Introduction to Java

- ○ Developed by Sun Microsystems (James Gosling)
- ○ Birth Date of Java 23rd May 1995
- ○ Development of Java was started under “Project Green”
- ○ First name was “Oak”, due to trademark issue it was renamed to Java.
- ○ A general-purpose object-oriented language
- ○ Based on C/C++
- ○ Designed for easy Web/Internet applications
- ○ Widespread acceptance

Features of Java

- Java is simple
  - ○ fixes some clumsy features of C++
  - ○ no pointers
  - ○ automatic garbage collection
  - ○ rich pre-defined class library
- Java is object-oriented
  - ○ Focus on the data (objects) and methods manipulating the data
  - ○ All functions are associated with objects
  - ○ Almost all datatypes are objects (file, strings, etc.)
  - ○ Potentially better code organization and reuse
  - ○ It supports concept of Inheritance, Polymorphism, Encapsulation, data hiding etc.
- Java is distributed
  - ○ Java Supports network programming like client-server paradigm and implementation of simple client server applications
  - ○ It also support web service protocol
- Java is interpreted
  - ○ Java compiler generate byte-codes, not native machine code
  - ○ the compiled byte-codes are platform-independent
  - ○ java bytecodes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
- Java is robust
  - ○ Extensive compile-time and runtime error checking
  - ○ No pointers but real arrays.
  - ○ Memory corruptions or unauthorized memory accesses are impossible
  - ○ Automatic garbage collection tracks objects usage over time
- Java is secure
  - ○ usage in networked environments requires more security
  - ○ memory allocation model is a major defense
  - ○ access restrictions are forced (private, public)
- Java is portable
  - ○ same application runs on all platforms
  - ○ the sizes of the primitive data types are always the same
  - ○ the libraries define portable interfaces
- Java is multithreaded
  - ○ Multiple concurrent threads of executions can run simultaneously
  - ○ Utilizes a sophisticated set of synchronization primitives (based on monitors and condition variables paradigm) to achieve this
- Java is dynamic
  - ○ Java is designed to adapt to evolving environment
  - ○ Libraries can freely add new methods and instance variables without any effect on their clients
  - ○ interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented
  - ○ Can check the class type in runtime

Object Oriented Programming Concepts

What is an OOP?

- A software design method that models the characteristics of real or abstract objects using software classes and objects.
- Characteristics of objects:
  - State (what the objects have)
  - Behavior (what the objects do)
  - Identity (what makes them unique)
- Definition: an object is a software bundle of related *fields* (variables) and *methods*.
- In OOP, a program is a collection of objects that act on one another (vs. procedures).
- For example, a car is an object.
  - Its state includes current:
    - Speed
    - RPM
    - Gear
    - Direction
    - Fuel level
    - Engine temperature
  - Its behaviors include:
    - Change Gear
    - Go faster/slower
    - Go in reverse
    - Stop
    - Shut-of
  - Its identity is:
    - License Plate

Classes in Java

- Everything in Java is defined in a class.
- In its simplest form, a class just defines a collection of data (like a record or a C struct). For example:

```
class Employee {

   String name;

   String ssn;

   String emailAddress;

   int yearOfBirth;

}
```

- The order of data fields and methods in a class is not significant.
- If you recall, each class must be saved in a file that matches its name, for example: Employee.java
- There are a few exceptions to this rule (for non-public classes), but the accepted convention is to have one class defined per source file.
- Note that in Java, Strings are also classes rather than being implemented as primitive types.
- Example
  - Person
  - Flower
  - Cricketer

Objects in Java

- To create an object (instance) of a particular class, use the new operator, followed by an invocation of a *constructor* for that class, such as:

```
new MyClass()
```

  - The constructor method initializes the state of the new object.
  - The new operator returns a *reference* to the newly created object.
- As with primitives, the variable type must be compatible with the value type when using object references, as in:

```
Employee e = new Employee();
```

- To access member data or methods of an object, use the dot (.) notation: *variable.field* or *variable.method*()

Example of Object Oriented

```
class A
{
      private int i; // Data Hiding
      A(){
      //Default Constructor
      }
      A(int i){
      //Constructor Overloaded
      }
      void setData(int i){}
}
class B extends A // Inheritance
{
      B(){}
      void setData(float f)
      {
            //Method Overloading
      }
      void setData(int f)
      {
            //Method Overriding
      }

}
```

- Method Overloading and Constructor overloading is known as polymorphism
- Wrapping up of data members and methods in one single unit called class is known as Encapsulation

