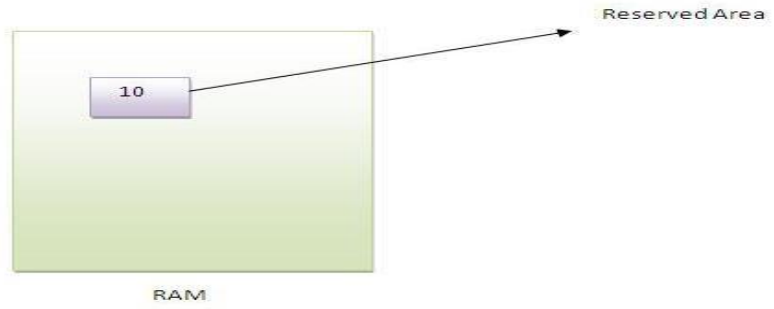
UNIT – II

1. VARIABLES
2. PRIMITIVE DATA TYPES
3. IDENTIFIERS – NAMING CONVENTIONS
4. KEYWORDS
5. LITERALS
6. OPERATORS
7. EXPRESSIONS
8. PRECEDENCE RULES & ASSOCIATIVITY
9. TYPE CONVERSION & TYPE CASTING
10. BRANCHING
11. CONDITIONAL STATEMENTS
12. LOOPS
13. CLASSES
14. OBJECTS
15. METHODS
16. CONSTRUCTIORS
17. CONSTRUCTIOR OVERLOADING
18. GARBAGE COLLECTOR
19. STATIC KEYWORD
20. THIS KEYWORD
21. ARRAYS
22. COMMAND LINE ARGUMENTS

VARIABLES

Variable is name of reserved area allocated in memory.



Ex: int data=50;//Here data is variable

Types of Variables

There are three types of variables in java

- local variable
- instance variable
- static variable

Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable. It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

Example to understand the types of variables:

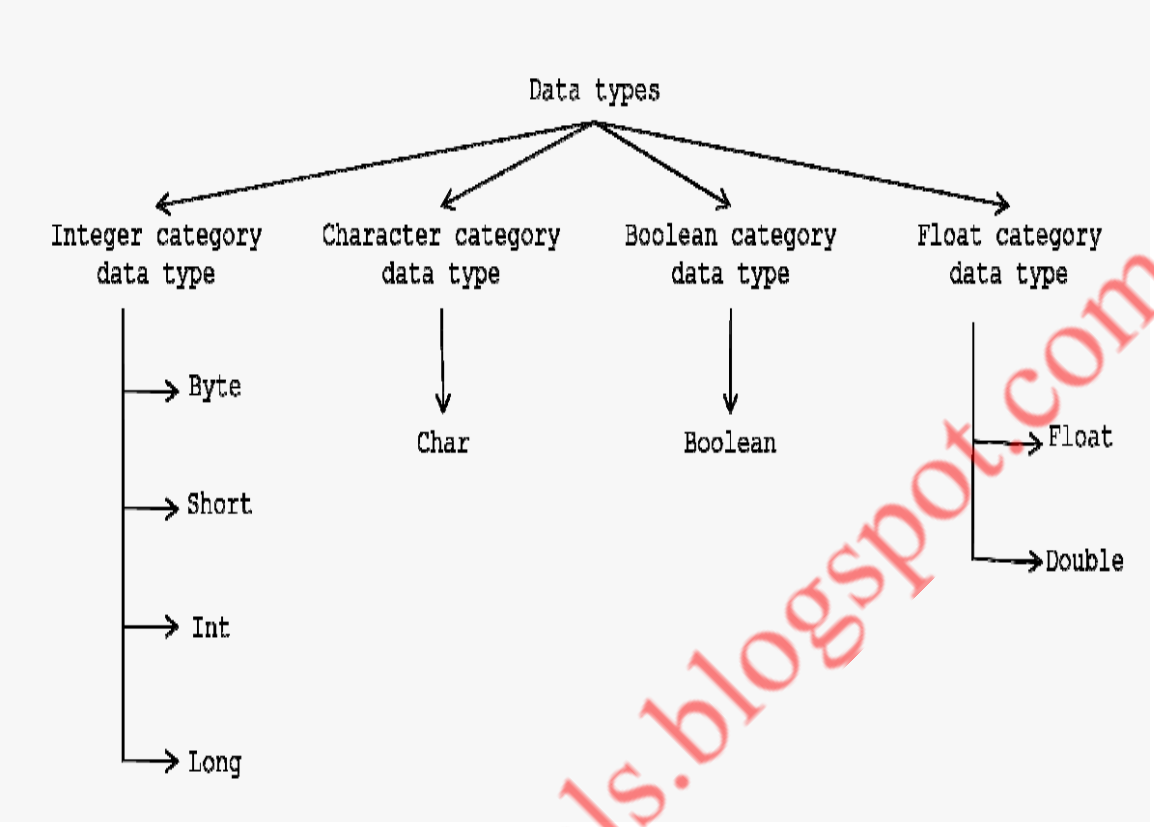
```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

DATA TYPES

"Data types are used for representing the data in main memory of the computer.

In java, there are two types of data types

- Primitive data types
- Non-primitive data types.



In JAVA, we have eight data types which are organized in four groups. They are integer category data types, float category data types, character category data types and Boolean category data types.

1. Integer category data types: These are used to represent integer data. This category of data type contains four data types which are given in the following table:

Type	Contains	Default	Size	Range
byte	Signed integer	0	8 bit or 1 byte	-2^7 to 2^7-1 or -128 to 127
short	Signed integer	0	16 bit or 2 bytes	-2^{15} to $2^{15}-1$ or -32768 to 32767
int	Signed integer	0	32 bit or 4 bytes	-2^{31} to $2^{31}-1$ or -2147483648 to 2147483647
long	Signed integer	0L	64 bit or 8 bytes	-2^{63} to $2^{63}-1$ or -9223372036854775808 to 9223372036854775807

2. Float category data types: Float category data types are used for representing the data in the form of scale, precision i.e., these category data types are used for representing float values. This category contains two data types; they are given in the following table:

Type	Contains	Default	Size	Range	No of decimals
Float	IEEE 754 floating point single-precision	0.0f	32 bit or 4 bytes	1.4E-45 to 3.4028235E+38	8
double	IEEE 754 floating point double-precision	0.0d	64 bit or 8 bytes	439E-324 to 1.7976931348623157E+308	16

3. CHAR:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

IDENTIFIERS – NAMING CONVENTIONS

Identifiers are symbolic names given to classes, methods and variables.
Identifiers should follow certain rules:

1. Identifiers can contain capital letters, small letters and numbers.

   Examples of Valid identifiers:
   Car; AnObject; myPROgrAM123

2. Identifiers can contain only the special characters Underscore (_), Dollar ($).

   Examples of valid identifiers:
   _aName; Student_Class; Student_; _$_$$_; $ant

   Invalid identifiers: Car>Class; %in;

3. Identifiers should not start with a number.

   Ex: 1Plane; 1245
4. Identifiers should not contain spaces.

Ex: Student class; A class

5. Keywords cannot be used as identifiers. Ex: class, for

6. Identifiers are case sensitive.

Ex: Car, CAR, car, cAR, CaR

KEYWORDS

Keywords are special tokens in the language which have reserved use in the language. Keywords may not be used as identifiers in Java — you cannot declare a field whose name is a keyword, for instance.

Examples of keywords are the primitive types, **int** and **boolean**; the control flow statements **for** and **if**; access modifiers such as **public**, and special words which mark the declaration and definition of Java classes, packages, and interfaces: **class**, **package**, **interface**.

Below are all the Java language keywords:

- **abstract**
- **assert** (since Java 1.4)
- **boolean**
- **break**
- **byte**
- **case**
- **catch**
- **char**
- **class**
- **const** (not used)
- **continue**
- **default**
- **do** ; **double**
- **else**
- **enum** (since Java 5.0)
- **extends**
- **final**
- **finally**
- **float**
- **for**
- **goto** (not used)
- **if**
- **implements**
- **import**
- **instanceof**
- **int**
- **interface**
- **long**
- **native**
- **new**
- **package**
- **private**
- **protected**
- **public**
- **return**
- **short**
- **static**
- **strictfp** (since Java 1.2)
- **super**

- **switch**
- **synchronized**
- **this**
- **throw**
- **throws**
- **transient**
- **try** ;
- **void**;
- **volatile**;
- **while**.

LITERALS

A literal is the source code representation of a fixed value.Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables.

Java language specifies five major types of literals. Literals can be any number, text, or other information that represents a value.

This means what you type is what you get. We will use literals In addition to variables in Java statement.

While writing a source code as a character sequence, we can specify any value as a literal such as an integer.

They are:

- **Integer literals**
- **Floating literals**
- **Character literals**
- **String literals**
- **Boolean literals**

Each of them has a type associated with it. The type describes how the values behave and how they are stored.

Integer literals:

Integer data types consist of the following primitive data types: int, ong, byte, and short. Byte, int, long, and short can be expressed in decimal (base 10), hexadecimal (base 16) or octal (base 8) number systems as well. Prefix 0 is used to indicate octal and prefix **0x** indicates hexadecimal when using these number systems for literals.

Examples:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

Floating-point literals:

Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001.

Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as: $0.0314 * 10^2$ (i.e 3.14). 6.5E+32 (or 6.5E32) Double-precision floating-point literal 7D Double-precision floating-point literal .01f Floating-point literal

Character literals:

char data type is a single 16-bit Unicode character. We can specify a character literal as asingle printable character in a pair of single quote characters such as 'a', '#', and '3'. You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. Below table shows a set of these special characters.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

If we want to specify a single quote, a backslash, or a non-Printable character as a character literal use an escape sequence. An escape sequence uses a special syntax to represents a character.

The syntax begins with a single backslash character. You can see the below table to view the character literals use Unicode escape sequence to represent printable and non-printable characters.

'u0041'	Capital letter A
'\u0030'	Digit 0
'\u0022'	Double quote "
'\u003b'	Punctuation ;
'\u0020'	Space
'\u0009'	Horizontal Tab

String Literals:

The set of characters in represented as String literals in Java. Always use "double quotes" for String literals. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same values.

""	The empty string
"\""	A string containing
"This is a string"	A string containing 16 characters
"This is a " + "two-line string"	actually a string-valued constant expression, formed from two string literals

Null Literals

The final literal that we can use in Java programming is a Null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal.
The type of the null literal is always null. We typically assign null literals to object reference variables. For instance
s = null;

Boolean Literals:

The values true and false are treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C,
We can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value.
Example
boolean chosen = true;

JAVA OPERATORS:

- In Java Variables are Value Container used to store some value.
- In order to Perform some operation on the Data we use different operators such as arithmetic Operator,Logical Operator etc.

Types of Operators:

1. Assignment Operator
2. Arithmetic Operator
3. Unary Operators

4. Relational Operator
5. Conditional Operators
6. Bitwise and Bit Shift Operators

Operators and its Precedence Table in Java:

Operators	**Precedence**
postfix	***expr++ expr--***
unary	***++expr --expr +expr -expr*** **~ !**
multiplicative	*** / %**
additive	**+ -**
shift	**<< >> >>>**
relational	**< > <= >= instanceof**
equality	**== !=**
bitwise AND	**&**
bitwise exclusive OR	**^**
bitwise inclusive OR	**|**

logical AND	**&&**
logical OR	**||**
ternary	**? :**
assignment	**= += -= *= /= %= &= ^= |= <<= >>= >>>=**

1.Java Assignment Operator:

- "=" is simple assignment operator in Java Programming.
- "Assignment Operator" is binary Operator as it operates on two operands.
- It have two values associated with it i.e Left Value and Right Value.
- It is used to Assign Literal Value to the Variable.
- Variable of Primitive Data Type
- Array Variable
- Object

Syntax and Example:

