2. Run the installer and follow the installation steps.
3. Set up a root password and configure any other settings as needed.

Installing MySQL Connector/J

The MySQL Connector/J is the JDBC driver for MySQL, which is required to connect Java applications to a MySQL database.

1. Download the MySQL Connector/J from the MySQL Connector/J download page.
2. Extract the downloaded ZIP file, and locate the mysql-connector-java-<version>.jar file.
3. Add this .jar file to your project's classpath. In IDEs like IntelliJ or Eclipse, you can do this by right-clicking your project, selecting "Add External JARs," and choosing the Connector/J JAR file.

4. JDBC Environment Setup

To set up the JDBC environment in a Java application:

1. Ensure the MySQL Connector/J JAR file is in your project's classpath.
2. Import necessary JDBC packages:

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.SQLException;
```

5. Establishing JDBC Database Connections

To establish a connection with a database in Java, follow these steps:

1. **Load the JDBC Driver** (optional since JDBC 4.0):

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

2. **Establish a Connection**:
    - Use DriverManager.getConnection() with the JDBC URL, username, and password.
    - The JDBC URL format for MySQL is:

```
jdbc:mysql://<hostname>:<port>/<database_name>
```

For example:

```
String url = "jdbc:mysql://localhost:3306/mydatabase";

String username = "root";

String password = "password";


Connection connection = DriverManager.getConnection(url, username, password);
```

3. **Create a Statement**:

Statement statement = connection.createStatement();

4. **Execute Queries**:

ResultSet resultSet = statement.executeQuery("SELECT * FROM users");

5. **Close the Connection**:

resultSet.close();

statement.close();

connection.close();

6. ResultSet Interface

The ResultSet interface represents the result set obtained by executing a SQL query and provides methods to navigate and retrieve data from it. A ResultSet can be thought of as a table of data, with rows representing each record returned by the query.

Commonly Used Methods of the ResultSet Interface

1. **Navigating the ResultSet**:
    - next(): Moves the cursor to the next row. Returns false if there are no more rows.
    - previous(): Moves the cursor to the previous row (only if ResultSet is scrollable).
    - first(), last(): Moves to the first or last row.
2. **Retrieving Data**:
    - getString(columnLabel): Retrieves a column as a String.
    - getInt(columnLabel): Retrieves a column as an int.
    - getDouble(columnLabel): Retrieves a column as a double.
    - Column labels can be the column name or the column index.

Example Program Using JDBC to Query MySQL Database

Here's an example program that connects to a MySQL database, retrieves data from a table, and displays it.

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.SQLException;

```
public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try (Connection connection = DriverManager.getConnection(url, username, password)) {
            System.out.println("Database connected successfully!");

            Statement statement = connection.createStatement();
            String query = "SELECT id, name, email FROM users";
            ResultSet resultSet = statement.executeQuery(query);

            System.out.println("User Details:");
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");

                System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
            }

            resultSet.close();
            statement.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

This program:

- Connects to a MySQL database.
- Queries a table named users.
- Iterates over the ResultSet to print each user's id, name, and email.

JavaFX Scene Builder

JavaFX Scene Builder is a **visual design tool** used for building the user interface (UI) of JavaFX applications without manual coding. Instead of writing Java code for UI layouts, Scene Builder allows you to visually design the interface and automatically generates an FXML file to represent the structure.

Key Features

1. **Drag-and-Drop Interface**
    - You can easily add UI components like buttons, labels, and text fields by dragging them from the toolbox onto the design canvas.
2. **Set Properties for Controls**
    - Configure UI components by setting properties such as text, size, alignment, and style directly in Scene Builder.
3. **FXML Code Generation**
    - Automatically generates an FXML file based on the layout you design. This file can be loaded in your JavaFX application for rendering.
4. **Link to Controller Classes**
    - Allows you to assign event handlers and bind UI components to your JavaFX application's controller class.

How to Use

1. **Download and Install Scene Builder**
    - Download Scene Builder from the official Gluon website.
    - Install it on your computer.
2. **Design the User Interface**
    - Open Scene Builder and start a new design.
    - Add nodes like buttons, text fields, or labels by dragging them from the toolbox to the design area.
    - Arrange and configure properties for each node using the Properties panel.
3. **Save as FXML**
    - Once the design is complete, save it as an .fxml file.
    - Example: MainUI.fxml.

4. **Integrate FXML with Your JavaFX Application**
    - Use the FXMLLoader class in your JavaFX code to load the saved FXML file.

