



# ***Java Reference Manual***

## **Introduction to history of java and how java works**

---

java is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems.

Java version history

- 1 JDK 1.0 (January 23, 1996)
- 2 JDK 1.1 (February 19, 1997)
- 3 J2SE 1.2 (December 8, 1998)
- 4 J2SE 1.3 (May 8, 2000)
- 5 J2SE 1.4 (February 6, 2002)
- 6 J2SE 5.0 (September 30, 2004)
- 7 Java SE 6 (December 11, 2006)

### **Key Benefits of java:**

1. *Write Once, Run Anywhere (WORA) slogan created by Sun Micro Systems:*

This means Java code can be developed on any device and compiled into a standard byte code and it can run on any device equipped with a Java virtual machine (JVM).

### ***Java Reference Manual – Code Archives KnowledgeBase***

We can develop java code on windows environment and can be execute on any other operating system such as (Linux, UNIX, Mac, sun Solaris etc.) **vice-versa**

#### ***2. Java Byte Code is Platform Independent or Architecture Neutral :***

Byte Code is generated by a Java **compiler** (at compile time) and execution of byte code was taken care by **java virtual machine** at run time.

The main advantage of byte code execution of the program is not restricted to a particular operating system; it can be executed on any operating system

#### ***3. Java is Humble :***

Java is humble because it omits several rarely used and confusing features of C++

- Operator Overloading
- Multiple Inheritance
- No Destructor code needed (added automatic garbage collection)

#### ***4. Java is Robust :***

Robust intention is to code a program by eliminating bugs. So it puts a lot of importance on early checking for eliminating the compile time errors and later dynamic checking for eliminating the runtime exceptions

5. *Network-Savvy :*

Java is useful to write internet programming to share the objects via network

6. *Java is secure :*

Java is enabled to use in Networked/Distributed environments, so security is an important part in Java to construct the virus-free and tamper-free systems

7. *Java is portable :*

Unlike C and C++ the sizes of primitive data types are fixed in java

For Example in C-Lang:

For 16-bit processor size of int is 2 bytes

For 32-bit processor size of int is 4 bytes but in java int size is 4 bytes (fixed)

8. *Java is object-oriented:*

Object orientation makes coding easy for programmer by applying features like inheritance, polymorphism, encapsulation, abstraction, Collection frame work and generics

9. *Java is multithreaded :*

Multithreading concept is useful in real time applications for better interactive and responsiveness for shared resource

***How to Install Java and Set Environment Variables***

Download java development kit from <http://java.sun.com/javase/downloads>

If you interested to download JDK 6

<http://java.sun.com/javase/downloads/widget/jdk6.jsp>

Install java

***Set Environment Variables:***

### ***Java Reference Manual – Code Archives KnowledgeBase***

- 1) Right click on my computer select properties
- 2) Select Advanced tab on System properties window
- 3) Click Environment variables
- 4) Click new button under user variables

Variable name: PATH

Variable value: C:\Program Files\Java\jdk1.6.0\_18\bin

Click ok--ok—ok

### **How Java Works:**

Step 1:

Open notepad and type plain java source code and save file with .java extension

Step 2:

Now compile plain java source code using command **Javac classname.java**, now compiler checks for any errors and wont let you compile until it satisfied

Then compiler creates new document with **classname.class** extension, it contains byte code instructions which can be executed on any operating system

Step 3:

Run the byte code instructions by starting java virtual machine using command

**Java classname**

Now the JVM translates the byte code in to the underlying platform standards and runs the program

**What is the difference between JRE, JDK and JVM?**

**JRE  
(Java Runtime environment)**

It is an implementation of the Java Virtual Machine\* which actually executes Java programs.

Java Run Time Environment is a plug-in needed for running java programs.

JRE is smaller than JDK so it needs less Disk space.

JRE can be downloaded/supported freely from [java.com](http://java.com)

It includes JVM , Core libraries and other additional components to run applications and applets written in Java.

**JDK  
(Java Development Toolkit)**

It is a bundle of software that you can use to develop Java based applications.

Java Development Kit is needed for developing java applications.

JDK needs more Disk space as it contains JRE along with various development tools.

