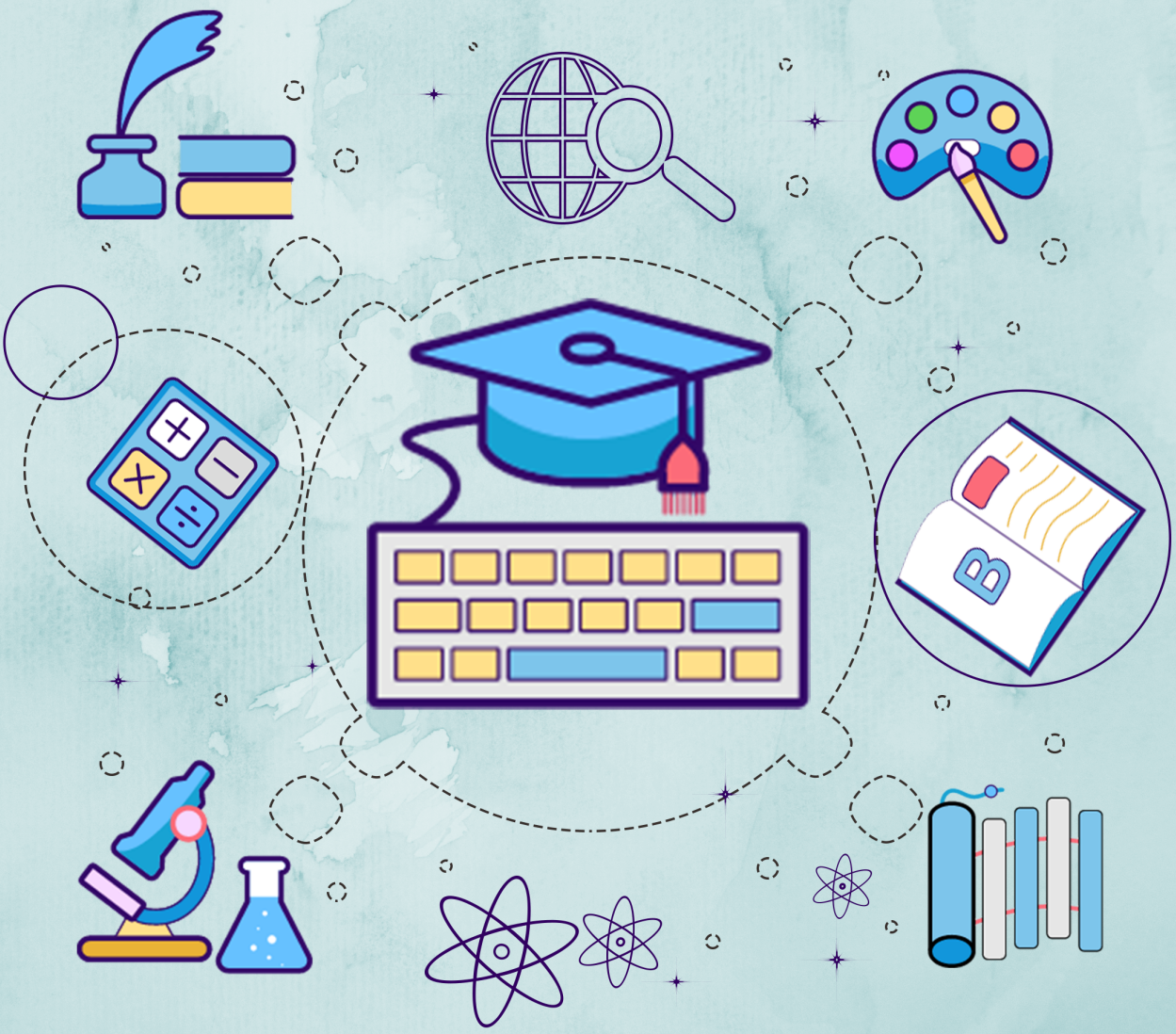
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

OBJECT ORIENTED PROGRAMMING USING JAVA

CST 205

Module 2

Related Link :

- KTU S3 STUDY MATERIALS
- KTU S3 NOTES
- KTU S3 SYLLABUS
- KTU S3 TEXTBOOK PDF
- KTU S3 PREVIOUS YEAR SOLVED QUESTION PAPER

MODULE 2
CORE JAVA FUNDAMENTALS

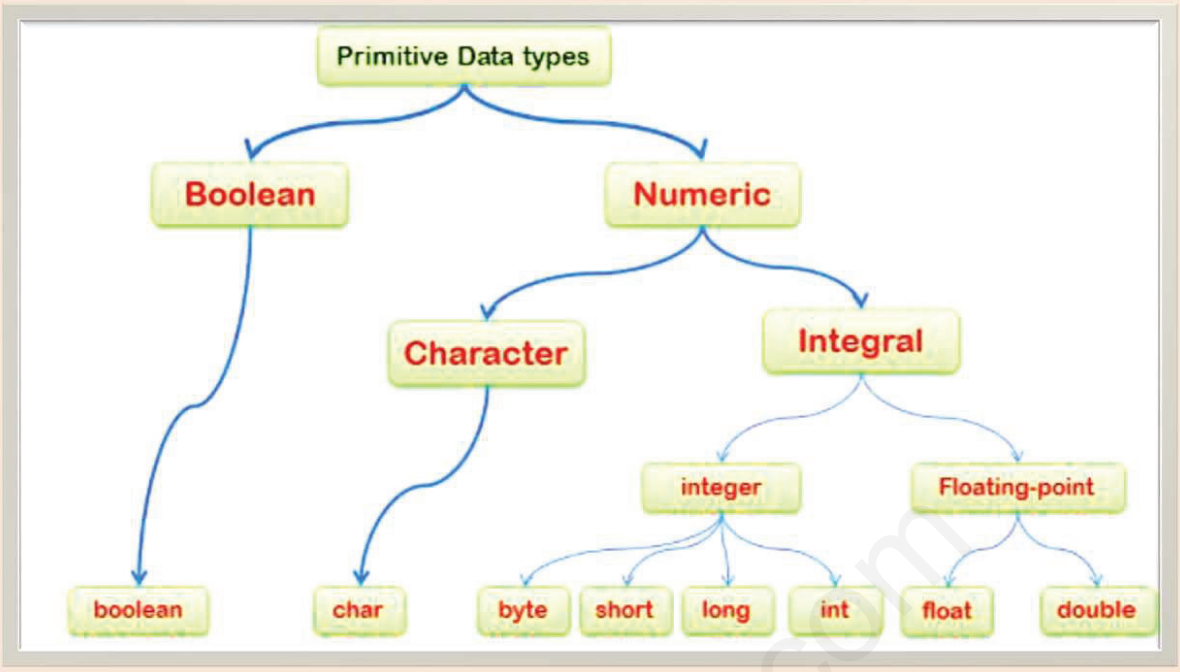
CHAPTER 1
DATA TYPES, OPERATORS
&
CONTROL STATEMENTS

DATA TYPES

- **Data type** defines the values that a variable can take, for example if a variable has int data type, it can only take integer values.
- Data types specify the different sizes and values that can be stored in the variable.
- There are two types of data types in Java:

  Primitive data types

  Non-primitive data types

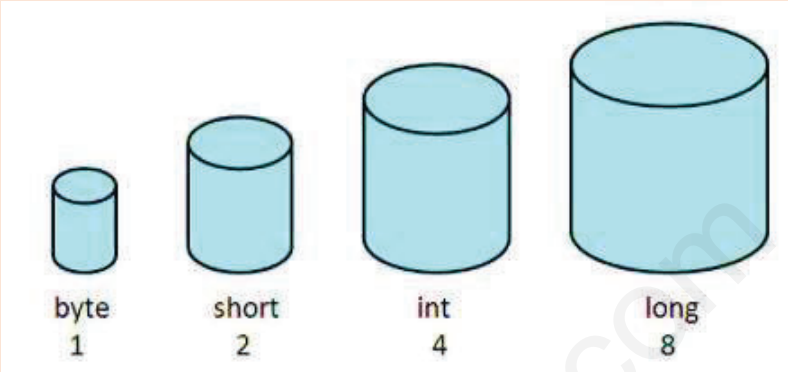


❖ Primitive Data Types (Fundamental Data Types)

- Primitive Data Types are predefined and available within the Java language. There are 8 types of primitive data types:

Data Type	Default Value	Default size
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0L	8 bytes
float	0.0f	4 bytes
double	0.0d	8 bytes
boolean	false	1 bit
char	'\u0000'	2 bytes

- **byte**, **short**, **int** and **long** data types are used for storing whole numbers.
- **float** and **double** are used for fractional numbers.
- **char** is used for storing characters(letters).
- **boolean** data type is used for variables that holds either true or false.