Example Code Integration

FXML File (MainUI.fxml):

xml

Copy code

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.StackPane?>

<StackPane xmlns:fx="http://javafx.com/fxml">
   <Button text="Click Me!" fx:id="myButton"/>
</StackPane>
```

Java Application:

java

Copy code

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("MainUI.fxml"));
        Scene scene = new Scene(root, 400, 300);
```

```
        primaryStage.setTitle("JavaFX with Scene Builder");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

2. JavaFX App Window Structure

A JavaFX application follows a hierarchical structure where components are organized into three main layers: **Stage**, **Scene**, and **Nodes**. Let's explore these components in detail:

1. Stage

- **What is it?**
  The **Stage** is the top-level container that represents the application window. It is automatically created when a JavaFX application starts.
- **Key Features:**
  - Controls the window's title, size, and visibility.
  - Acts as the main entry point for displaying the user interface.

Example:

```
primaryStage.setTitle("My JavaFX Application");
primaryStage.setWidth(800);
primaryStage.setHeight(600);
```

2. Scene

- **What is it?**
  The **Scene** holds all the visual elements (nodes) of the application and represents the content to be displayed in the Stage.
- **Key Features:**
  - Acts as a container for the **Scene Graph**, which is a hierarchical tree of nodes.

- Defines properties like dimensions and styling.
- A Stage can have only one Scene at a time, but the Scene can be swapped dynamically.

Example:

Scene scene = new Scene(rootNode, 400, 300);

primaryStage.setScene(scene);

3. Nodes

- **What are they?**
  Nodes are the building blocks of the Scene Graph. They are individual components like buttons, labels, text fields, and layout panes.
- **Types of Nodes:**
  - **Root Node:** The top-most node in the Scene Graph (e.g., layout panes like StackPane, VBox, etc.).
  - **Child Nodes:** UI elements (e.g., Button, Label, Text, ImageView) added to the Root Node or other containers.

Example:

Label label = new Label("Hello, JavaFX!");

Button button = new Button("Click Me");

VBox rootNode = new VBox(10, label, button);

Complete Example:

Here's a simple JavaFX application demonstrating the structure:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;


public class JavaFXAppStructure extends Application {
    @Override
```

```
    public void start(Stage primaryStage) {
        // Create Nodes
        Label label = new Label("Welcome to JavaFX!");
        Button button = new Button("Click Me");

        // Create Root Node (Layout Pane)
        VBox rootNode = new VBox(10, label, button);

        // Create Scene and Set Dimensions
        Scene scene = new Scene(rootNode, 400, 300);

        // Set Scene to the Stage
        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App Window Structure");
        primaryStage.show();
    }

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

3. Displaying Text and Images in JavaFX

JavaFX provides straightforward ways to display both **text** and **images** in a user interface. Here's how you can use the Label, Text, and ImageView nodes effectively:

Displaying Text

Options:

1. **Label**
    - Used for short, non-editable text.
    - Often used in forms or as a description for UI components.

Example:

```
Label label = new Label("Welcome to JavaFX!");
```

2. **Text**
    - More flexible than Label, allowing custom fonts, styles, and multi-line text.
    - Used for rich text display or larger content.

Example:

```
Text text = new Text("Hello, JavaFX Text Node!");
text.setStyle("-fx-font-size: 20px; -fx-fill: blue;");
```

Code Example for Text Display:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class DisplayTextExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("This is a Label!");
        Text text = new Text("This is a Text node!");

        VBox root = new VBox(10, label, text);

        Scene scene = new Scene(root, 300, 200);
        primaryStage.setTitle("Displaying Text");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

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

Displaying Images

Using Image and ImageView:

1. **Image:** Represents the image file loaded from a URL or local file.
2. **ImageView:** Displays the image in the scene.

Steps:

- Create an Image object.
- Pass it to an ImageView.

Code Example for Image Display:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class DisplayImageExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Load image from a file (adjust the file path as needed)
        Image image = new Image("file:your_image_path.jpg");
        ImageView imageView = new ImageView(image);

        // Optional: Set image dimensions
        imageView.setFitWidth(200);
        imageView.setPreserveRatio(true);
```

```
        VBox root = new VBox(imageView);

        Scene scene = new Scene(root, 300, 300);
        primaryStage.setTitle("Displaying Image");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

4. Event Handling in JavaFX

Event handling in JavaFX allows you to define actions or behaviors that occur when a user interacts with UI components, such as clicking a button, pressing a key, or moving the mouse. JavaFX uses an event-driven model to handle these interactions.

Key Components of Event Handling

1. **Event Source**
   The UI component that generates the event (e.g., Button, TextField).
2. **Event Handler**
   A method or lambda expression that defines the response to the event.
3. **Event Object**
   Provides information about the event, such as the source of the event and event type.

Steps to Handle Events

1. Set an Event Handler

You can set an event handler for a UI component using:

- A **Lambda Expression**
- An **Anonymous Class**
- A **Separate Method**

2. Use Event Methods

The most common method for handling events is setOnAction, which is used for buttons and similar controls.

Examples

1. Button Click Event

Using a **Lambda Expression**:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ButtonEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(e -> System.out.println("Button clicked!"));

        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Button Click Event");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

2. Handling Mouse Events

JavaFX provides methods like setOnMouseEntered and setOnMouseClicked for handling mouse interactions.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MouseEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Text text = new Text("Hover over me!");
        text.setOnMouseEntered(e -> text.setText("Mouse Entered!"));
        text.setOnMouseExited(e -> text.setText("Hover over me!"));

        StackPane root = new StackPane(text);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Mouse Event Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

3. Handling Events with a Separate Method

You can define a separate method to handle the event.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class SeparateMethodEventExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(this::handleButtonClick);

        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Event Handling with Separate Method");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private void handleButtonClick(javafx.event.ActionEvent event) {
        System.out.println("Button was clicked!");
    }

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

}

Event Types

- **ActionEvent**: Triggered by actions like button clicks or menu item selection.
- **MouseEvent**: Triggered by mouse actions like clicks or movement.
- **KeyEvent**: Triggered by keyboard actions like key presses or releases.
- **WindowEvent**: Triggered by changes in the application window (e.g., close or resize).

---

5. Laying Out Nodes in the Scene Graph

In JavaFX, layout panes are used to organize and position nodes (UI components) within the Scene Graph. Each layout pane provides a specific way to arrange its children.

Common Layout Panes

1. HBox (Horizontal Layout)

- **Description:** Arranges its children in a single horizontal row.
- **Use Case:** Useful for toolbars or placing buttons side-by-side.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class HBoxExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");

        HBox hbox = new HBox(10, btn1, btn2, btn3); // Spacing between nodes
```

```
        Scene scene = new Scene(hbox, 300, 100);

        primaryStage.setTitle("HBox Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

2. VBox (Vertical Layout)

- **Description:** Arranges its children in a single vertical column.
- **Use Case:** Useful for forms, menus, or stacked controls.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class VBoxExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");

        VBox vbox = new VBox(10, btn1, btn2, btn3); // Spacing between nodes
```

```
        Scene scene = new Scene(vbox, 200, 150);

        primaryStage.setTitle("VBox Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

3. GridPane (Grid Layout)

- **Description:** Arranges children in a flexible grid of rows and columns.
- **Use Case:** Useful for complex forms or tables.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class GridPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");
        Button btn4 = new Button("Button 4");
```

```
        GridPane grid = new GridPane();
        grid.setHgap(10); // Horizontal gap between columns
        grid.setVgap(10); // Vertical gap between rows

        // Adding buttons to the grid (column, row)
        grid.add(btn1, 0, 0);
        grid.add(btn2, 1, 0);
        grid.add(btn3, 0, 1);
        grid.add(btn4, 1, 1);

        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setTitle("GridPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

---

4. BorderPane (Border Layout)

- **Description:** Divides the layout into five regions: top, bottom, left, right, and center.
- **Use Case:** Useful for creating applications with a header, footer, sidebar, and main content.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
```

```
public class BorderPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button topButton = new Button("Top");
        Button bottomButton = new Button("Bottom");
        Button leftButton = new Button("Left");
        Button rightButton = new Button("Right");
        Button centerButton = new Button("Center");

        BorderPane borderPane = new BorderPane();
        borderPane.setTop(topButton);
        borderPane.setBottom(bottomButton);
        borderPane.setLeft(leftButton);
        borderPane.setRight(rightButton);
        borderPane.setCenter(centerButton);

        Scene scene = new Scene(borderPane, 400, 300);
        primaryStage.setTitle("BorderPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

6. Handling Mouse Events in JavaFX

JavaFX provides a rich set of mouse events to handle interactions such as clicks, drags, and hover actions. These events are defined in the MouseEvent class, and you can attach event handlers to any node in your scene.

Common Mouse Events

1. **Mouse Click Events**
   - setOnMouseClicked: Triggered when a mouse button is clicked on a node.
2. **Mouse Hover Events**
   - setOnMouseEntered: Triggered when the mouse enters a node.
   - setOnMouseExited: Triggered when the mouse leaves a node.
3. **Mouse Drag Events**
   - setOnMouseDragged: Triggered when the mouse is dragged while pressing a button.
   - setOnMousePressed / setOnMouseReleased: Triggered when the mouse button is pressed/released.

Example 1: Handling a Mouse Click

This example changes the text of a Label when clicked.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class MouseClickExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("Click Me!");

        // Set Mouse Click Event
        label.setOnMouseClicked(e -> label.setText("Label Clicked!"));
```

```
        StackPane root = new StackPane(label);
        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Mouse Click Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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