JDK can be downloaded/supported freely from [java.sun.com](http://java.sun.com)

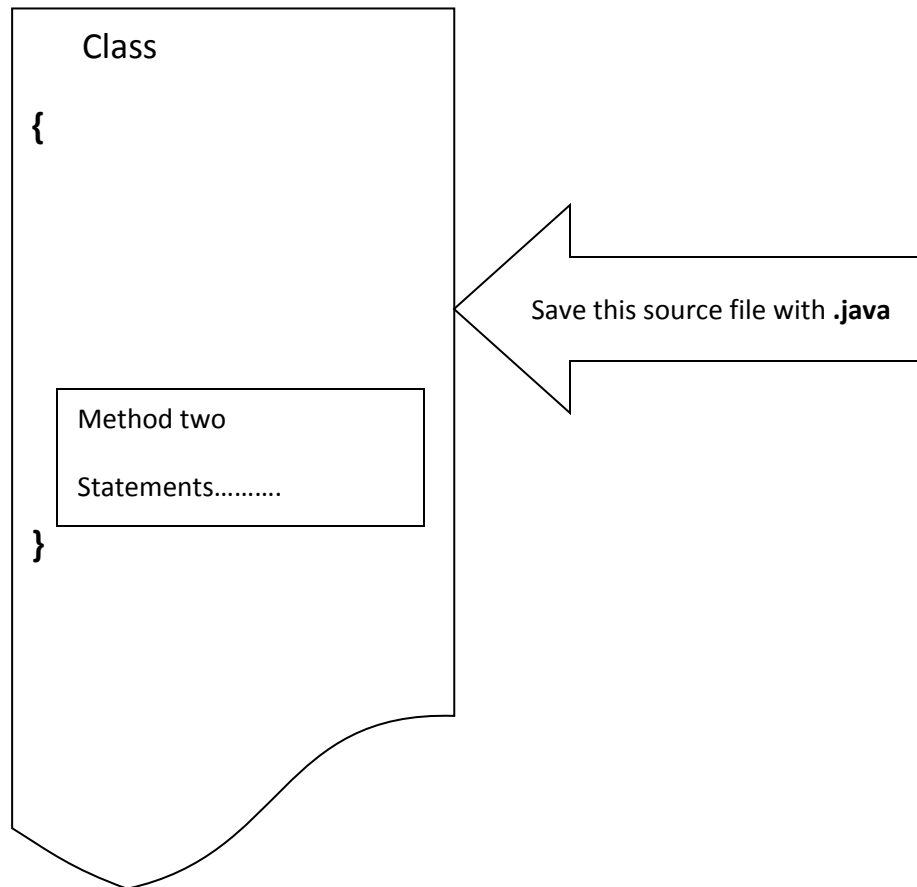
It includes JRE, set of API classes, Java compiler, Web start and additional file

### **Code structure of a java:**

To avoid confusion for java beginners a brief notes about a code structure of java

Package (pack the one or more java files in a single package )

Import statement (for importing packages)





## **What goes inside source file?**

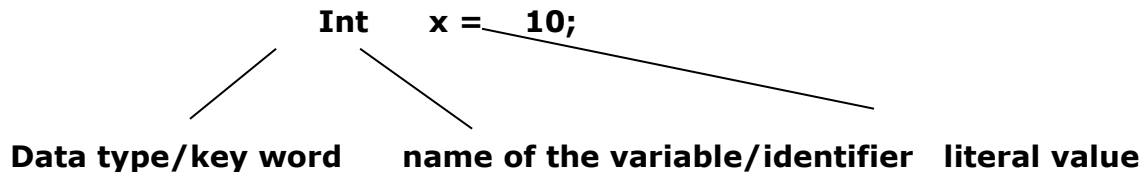
Plain java class

## **What goes inside class?**

A class can have one or more methods (main is also method)

## **What goes inside a method?**

With in curly braces of method write instructions for how method should perform



## **Data types in java:**

For our convenience data types categorized in to 4 types

Integers type:

byte, int, short ,long

floating type:

float, double

boolean type:

boolean

character type:

char

empty set:

void

Data types	Wrapper class	Size (bytes)	Range	Default value
byte	Byte	1	-128 to 127	0
short	Short	2	-32768 to 32767	0
int	Integer	4	$-2^{31}$ to $2^{31}-1$	0
long	Long	8	$-2^{63}$ to $2^{63}-1$	0
float	Float	4	-3.4e38 to 3.4e38	0.0
double	Double	8	-1.7e308 to 1.7e308	0.0
boolean	Boolean	n/a	n/a	false
char	Character	2	0 to 65535	0(blank space)
void	Void	n/a	n/a	n/a