Creating an Application in Java

1. Write the program in Java
2. Compile the source code
3. Run the program

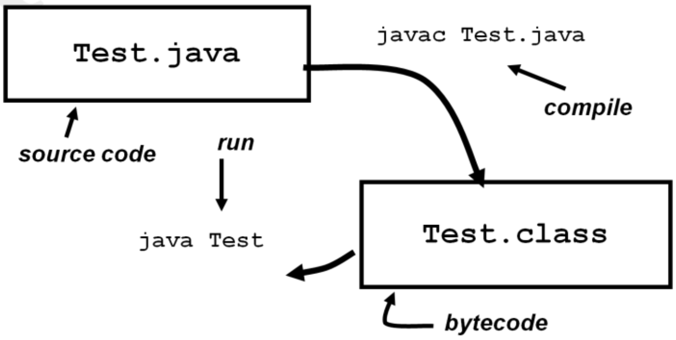
```
/*  Test.java
public class Test {
   public static void main(String[] args)
   {
      System.out.println("Hello ALL");
   }
}
```

- **To Compile**
  - `javac  file_name.java`
- **To Run**
  - `java class_name_with_main_method`

javac Test.java

java Test

Compiling and Running



Program Comments

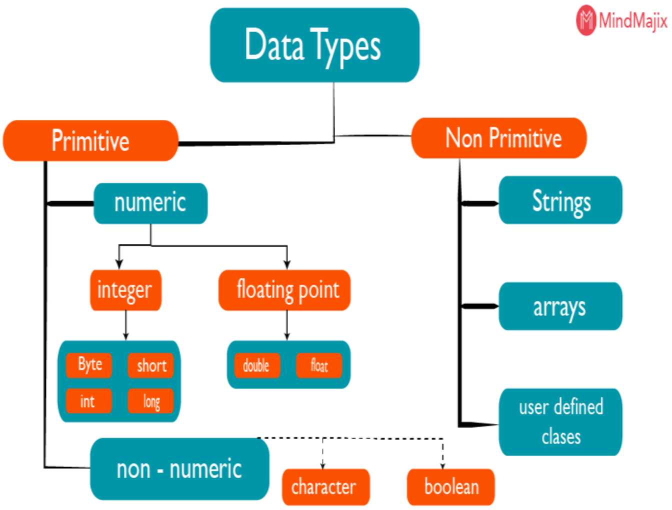
- It is good practice to write comments explaining your code.
- There are two types of comments in Java, both with syntax similar to comments in C and C++.
- Traditional comments: Enclose a traditional comment in /* and */.
- End-of-line comments: Use double slashes (//) which causes the rest of the line ignored by the compiler.
- Traditional comments do not nest, which means

```
/*
 /* comment 1 */
 comment 2 */
```

- is invalid because the first */ after the first /* will terminate, which will generate a compiler error
- End-of-line comments can contain anything, including the sequences of characters /* and */, such as this:

```
// /* this comment is okay */

/* multiple lines of comment
another line
 */
public class MainClass{
// this is single line comment
  public static void main(String[] arg)
  {
  }
}
```



Primitive Data types

- Java has eight primitive types of data: byte, short, int, long, char, float, double, and boolean.
- These can be put in four groups:
  1. Integers includes byte, short, int, and long
  2. Floating-point numbers includes float and double
  3. Characters includes char, like letters and numbers.
  4. Boolean includes boolean representing true/false values.

byte

- The smallest integer type
- a range **from** -128 to 127.
- useful when working with a stream of data **from** a network or file.
- Byte variables are declared by use of the **byte** keyword.