```
int javaProgramming;  //Default Assignment
0int distance = 10;    //Assign 10
int price = 1990;     //Assign 1990
```

Ways of Assignment:

Way 1 : Declare and Assign

```
class AssignmentDemo {

  public static void main (String[] args){
    int ivar;
    ivar = 10;

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

- Firstly we are going to declare variable of primitive data type.
- After declaring variable will have default value inside it (i.e 0).
- 10 is assigned to Variable "ivar".

Way 2 : Declare and Assign in Single Statement

```
class AssignmentDemo {

  public static void main (String[] args){

    int ivar = 10;
    System.out.println(ivar);
  }
}
```

Way 3 : Assignment & Arithmetic Operation

```
class AssignmentAndArithmetic {

  public static void main (String[] args){

    int num1 = 10;
    int num2 = 20;
    int num3 = num1 + num2;

    System.out.println(num1);
  }
}
```

2.Arithmetic Operators in Java Programming:

- The basic arithmetic operations in Java Programming are addition, subtraction, multiplication, and division.
- Arithmetic Operations are operated on Numeric Data Types as expected.
- Arithmetic Operators can be Overloaded.
- Arithmetic Operators are "Binary" Operators i.e they operates on two operands.

Arithmetic Operators used in Java :

Operator	Use of Operator
+	Use to Add Two Numbers and Also used to Concatenate two strings
-	Used for Subtraction
*	Used to multiply numbers
/	Used for Division
%	Used for Finding Mod (Remainder Operator)

Example 1 : Arithmetic Operators

```
class ArithmeticOperationsDemo {

  public static void main (String[] args){

    // answer is now 3
    int answer = 1 + 2;
    System.out.println(answer);

    // answer is now 2
    answer = answer - 1;
    System.out.println(answer);

    // answer is now 4
    answer = answer * 2;
    System.out.println(answer);

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

    // answer is now 10
    answer = answer + 8;

    // answer is now 3
    answer = answer % 7;
        System.out.println(answer);
  }
}
```

Example 2 : Use of Modulus Operator

```
class ModulusOperatorDemo {
 public static void main(String args[]) {
  int   x = 92;
  double y = 92.25;

  System.out.println("x mod 10 = " + x % 10);
  System.out.println("y mod 10 = " + y % 10);
 }
}
```

Output :

```
x mod 10 = 2
y mod 10 = 2.25
```

Example : Arithmetic Compound Assignment Operators in Java

```
class CompoundAssignmentDemo {
 public static void main(String args[]) {
  int a = 1;
  int b = 2;
  int c = 3;

  a += 1;
  b *= 1;
  c %= 1;

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

Output :

```
a = 2
b = 2
c = 0
```

3.Bitwise Operators in Java Programming :

- Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

- Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

  a = 0011 1100

  b = 0000 1101

  ----

  a&b = 0000 1100

  a|b = 0011 1101

  a^b = 0011 0001

  ~a = 1100 0011

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

&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Operators : Bitwise unary NOT

- Bitwise Unary Not is also called as bitwise complement.
- It is Unary Operator because it operates on single Operand.
- Unary Not Operator is denoted by – '~'.
- The Unary NOT operator inverts all of the bits of its operand.

Example 1 : Bitwise unary NOT : Bitwise Operators

```
class BitwiseNOT{
 public static void main(String args[]){
 int ivar =42;
 System.out.println("~ ivar = " + ~ivar);
 }
}
Output :
~ ivar = -43
```

Example 2 : Unary Not Operator 0-5

```
class BitwiseNOT{
  public static void main(String args[]){
  System.out.println("~ 0 = " + ~0);
  System.out.println("~ 1 = " + ~1);
  System.out.println("~ 2 = " + ~2);
  System.out.println("~ 3 = " + ~3);
     }
}
Output :
~ 0 = -1
~ 1 = -2
~ 2 = -3
~ 3 = -4
```

Bitwise AND Operator is –

- Binary Operator as it Operates on 2 Operands.
- Denoted by – &
- Used for : Masking Bits.

Bitwise AND Summary Table :

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Example 1 : ANDing 42 and 15

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

  int num1 = 42;
  int num2 = 15;

  System.out.println("AND Result =" +(num1&num2));

  }
}
```

Output :
AND Result = 10
Explanation of Code :

Num1 : 00101010 42
Num2 : 00001111 15
======================
AND : 00001010 10
42 is represented in Binary format as -> 00101010
15 is represented in Binary format as -> 00001111
According to above rule (table) we get 00001010 as final structure.
println method will print decimal equivalent of 00001010 and display it on screen.

Bitwise OR Operator is –

- The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1
- Binary Operator as it Operates on 2 Operands.
- Denoted by – |

Bitwise OR Summary Table :

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Example 1 : ORing 42 and 15

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

 int num1 = 42;
 int num2 = 15;

 System.out.println("OR Result =" +(num1 | num2));

 }
}
Output :
OR Result = 47
```

Explanation of Code :

Num1 : 00101010 42
Num2 : 00001111 15
======================
OR : 00101111 47
42 is represented in Binary format as -> 00101010

15 is represented in Binary format as -> 00001111
According to above rule (table) we get 00101111 as final structure.
println method will print decimal equivalent of 00101111 and display it on screen.

Bitwise XOR Operator is –

- The XOR operator (^) combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1
- Binary Operator as it Operates on 2 Operands.
- Denoted by : ^

Bitwise XOR Summary Table :

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Example 1 : XORing 42 and 15

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

  int num1 = 42;
  int num2 = 15;

  System.out.println("XOR Result =" +(num1 ^ num2));

  }
}
```

Output :
XOR Result = 37

Explanation of Code :
Num1 : 00101010 42
Num2 : 00001111 15
==========
XOR : 00100101 37
42 is represented in Binary format as -> 00101010

15 is represented in Binary format as -> 00001111
According to above rule (table) we get 00100101 as final structure.
println method will print decimal equivalent of 00100101 and display it on screen.