### **Identifier:**

Identifier represents name in java prog

It can be a class name, variable, method name

Note: rules for defining identifiers

- 1) Identifier never starts with digit

## ***Java Reference Manual – Code Archives KnowledgeBase***

- 2) Identifier never a reserve word or keyword
- 3) No special characters are allowed except \$ / \_
- 4) Variable names should not start with underscore \_ or dollar sign \$ characters, even though both are allowed

## **Literals:**

Now we are discussing about literals there are 5 types of literals

1. Integral literal
2. Floating point literal
3. Boolean literal
4. Character literal
5. String literal

## **Integral literal:**

We can specify integral literals in the following 3 ways

Int x=010; //Octal literal (allowed digits are 0 -7)

//literal value prefixed with o

```
Int x=10; //Decimal literal
```

```
Int x=0x10; //Hex decimal literal (allowed digits 0 – 15)
```

```
//literal value prefixed with 0x
```

**Program:**

```
public class Integral    {  
    public static void main(String[] args)  {  
  
        int x=010;  
  
        int x1=10;  
  
        int x2=0x10;  
  
        System.out.println(x+"....."+x1+"....."+x2);  
  
    } }
```

### **Floating point literal :**

Floating point literals are by default double

```
float f=143.44; // error
```

```
float f =143.44f; // compiles
```

```
double d =143.44; // compiles
```

```
double d =143.44d; // compiles
```

note :we can assign integral literals to floating type but we cant assign floating literals to integral type

### **Boolean literals :**

Only allowed boolean literals are true/false

```
boolean b =true;
```

```
boolean b =false;
```

### **Character literals :**

Character literal can be specified as single character with in single quotes

```
char c='a';
```

We can assign integral literals to char data type, this represents the Unicode, but allowed range is 0 - 65535

```
char c=010; //octal literal
```

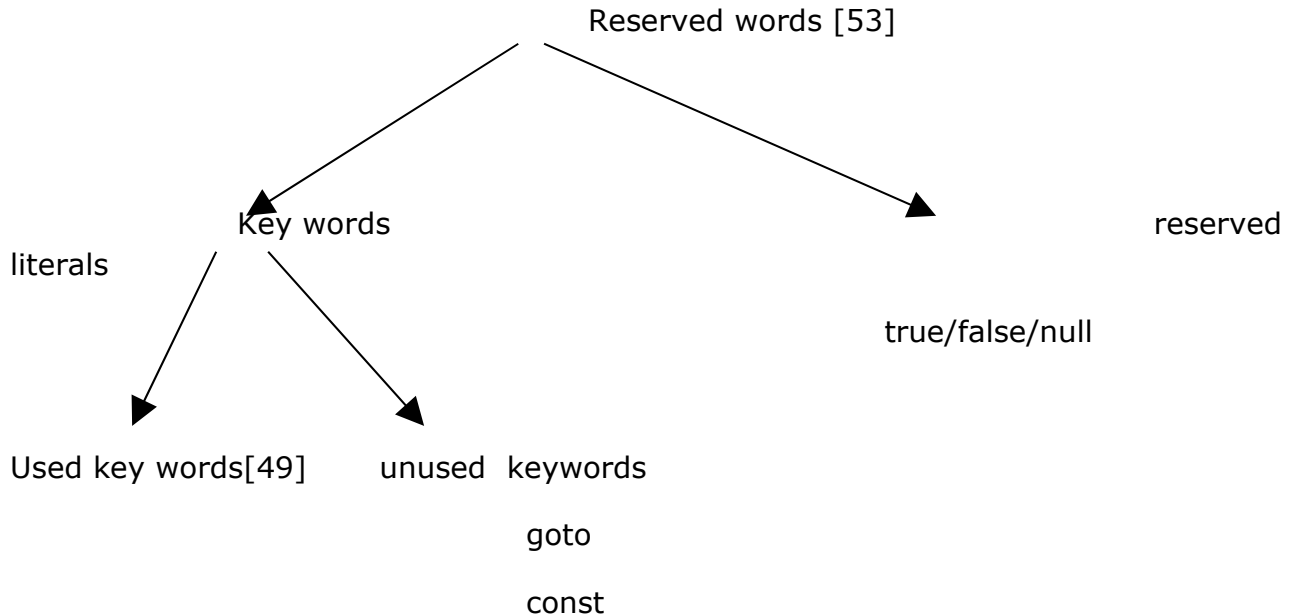
```
char c =97; // decimal
```

```
char c=0xface; //hex
```