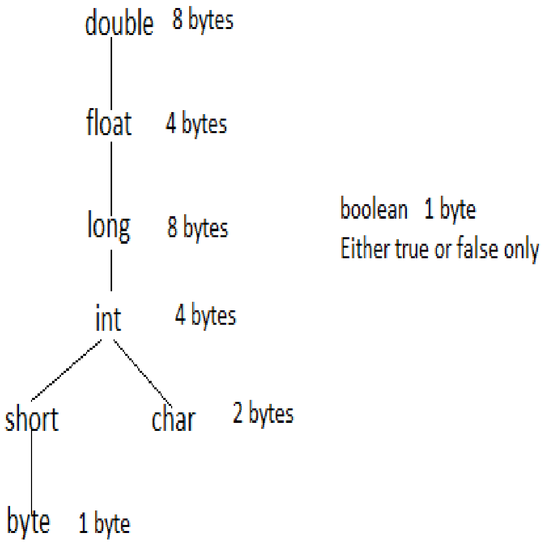
```
byte b, c;
```

int

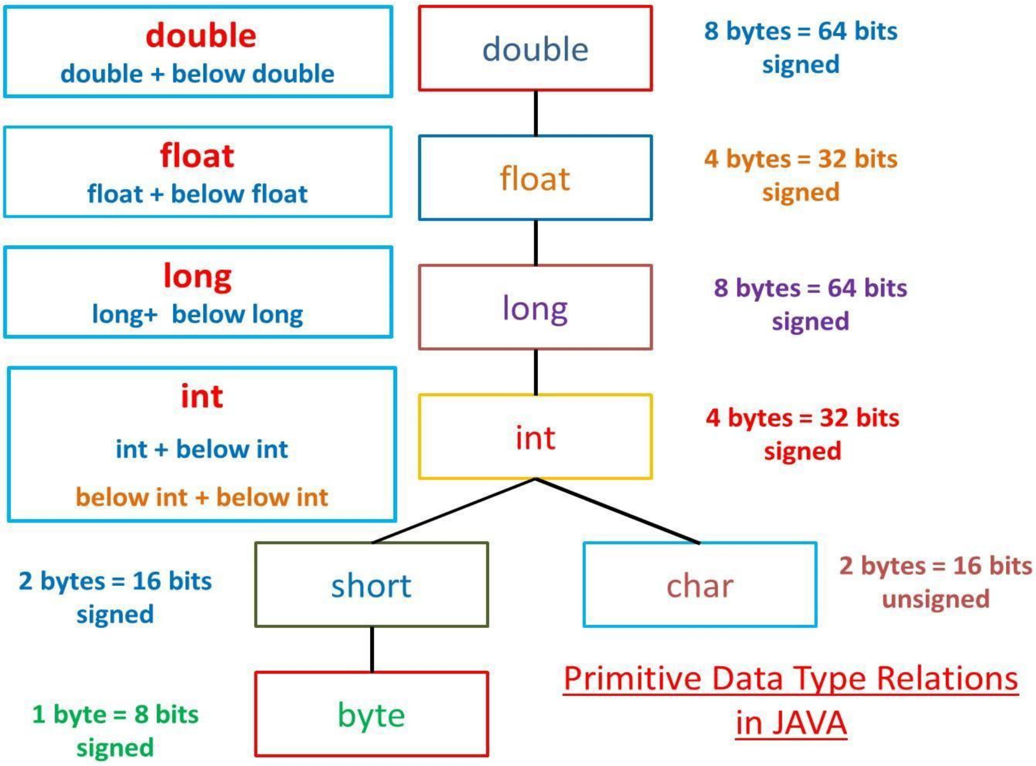
- The most commonly used integer type
- a signed 32-bit type
- Ranging from -2,147,483,648 to 2,147,483,647
- used to control loops and to index arrays.
- the most efficient type

long

a signed 64-bit type



Super type and sub type relations in primitive data types



Primitive Data Type Relations in JAVA

```
class DatatypeRange
{
          public static void main(String args[])
          {
                    System.out.println("        MIN VALUE           MAX VALUE     BIT     BYTE      TYPE");

          System.out.println("=====================================================");
                    System.out.printf(" %20d %20d %8d %8d
%9s\n",Byte.MIN_VALUE,Byte.MAX_VALUE,Byte.SIZE,((Byte.SIZE)/8),Byte.TYPE);
                    System.out.printf(" %20d %20d %8d %8d
%9s\n",Integer.MIN_VALUE,Integer.MAX_VALUE,Integer.SIZE,((Integer.SIZE)/8),Integer.TYPE)

                    System.out.printf(" %20d %20d %8d %8d
%9s\n",Long.MIN_VALUE,Long.MAX_VALUE,Long.SIZE,((Long.SIZE)/8),Long.TYPE);
                    System.out.printf(" %20d %20d %8d %8d
%9s\n",Short.MIN_VALUE,Short.MAX_VALUE,Short.SIZE,((Short.SIZE)/8),Short.TYPE);
                    System.out.printf(" %20e %20e %8d %8d
%9s\n",Float.MIN_VALUE,Float.MAX_VALUE,Float.SIZE,((Float.SIZE)/8),Float.TYPE);
                    System.out.printf(" %20e %20e %8d %8d
%9s\n",Double.MIN_VALUE,Double.MAX_VALUE,Double.SIZE,((Double.SIZE)/8),Double.TYPE);

          }
}
```