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

        byte num;

        num = 113;
        System.out.println(num);
    }
}
```

Output 113

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

        short num;

        num = 150;
        System.out.println(num);
    }
}
```

Output 150

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

        boolean b = false;
        System.out.println(b);
    }
}
```

Output false

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

        char ch = 'Z';
        System.out.println(ch);
    }
}
```

Output Z

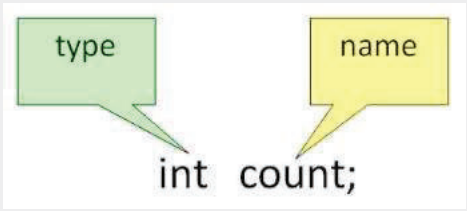
Variables In JAVA

- **Variable in Java** is a data container that stores the data values during Java program execution.
- Variable is a memory location name of the data.
- variable="vary + able" that means its value can be changed.
- In order to use a variable in a program we need to perform 2 steps
  1. **Variable Declaration**
  2. **Variable Initialization**

1. Variable Declaration

Syntax: data_type variable_name ;

Eg: int a,b,c;
float pi;
double d;

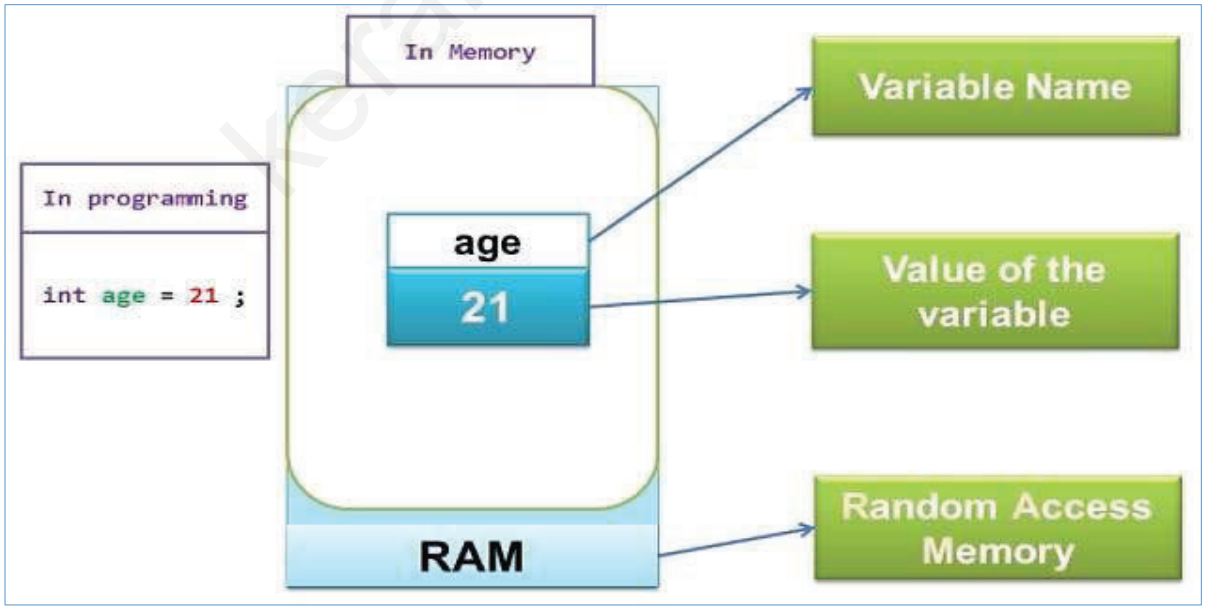


2. Variable Initialization

Syntax : data_type variable_name = value;

Eg: int a=2,b=4,c=6;
float pi = 3.14f;
double val = 20.22d;
char a = 'v';

int num = 45.66;



Types of variables

1. **Local variables -** declared inside the method.
2. **Instance Variable -** declared inside the class but outside the method.
3. **Static variable** - declared as with static keyword.

Example:

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```


Java Type Casting or Type Conversion

- Type casting is when you assign a value of one primitive data type to another type.
- In Java, there are two types of casting:

1. Widening Casting (automatically) – converting a smaller type to a larger type size (called Type Conversion)
   byte -> short -> char -> int -> long -> float -> double
2. Narrowing Casting (manually) – converting a larger type to a smaller size type (called Type Casting)
   double -> float -> long -> int -> char -> 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
```

Widening

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);
  }
}
```

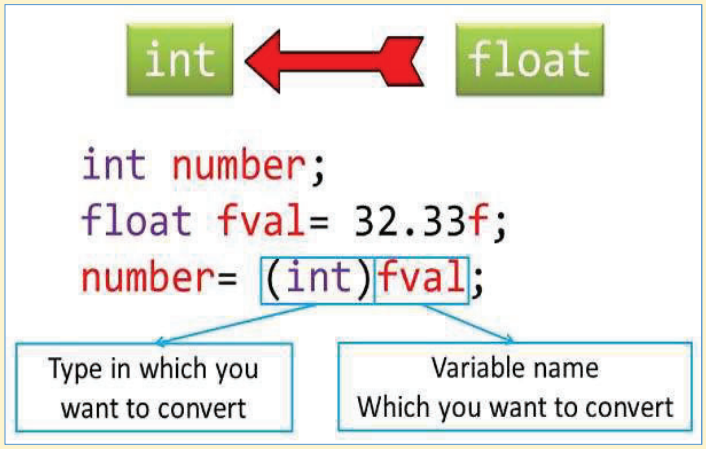
Output

```
The double value: 10.99
The integer value: 10
```

Narrowing

❖Truncation

- ➢when a floating-point value is assigned to an integer type: truncation takes place, As you know, integers do not have fractional components
- ➢Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- ➢For example, if the value 45.12 is assigned to an integer, the resulting value will simply be 45. The 0.12 will have been truncated.
- ➢No automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.




```
class Casting{
public static void main(String[] args){
        int number;
        float fval= 32.33f;
        number= (int)fval;
        System.out.println(number);
        }
}
```

Output:

```
32
Press any key to continue . . .
```


OPERATORS

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.
- Java operators can be divided into following categories:
  - Arithmetic Operators
  - Relational Operators
  - Bitwise Operators
  - Logical Operators
  - Assignment Operators
  - conditional operator (Ternary)