### **String literals :**

String literals can be specified as sequence of characters with in the double quotes

```
String str = "haisekhar";
```



### **49 java key words ☹)(java reserved words are lower case)**

#### **Key words from primitive data types (9)**

byte, int, short, long, float, double, boolean, char , void

#### **key words from flow ctrl (11)**

if , else , switch , case , default , for , do , while , break , continue , return

#### **key words from exception handling (6)**

try , catch , finally , throw , throws , assert

### **key words from modifiers (12)**

public , protected , default , private , static , final , transient , volatile , native , strictfp ,  
abstract , synchronized

### **key words for class (6)**

package , import , class , extends , interface , implements

### **object related key words (5)**

new , instanceof , super , this, enum

### **Summary of Operators**

The following quick reference summarizes the operators supported by the Java programming language.

#### **Simple Assignment Operator**

= Simple assignment operator

#### **Arithmetic Operators**

+ Additive operator (also used for String concatenation)

- Subtraction operator

\* Multiplication operator

/ Division operator

% Remainder operator

#### **Unary Operators**

+ Unary plus operator; indicates positive value (numbers are positive without this, however)

- Unary minus operator; negates an expression

++ Increment operator; increments a value by 1

-- Decrement operator; decrements a value by 1

! Logical compliment operator; inverts the value of a boolean

#### **Equality and Relational Operators**

== Equal to

!= Not equal to



## ***Java Reference Manual – Code Archives KnowledgeBase***

> Greater than  
>= Greater than or equal to  
< Less than  
<= Less than or equal to

### **Conditional Operators**

&& Conditional-AND  
|| Conditional-OR  
?: Ternary (shorthand for if-then-else statement)

### **Bitwise and Bit Shift Operators**

~ Unary bitwise complement  
<< Signed left shift  
>> Signed right shift  
>>> Unsigned right shift  
& Bitwise AND  
^ Bitwise exclusive OR  
| Bitwise inclusive OR

## **The Simple Assignment Operator**

One of the most common operators that you'll encounter is the simple assignment operator "=". **It assigns the value on its right to the operand on its left:**

```
int cadence = 0;
```

This operator can also be used on objects to assign *object references*, as discussed in [Creating Objects](#).

## **The Arithmetic Operators**

The Java programming language provides operators that perform addition, subtraction, multiplication, and division.

### ***Java Reference Manual – Code Archives KnowledgeBase***

+        additive operator (also used for String concatenation)  
-        subtraction operator  
\*        multiplication operator  
/        division operator  
%        remainder operator

### **Sample program :**

```
class ArithmeticDemo {  
  
    public static void main (String[] read){  
  
        int result = 1 + 2; // result is now 3  
        System.out.println (result);  
  
        result = result - 1; // result is now 2  
        System.out.println(result);  
  
        result = result * 2; // result is now 4  
        System.out.println(result);  
  
        result = result / 2; // result is now 2  
        System.out.println(result);  
  
        result = result + 8; // result is now 10  
        result = result % 7; // result is now 3  
        System.out.println(result);  
  
    }  
}
```

You can also **combine the arithmetic operators with the simple assignment operator to create compound assignments**. For example, `x+=1;` and `x=x+1;` both increment the value of `x` by 1.

The **+** operator can also be used for concatenating (joining) two strings together

```
class ConcatDemo {  
    public static void main(String[] ping){  
        String firstString = "This is";  
        String secondString = " a concatenated string.";  
  
        String thirdString = firstString+secondString;  
  
        System.out.println(thirdString);  
    }  
}
```

By the end of this program, the variable thirdString contains "This is a concatenated string.", which gets printed to standard output.

## **The Unary Operators**

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

- +       Unary plus operator; indicates positive value (numbers are positive without this, however)
- Unary minus operator; negates an expression
- ++      Increment operator; increments a value by 1

--      Decrement operator; decrements a value by 1  
!        Logical complement operator; inverts the value of a boolean

### **Program on inc / dec operators**

```
public class Sample {  
    public static void main(String[] yahoo) {  
        int x=1;  
        int j=1;  
        int k=++x; //pre inc  
        System.out.println(k);//2  
        int f=j++; //post inc  
        System.out.println(f);//1  
    }  
}
```

### **The Equality and Relational Operators**

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

==	equal to	} Equality operators
!=	not equal to	

>	greater than	} Relational operators
>=	greater than or equal to	
<	less than	
<=	less than or equal to	