USING byte DATA TYPE And Byte Class

```
class Test {
  public static void main(String[] a){
    byte luckyNumber = 7;
    System.out.println(luckyNumber);
  }
}
```

Convert byte to String: Using the static toString method of the Byte Class

```
class Test {
  public static void main(String[] a){
        byte b = 65;
System.out.println(Byte.toString(b));
    }
}
```

Convert byte to String: Using simple concatenation with an empty String

```
class Test {
  public static void main(String[] a){
        byte b = 65;
        System.out.println(b + "");
    }
}
```

Convert String to byte

```
class Test {
 public static void main(String[] a) {
    String s = "65";

    byte b = Byte.valueOf(s);

    System.out.println(b);

/* Causes a NumberFormatException
since the value is out of range*/
System.out.println(Byte.valueOf("129"));
  }
}
```

/* **OUTPUT**

```
65
Exception in thread "main"
java.lang.NumberFormatException: Value out of
range. Value:"129" Radix:10
  at java.lang.Byte.parseByte(Byte.java:153)
  at java.lang.Byte.valueOf(Byte.java:184)
  at java.lang.Byte.valueOf(Byte.java:208)
  at Main.main(Main.java:11)
```

*/

Convert Byte to numeric primitive data types example

```
class Test {
  public static void main(String[] a)
  {
    Byte bObj = new Byte("10");
    byte b = bObj.byteValue();
    System.out.println(b);
    short s = bObj.shortValue();
    System.out.println(s);
    int i = bObj.intValue();
    System.out.println(i);
    float f = bObj.floatValue();
    System.out.println(f);
    double d = bObj.doubleValue();
    System.out.println(d);
    long l = bObj.longValue();
    System.out.println(l);
  }
}
```

USING int DATA TYPE AND Integer class

Integer: MAX, MIN VALUE

- A constant holding the maximum value an int can have, 2^31-1.
- A constant holding the minimum value an int can have, -2^31.

```
class Test {

  public static void main(String[] a){
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);
System.out.println(Integer.SIZE);
  }
}
```

Create an Integer object

```
class Test {
 public static void main(String[] a){
  Integer intObj1 = new Integer(10);
  Integer intObj2 = new Integer("10");
  System.out.println(intObj1);
  System.out.println(intObj2);
  }
}
```

Read Integers from console and calculate

```
import java.util.Scanner;
class Test {
 public static void main(String a[]){
   Scanner input = new
              Scanner( System.in );

int x,y,z,result;

  System.out.print( "Enter 1st no :");
  x = input.nextInt();
  System.out.print( "Enter 2nd no :");
  y = input.nextInt();
  System.out.print( "Enter 3rd no :");
  z = input.nextInt();

  result = x * y * z;

 System.out.printf( "Product is %d\n",
result );
   }
}
```

Convert an int value to String: Integer.toString(i)

```
class Test {
  public static void main(String[] a){
    int i = 50;

    String str = Integer.toString(i);
    System.out.println(str + " : " +
Integer.toString(i).getClass());
  }
}
```

USING char DATA TYPE

Display printable Characters

```
class TestClass {
  public static void main(String[] a){

 for (int i = 32; i < 127; i++) {
      System.out.write(i);
// break line after every eight char.
      if (i % 8 == 7)
        System.out.write('\n');
      else
        System.out.write('\t');
    }
    System.out.write('\n');

  }
}
```

char variables behave like integers

```
class TestClass {
  public static void main(String a[]){
    char ch1;
    ch1 = 'X';
    System.out.println("ch1 contains "
+ ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " +
ch1);
  }
}
```

Assign int value to char variable

```
class TestClass {
 public static void main(String a[]) {
    char ch1, ch2;
    ch1 = 88; // code for X
    ch2 = 'Y';

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

Storing Characters

Variables of type char

1. store a single character code.
2. occupy 16 bits, or 2 bytes,
3. all characters in Java are stored as Unicode.