Bit Shift operator in Java

- The bitwise shift operators take two operands.
- The first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted.
- The direction of the shift operation is controlled by the operator used. Shift operators convert their operands to 32 or 64 bits and return a result of the same type as the left operator.

Shift Operators			
Operator	**Name**	**Use**	**Description**
<<	Shift left	**op1 << op2**	Shifts bits of op1 left by distance op2; fills with 0 bits on the right side
>>	Shift right	**op1 >> op2**	Shifts bits of op1 right by distance op2; fills with highest (sign) bit on the left side
>>>	Shift right zero fill	**op1 >>> op 2**	Shifts bits of op1 right by distance op2; fills with 0 bits on the left side

Shifting is basically taking the binary equivalent of a number and moving the bit pattern left or right.

Example:

```
class Bit{
public static void main(String []args)
{
int a =2;
System.out.println(" rightshift by 1 is:" +(a>>1));
System.out.println(" leftshift by 1 is:" +(a<<1));
System.out.println(" Shift right zero fill by 1 is:" +(a>>1));
}
}
```

Out Put:

```
C:\>javac Bit.java

C:\>java Bit
 rightshift by 1 is:1
```

leftshift by 1 is:4
Shift right zero fill by 1 is:1

Example: Bitwise AND assignment, Bitwise OR assignment, Bitwise exclusive OR assignment, Shift right assignment, Shift right zero fill assignment and Shift left assignment

```
public class Bitwise_assign {
 public static void main(String args[]) {
  int a = 1;
  int b = 2;
  int c = 3;

  a |= 4;
  b >>= 1;
  c <<= 1;
  a ^= c;
  System.out.println("a = " + a);
  System.out.println("b = " + b);
  System.out.println("c = " + c);
 }
}
```

OutPut:

```
C:\>javac Bitwise_assign.java

C:\>java Bitwise_assign
a = 3
b = 1
c = 6
```

4.Unary Operators in Java Programming :

- The unary operators require only one operand.
- There are 5 unary operators in Java Programming Language.
- Unary Operators are :
  - Unary Plus
  - Unary Minus
  - Logical Compliment Operator
  - Increment Operator
  - Decrement Operator

Example: Unary Operator

```
class UnaryDemo {

   public static void main(String[] args){
      // result is now 1
      int result = +1;
      System.out.println(result);

      // result is now -1
      result = -result;
      System.out.println(result);

      // false
      boolean success = false;
      System.out.println(success);

      // true
      System.out.println(!success);
   }
}
```

Out Put:

```
C:\>javac UnaryDemo.java
C:\>java UnaryDemo
1
-1
false
true
```

Increment and Decrement Operator in Java :

- It is one of the variation of "Arithmetic Operator".
- Increment and Decrement Operators are Unary Operators.
- Unary Operator Operates on One Operand.
- Increment Operator is Used to Increment Value Stored inside Variable on which it is operating.
- Decrement Operator is used to decrement value of Variable by 1 (default).

Types of Increment and Decrement Operator :

- Pre Increment / Pre Decrement Operator
- Post Increment / Post Decrement Operator

Syntax :

++ Increment operator : increments a value by 1 -- Decrement operator : decrements a value by 1

Example:

```
class Inc_Dec {
  public static void main(String args[]) {
    int num1 = 1;
    int num2 = 1;
    num1++;
    num2++;
    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
    ++num1;
    ++num2;
    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
    num1--;
    num2--;
    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
    --num1;
    --num2;
    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
     }
}
```

OutPut:

```
C:\>javac Inc_Dec.java

C:\>java Inc_Dec
num1 = 2
num2 = 2
num1 = 3
num2 = 3
num1 = 2
num2 = 2
```

num1 = 1
num2 = 1

5.Logical Operators:

Every programming language has its own logical operators, or at least a way of expressing logic. Java's logical operators are split into two subtypes, relational and conditional.

Relational Operators in Java Programming:

- Relational Operators are used to check relation between two variables or numbers.
- Relational Operators are Binary Operators.
- Relational Operators returns "Boolean" value .i.e it will return true or false.
- Most of the relational operators are used in "If statement" and inside Looping statement in order to check truthness or falseness of condition.

Operator	Use	Description
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Logical Operators:

There are four logical operators in Java, however one of them is less commonly used and I will not discuss here. It's really optional compared to the ones I will introduce, and these will be a whole heck of a lot useful to you now.

Here's your table of logical operators:

Conditional symbols and their meanings	
Symbol	**Condition**
&&	**AND**
||	**OR**
!	**NOT**

Example:

```
public class MainClass {
 public static void main(String[] args) {
     int i = 5;
  int j = 6;
  System.out.println("i = " + i);
  System.out.println("j = " + j);
  System.out.println("i > j is " + (i > j));
  System.out.println("i < j is " + (i < j));
  System.out.println("i >= j is " + (i >= j));
  System.out.println("i <= j is " + (i <= j));
  System.out.println("i == j is " + (i == j));
  System.out.println("i != j is " + (i != j));

  System.out.println("(i < 10) && (j < 10) is " + ((i < 10) && (j <
10)));
  System.out.println("(i < 10) || (j < 10) is " + ((i < 10) || (j <
10)));
 }
}
```

Out Put:

```
C:\>javac MainClass.java
C:\>java MainClass
i = 5
j = 6
i > j is false
i < j is true
i >= j is false
i <= j is true
i == j is false
i != j is true
(i < 10) && (j < 10) is true
(i < 10) || (j < 10) is true
```

6.Ternary operator:

Ternary operator is also known as the Conditional operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Example:

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

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

OutPut:

```
C:\>javac Ternary.java
C:\>java Ternary
Value of b is : 30
Value of b is : 20
```

Expressions

- An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.
- Examples of expressions are in bold below:
  - int number = 0;
  - int anArray[0] = 100;
  - System.out.println ("Element 1 at index 0: " + anArray[0]);
  - int result = 1 + 2; // result is now 3 if(value1 == value2)
  - System.out.println ("value1 == value2");

- The data type of the value returned by an expression depends on the elements used in the expression.
- The expression number = 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, number is an int.
- As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.

- The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Here's an example of a compound expression: 1 * 2 * 3.

Precedence and Associativity Rules for Operators:

Precedence and associativity rules are necessary for deterministic evaluation of expressions.

- The operators are shown with decreasing precedence from the top of the table.
- Operators within the same row have the same precedence.
- Parentheses, ( ), can be used to override precedence and associativity.
- The unary operators, which require one operand, include the postfix increment (++) and decrement (--) operators from the first row, all the prefix operators (+, -, ++, --, ~, !) in the second row, and the prefix operators (object creation operator new, cast operator (type)) in the third row.
- The conditional operator (? :) is ternary, that is, requires three operands.
- All operators not listed above as unary or ternary, are binary, that is, require two operands.
- All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
- Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.
- Precedence rules are used to determine which operator should be applied first if there are two operators with different precedence, and these follow each other in the expression. In such a case, the operator with the highest precedence is applied first.
- 2 + 3 * 4 is evaluated as 2 + (3 * 4) (with the result 14) since * has higher precedence than +.
- Associativity rules are used to determine which operator should be applied first if there are two operators with the same precedence, and these follow each other in the expression.
- Left associativity implies grouping from left to right:
- 1 + 2 - 3 is interpreted as ((1 + 2) - 3), since the binary operators + and - both have same precedence and left associativity.

- Right associativity implies grouping from right to left:
- - - 4 is interpreted as (- (- 4)) (with the result 4), since the unary operator - has right associativity.
- The precedence and associativity rules together determine the evaluation order of the operators.

Evaluation Order of Operands:

In order to understand the result returned by an operator, it is important to understand theevaluation order of its operands. Java states that the operands of operators are evaluated from left to right.

Java guarantees that all operands of an operator are fully evaluated before the operator is applied. The only exceptions are the short-circuit conditional operators &&, ||, and ?:.

In the case of a binary operator, if the left-hand operand causes an exception the right-hand operand is not evaluated. The evaluation of the left-hand operand can have side effects that can influence the value of the right-hand operand. For example, in the following code:

```
int b = 10;
System.out.println((b=3) + b);
```

The value printed will be 6 and not 13. The evaluation proceeds as follows:

```
(b=3) + b
 3 + b b is assigned the value 3
 3 + 3
 6
```

The evaluation order also respects any parentheses, and the precedence and associativity rules of operators.