❖Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds two operands	5 + 10 =15
- (Subtraction)	Subtract second operands from first. Also used to Concatenate two strings	10 - 5 =5
* (Multiplication)	Multiplies values on either side of the operator.	10 * 5 =50
/ (Division)	Divides left-hand operand by right-hand operand.	10 / 5 =2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	5 % 2 =1
++ (Increment)	Increases the value of operand by 1.	2++ gives 3
-- (Decrement)	Decreases the value of operand by 1.	3-- gives 2


```
class ArithmeticOperations {

    public static void main (String[] args){

        int answer = 2 + 2;
        System.out.println(answer);

        answer = answer - 1;
        System.out.println(answer);

        answer = answer * 2;
        System.out.println(answer);

        answer = answer / 2;
        System.out.println(answer);

        answer = answer + 8;
        System.out.println(answer);


        answer = answer % 7;
        System.out.println(answer);
    }
}
```

Output

4
3
6
3
11
4

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

        int x= 5;
        System.out.println(x++);
        System.out.println(++x);
        System.out.println(x--);
        System.out.println(--x);
        }
}
```

Output

```
5
7
7
5
Press any key to continue . . .
```

X++ is Use –Then - Change

```
class IncrementDecrementExample{

        public static void main(String args[]){
        int p=10;
        int q=10;
        System.out.println(p++ + ++p);//10+12=22
        System.out.println(q++ + q++);//10+11=21

        }
}
```

Output

```
22
21
Press any key to continue . . .
```

++x is Change – Then - Use

Use of Modulus Operator

```
class ModulusOperator {
   public static void main(String args[]) {
      int     R = 42;
      double S = 62.25;

      System.out.println("R mod 10 = " + R % 10);
      System.out.println("S mod 10 = " + S % 10);
   }
}
```

Output

```
R mod 10 = 2
S mod 10 = 2.25
Press any key to continue . . .
```

Joining or Concatenate two strings

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

        String firstName = "Rahim";
        String lastName  = "Ramboo";

        String fullName  = firstName + lastName;
        System.out.println(fullName);
     }
}
```

Output

```
RahimRamboo
Press any key to continue . . .
```

❖Relational Operators

Operators	Descriptions	Examples
== (equal to)	This operator checks the value of two operands, if both are **equal**, then it returns true otherwise false.	(2 == 3) is not true.
!= (not equal to)	This operator checks the value of two operands, if both are **not equal**, then it returns true otherwise false.	(4 != 5) is true.
> (greater than)	This operator checks the value of two operands, if the left side of the operator is **greater**, then it returns true otherwise false.	(5 > 56) is not true.
< (less than)	This operator checks the value of two operands if the left side of the operator is **less**, then it returns true otherwise false.	(2 < 5) is true.
>= (greater than or equal to)	This operator checks the value of two operands if the left side of the operator is **greater or equal**, then it returns true otherwise false.	(12 >= 45) is not true.
<= (less than or equal to)	This operator checks the value of two operands if the left side of the operator is **less or equal**, then it returns true otherwise false.	(43 <= 43) is true.


```
public class RelationalOperator {

    public static void main(String args[]) {
        int p = 5;
        int q = 10;

        System.out.println("p == q = " + (p == q) );
        System.out.println("p != q = " + (p != q) );
        System.out.println("p > q = " + (p > q) );
        System.out.println("p < q = " + (p < q) );
        System.out.println("q >= p = " + (q >= p) );
        System.out.println("q <= p = " + (q <= p) );
    }
}
```

Output

```
p == q = false
p != q = true
p > q = false
p < q = true
q >= p = true
q <= p = false
Press any key to continue
```

❖ Bitwise Operators

Operator	Description
& (bitwise and)	Bitwise AND operator give true result if both operands are true. otherwise, it gives a false result.
| (bitwise or)	Bitwise OR operator give true result if any of the operands is true.
^ (bitwise XOR)	Bitwise Exclusive-OR Operator returns a true result if both the operands are different. otherwise, it returns a false result.
~ (bitwise compliment)	Bitwise One's Complement Operator is unary Operator and it gives the result as an opposite bit.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.


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

        int A = 10;
        int B = 3;
        int Y;
        Y = A & B;
        System.out.println(Y);

    }
}
```

Output

```
2
Press any key to continue . . .
```

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

        int A = 10;
        int B = 3;
        int Y;
        Y = A | B;
        System.out.println(Y);

    }
}
```

Output

```
11
Press any key to continue . . .
```

❖Logical Operators

Operator	Description	Example
&& **(logical and)**	If both the operands are non-zero, then the condition becomes true.	(0 && 1) is false
|| **(logical or)**	If any of the two operands are non-zero, then the condition becomes true.	(0 || 1) is true
! **(logical not)**	Logical NOT Operator Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(0 && 1) is true


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

      System.out.println("b1 && b2: " + (b1&&b2));
      System.out.println("b1 || b2: " + (b1||b2));
      System.out.println("!(b1 && b2): " + !(b1&&b2));
   }
}
```

Output:

```
b1 && b2: false
b1 || b2: true
!(b1 && b2): true
```

❖Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3

❖conditional Operator / Ternary Operator (? :)

Expression1 ? Expression2 : Expression3

Expression ? value if true : value if false

```
public class ConditionalOperator {

    public static void main(String args[]) {
        int a, b;
        a = 20;
        b = (a == 1) ? 10: 25;
        System.out.println( "Value of b is : " +  b );
        b = (a == 20) ? 20: 30;
        System.out.println( "Value of b is : " + b );
    }
}
```

Output

```
Value of b is : 25
Value of b is : 20
Press any key to continue . . .
```

```
public class TernaryOperatorDemo {

   public static void main(String args[]) {
        int num1, num2;
        num1 = 25;
        /* num1 is not equal to 10 that's why
         * the second value after colon is assigned
         * to the variable num2
         */
        num2 = (num1 == 10) ? 100: 200;
        System.out.println( "num2: "+num2);

        /* num1 is equal to 25 that's why
         * the first value is assigned
         * to the variable num2
         */
        num2 = (num1 == 25) ? 100: 200;
        System.out.println( "num2: "+num2);
   }
}
```

Output:

```
num2: 200
num2: 100
```


Operator Precedence

- Evaluate 2*x-3*y ?

  (2x)-(3y) or 2(x-3y) which one is correct??????

- Evaluate A / B * C

  A / (B * C) or (A / B) * C Which one is correct?????

➢ To answer these questions satisfactorily one has to understand the priority or precedence of operations.

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

- **Precedence order -** When two operators share an operand the operator with the higher precedence goes first.
- **Associativity -** When an expression has two operators with the same precedence, the expression is evaluated according to its associativity.

➢Larger number means higher precedence

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right

Larger number means higher precedence

10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Evaluate i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8	operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8	operation: /
i = 1 + 1+ 8 - 2 + 5 / 8	operation: /
i = 1 + 1+ 8 - 2 + 0	operation: /
i = 2 + 8- 2 + 0	operation: +
i = 10 - 2 + 0	operation: +
i = 8 + 0	operation : -
i = 8	operation: +

SELECTION STATEMENTS

- Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Also called decision making statements
- Java supports various selection statements, like if, if-else and switch
- There are various types of if statement in java.
- if statement
- if-else statement
- nested if statement
- if-else-if ladder

❖ If statement

- Use the if statement to specify a block of Java code to be executed if a condition is true.

Syntax

```
if (condition)
    {
        // block of code to be executed if the condition is true
     }
```

Example

```
class SampleIf
{
   public static void main(String args[])
   {
      int a=10;
      if (a > 0) {
         System.out.println("a is greater than 0");
      }
   }
}
```

Output:

a is greater than 0

❖ if-else Statement

➢If-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax

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

Example