```
class TestClass{

  public static void main(String[] a){
     char myCharacter = 'X';
     System.out.println(myCharacter);
  }

}
```

Character: is Upper Case

```
class TestClass {
  public static void main(String[] a){
    char symbol = 'A';
    if(Character.isUpperCase(symbol)){
      System.out.println("true");
    }else{
      System.out.println("false");
    }
  }
}
```

USING short DATA TYPE

```
class Test {

 public static void main(String[] a) {
    short s1 = 50;
    short s2 = 42;
    System.out.println("Value of short
variable b1 is :" + s1);
    System.out.println("Value of short
variable b1 is :" + s2);
  }
}

class Test {
  public static void main(String[] a){
    short smallNumber = 1234;
    System.out.println(smallNumber);
  }
}
```

Min and Max values of datatype short

```
class Test {
 public static void main(String[] a)
{
System.out.println(Short.MIN_VALUE);
System.out.println(Short.MAX_VALUE);
}
}
```

Cast back to short

```
class Test {
  public static void main(String[] a){
    short a, b, c;
    c = 2;
    b = 9;
    a = (short) (b + c);
    System.out.println("a is " + a);
  }
}
```

Cast result of plus opertion to byte

```
class Test {
  public static void main(String[] a){
    short a, b, c;
    c = 2;
    byte s;
    s = (byte) c;
    System.out.println("s is " + s);
  }
}
```

Convert Java String to Short

```
class Test {
  public static void main(String[] a){
    Short sObj1 = new Short("100");
    System.out.println(sObj1);
    String str = "100";
    Short sObj2 = Short.valueOf(str);
    System.out.println(sObj2);
  }
}
```

Convert String to short primitive

```
class Test {
  public static void main(String[] a){
    short s = Short.parseShort("10");
    System.out.println(s);
  }
}
```

Convert Short to numeric primitive data types

```
class Test {
  public static void main(String[] a){

    Short sObj = new Short("10");
    byte b = sObj.byteValue();
    System.out.println(b);

    short s = sObj.shortValue();
    System.out.println(s);

    int i = sObj.intValue();
    System.out.println(i);

    float f = sObj.floatValue();
    System.out.println(f);

    double d = sObj.doubleValue();
    System.out.println(d);

    long l = sObj.longValue();
    System.out.println(l);
  }
}
```

// Demonstrate boolean values.

```
class BoolTest {
public static void main(String a[]) {
    boolean b;

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

 //a boolean value can control the if

    if(b)
System.out.println("This is executed.");

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

System.out.println("10 > 9 is" + (10>9));
  }
}
```

Types of Conversion in Java

1. **Widening Conversion**
2. **Narrowing Conversion**
3. **Mixed Conversion**

Widening Conversion

Conversion of data type from sub-data type to super data type is known as widening conversion.

- Byte to short,int,long,float,double
- Short to int , float,ling, double
- Int to float, long ,dounble
- Float to long double
- Long to double

Example

```
int i = 50;
long l = i;
```

Narrowing Conversion

Conversion of dta type from super data type to sub data type is called Narrowing Conversion.

- Double to float,long,int,char,short,byte
- Float to long,int,char,short,byte
- Lont to int,cahr,short,buyte
- Int to char,short,byte
- Chat to short,byte
- Short to byte

Example

```
long l = 50;
int i=l;
```

```
// Demonstrate casts.
class Conversion {
 public static void main(String a[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion
of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i
+ " " + b);

    System.out.println("\nConversion
of double to int.");
    i = (int) d;
    System.out.println("d and i " + d +
" " + i);

    System.out.println("\nConversion of
double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d +
" " + b);
  }
}
```

```
// Type Promotion
class Promote {
  public static void main(String
args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) -
(d * s);
    System.out.println((f * b) + " + "
+ (i / c) + " - " + (d * s));
    System.out.println("result = " +
result);
  }
}
```

Array

- An array is a type of variable that can store multiple values.
- The array is an object in Java that contains similar data type values.
- **A few main points about arrays in Java:**
- **Java Array** is a data structure in java that can hold one or more values in a single variable.
- Array in java is a collection of similar type of values.
- Java has two types of arrays,
  - single dimensional and
  - multidimensional java arrays.
- Java array index starts at 0.
- This is how an array in java can be declared:

ArrayDataType[] ArrayName;
OR
ArrayDataType ArrayName[];

Where ArrayDataType defines the data type of **java array** element like int, double etc.

```
class array_ex {

public static void main(String []args)
{
//Declaring and initializing an array
        int arrex[] = {10,20,30};

        for (int i=0;i<arrex.length;i++){
              System.out.println(arrex[i]);
            }
    }
}
```

// Demonstrate a one-dimensional array.