Examples illustrating how the operand evaluation order influences the result returned by an operator.

Operator	Description	Level	Associativity
[] . () ++ --	access array element access object member invoke a method post-increment post-decrement	1	left to right

++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2	right to left
() new	cast object creation	3	right to left
* / %	Multiplicative	4	left to right
+ - +	additive string concatenation	5	left to right
<< >> >>>	Shift	6	left to right
< <= > >= instanceof	relational type comparison	7	left to right
== !=	Equality	8	left to right
&	bitwise AND	9	left to right
^	bitwise XOR	10	left to right
|	bitwise OR	11	left to right
&&	conditional AND	12	left to right
||	conditional OR	13	left to right

?:	Conditional	14	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	Assignment	15	right to left

Java Data Type Casting Type Conversion

Summary: By the end of this tutorial "Java Data Type Casting Type Conversion", you will be comfortable with converting one data type to another either implicitly or explicitly.

Java supports two types of castings – **primitive data type casting** and **reference type casting**. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

Java data type casting comes with 3 flavors.

1. **Implicit casting**
2. **Explicit casting**
3. **Boolean casting.**

1. Implicit casting (widening conversion)

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as **automatic type conversion**.

Examples:

```
        int x = 10;               // occupies 4 bytes
    double y = x;              // occupies 8 bytes
    System.out.println(y);      // prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

Example:

```
public class Test
{
  public static void main(String[] args)
  {
    int i = 100;
    long l = i;    //no explicit type casting required
    float f = l;    //no explicit type casting required
    System.out.println("Int value "+i);
    System.out.println("Long value "+l);
    System.out.println("Float value "+f);
  }
}
```

Out Put:

```
Int value 100

Long value 100

Float value 100.0
```

2. Explicit casting (narrowing conversion)

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;          // 8 bytes
    int y = x;            // 4 bytes ;  raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x = 10.5;
    int y = (int) x;
```

The double **x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.

```
public class Test
{
  public static void main(String[] args)
  {
   double d = 100.04;
   long l = (long)d;  //explicit type casting required
   int i = (int)l;              //explicit type casting required

   System.out.println("Double value "+d);
   System.out.println("Long value "+l);
   System.out.println("Int value "+i);
     }
  }
```

Out Put:

```
Double value 100.04

Long value 100

Int value 100
```

3. Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is **incompatible** for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

```
boolean x = true;
    int y = x;              // error
boolean x = true;
    int y = (int) x;        // error
```

byte –> short –> int –> long –> float –> double

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte **can be assigned to** short **implicitly but** short **to** byte **requires explicit casting.**

Following char operations are possible

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

```
    char ch1 = 'A';
    double d1 = ch1;

    System.out.println(d1);          // prints 65.0
    System.out.println(ch1 * ch1);     // prints 4225 , 65 * 65

    double d2 = 66.0;
    char ch2 = (char) d2;
    System.out.println(ch2);          // prints B
  }
}
```

Pass your comments for the betterment of this tutorial "Java Data Type Casting Type Conversion".

Your one stop destination for all data type conversions

byte TO	short	int	long	float	double	char	boolean
short TO	byte	int	long	float	double	char	boolean
int TO	byte	short	long	float	double	char	boolean
float TO	byte	short	int	long	double	char	boolean
double TO	byte	short	int	long	float	char	boolean
char TO	byte	short	int	long	float	double	boolean
boolean TO	byte	short	int	long	float	double	char

String and data type conversions

String TO	byte	short	int	long	float	double	char	boolean
byte	short	int	long	float	double	char	boolean	**TO String**

Java Control Flow Statements

- All the programs we have written till now had a sequential flow of control i.e. the statements were executed line by line from the top to bottom in an order.
- Nowhere were any statements skipped or a statement executed more than once. We will now look into how this can be achieved using control structures.
- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:

    - ***selection*** statements allow the program to choose different parts of the execution based on the outcome of an expression

- ***iteration*** statements enable program execution to repeat one or more statements
- ***jump*** statements enable your program to execute in a non-linear fashion

Selection Statements

- Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.
- Java provides four selection statements:

  - 1) if
  - 2) if-else
  - 3) if-else-if
  - 4) switch

if

Statement(s) between the set of curly braces '{ }' will be executed only if the condition(s), between the set of brackets '( )' after 'if' keyword, is/are true.

Syntax:

```
if (Condition) {
   // statements;
}
```

if-else

If the condition(s) between the brackets '( )' after the 'if' keyword is/are true then the statement(s) between the immediately following set of curly braces '{ }' will be executed else the statement(s) under, the set of curly braces after the 'else' keyword will be executed.

Example:

Syntax:

```
if (condition) {
// statements;
}
 else {
   // statements;}
```

```
class Example_if_else
{
   public static void main(String Args[])
  {

        if( a > b)
```

```
        {
                System.out.println("A = " + a + "\tB = " + b);
                System.out.println("A is greater than B");
        }
        else
        {
                System.out.println("A = " + a + "\tB = " + b);
                System.out.println("Either both are equal or B is greater");
        }
    }
}

class Example_nested_if _else
{
  public static void main(String Args[])
  {
        int a = 3;
         if (a <= 10 && a > 0)
         {
                System.out.println("Number is valid.");
                if ( a < 5)
                        System.out.println("From 1 to 5");
                else
                        System.out.println("From 5 to 10");
         }
        else
            System.out.println("Number is not valid");
    }
}

class  Example_if_elseif_else
{
   public static void main (String Args[])
  {
        int a = 5;
        boolean val = false;
         if(val )
                System.out.println("val is false, so it won't execute";
        else if (a < 0 )
                System.out.println("A is a negative value");
        else if (a > 0)
                System.out.println ("A is a positive value");
        else
                System.out.println ("A is equal to zero");
    }
}
```

switch

When there is a long list of cases & conditions, then if/if-else is not good choice as the code would become complicated.

Syntax:

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

The moment user enters his choice, it will be matched with the cases' names & program execution will jump to the matching 'Case' & the statement under that case will be executed till the keyword 'break' comes. It is very important else the other unwanted cases will also get executed. After the last case there is 'default' keyword. Statements between 'default:' and the closing bracket of switch-case region will be executed only if the user has entered any wrong value as his choice i.e. other than the cases' names.

Example program :

```
class Example_switch
{
    public static void main(String Args[])
   {
       int month = 3;
      switch (month)
       {
       case 1:
                System.out.println("The month of January");
                break;
       case 2:
                System.out.println("The month of February");
                break;
       case 3:
                System.out.println("The month of March");
                break;
```

```
            case 4:
                    System.out.println("The month of April");
                    break;
            case 5:
                    System.out.println("The month of May");
                    break;
            case 6:
                    System.out.println("The month of June");
                    break;
            case 7:
                    System.out.println("The month of July");
                    break;
            case 8:
                    System.out.println("The month of August");
                    break;
            case 9:
                    System.out.println("The month of September");
                     break;
            case 10:
                    System.out.println("The month of October");
                    break;
            case 11:
                    System.out.println("The month of November");
                    break;
            case 12:
                    System.out.println("The month of December");
                    break;
            default:
                    System.out.println("Invalid month");
            }
    }
}
```

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

- Java provides three iteration statements:

1) while
2) do-while
3) for

while

while statement continually executes a block of statements while a particular condition is true. Entry controlled

Syntax:

```
while(conditions)
{
//Loop body
}
```

```
public class WhileExample
{
  public static void main (String[ ] args)
  {
        int i =0;
         while (i < 4)
         {
               System.out.println ("i is : " + i);

               i++;
         }
  }
}
```

do-while

It will enter the loop without checking the condition first and checks the condition after the execution of the statements. That is it will execute the statement once and then it will evaluate the result according to the condition. Exit controlled

Syntax:

```
do
{
//Loop body
}while(condition);
```

```
public class DoWhileExample
{
   public static void main (String[ ] args)
   {
         int i =0;
         do {
               System.out.println ("i is : " + i);
               i++;
            } while (i < 4);
   }
}
```

6. for

The concept of Iteration has made our life much easier. Repetition of similar tasks is what Iteration is and that too without making any errors. Until now we have learnt how to use selection statements to perform repetition.

Syntax:

```
    for(initialization;test condition;increment)
    {
    //Loop body
    }
public class ForExample
{
  public static void main (String[ ] args)
 {
     for(  int i =0;i < 4;i++)
       {
            System.out.println ("i is : " + i);
       }
 }
}
```

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
  - break
  - continue
  - return
- In addition, Java supports exception handling that can also alter the control flow of a program.
- Java programs.

Continue Statement in Java:

Continue Statement in Java is used to skip the part of loop. Unlike break statement it does not terminate the loop , instead it skips the remaining part of the loop and control again goes to check the condition again.