```
class SampleIfElse
{
   public static void main(String args[])
   {
      int a=10;
      if (a > 0) {
         System.out.println("a is greater than 0");
      }
      else
      {
         System.out.println("a is smaller than 0");
      }
   }
}
```

Output:

a is greater than 0


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

Nested if else Statement

Syntax

```
class SampleNestedIfElse
{
   public static void main(String args[])
   {
      int a=10,b=20,c=30;
      if (a>b)
      {
         if (a>c)
         {
             System.out.println("a is greatest.");
         }
         else
         {
              System.out.println("c is greatest.");
         }
      }
      else
      {
         if (b>c)
         {
             System.out.println("b is greatest.");
         }
         else
         {
              System.out.println("c is greatest.");
         }
      }
   }
}
```

Output:

c is greatest.

❖ if else if ladder

Syntax

```
if (condition)
 {
   // block of code to be executed if the condition is true
 }
else if (condition)
 {
   // block of code to be executed if the condition is true
 }
else
 {
  // block of code to be executed if the condition is true
 }
```

```java
class IfElseIfLadder {
    public static void main(String[] args){
        double score = 55;

        if (score >= 90.0)
        System.out.println('A');
        else if (score >= 80.0)
        System.out.println('B');
        else if (score >= 70.0)
        System.out.println('C');
        else if (score >= 60.0)
        System.out.println('D');
        else
      System.out.println('F');
    }
}
```

Output

```
F
Press any key to continue . . .
```



```java
class SampleLadderIfElse
{
    public static void main(String args[])
    {
        int a=10;
        if (a > 0) {
            System.out.println("a is +ve");
        }
        else if (a < 0) {
            System.out.println("a is -ve");
        }
        else {
            System.out.println("a is zero");
        }
    }
}
```

Output:

```
a is +ve
```

❖ If...Else & Ternary Operator – A comparison

```java
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

```java
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

❖ switch case

- The if statement in java, makes selections based on a single true or false condition. But switch case have multiple choice for selection of the statements
- It is like if-else-if ladder statement
- **How to Java switch works:**

- Matching each expression with case
- Once it match, execute all case from where it matched.
- Use break to exit from switch
- Use default when expression does not match with any case

Syntax

```
 switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

```
class SampleSwitch
{
   public static void main(String args[])
   {
        int day = 4;
        switch (day) {
          case 1:
                System.out.println("The day is Monday");
                break;
          case 2:
                System.out.println("The day is Tuesday");
                break;
          case 3:
                System.out.println("The day is Wednesday");
                break;
          case 4:
                System.out.println("The day is Thursday");
                break;
          case 5:
                System.out.println("The day is Friday");
                break;
          case 6:
                System.out.println("The day is Saturday");
                break;
          case 7:
                System.out.println("The day is Sunday");
                break;
          deafault:
                System.out.println("Please enter between 1 to 7.");
        }
   }
}
```

Output

The day is Thursday

Why break is necessary in switch statement ?

- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement
- This has the effect of jumping out of the switch.
- The break statement is optional. If you omit the break, execution will continue on into the next case.

Nested Switch

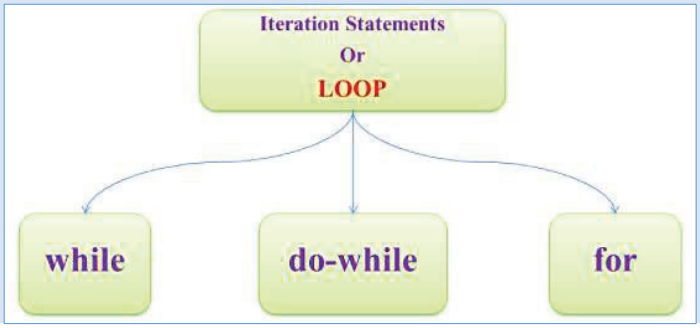
```
class NestedSwitchCase {
public static void main(String args[]) {
int count = 1;
int target = 1;
 switch(count) {
         case 1:
                switch(target) { // nested switch
                        case 0:
                         System.out.println("target is zero inner switch");
                         break;
                        case 1: // no conflicts with outer switch
                         System.out.println("target is one inner switch");
                         break;
                        }
         break;

case 2:
     System.out.println("case 2 outer switch");
        }
 }
}
```

```
target is one inner switch
Press any key to continue . . .
```


Iteration Statements (Loop)

- A loop can be used to tell a program to execute statements repeatedly
- A loop repeatedly executes the same set of instructions until a termination condition is met.



❖ While Loop

➢In **while loop** first checks the condition if the condition is true then control goes inside the loop body otherwise goes outside of the body.

Syntax

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


Example - 1

```
class WhileLoopExample
{
     public static void main(String args[])
     {
          int count = 0;
             while(count < 100){
             System.out.println("Welcome to atnyla!");
             count++;
          }
     }
}
```

Output

```
Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!
.....
.....
.....
Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!
Press any key to continue . . .
```

Example - 2

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

Output

```
1
2
3
4
5
6
7
8
9
10
Press any key to continue . .
```


Example - 3

```
class WhileLoopSingleStatement {
     public static void main(String[] args){
          int count = 1;
          while (count <= 11)
           System.out.println("Number Count : " + count++);
     }
}
```

Output

```
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
Number Count : 5
Number Count : 6
Number Count : 7
Number Count : 8
Number Count : 9
Number Count : 10
Number Count : 11
Press any key to continue . .
```

Example - 4

```
public class WhileInfiniteLoop {
public static void main(String[] args) {
      while(true){
            System.out.println("infinitive while loop");
      }
   }
}
```

Output

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
.....
.....
.....
.....
infinite time it will print like this
```


Example - 5 (Boolean Condition inside while loop)

```
class WhileLoopBoolean {
     public static void main(String[] args){
          boolean a = true;
          int count = 0 ;
          while (a)
               {
               System.out.println("Number Count : " + count);
               count++;
               if(count==5)
                   a = false;
               }
     }
}
```

Output

```
Number Count : 0
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
Press any key to continue . . .
```

❖ do...while loop

➢ A do while loop is a control flow statement that executes a block of code at least once, and then repeatedly executes the block, or not, depending on a given condition at the end of the block (in while).

Syntax

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


Example -1

```
class DoWhile {
public static void main(String args[]) {
        int n = 0;
        do {
                System.out.println("Number " + n);
                n++;
            } while(n < 10);
    }
}
```

Output

```
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Press any key to continue . . .
```

Example -2 (Infinitive do-while Loop)

```
public class InfiniteDoWhileLoop {
public static void main(String[] args) {
     do{
         System.out.println("infinitive do while loop");
      }while(true);
   }
}
```

Output

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
infinitive do while loop
.....
.....
.....
.....
infinite time it will print like this
```


Difference Between while and do-while Loop

BASIS FOR COMPARISON	WHILE	DO-WHILE
General Form	`while ( condition) { statements; //body of loop }`	`do{ . statements; // body of loop. . } while( Condition );`
Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

❖ for loop

➢For Loop is used to execute set of statements repeatedly until the condition is true.

Syntax

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

Initialization : It executes at once.
Condition : This check until get true.
Increment/Decrement: This is for increment or decrement.