```
class Array {
 public static void main(String a[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " +
month_days[3] + " days.");
  }
}
```

/* An improvied version of the previous program.*/

```
class AutoArray {
  public static void main(String a[]){
    int month_days[] = { 31, 28, 31,
30, 31, 30, 31, 31, 30, 31, 30, 31 };

    System.out.println("April has " +
month_days[3] + " days.");
  }
}
```

// Average an array of values.

```
class Average {
  public static void main(String a[]){
    double nums[] = {10.1, 11.2, 12.3,
13.4, 14.5};
    double result = 0;
    int i;

    for(i=0; i<nums.length; i++)
      result = result + nums[i];

    System.out.println("Average is " +
result / 5);
  }
}
```

```
// Demonstrate a two-dimensional
array.
class TwoDArray {
  public static void main(String
args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] +
" ");
      System.out.println();
    }
  }
}
```

```
// Manually allocate differing size
second dimensions.

class TwoDAgain {
  public static void main(String
args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] +
" ");
      System.out.println();
    }
  }
}
```

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String a[]){
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + "
");
      System.out.println();
    }
  }
}
```

```
// Demonstrate a three-dimensional
array.
class ThreeDMatrix {
  public static void main(String a[]){
    int threeD[][][] = new
int[3][4][5];
    int i, j, k;

    for(i=0; i<threeD.length; i++)
     for(j=0; j<threeD[i].length; j++)
  for(k=0; k<threeD[i][j].length; k++)
        threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)

System.out.print(threeD[i][j][k] + "");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

Operators in Java

Following are the operators of java:-

1) Arithmetic Operators
2) Relational Operators
3) Logical Operators
4) Assignment Operators
5) Increment-Decrement Operators
6) Bitwise Operators
7) Conditional or Ternary Operators
8) instanceof operator

1) Arithmetic operators:-

Operator	Description	Example
*	Multiplication	oprd1 * oprd2
/	Division	oprd1 / oprd2
%	Modulus (remainder)	oprd1 % oprd2
+	Addition	oprd1+ oprd2
-	Subtraction	oprd1 - oprd2

- *,/,% operators has same priority it is evaluated from left to right.
- + and - has same prioriy.
- % can not use for float and double data type.

2) Relational operator:-

- Relational operators are used in boolean conditions or expressions.
- Boolean conditions or expressions returns either TRUE or FALSE.
- The relational operator returns ZERO or NONZERO value.
- The ZERO value is taken as FALSE while the nonzero value taken as TRUE.
- The relational value is taken as FALSE while the nonzero value taken as TRUE.
- The relational operators are as follows:
- <, <=, >, >=, ==, !=.
- The priority of the first four operators is higher than that of the later two operators.
- It returns ZERO values if condition is FALSE or NONZERO it it is TRUE.

Operator	Description	Example
<	Less than	a < b
<	Greater than	a > b
<=	Leass than or Equal	a <= b
>=	Greater than or equal	a > = b
==	comparision	a==b
!=	Not Equal	a!=b

3) Logical operator:-

Operator	Description	Example
&&	Logical AND	a > b && a>c
||	Loggical OR	a>b || a>c
!	Logical NOT	a!(a > b)

Logical AND [&& operator]:-

- && operator evaluate a boolean expression from left to right.
- Both sides of the operator must evaluate to TRUE before the entire expression becomes TRUE.
- Logical AND produces 0 if one or both its operands evaluates to 0. otherwise it produces 1.

Logical OR [|| operator]:-

- It returns TRUE if any of the expressions is TRUE.
- Logical OR produces 0 if both its operands evaluate tp 0.. otherwise it prodices 1.

Negation [! operator]:-

- It returns complements of values of relational expressions.

- Logical negation is a unary operator that negates the logical value of its single operand.

- If its operand is non-zero, it produces 0, and if it is 0, it produces 1.

R1	R2	R1 && R2	R1 || R2	! R1
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

4) Assignment operator:-

- Assigning the value of an expression to a variable
- var = expression

- You can use formats such as
- var + = expression
- var = var + expressions
- Assignment operators have the lowest priority.
- It evaluated from left to right.

Operators	Descriptions	Examples
=	Equal assignment	result = value
+=	Plus & assignment	var += value
-=	Minus & assign	var - = value
*=	Multiplication&ass	var * =value
/=	Divide & ass	var/ = value
%=	Modulo & assi	var %=value

5) Increment - Decrement operator:-

- It is used for increse or decrease the value of variable.

- It is called unary operator.

PREFIX:- The value is incremented / decremented first and than applied.

POSTFIX:- The value is applied and the value is incremented / decremented.

- The unary '++' and '—' operator increment or decrement the value in a variable 1.

- var++ : increment 'Post' variant
- ++var : increment 'Pre' varient
- var-- : decrement 'Post' varient
- --var : decrement 'Pre' varient.

6) Bitwise operator :-

- Java has a distinction of supporting special operator known as Bitwise operators for manipulating of data at bit level.
- These operator are used for individual bits in an integer quantity.
- Bitwisw negation is a unary operator that complements the bits in its operands.
- Bitwise AND compares the corresponding bits of its operands and produces a 1 when bothbits are 1 and 0 otherwise.
- Bitwise OR compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise.
- Bitwise exclusive or compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0 and 1 otherwise.

Operators	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Bitwise Left shift
>>	Bitwise Right shift
>>>	Unsigned Shift Right
~	Bitwise Complement

7) Ternary operator / ? : / Conditional operator :-

- A ternary operator pair “ ?: “ is available in C to construct conditional expressions of the form.
- Exp1? Exp 2: exp3
- Where exp1, exp2, exp3 are expressions.

The operator? : works as follows:-

- Exp1 is evaluated first. If it is non zero (or true)), than the expression exp2 is evaluated. If exp1 is false, exp3 is evaluated.
- Only one os the expressions (either exp1 or exp3) is evaluated.