Syntax:

```
{
//loop body
----------
----------
----------
continue;
```

```
----------
----------
}
```

- Continue Statement is **Jumping Statement in Java Programming** like break.
- Continue Statement skips the Loop and **Re-Executes Loop with new condition**.
- Continue Statement **can be used only in Loop Control Statements** such as For Loop | While Loop | do-While Loop.
- **Continue** is Keyword in Java Programming.

Example:

```
public class continue_demo {
    public static void main(String[] args) {
        int j;
        for (j = 1; j < 5; j++) {
            if (j == 3) {
                System.out.println("continue!");
             continue;
            }
            System.out.println(j);
        }
    }
}
```

Out Put:

```
C:\>java continue_demo
1
2
continue!
4
```

return statement in java:

- Return statement in used to explicitly return from a method. Return causes program control to transfer back to the caller of the method.
- The return statement immediately terminates the method in which it is executed.
- We can specify return type of the method as **"Primitive Data Type" or "Class name"**.
- Return Type can be "Void" means **it does not return any value**.

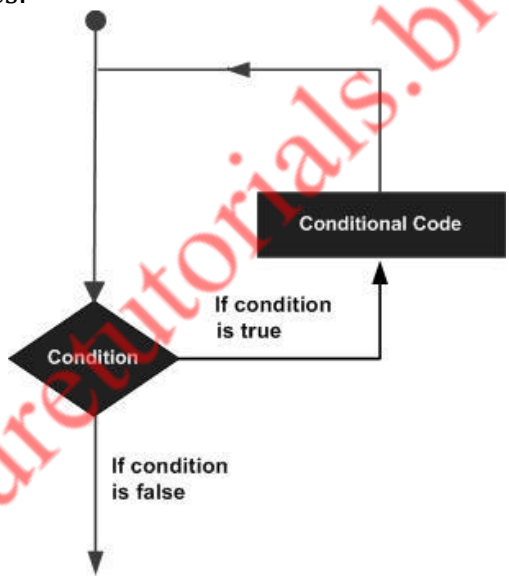
- Method can return a value by using "**return**" keyword.

Syntax:-

return;

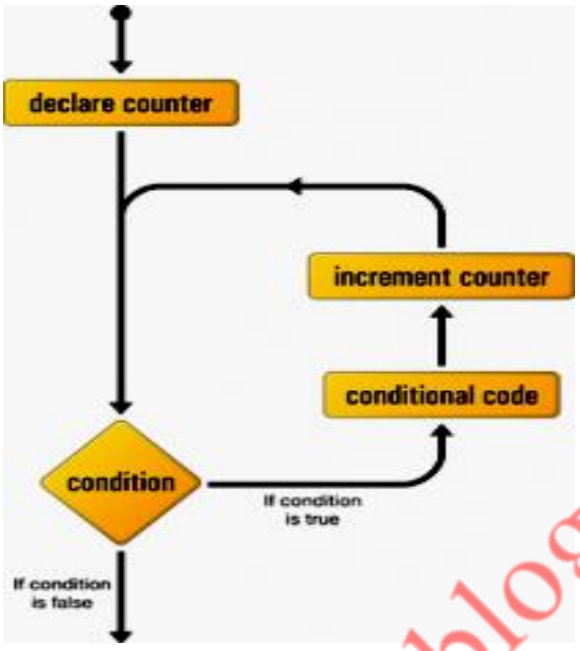
ITERATIVE STATEMENT (LOOP CONTROL STATEMENT)

- There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



For Loop in Java Programming :

- For Loop is one of the looping statement in java programming.
- For Loop is used to execute set of statements repeatedly until the condition is true.
- For Loop checks the contrition and executes the set of the statements , It is loop control statement in java.
- For Loop contain the following statements such as "**Initialization**" , "**Condition**" and "**Increment/Decrement**" statement.



Example : For Loop Statement

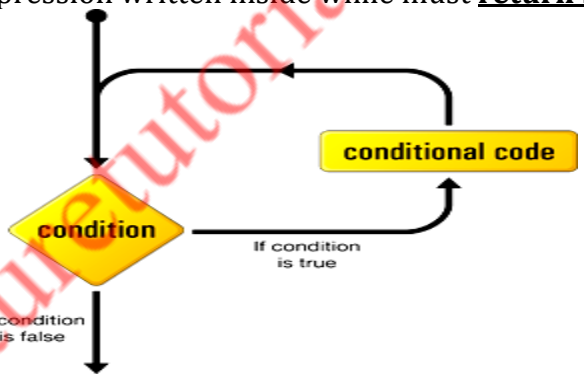
```
class ForDemo {
  public static void main(String[] args){
    for(int i=1; i<11; i++){
      System.out.println("Count is : " + i);
    }
  }
}
```

Output :

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

While Loop statement in Java :

- In java "**while**" is iteration statements like for and do-while.
- It is also called as "**Loop Control Statement**".
- "**While Statement**" repeatedly executes the same set of instructions until a termination condition is met.
- While loop is **Entry Controlled Loop** because condition is check at the entrance.
- If initial condition is true then and then only control enters into the while loop body
- In for loop initialization,condition and increment all three statements are combined into the one statement , in "**while loop**" all these statements are written **as separate statements**.
- Conditional Expression written inside while must **return boolean value**.



Syntax :

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

- The condition is any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.

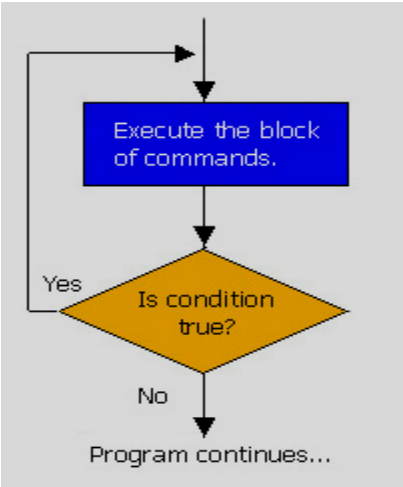
- When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated.

Example :

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

Java do-while loop:

- In java "**Do-while**" is iteration statements like for loop and while loop.
- It is also called as "**Loop Control Statement**".
- "**Do-while Statement**" is **Exit Controlled Loop** because condition is check at the last moment.
- Irrespective of the condition , control enters into the do-while loop , after completion of body execution , condition is checked whether true/false. If condition is false then it will jump out of the loop.
- Conditional Expression written inside while must **return boolean value**.
- In do-while loop body gets executed once whatever may be the condition but condition must be true if you need to execute body for second time.



Syntax : Do-While Loop

```
do {
   Statement1;
   Statement2;
   Statement3;
   ..
   .
   StatementN;
} while (expression);
```

Example 1 : Printing Numbers

```
class DoWhile {
  public static void main(String args[]) {
   int n = 5;
   do
   {
   System.out.println("Sample : " + n);
   n--;
   }while(n > 0);
}
```

Lab Programs:

1. Write a JAVA program to display default value of all primitive data types of JAVA

```
class  DefaultValues
{
   static byte b;
   static short s;
   static int i;
   static long l;
   static float f;
   static double d;
   static char c;
   static boolean bl;
   public static void main(String[] args)
   {
      System.out.println("Byte :"+b);
      System.out.println("Short :"+s);
      System.out.println("Int :"+i);
      System.out.println("Long :"+l);
      System.out.println("Float :"+f);
      System.out.println("Double :"+d);
      System.out.println("Char :"+c);
      System.out.println("Boolean :"+bl);
   }
}
```

Out Put:

```
E:\JAVA>javac DefaultValues.java
E:\JAVA>java DefaultValues
Byte :0
Short :0
Int :0
Long :0
Float :0.0
```

Double :0.0

Char :

Boolean :false

2. Write a JAVA program to get the minimum and maximum value of a primitive data types

```
public class MinMaxExample
{
 public static void main(String[] args)
 {
  System.out.println("Byte.MIN = " + Byte.MIN_VALUE);
  System.out.println("Byte.MAX = " + Byte.MAX_VALUE);
  System.out.println("Short.MIN = " + Short.MIN_VALUE);
  System.out.println("Short.MAX = " + Short.MAX_VALUE);
  System.out.println("Integer.MIN = " + Integer.MIN_VALUE);
  System.out.println("Integer.MAX = " + Integer.MAX_VALUE);
  System.out.println("Long.MIN = " + Long.MIN_VALUE);
  System.out.println("Long.MAX = " + Long.MAX_VALUE);
  System.out.println("Float.MIN = " + Float.MIN_VALUE);
  System.out.println("Float.MAX = " + Float.MAX_VALUE);
  System.out.println("Double.MIN = " + Double.MIN_VALUE);
  System.out.println("Double.MAX = " + Double.MAX_VALUE);
 }
}
```