Example 1

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

}
```

Output

```
1
2
3
4
5
6
7
8
9
10
Press any key to continue . . .
```

Example 2

```
/*
Demonstrate the for loop.
Call this file "ForLoopExample.java".
*/

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

        for(int x = 15; x < 25; x = x + 1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
         }

}
```

Output

```
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
value of x : 20
value of x : 21
value of x : 22
value of x : 23
value of x : 24
Press any key to continue . .
```


❖ For-each or Enhanced For Loop

➢ The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

Syntax

```
for (type variableName : arrayName)
 {
    // code block to be executed
 }
```

Example

```
/*
Demonstrate the for each loop.
save file "ForEachExample.java".
*/

public class ForEachExample {
public static void main(String[] args) {
      int array[]={10,11,12,13,14};
      for(int i:array){
            System.out.println(i);
      }
   }
}
```

Output

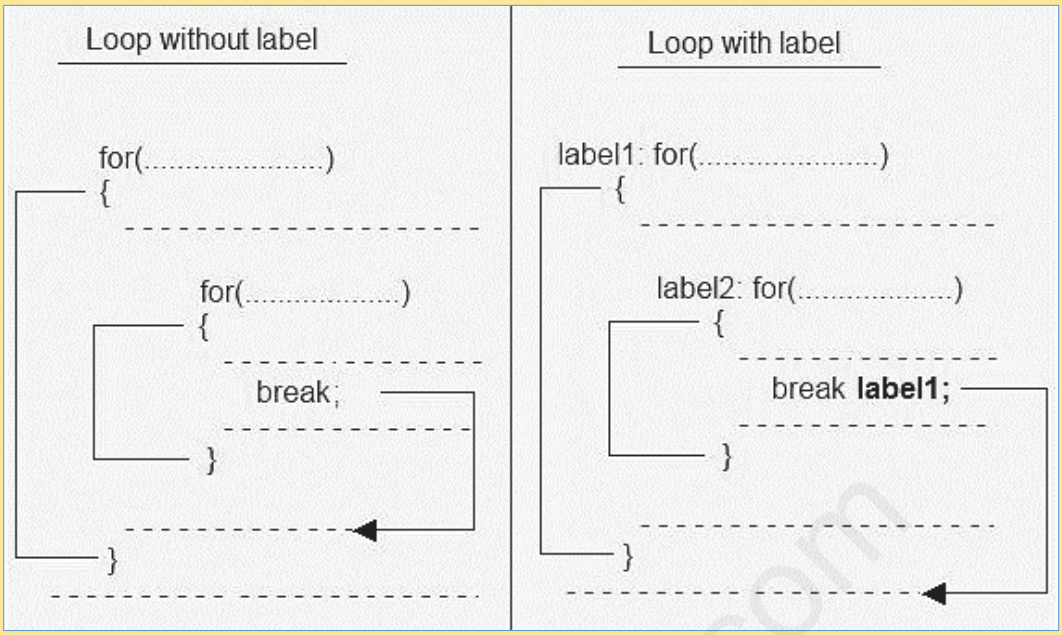
```
10
11
12
13
14
Press any key to continue . .
```


❖Labeled For Loop

- According to nested loop, if we put break statement in inner loop, compiler will jump out from inner loop and continue the outer loop again.
- What if we need to jump out from the outer loop using break statement given inside inner loop? The answer is, we should define **label** along with colon(:) sign before loop.

Syntax

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



Example without labelled loop

```
class WithoutLabelledLoop
{
     public static void main(String args[])
     {
          int i,j;
          for(i=1;i<=10;i++)
          {
               System.out.println();
               for(j=1;j<=10;j++)
               {
                    System.out.print(j + " ");
                    if(j==5)
                    break;              //Statement 1
               }
          }
     }
}
```

Output

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 Press any key to continue . . .
```

Example with labelled loop

```
class WithLabelledLoop
{
     public static void main(String args[])
     {
          int i,j;
          loop1:    for(i=1;i<=10;i++)
          {
               System.out.println();
               loop2:    for(j=1;j<=10;j++)
               {
                    System.out.print(j + " ");
                    if(j==5)
                         break loop1;      //Statement 1
               }
          }
     }
}
```

Output

```
1 2 3 4 5 Press any key to continue . . .
```


Jump Statements

❖ Java Break Statement

- The Java break statement is used to break loop or switch statement
- It breaks the current flow of the program at specified condition
- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- In case of inner loop, it breaks only inner loop.

Example 1

```
class SampleBreak
{
   public static void main(String args[])
   {
        int num= 1;
        while (num <= 10) {
          System.out.println(num);
          if(num==5)
          {
              break;
          }
          num++;
        }
   }
}
```

Output

```
1
2
3
4
5
```


Example 2

```
//Java Program to demonstrate the use of break statement
//inside the for loop.
public class BreakExample {
public static void main(String[] args) {
   //using for loop
   for(int i=1;i<=10;i++){
      if(i==5){
         //breaking the loop
         break;
      }
      System.out.println(i);
   }
}
}
```

Output:

```
1
2
3
4
```

Example 3

```
//Java Program to illustrate the use of break statement
//inside an inner loop
public class BreakExample2 {
public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
              if(i==2&&j==2){
                //using break statement inside the inner loop
                break;
              }
              System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```


Example 4

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
   //declaring variable
   int i=1;
   //do-while loop
   do{
      if(i==5){
        //using break statement
        i++;
        break;//it will break the loop
      }
      System.out.println(i);
      i++;
   }while(i<=10);
}
}
```

Output:

```
1
2
3
4
```

❖ Java Continue Statement

- The Java continue statement is used to continue the loop
- The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately
- It continues the current flow of the program and skips the remaining code at the specified condition.
- In case of an inner loop, it continues the inner loop only.

Example 1

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
   //for loop
   for(int i=1;i<=10;i++){
      if(i==5){
         //using continue statement
         continue;//it will skip the rest statement
      }
      System.out.println(i);
   }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Example 2

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement inside inner loop
                    continue;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```


Example 3

```
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
public static void main(String[] args) {
   //while loop
   int i=1;
   while(i<=10){
      if(i==5){
         //using continue statement
         i++;
         continue;//it will skip the rest statement
      }
      System.out.println(i);
      i++;
   }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Example 4

```
class myClass {
    public static void main( String args[] ) {
      label:
      for (int i=0;i<6;i++)
      {
        if (i==3)
        {
          continue label; //skips 3
        }
        System.out.println(i);
      }
    }
}
```

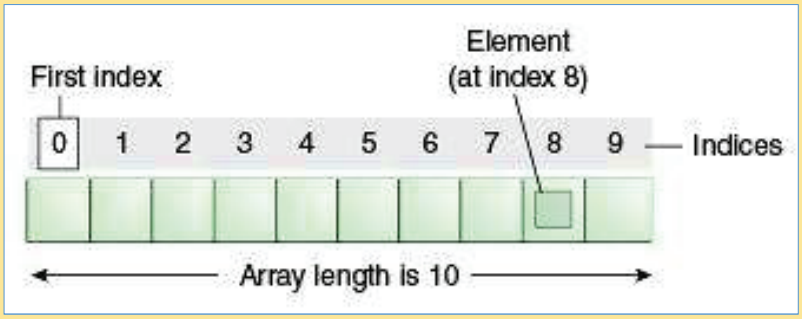
Output

```
0
1
2
4
5
```


ARRAY

- An array is a collection of similar data types.
- **Java array** is an object which contains elements of a similar data type.
- The elements of an array are stored in a contiguous memory location
- the size of an array is fixed and cannot increase to accommodate more elements
- It is also known as static data structure because size of an array must be specified at the time of its declaration.
- Array in Java is index-based, the first element of the array is stored at the 0th index

- Java provides the feature of anonymous arrays which is not available in C/C++.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store the only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Features of Array

- It is always indexed. The index begins from 0.
- It is a collection of similar data types.
- It occupies a contiguous memory location.

Types of Java Array

- Single Dimensional Array
- Multidimensional Array

❖ Single Dimensional Array in java

➢Array Declaration

Syntax: datatype[ ] arrayname;

Eg: int[ ] arr;
char[ ] name;
short[ ] arr;
long[ ] arr;
int[ ][ ] arr; //two dimensional array

In C program datatype arrayname[];

➢Initialization of Array

new operator is used to initializing an array.

Eg 1: int[ ] arr = new int[10];

or

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

Eg 2: String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Eg 3: double[] myList = new double[10];

➢**Accessing array element**

Example: To access 4th element of a given array