```
Ex: -   a=10;
        B=15;
        X= (a>b)? a:b ;
```

8) instanceof Operator

- instanceof operator return boolean value.
- It returns true if specified instance is of specified type otherwise returns false.
- Syntax
  - instance_name instanceof Classname

```
class A
{
}
class B extends A
{
}
class TestDemo
{
        public static void main(String []args)
        {
                A a = new A();
                B b = new B();
                System.out.println(a instanceof A);
                System.out.println(b instanceof B);
                System.out.println(a instanceof B);
                System.out.println(b instanceof A);
        }
}

//Output
true
true
false
true
```

Defining Class

Member of the classes are

1. Instance Variable
2. Class Variable (static)
3. Instance Methods
4. Class Methods (static)
5. Constructor
6. Instance initializer block
7. Class initializer block (static block)

```
class Test
{
        int i=500; // Instance Variable
        static int p=100; // Class Variable
        Test()
        {
                //Default Constructor
        }
        Test(int i)
        {
                //Parameterized constructor
                this.i=i;
        }

        {
        System.out.println("Initializer Block"+i);
        }
        static
        {
        System.out.println("Class Initializer Block"+p);
        }
        static void display() // static method
        {
        System.out.println("It's Interestring"+p);
        }
        void show()//non static method
        {
        }
        void show(int i){
        //Method Overloading
        }
}
class TestDemo
{
        public static void main(String []args)
        {
                Test t1,t2; // Reference Variable
                 for(int i=0;i<10;i++)
                        new Test();
        }
}
```

Instance Variable and static variable with example.

Instance Variable:

- These are the variable whose separate copy is created for each instance in class.
- This variable does not exists without an instance.
- We can use this variable using this keyword.

Example

```
class Student
{
        int rno;        // instance variable
        String name; // instance variable
        Student(int rno, String name)
        {
                this.rno=rno;
                this.name=name;
        }
}
```

static variables

- Normally a class member must be accessed only with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, declaration with the keyword **static**.
  - The most common example of a **static** member is **main( )**.

```
class Stuff
 {
        public static int val=100;
}

class Application
{
        public static void main(String[] args)
        {
        System.out.println(Stuff.val);
        }
}
```

Initalizer Block and Class Initializer block.

Initializer block

- It is used to define activity that is required to carried out whenever any instance is created for the class.
- There can be any number of initializer block in a class.
- This code executed just before constructor code is executed

Example :

```
class Test
{
     {
       System.out.println("Empty  block");
     }

     static {
       System.out.println("Static  block");
     }
   public static void main(String[] args)
     {
        Test t = new Test();
     }
}
```

Initializer Block	Class Initializer Block
It is used to define activity that is carried out whenever any instance is created for the class.	It is used to initialize class variables.
Initializer Block is a execute just before constructor is called.	It will execute when class is loaded.
This Block is called for each instance separately.	It is execute **only once** when class is loaded

Syntex:
class Test
{
{
// initializer block
}
}
	Syntex:
	class Test
	{
	static
	{
	// Class initializer block
	}
	}

Example:
class Test
{
static int p=0;
int i=50;
{
i++;
p++;
}
}
	Example:
	class Test
	{
	static int p=0;
	int i=50;
	static
	{
	p++;
	// i is not accessible
	}

Use of " this" keyword in Java

- In java, this is a **reference variable** that refers to the current object.
- this keyword can be used to refer current class instance variable.
- this() can be used to invoke current class overloaded constructor.
- this keyword can be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this keyword can also be used to return the current class instance.