Out Put:

E:\JAVA>javac MinMaxExample.java

E:\JAVA>java MinMaxExample

Byte.MIN = -128

Byte.MAX = 127

Short.MIN = -32768

Short.MAX = 32767

Integer.MIN = -2147483648

Integer.MAX = 2147483647

Long.MIN = -9223372036854775808

Long.MAX = 9223372036854775807

Float.MIN = 1.4E-45

Float.MAX = 3.4028235E38

Double.MIN = 4.9E-324

Double.MAX = 1.7976931348623157E308

4. Write a JAVA program to display the Fibonacci sequence

```
class FibonacciExample1{
public static void main(String args[])
{
 int n1=0,n2=1,n3,i,count=10;
 System.out.print(n1+" "+n2);
  for(i=2;i<count;++i)
 {
  n3=n1+n2;
  System.out.print(" "+n3);
  n1=n2;
  n2=n3;
 }
  }
 }
```

Out Put:

E:\JAVA>javac FibonacciExample1.java

E:\JAVA>java FibonacciExample1

0 1 1 2 3 5 8 13 21 34

4. Write a JAVA program give example for command line arguments.

```
class CommandLineExample{
```

```
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
System.out.println("Your first argument is: "+args[1]);
System.out.println("Your first argument is: "+args[2]);
}
}
```

Out Put:

E:\JAVA>javac CommandLineExample.java

E:\JAVA>java CommandLineExample rise prakasam cse

Your first argument is: rise

Your first argument is: prakasam

Your first argument is: cse

5. Write java Program which is capable of adding any number of integers passed as command line arguments.

```
public class Add {

  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < args.length; i++) {
      sum = sum + Integer.parseInt(args[i]);
    }
    System.out.println("The sum of the arguments passed is " + sum);
  }
}
```

OutPut:

E:\JAVA>javac Add.java

E:\JAVA>java Add 1 2 5

The sum of the arguments passed is 8

Classes:

- Class is a template for creating objects which defines its **state** and **behavior**. A class contains *field* and *method* to define the *state* and behavior of its object.
- In Java everything is encapsulated under classes. Class is the core of Java language.
- Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity.
- A class defines new data type. Once defined this new type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.
- A class is declared using **class** keyword. A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.
- A class in java can contain:
    - **data member**
    - **method**
    - **constructor**
    - **block**
    - **class and interface**

Rules for Java Class:

- A class can have only public or default(no modifier) access specifier.
- It can be either abstract, final or concrete (normal class).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces {}.
- Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.
- By convention, class names capitalize the initial of each word.
- For example: Employee, Boss, DateUtility, PostOffice, RegularRateCalculator.
- This type of naming convention is known as Pascal naming convention.
- The other convention, the camel naming convention, capitalize the initial of each word, except the first word.
- Method and field names use the camel naming convention.

Syntax to declare a class:

```
class <class_name>
{
  data member;
  method;
}
```

Example:

```
class Student.
{
 String name;
 int rollno;
 int age;
}
```

Object in Java:

- An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.
- An object has three characteristics:
    - **state:** represents data (value) of an object.
    - **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
    - **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.
- For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
- Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Creating an Object:

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java, the new key word is used to create new objects.

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.
- To create object of a class ***<new> Keyword*** can be used.

Syntax:

<Class_Name> ClassObjectReference = new <Class_Name>();

Here constructor of the class(*Class_Name*) will get executed and object will be created(ClassObjectRefrence *will hold the reference of created object in memory*).

The new() operator

Using Person class, you can create any number of objects using new() operator.The syntax is

classname objectname = new classname();

When a reference is made to a particular student with its property then it becomes an **object**, physical existence of Student class.

Student std=new Student();

After the above statement **std** is instance/object of Student class. Here the **new** keyword creates an actual physical copy of the object and assign it to the **std** variable. It will have physical existence and get memory in heap area. **The new operator dynamically allocates memory for an object.**

Simple Example of Object and Class:

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
class Student1{
 int id=100;//data member (also instance variable)
int marks=500 ;//data member(also instance variable)
  public static void main(String args[]){
 Student1 s1=new Student1();//creating an object of Student
 System.out.println(s1.id);
 System.out.println(s1.marks);
```

```
    }
}
```

Out Put:

```
E:\JAVA>javac Student1.java
E:\JAVA>java Student1
100
500
```

METHODS IN JAVA:

- Method describe behavior of an object. A method is a collection of statements that are group together to perform an operation.

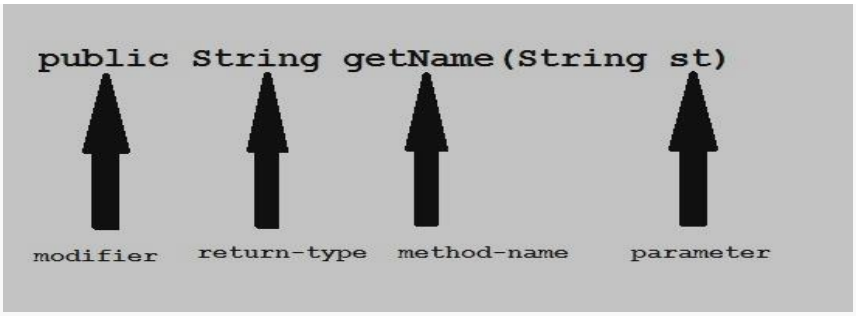
- A method is like function i.e. used to expose behavior of an object.
- Advantage of Method
  - Code Reusability
  - Code Optimization

Syntax :

```
return-type methodName(parameter-list)
{
 //body of method
}
```

Example of a Method

```
public String getName(String st)
{
 String name="StudyTonight";
 name=name+st;
 return name;
}
```



Modifier : Modifier are access type of method. We will discuss it in detail later.

Return Type : A method may return value. Data type of value return by a method is declare in method heading.

Method name : Actual name of the method.

Parameter : Value passed to a method.

Method body : collection of statement that defines what method does.

- Methods define actions that a class's objects (or instances) can do.
- A method has a declaration part and a body.
- The declaration part consists of a return value, the method name, and a list of arguments.
- The body contains code that perform the action.
- The return type of a method can be a primitive, an object, or void.
- The return type void means that the method returns nothing.
- The declaration part of a method is also called the signature of the method.

Example:

```
public class MethodExample{
public static void PrintLine() {
System.out.println("This is a line of text.");
}
public static void main(String[] args) {
System.out.println("Start Here");
PrintLine();
System.out.println("Back to the Main");
PrintLine();
System.out.println("End Here");
```

```
}
}
```

Out Put:

```
E:\JAVA>javac MethodExample.java
E:\JAVA>java MethodExample
Start Here
This is a line of text.
Back to the Main
This is a line of text.
End Here
```

Example: Method with arguments:

```
class Method
{
public static void main(String args[])
  {
     Method obj = new Method();
     obj.disp('a');
     obj.disp('a',10);
  }
   public void disp(char c)
   {
      System.out.println(c);
   }
   public void disp(char c, int num)
   {
       System.out.println(c + " "+num);
   }
}
```

OutPut:

```
E:\JAVA>javac Method.java
```

E:\JAVA>java Method

a

a 10

Access Control:

- Java provides control over the visibility of variables and methods.
- Encapsulation, safely sealing data within the capsule of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

Access Modifiers:

- Public
- Private
- Protected
- Default

- Access modifiers help you set the level of access you want for your class, variables as well as methods.
- Access modifiers (Some or All) can be applied on Class, Variable, Methods

Java Access Modifiers Table for Class

Visibility	Public Access Modifier	Default Access Modifier
Within Same Package	Yes	Yes
From Outside the Same Package	Yes	No

Public

When set to public, the given class will be accessible to all the classes available in Java world.

Default

When set to default, the given class will be accessible to the classes which are defined in the same package.

Access Modifiers for Variable (Instance / Static Variable)

- Default
- Public
- Protected
- Private

Note*: Visibility of the class should be checked before checking the visibility of the variable defined inside that class. If the class is visible only then the variables defined inside that class will be visible . If the class is not visible then no variable will be accessible, even if it is set to public.

Default

If a variable is set to default, it will be accessible to the classes which are defined in the same package. Any method in any class which is defined in the same package can access the variable v*ia* ***Inheritance*** *or* ***Direct access.***

Public

If a variable is set to public it can be accessible from any class available in the Java world. Any method in any class can access the given variable v*ia* ***Inheritance*** *or* ***Direct access.***

Protected

If a variable is set to protected inside a class, it will be accessible from its sub classes defined in the same or different package only via ***Inheritance***.

Note: The only difference between protected and default is that protected access modifiers respect **class subclass relation** while default does not.

Private

A variable if defined private will be accessible only from within the class it is defined. Such variables are not accessible from outside the defined class, not even its subclass .

Java Access Modifiers Table for Variable

Visibility	**Public Access Modifier**	**Private Access Modifier**	**Protected Access Modifier**	**Default Access Modifier**
Within Same Class	Yes	Yes	Yes	Yes
From Any Class in Same Package	Yes	No	Yes	Yes
From Any Sub Class in Same Package	Yes	No	Yes	Yes
From Any Sub Class from Different Package	Yes	No	Yes(**Only By Inheritance**)	No
From Any Non Sub Class in Different Package	Yes	No	No	No

Access Modifiers for Methods

Methods are eligible for all of the above mentioned modifiers.

Default

When a method is set to default it will be accessible to the class which are defined in the same package. Any method in any class which is defined in the same package can access the given method v*ia* ***Inheritance*** *or* ***Direct access****.*

Public

When a method is set to public it will be accessible from any class available in the Java world. Any method in any class can access the given method v*ia* ***Inheritance*** *or* ***Direct access*** depending on class level access.

Protected

If a method is set to protected inside a class, it will be accessible from its sub classes defined in the same or different package.

Note:* The only difference between protected and default is that protected access modifiers respect **class subclass relation** while default does not.

Private

A method if defined private will be accessible only from within the class it is defined. Such methods are not accessible from outside the defined class, not even its subclass.

Java Access Modifiers Table for Method

Visibility	**Public Access Modifier**	**Private Access Modifier**	**Protected Access Modifier**	**Default Access Modifier**
Within Same Class	Yes	Yes	Yes	Yes
From Any Class in Same Package	Yes	No	Yes	Yes
From Any Sub Class in Same Package	Yes	No	Yes	Yes
From Any Sub Class from Different Package	Yes	No	Yes(**Only By Inheritance**)	No
From Any Non Sub Class in Different Package	Yes	No	No	No

CONSTRUCTORS in java:

A constructor is a special member method which will be called by the JVM implicitly (automatically) for placing user/programmer defined values instead of placing default values.Constructors are meant for initializing the object.

ADVANTAGES of constructors:

- A constructor eliminates placing the default values.
- A constructor eliminates calling the normal method implicitly.

RULES/PROPERTIES/CHARACTERISTICS of a constructor:

- Constructor name must be similar to name of the class.
- Constructor should not return any value even void also (if we write the return type for theconstructor then that constructor will be treated as ordinary method).
- Constructors should not be static since constructors will be called each and every time whenever an object is creating.
- Constructor should not be private provided an object of one class is created in another class (constructor can be private provided an object of one class created in the same class).
- Constructors will not be inherited at all.
- Constructors are called automatically whenever an object is creating.

TYPES of constructors:

Based on creating objects in JAVA we have two types of constructors.

They are

- default/parameter less/no argument constructor
- parameterized constructor.

A default constructor is one which will not take any parameters.

Syntax:

```
class <clsname>
{
clsname () //default constructor
{
Block of statements;
```

```
……….;
……….;
}
……….;
……….;
};
```

For example:

```
class Test
{
int a, b;
Test ()
{
System.out.println ("I AM FROM DEFAULT CONSTRUCTOR...");
a=10;
b=20;
System.out.println ("VALUE OF a = "+a);
System.out.println ("VALUE OF b = "+b);
}
};
class TestDemo
{
public static void main (String [] args)
{
Test t1=new Test ();
}
};
```

OutPut:

```
E:\JAVA>javac TestDemo.java
E:\JAVA>java TestDemo
I AM FROM DEFAULT CONSTRUCTOR...
VALUE OF a = 10
```

VALUE OF b = 20

RULE-1:

Whenever we create an object only with default constructor, defining the default constructor is optional. If we are not defining default constructor of a class, then JVM will call automatically system defined default constructor (SDDC). If we define, JVM will call user/programmer defined default constructor (UDDC).

A parameterized constructor is one which takes some parameters.

Syntax:

```
class <clsname>
{
……..;
……..;
<clsname> (list of parameters) //parameterized constructor
{
Block of statements (s);
}
……..;
……..;
}
```

For example:

```
class Test
{
int a, b;
Test (int n1, int n2)
{
System.out.println ("I AM FROM PARAMETER CONSTRUCTOR...");
a=n1;
b=n2;
System.out.println ("VALUE OF a = "+a);
System.out.println ("VALUE OF b = "+b);
}
```

```
};
class TestDemo1
{
public static void main (String k [])
{
Test t1=new Test (10, 20);
}
};
```

OutPut:

```
E:\JAVA>javac TestDemo1.java
E:\JAVA>java TestDemo1
I AM FROM PARAMETER CONSTRUCTOR...
VALUE OF a = 10
VALUE OF b = 20
```

RULE-2:

- Whenever we create an object using parameterized constructor, it is mandatory for the JAVA programmer to define parameterized constructor otherwise we will get compile time error.

Overloaded constructor:

Overloaded constructor is one in which constructor name is similar but its signature is different. Signature represents number of parameters, type of parameters and order of parameters. Here, at least one thing must be differentiated.

For example:

```
Test t1=new Test (10, 20);
Test t2=new Test (10, 20, 30);
Test t3=new Test (10.5, 20.5);
Test t4=new Test (10, 20.5);
Test t5=new Test (10.5, 20);
```

RULE-3:

Whenever we define/create the objects with respect to both parameterized constructor and default constructor, it is mandatory for the JAVA programmer to define both the constructors.

NOTE:

When we define a class, that class can contain two categories of constructors they are single default constructor and 'n' number of parameterized constructors (overloaded constructors).

Example:

```
class Test
{
int a, b;
Test ()
{
System.out.println ("I AM FROM DEFAULT CONSTRUCTOR...");
a=1;
b=2;
System.out.println ("VALUE OF a ="+a);
System.out.println ("VALUE OF b ="+b);
}
Test (int x, int y)
{
System.out.println ("I AM FROM DOUBLE PARAMETERIZED CONSTRUCTOR...");
a=x;
b=y;
System.out.println ("VALUE OF a ="+a);
System.out.println ("VALUE OF b ="+b);
}
Test (int x)
{
```

```
System.out.println ("I AM FROM SINGLE PARAMETERIZED
CONSTRUCTOR...");
a=x;
b=x;
System.out.println ("VALUE OF a ="+a);
System.out.println ("VALUE OF b ="+b);
}
Test (Test T)
{
System.out.println ("I AM FROM OBJECT PARAMETERIZED
CONSTRUCTOR...");
a=T.a;
b=T.b;
System.out.println ("VALUE OF a ="+a);
System.out.println ("VALUE OF b ="+b);
}
};
class TestDemo2
{
public static void main (String k [])
{
Test t1=new Test ();
Test t2=new Test (10, 20);
Test t3=new Test (1000);
Test t4=new Test (t1);
}
};
```

Output:

```
E:\JAVA>javac TestDemo2.java
E:\JAVA>java TestDemo2
I AM FROM DEFAULT CONSTRUCTOR...
```

VALUE OF a =1

VALUE OF b =2

I AM FROM DOUBLE PARAMETERIZED CONSTRUCTOR...

VALUE OF a =10

VALUE OF b =20

I AM FROM SINGLE PARAMETERIZED CONSTRUCTOR...

VALUE OF a =1000

VALUE OF b =1000

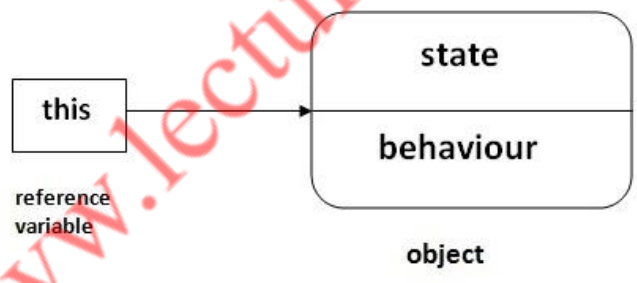
I AM FROM OBJECT PARAMETERIZED CONSTRUCTOR...

VALUE OF a =1

VALUE OF b =2

'this ':

In java, this is a reference variable that refers to the current object.



- 'this' is an internal or implicit object created by JAVA for two purposes. They are
- 'this' object is internally pointing to current class object.

- Whenever the formal parameters and data members of the class are similar, to differentiate the data members of the class from formal parameters, the data members of class must be proceeded by 'this'.
- this (): this () is used for calling current class default constructor from current class parameterized constructors.
- this (...): this (...) is used for calling current class parameterized constructor from other category constructors of the same class.

Usage of java this keyword

1) The this keyword can be used to refer current class instance variable.
2) this() can be used to invoked current class constructor.

Rule for 'this':

- Whenever we use either this () or this (...) in the current class constructors, that statements must be used as first statement only.
- The order of the output containing this () or this (...) will be in the reverse order of the input which we gave as inputs.

NOTE:

Whenever we refer the data members which are similar to formal parameters, the JVM gives first preference to formal parameters whereas whenever we write a keyword this before the variable name of a class then the JVM refers to data members of the class. this methods are used for calling current class constructors.

NOTE:

• If any method called by an object then that object is known as source object.

• If we pass an object as a parameter to the method then that object is known as target object.

Examples:

1) The this keyword can be used to refer current class instance variable.