```
int[ ] arr = {10,24,30,50};
System.out.println("Element at 4th place" + arr[3]);
```

➢To find the length of an array, we can use the following syntax: array_name.length

Example:

```
public class MyClass
{
  public static void main(String[] args)
  {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    System.out.println(cars.length);
  }
}
```

Output 4

➢**Loop Through an Array**

```
public class MyClass
{
  public static void main(String[] args)
    {
      String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
      for (int i = 0; i < cars.length; i++)
        {
          System.out.println(cars[i]);
        }
    }
}
```

```
Volvo
BMW
Ford
Mazda
```

➢Loop Through an Array with For-Each

```
public class MyClass
{
   public static void main(String[] args)
      {
       String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
       for (String i : cars)
        {
            System.out.println(i);
        }
      }
}
```

```
Volvo
BMW
Ford
Mazda
```



```
class ArrayDemo{
      public static void main(String args[]){
          int array[] = new int[7];
          for (int count=0;count<7;count++){
              array[count]=count+1;
          }

          for (int count=0;count<7;count++){
               System.out.println("array["+count+"] = "+array[count]);
          }

      }
}
```

Output

```
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
array[5] = 6
array[6] = 7
```

```java
public class ArrayExample {

    public static void main(String[] args) {
        double[] myList = {3.9, 5.9, 22.4, 31.5};

        // Print all the array elements
        for (int i = 0; i < myList.length; i++) {
            System.out.println(myList[i] + " ");
        }

        // Summing all elements
        double total = 0;
        for (int i = 0; i < myList.length; i++) {
            total += myList[i];
        }
        System.out.println("Total is " + total);

        // Finding the largest element
        double max = myList[0];
        for (int i = 1; i < myList.length; i++) {
            if (myList[i] > max) max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}
```

Output

```
3.9
5.9
22.4
31.5
Total is 63.7
Max is 31.5
```


❖Two Dimensional array

➢Array Declaration

Syntax : datatype[ ][ ] arrayname;

Eg: int[][] myNumbers ;

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

➢Array Initialization

```
int[ ][ ] arrName = new int[10][10];
```

Or

```
int[ ][ ] arrName = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};  // 3 by 5 is the size of the array.
```

```java
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
  System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output

```
1 2 3
2 4 5
4 4 5
```



```java
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

Output

```
2 6 8
6 8 10
```

STRING

- Strings are used for storing text
- A String variable contains a collection of characters surrounded by double quotes

Eg: Create a variable of type String and assign it a value

String greeting = "Hello";

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String greeting = "Hello";
    System.out.println(greeting);
  }
}
```

Output

Hello

- In Java, string is basically an object that represents sequence of char values
- An array of characters works same as Java string. For example:

```
char[] ch={'j','o','s','e','p','h'};
String s=new String(ch);  //converting char array to string
```

is same as

```
String s="joseph";  //creating string by java string literal
```

❖String Length

- The length of a string can be found with the length() method

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    System.out.println("The length of the txt string is: " + txt.length());
  }
}
```

Output

The length of the txt string is: 26

❖toUpperCase() and toLowerCase()

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String txt = "Hello World";
    System.out.println(txt.toUpperCase());
    System.out.println(txt.toLowerCase());
  }
}
```

Output

HELLO WORLD

hello world

❖Finding a Character in a String

➢The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace)

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String txt = "Please locate where 'locate' occurs!";
    System.out.println(txt.indexOf("locate"));
  }
}
```

Output 7

❖String Concatenation

• The + operator can be used between strings to combine them. This is called concatenation

MyClass.java

```
public class MyClass {
  public static void main(String args[]) {
    String firstName = "John";
    String lastName = "Doe";
    System.out.println(firstName + " " + lastName);
  }
}
```

Output John Doe

❖concat() method

- We can also use the concat() method to concatenate two strings:

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String firstName = "John ";
    String lastName = "Doe";
    System.out.println(firstName.concat(lastName));
  }
}
```

Output John Doe

❖Special Characters

Consider the following example

String txt = "We are the so-called "Vikings" from the north.";

- Because strings must be written within quotes, Java will misunderstand this string
- The solution to avoid this problem, is to use the backslash escape character

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String txt = "We are the so-called \"Vikings\" from the north.";
    System.out.println(txt);
  }
}
```

Output We are the so-called "Vikings" from the north.

- The sequence \" inserts a double quote in a string
- The sequence \' inserts a single quote in a string
- The sequence \\ inserts a single backslash in a string

❖Adding Numbers and Strings

- Java uses the + operator for both addition and concatenation.
- If we add two strings, the result will be a string concatenation

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String x = "10";
    String y = "20";
    String z = x + y;
    System.out.println(z);
  }
}
```

Output

1020

- If we add a number and a string, the result will be a string concatenation

MyClass.java

```
public class MyClass {
  public static void main(String[] args) {
    String x = "10";
    int y = 20;
    String z = x + y;
    System.out.println(z);
  }
}
```

Output

1020

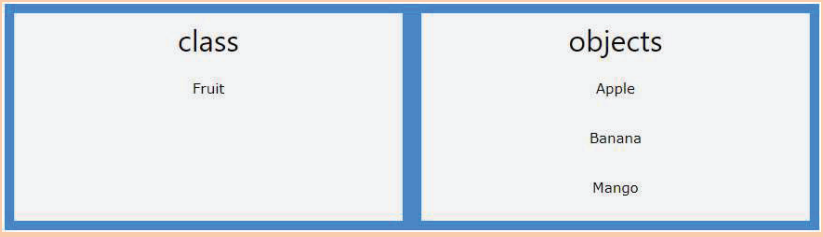
CHAPTER 2

OBJECT ORIENTED PROGRAMMING IN JAVA

Class Fundamentals

❖Classes and Objects

- Classes and objects are the two main aspects of object-oriented programming.



- So, a class is a template for objects, and an object is an instance of a class.
- A Class is a "blueprint" for creating objects.

❖Create a Class

- To create a class, use the keyword class

❖Create an Object

- To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new

```
public class MyClass {
  int x = 5;
}
```

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```


❖Multiple Objects

- You can create multiple objects of one class

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj1 = new MyClass();  // Object 1
    MyClass myObj2 = new MyClass();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

❖ Initialize the object through a reference variable

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output

101 sonoo

❖Using Multiple Classes

- We can also create an object of a class and access it in another class.
- This is often used for better organization of classes
- One class has all the attributes and methods, while the other class holds the main() method (code to be executed).
- Remember that the name of the java file should match the class name.
- In the following example, we have created two files in the same directory/folder:

  MyClass.java

  OtherClass.java

MyClass.java

```
public class MyClass {
  int x = 5;
}
```

OtherClass.java

```
class OtherClass {
  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```


JAVA CLASS ATTRIBUTES

- Class attributes are variables within a class

Example

Create a class called "MyClass" with two attributes x and y

```
public class MyClass {
  int x = 5;
  int y = 3;
}
```

- Another term for class attributes is fields / Data members

❖Accessing Attributes

- We can access attributes by creating an object of the class, and by using the dot syntax (.)

Example

Create an object called "myObj" and print the value of x

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

Output

5

❖Modify Attributes

- We can also modify attribute values

Example

Set the value of x to 40

```
public class MyClass {
  int x;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Output

40

❖Override existing values

Example

Change the value of x to 25

```
public class MyClass {
  int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

Output

25

- If you don't want the ability to override existing values, declare the attribute as **final**

```
public class MyClass {
  final int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

- The final keyword is useful when we want a variable to always store the same value, like PI (3.14159...)
- The final keyword is called a "modifier".