The following program, [Comparison Demo](#), tests the comparison operators:

```
class ComparisonDemo {
```

### ***Java Reference Manual – Code Archives KnowledgeBase***

```
public static void main(String[] args){  
    int value1 = 1;  
    int value2 = 2;  
    if (value1 == value2) System.out.println("value1 == value2");  
    if(value1 != value2) System.out.println("value1 != value2");  
    if (value1 > value2) System.out.println("value1 > value2");  
    if(value1 < value2) System.out.println("value1 < value2");  
    if(value1 <= value2) System.out.println("value1 <= value2");  
}  
}
```

Output:

```
value1! = value2  
value1 < value2  
value1 <= value2
```

### **Logical operators**

& -logical AND operator

| -logical OR operator

### **The Conditional Operators OR Short –Circuit operators**

The && and || operators perform *Conditional-AND* and *Conditional-OR* operations on two Boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

&& Conditional-AND

|| Conditional-OR

The following program tests these operators:

```
public class Sample {  
    public static void main(String[] args) {  
        int x= 10;
```

### ***Java Reference Manual – Code Archives KnowledgeBase***

```
int y=10;
if(++x>10 & ++y<10){
x++;
}
else {
y++;
}
System.out.println(x+","+y);
}
}
```

<b>&amp;</b> 11, 12	<b> </b> 12 ,11
<b>&amp;&amp;</b> 11 ,12	<b>  </b> 12, 10

### **Another conditional operator is ?:,which is also called ternary operator**

Syntax:

Variable =    condition? value1: value2

How it works?

If the condition returns true, value1 assign to variable

If the condition returns false, value2 assign to variable

### Sample program on ternary operator

```
class Ternary
{
    public static void main(String[] args)
    {
        int a=10;
```

## Java Reference Manual – Code Archives KnowledgeBase

```
int b=20;
int max;

max=(a>b)?a:b;
System.out.println(max);
}
}
```

### Bitwise and Bit Shift Operators:

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

#### The bitwise operators

Operator	Name	Example	Result	Description
$a \& b$	and	$3 \& 5$	1	1 if both bits are 1.
$a   b$	or	$3   5$	7	1 if either bit is 1.
$a \wedge b$	xor	$3 \wedge 5$	6	1 if both bits are different.
$\sim a$	not	$\sim 3$	-4	Inverts the bits.
$n \ll p$	left shift	$3 \ll 2$	12	Shifts the bits of $n$ left $p$ positions. Zero bits are shifted into the low-order positions.
$n \gg p$	right shift	$5 \gg 2$	1	Shifts the bits of $n$ right $p$ positions. If $n$ is a 2's complement signed number, the sign bit is shifted into the high-order positions.
$n \ggg p$	right shift	$-4 \ggg 28$	15	Shifts the bits of $n$ right $p$ positions. Zeros are shifted into the high-order positions.

## **Expressions, Statements, and Blocks**

Now that you understand data types, variables and operators, it's time to learn about *expressions*, *statements*, and *blocks*.

### Expressions

An *expression* is a construct made up of variables, operators, and method invocations

(\*) Multiplication operator (/) Division operator (%) Remainder operator will have same priority

(+) addition (-) minus operator both will have same priority

10\*20/20;  
10/20\*20; Arithmetic expression evaluates from left to right  
10+20\*10/2;

```
class Expressions
{
    public static void main(String[] args)
    {
        int x=10;
        System.out.println(10*20/20);
        System.out.println(10/20*20);
        System.out.println(10+20*10/2);
    }
}
```

Output:

10  
40



## **Statements**

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

## **Block**

A *block* is a group of zero or more statements between balanced braces

```
class BlockDemo {  
    public static void main(String[] args) {  
        boolean condition = true;  
        if (condition) { // begin block 1  
            System.out.println("Condition is true.");  
        } // end block one  
        else { // begin block 2  
            System.out.println("Condition is false.");  
        } // end block 2  
    }  
}
```