```
class Student{
   int id;
   int marks;

   Student(int id, int marks){
  this.id = id;
   this.marks = marks;
   }
   void display(){System.out.println(id+" "+marks);}
   public static void main(String args[]){
   Student s1 = new Student(111,1500);
   Student s2 = new Student(222,2000);
   s1.display();
   s2.display();
}
}
```

Out Put:

```
E:\JAVA>javac Student.java
E:\JAVA>java Student
111 1500
222 2000
```

2) this() can be used to invoked current class constructor.

```
class Student{
   int id;
   int marks;
 Student()
{System.out.println("default constructor is invoked");}

   Student(int id, int marks){
this();
  this.id = id;
   this.marks = marks;
   }
   void display(){System.out.println(id+" "+marks);}
   public static void main(String args[]){
   Student s1 = new Student(111,1500);
   Student s2 = new Student(222,2000);
   s1.display();
   s2.display();
```

```
    }
    }
```

Out Put:

```
E:\JAVA>javac Student.java
E:\JAVA>java Student
default constructor is invoked
default constructor is invoked
111 1500
222 2000
```

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

- It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
class Student{
   int rollno;
   String name;
   String college="ITS";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and

name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

Example of static variable

```
class Student8{
  int rollno;
  String name;
  static String college ="RISE";

  Student8(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+" "+college);}
  public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");
  s1.display();
 s2.display();
 }
}
```

Output:111 Karan RISE
222 Aryan RISE

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
class Student9{
   int rollno;
   String name;
   static String college = "RISE";
```

```
    static void change(){
    college = "RPRA";
    }
     Student9(int r, String n){
    rollno = r;
    name = n;
    }
     void display (){System.out.println(rollno+" "+name+" "+college);}
     public static void main(String args[]){
    Student9.change();
     Student9 s1 = new Student9 (111,"Karan");
    Student9 s2 = new Student9 (222,"Aryan");
    Student9 s3 = new Student9 (333,"Sonoo");
     s1.display();
    s2.display();
    s3.display();
    }
}
Output:111 Karan RPRA
     222 Aryan RPRA
     333 Sonoo RPRA
```

Restrictions for static method

There are two main restrictions for the static method. They are:

- The static method can not use non static data member or call non-static method directly.
- this cannot be used in static context.

ARRAYS

Arrays are generally effective means of storing groups of variables. An array is a group of variables that share the same name and are ordered sequentially from zero to one less than the number of variables in the array. The number of variables that can be stored in an array is called the array's dimension. Each variable in the array is called an element of the array.

Creating Arrays:

There are three steps to creating an array, declaring it, allocating it and initializing it.

Declaring Arrays:

Like other variables in Java, an array must have a specific type like byte, int, String or double. Only variables of the appropriate type can be stored in an array. You cannot have an array that will store both ints and Strings, for instance.

Like all other variables in Java an array must be declared. When you declare an array variable you suffix the type with [] to indicate that this variable is an array. Here are some examples:

int[] k;
float[] yt;
String[] names;

In other words you declare an array like you'd declare any other variable except you append brackets to the end of the variable type.

Allocating Arrays

Declaring an array merely says what it is. It does not create the array. To actually create the array (or any other object) use the new operator. When we create an array we need to tell the compiler how many elements will be stored in it. Here's how we'd create the variables declared above: new

k = new int[3];
yt = new float[7];
names = new String[50];

The numbers in the brackets specify the dimension of the array; i.e. how many slots it has to hold values. With the dimensions above k can hold three ints, yt can hold seven floats and names can hold fifty Strings.

Initializing Arrays

Individual elements of the array are referenced by the array name and by an integer which represents their position in the array. The numbers we use to identify them are called subscripts or indexes into the array. Subscripts are consecutive integers beginning with 0. Thus the array k above has elements k[0], k[1], and k[2]. Since we started counting at zero there is no k[3], and trying to access it will generate an ArrayIndexOutOfBoundsException. subscripts indexes k k[0] k[1] k[2] k[3] ArrayIndexOutOfBoundsException

You can use array elements wherever you'd use a similarly typed variable that wasn't part of an array.

Here's how we'd store values in the arrays we've been working with:

k[0] = 2;
k[1] = 5;

```
k[2] = -2;
yt[6] = 7.5f;
names[4] = "Fred";
```

This step is called initializing the array or, more precisely, initializing the elements of the array. Sometimes the phrase "initializing the array" would be reserved for when we initialize all the elements of the array.For even medium sized arrays, it's unwieldy to specify each element individually. It is often helpful to use for loops to initialize the array. For instance here is a loop that fills an array with the squares of the numbers from 0 to 100.

```
float[] squares = new float[101];

for (int i=0; i <= 500; i++) {
 squares[i] = i*2;
}
```

Two Dimensional Arrays

Declaring, Allocating and Initializing Two Dimensional Arrays

Two dimensional arrays are declared, allocated and initialized much like one dimensional arrays. However we have to specify two dimensions rather than one, and we typically use two nested for loops to fill the array. for The array examples above are filled with the sum of their row and column indices. Here's some code that would create and fill such an array:

```
class FillArray {

 public static void main (String args[]) {

  int[][] M;
  M = new int[4][5];

   for (int row=0; row < 4; row++) {
    for (int col=0; col < 5; col++) {
     M[row][col] = row+col;
    }
   }

 }
 }
```

In two-dimensional arrays ArrayIndexOutOfBounds errors occur whenever you exceed the maximum column index or row index. Unlike two-dimensional C arrays, two-dimensional Java arrays are not just one-dimensional arrays indexed in a funny way.

Multidimensional Arrays

You don't have to stop with two dimensional arrays. Java lets you have arrays of three, four or more dimensions. However chances are pretty good that if you need more than three dimensions in an array, you're probably using the wrong data structure. Even three dimensional arrays are exceptionally rare outside of scientific and engineering applications.

The syntax for three dimensional arrays is a direct extension of that for two-dimensional arrays. Here's a program that declares, allocates and initializes a three-dimensional array:

```
class Fill3DArray {

 public static void main (String args[]) {

  int[][][] M;
  M = new int[4][5][3];

  for (int row=0; row < 4; row++) {
   for (int col=0; col < 5; col++) {
    for (int ver=0; ver < 3; ver++) {
     M[row][col][ver] = row+col+ver;
    }
   }
  }
 }
}
```

Strings

- Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.
- The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a string literal—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays hello.

String Length

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A palindrome is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the ith character in the string, counting from 0.

```
public class StringDemo {
  public static void main(String[] args) {
    String palindrome = "Dot saw I was Tod";
    int len = palindrome.length();
    char[] tempCharArray = new char[len];
    char[] charArray = new char[len];

    // put original string in an array of chars
    for (int i = 0; i < len; i++) {
      tempCharArray[i] = palindrome.charAt(i);
    }

    // reverse array of chars
    for (int j = 0; j < len; j++) {
      charArray[j] = tempCharArray[len - 1 - j];
    }
        String reversePalindrome =  new String(charArray);
    System.out.println(reversePalindrome);
  }
```

}

Running the program produces this output:

doT saw I was toD

To accomplish the string reversal, the program had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The String class includes a method, getChars(), to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the program above with

palindrome.getChars(0, len, tempCharArray, 0);

Concatenating Strings

The String class includes a method for concatenating two strings:

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

"My name is ".concat("Rumplestiltskin");

Strings are more commonly concatenated with the + operator, as in

"Hello," + " world" + "!"

which results in

"Hello, world!"

The + operator is widely used in print statements. For example:

String string1 = "saw I was ";

System.out.println("Dot " + string1 + "Tod");

which prints

Dot saw I was Tod

Java Command Line Arguments

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

class CommandLineExample{

public static void main(String args[]){

```
System.out.println("Your first argument is: "+args[0]);
}
}
```

OutPut:

```
D:\>javac CommandLineExample.java
D:\>java CommandLineExample Rise
Your first argument is: Rise
```

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{
public static void main(String args[]){
for(int i=0;i<args.length;i++)
System.out.println(args[i]);
 }
}
```

OutPut:

```
D:\>javac A.java
D:\>java A 1 2 3 Rise Cse
1
2
3
Rise
Cse
```