❖Multiple Objects

➢If we create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other

Example - Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj1 = new MyClass();  // Object 1
    MyClass myObj2 = new MyClass();  // Object 2
    myObj2.x = 25;
    System.out.println(myObj1.x);  // Outputs 5
    System.out.println(myObj2.x);  // Outputs 25
  }
}
```


Multiple Attributes

➢We can specify as many attributes as you want

```
public class Person {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Person myObj = new Person();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

JAVA CLASS METHODS

- Methods are declared within a class, and that they are used to perform certain actions

Example - Create a method named myMethod() in MyClass

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

- myMethod() prints a text (the action), when it is called.
- To call a method, write the method's name followed by two parentheses () and a semicolon;

Example

Inside main, call myMethod()

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }

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

// Outputs "Hello World!"
```

❖Static vs. Non-Static Methods

- Java programs have either static or public attributes and methods.
- Static method can be accessed without creating an object of the class
- Public methods can only be accessed by objects

Example

The differences between static and public methods


```
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Access Methods With an Object

```
public class Car {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Car myCar = new Car();      // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}

// The car is going as fast as it can!
// Max speed is: 200
```


Remember that..

- The dot (.) is used to access the object's attributes and methods.
- To call a method in Java, write the method name followed by a set of parentheses (), followed by a semicolon (;)

Using Multiple Classes

- It is a good practice to create an object of a class and access it in another class.
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

  Car.java

  OtherClass.java

Car.java

```
public class Car {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

OtherClass.java

```
class OtherClass {
  public static void main(String[] args) {
    Car myCar = new Car();      // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}
```


CONSTRUCTORS

- A constructor in Java is a **special method** that is used to initialize objects.
- The constructor is called when an object of a class is created.
- It can be used to set initial values for object attributes

Types of Java constructors

Default constructor (no-argument constructor)

Parameterized constructor

Syntax of default constructor:

<class_name>(){}

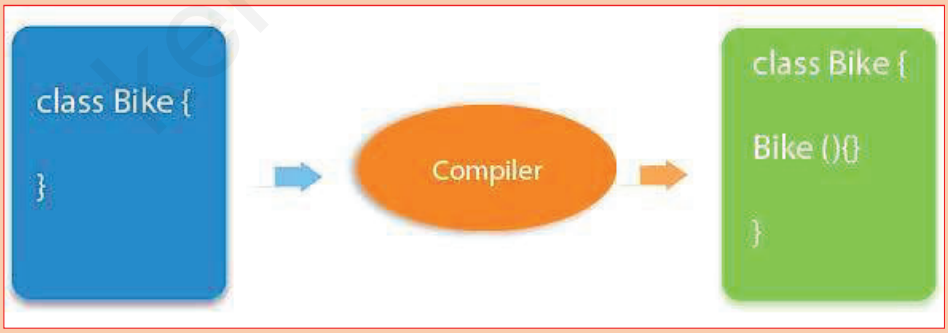
➢In the following example, we are creating the no-argument constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output

Bike is created

➢If there is no constructor in a class, compiler automatically creates a default constructor.



The purpose of a default constructor

- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```java
// Create a MyClass class
public class MyClass {
  int x;  // Create a class attribute

  // Create a class constructor for the MyClass class
  public MyClass() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(); // Create an object of class MyClass (This will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}

// Outputs 5
```


- The constructor name must match the class name, and it cannot have a return type (like void).
- The constructor is called when the object is created.
- All classes have constructors by default
- If you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters (Parameterized constructor)

- Constructors can also take parameters, which is used to initialize attributes

```
public class MyClass {
  int x;

  public MyClass(int y) {
    x = y;
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(5);
    System.out.println(myObj.x);
  }
}

// Outputs 5
```


Constructor Parameters - We can have as many parameters as you want

```
public class Car {
  int modelYear;
  String modelName;

  public Car(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Car myCar = new Car(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 Mustang
```

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Method Overloading

- With **method overloading**, multiple methods can have the same name with different parameters
- Method overloading is one of the ways that Java supports polymorphism.

There are two ways to overload the method in java

By changing number of arguments

By changing the data type

Example

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

❖ Advantage of method overloading

- The main advantage of this is cleanliness of code.
- Method overloading increases the readability of the program.
- Flexibility

Example - Consider the following example, which have two methods that add numbers of different type

```
static int plusMethodInt(int x, int y) {
  return x + y;
}

static double plusMethodDouble(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethodInt(8, 5);
  double myNum2 = plusMethodDouble(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

```
static int plusMethod(int x, int y) {
  return x + y;
}

static double plusMethod(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```


RECURSION

- Recursion is the technique of making a method call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Syntax

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

```
public class MyClass {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Working

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Example

Use recursion to add all of the numbers up to 10.


```
public class RecursionExample3 {
   static int factorial(int n){
       if (n == 1)
         return 1;
       else
         return(n * factorial(n-1));
   }

public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Working

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

Example

Factorial of a number

USING OBJECT AS A PARAMETER / ARGUMENT

```
class Operation2{
 int data=50;

 void change(Operation2 op){
 op.data=op.data+100;//changes will be in the instance variable
 }


 public static void main(String args[]){
  Operation2 op=new Operation2();

  System.out.println("before change "+op.data);
  op.change(op);//passing object
  System.out.println("after change "+op.data);

 }
}
```

```
Output:before change 50
       after change 150
```


THIS KEYWORD

➢There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

Usage of java this keyword

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output

```
0 null 0.0
0 null 0.0
```

Understanding the problem without this keyword


```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output

```
111 ankit 5000
112 sumit 6000
```

Solution of the problem with this keyword

- It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

➢ **this: to invoke current class method**

- You may invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method


```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output

```
hello n
hello m
```

JAVA INNER CLASS

- In Java, it is also possible to nest classes (a class within a class).
- The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- To access the inner class, create an object of the outer class, and then create an object of the inner class


```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

➢Access Outer Class From Inner Class

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

One advantage of inner classes, is that they can access attributes and methods of the outer class

Command-Line Arguments

- Sometimes we want to pass information into a program when we run it. This is accomplished by passing command-line arguments to main( ).
- The main method can receive string arguments from the command line
- To access the command-line arguments inside a Java program is quite easy— they are stored as strings in a String array passed to the args parameter of main( ).
- The first command-line argument is stored at args[0], the second at args[1], and so on.

```java
// Display all command-line arguments.
class CmdLine {
public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
        }
}
```

```
C:\Users\Hello World\Desktop\JAVA>javac CmdLine.java

C:\Users\Hello World\Desktop\JAVA>java CmdLine This is Commend-Line Argument Example
args[0]: This
args[1]: is
args[2]: Commend-Line
args[3]: Argument
args[4]: Example

C:\Users\Hello World\Desktop\JAVA>
```

MODULE - 2

CHAPTER – 3
INHERITANCE

INHERITANCE IN JAVA

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and attributes of the parent class. Moreover, you can add new methods and attributes in your current class also
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

❖Terms used in Inheritance

Class: A class is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the attributes and methods of the existing class when you create a new class. We can use the same attributes and methods already defined in the previous class.

❖Access Modifiers - There are four types of Java access modifiers:

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

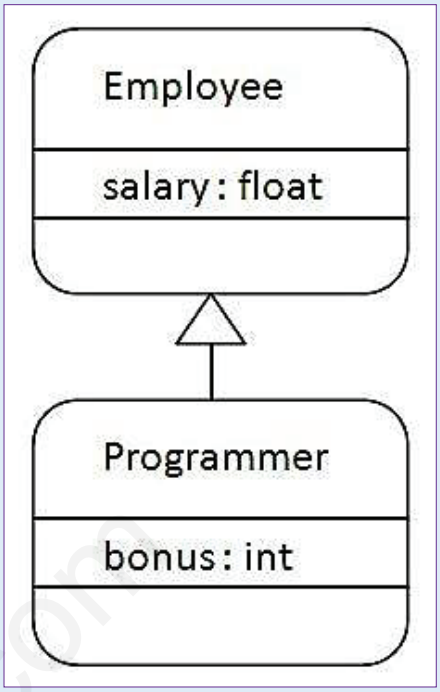
Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

❖The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The extends keyword indicates that you are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

- Programmer is the subclass (child class)
- Employee is the superclass (Parent class)
- The relationship between the two classes is Programmer IS-A Employee
- It means that Programmer is a type of Employee.