```
class Student
{
        int id;
        String name;
        Student()
        {
            System.out.println("Default Cons");
        }

        Student(int id,String name)
        {
          this(); // used to invoke default cons
          this.id = id;
          this.name = name;
        }
        void display()
        {
          System.out.println(id+" "+name);
        }
        public static void main(String args[])
        {
          Student s1 = new Student(111,"Sita");
          Student s2 = new Student(222,"Ram");
          s1.display();
          s2.display();
         }
}
```

- **Either this or super must be the first statement in constructor.**

Use of "super" keyword in Java.

Super keyword:

- ✓ super can be used to refer super class **variables** as: super.variablename.
- ✓ super can be used to refer super class **methods** as: super.methodname ().
- ✓ super can be used to refer super class **constructor** as: super (values).

Example:

```
class A
{
        int x;
        A (int x)
        {
                this.x = x;
        }
        void show( )
        {
System.out.println("super method: x = "+x);
        }
}
class B extends A
{
        int y;
        B (int x,int y)
        {
                super(x); // (or)this. x=x;
                this.y=y;
        }
        void show( )
        {
        super.show ();
        System.out.println ("y = "+y);
System.out.println (" super x = " + super.x);
        }
}
class SuperUse
{
        public static void main(String args[])
        {
                B ob = new B (10, 24);
                ob.show ( );
        }
}
```

- ✓ Super key word is used in **sub class** only.

Method Overloading with suitable Example.

- If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

- If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading?

- Method overloading **increases the readability of the program.**

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) By changing number of arguments Example

```
class Calculation{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[])
  {
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);
  }
}
```

2) By changing the data type

```
class Calculation2{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(double a,double b){System.out.println(a+b);}


  public static void main(String args[]){
  Calculation2 obj=new Calculation2();
  obj.sum(10.5,10.5);
  obj.sum(20,20);

  }
}
```

Method Overriding with suitable Example.

Method overriding:

✓ If subclass (child class) has the same method as declared in the parent class, it is known as method overriding.

Advantage of Method Overriding:

✓ Method Overriding is used to provide specific implementation of a method that is already provided by its super class.

✓ Method Overriding is used for Runtime Polymorphism.

Rules for Method Overriding:

✓ Method **must have same name** as in the parent class

✓ Method **must have same parameter** as in the parent class.

✓ **Must be** IS-A relationship **(inheritance**).

```
class Animal
{
        void move()
        {
        System.out.println ("Animals can move");
        }
}
class Dog extends Animal
{
        void move()
        {
System.out.println ("Dogs can walk and run");
        }
}
public class TestAnimal
{
        public static void main(String args[])
        {
        Animal a = new Animal ();
        // Animal reference and object
        Animal b = new Dog ();
        // Animal reference but Dog object
        a.move ();
        // Runs the method in Animal class
        b.move ();
        //Runs the method in Dog class
        }
}
```

Garbage Collection in Java
OR Explain use of finalize() method.

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically.

Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

1. By nulling the reference
2. By assigning a reference to another
3. By annonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();

e1=e2;//now the first object referred by
e1 is  eligible for garbage collection
```

3) By annonymous object:

```
new Employee();
```

finalize() method:

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.
- This method is defined in System class as:
  - protected void finalize(){}

gc() method:

- The gc() method is used to invoke the garbage collector to perform cleanup processing.
- The gc() is found in System and Runtime classes.
  - public static void gc(){

Simple Example of Garbage Collection

```
class Simple
{
   public void finalize()
   {
System.out.println("object is garbage collected");
   }

  public static void main(String args[])
  {
          Simple s1=new Simple();
          Simple s2=new Simple();
          s1=null;
          s2=null;
          System.gc();
  }
}
```

Points to Note:

- The Garbage collector of JVM collects only those objects that are created by new keyword.
- So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).
- Garbage collection is performed by a daemon thread called Garbage Collector(GC).
- This thread calls the finalize() method before object is garbage collected.
- Neither finalization nor garbage collection is guaranteed.