```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

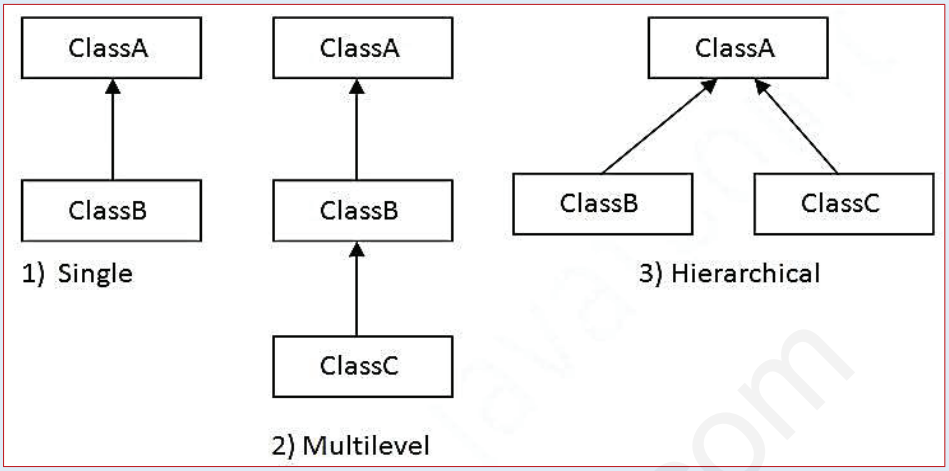
Output

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

- Programmer object can access the attribute of its own class as well as of Employee class i.e. code reusability.

❖Types of inheritance in java

➢On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.




```
class Vehicle {
  protected String brand = "Ford";         // Vehicle attribute
  public void honk() {                     // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

SUPER KEYWORD

➢The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

➢We can use super keyword to access the data member (attribute) of parent class. It is used if parent class and child class have same attribute.


```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

output

```
black
white
```

Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output

```
eating...
barking...
```

The super keyword can also be used to invoke(call) parent class method.

- In the above example Animal and Dog both classes have eat() method
- If we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.
- To call the parent class method, we need to use super keyword.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

The super keyword can also be used to invoke the parent class constructor.

Output

```
animal is created
dog is created
```


Calling order of constructors in inheritance

- Order of execution of constructors in inheritance relationship is from base (parent) class to derived (child)class.
- We know that when we create an object of a class then the constructors get called automatically.
- In inheritance relationship, when we create an object of a child class, then first base class constructor and then derived class constructor get called implicitly.
- In simple word, we can say that the parent class constructor get called first, then of the child class constructor.

```
class A {
  A() {
        System.out.println("Inside A's constructor.");
      }
 }
// Create a subclass by extending class A.
class B extends A {
  B() {
        System.out.println("Inside B's constructor.");
      }
}
// Create another subclass by extending B.
class C extends B {
  C() {
        System.out.println("Inside C's constructor.");
      }
}
public class Main{
    public static void main(String args[])
      {
        C c = new C();
      }
}
```

Output

```
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```


METHOD OVERRIDING

- If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Remember…….

- A static method cannot be overridden. It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.
- Can we override java main method? - No, because the main is a static method.

Example - method overriding

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output

```
Bike is running safely
```

we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

method overloading Vs. method overriding

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

FINAL KEYWORD

➢The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. **variable**
2. **method**
3. **class**

❖Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant)

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

```
Output:Compile Time Error
```

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

❖Java final method

➢If we make any method as final, **we cannot override** it

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Output:Compile Time Error
```

❖Java final class

➢If we make any class as final, we cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```


Is final method inherited?

➢ Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

```
Output:running...
```

❖ **Points to Remember**

1) A constructor cannot be declared as final.

2) Local final variable must be initializing during declaration.

3) We cannot change the value of a final variable.

4) A final method cannot be overridden.

5) A final class not be inherited.

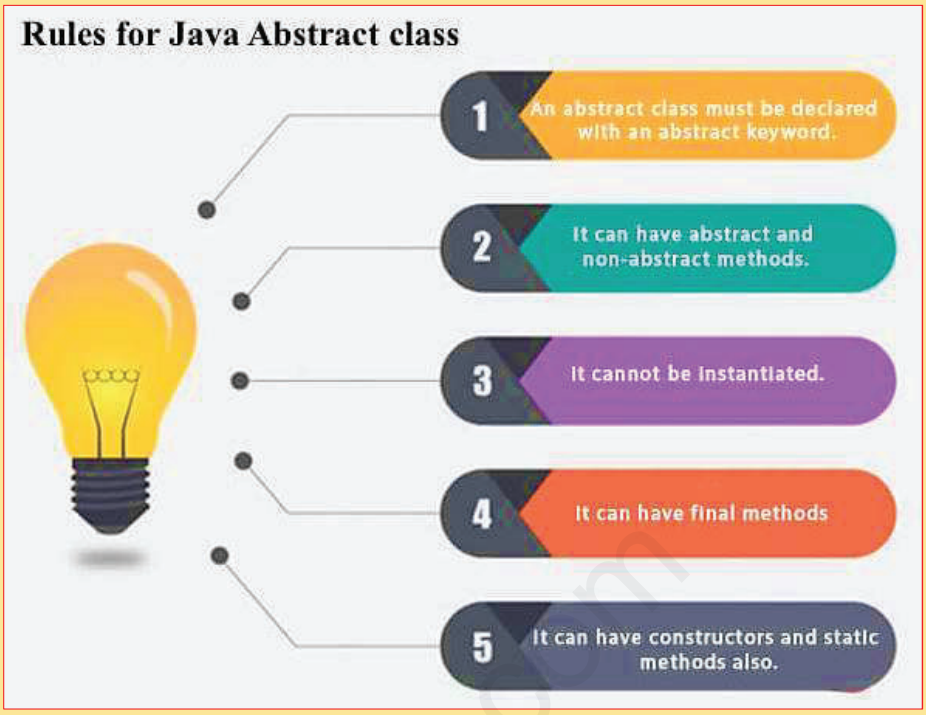
6) If method parameters are declared final then the value of these parameters cannot be changed.

7) final, finally and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during garbage collection.

ABSTRACT CLASSES AND METHODS

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- An abstract class can have both abstract and regular methods:

❖Abstract class




```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

- From the example above, it is not possible to create an object of the Animal class

  Animal myObj = new Animal(); // will generate an error

- To access the abstract class, it must be inherited from another class

Example

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```


Example - Here Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Output

```
running safely
```

THE OBJECT CLASS

- The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.
- Object class is present in java.lang package
- Every class in Java is directly or indirectly derived from the Object class

❖Methods of Object class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).