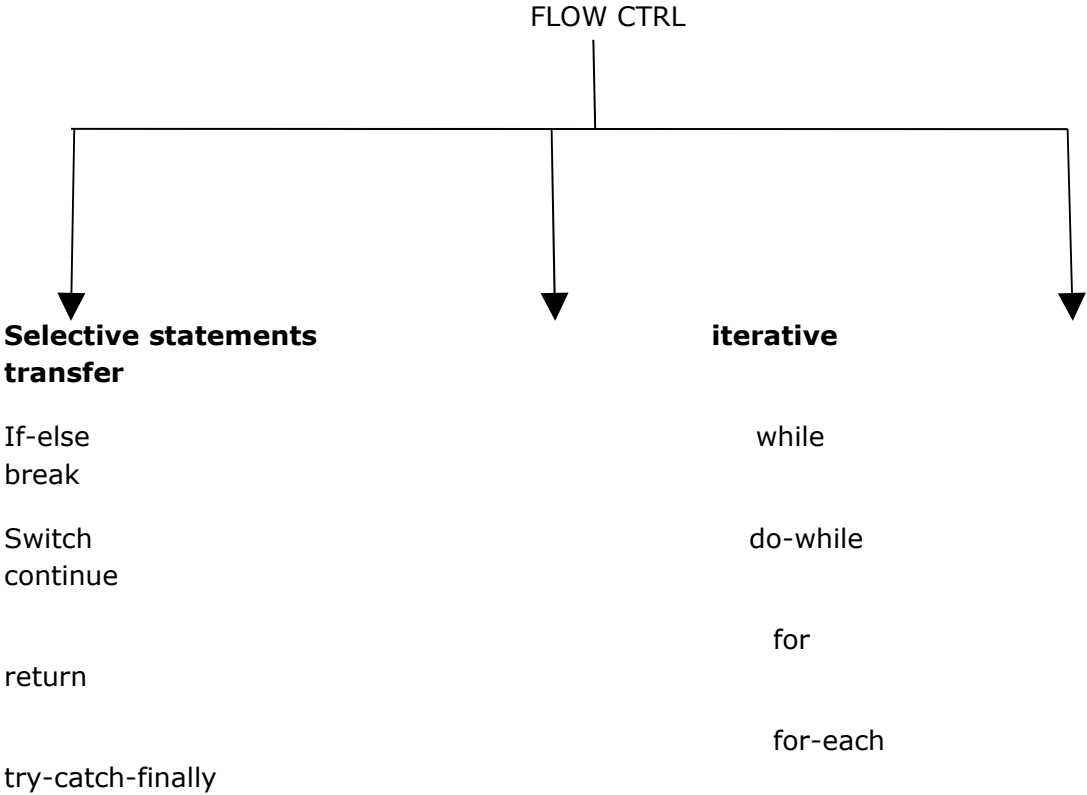
**Java naming conventions:**

Identifier Type	Rules for Naming	Examples
Packages	package name is always written in all-lowercase ASCII letters	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Classes	Class names should be nouns, In normal case first letter should be capital letter, In mixed case first letter of each internal word capitalized.	Class Raster class ImageSprite
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;
Methods	Methods should be verbs; In normal case first letter should start with small letter, In mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run(); runFast(); getBackground();
Variables	Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed.	int i; char c; float myWidth;

constants	Constant names should be in capital letters	final int MAX=10;
-----------	---	-------------------

Flow control

---



## Selective Statements:

### **The if Statement**

Syntax:

```
        If (condition)
        {
            Statements;
        }
```

If the condition returns **true** statements in the block will execute, if condition returns false control jumps out of the block and remaining program will execute normally

```
class SelectIf
{
    public static void main(String[] args)
    {
        int a=10;
        int b=4;
        if (a>b)//if condition true block will execute otherwise ctrl jumps out of the block
        {
            System.out.println(a);
            int x=9;
            System.out.println(x);
        }

        System.out.println("hai hello");
    }
}
```

### **The If-else statement**

Syntax:

```
        If (condition)
        {
            Statement;
```

```
    }  
    else  
    {  
        Statement;  
    }
```

If conditions are go on increasing than prefer if – else if selective statements

```
class IfElseDemo {  
    public static void main(String[] args) {  
  
        int testscore = 76;  
        char grade;  
  
        if (testscore >= 90) {  
            grade = 'A';  
        } else if (testscore >= 80) {  
            grade = 'B';  
        } else if (testscore >= 70) {  
            grade = 'C';  
        } else if (testscore >= 60) {  
            grade = 'D';  
        } else {  
            grade = 'F';  
        }  
        System.out.println("Grade = " + grade);  
    }  
}
```

### **The switch Statement:**

A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types*

Syntax:

switch (any primitive data type)

```
{
    Case 1:
        Statement;
        break;
    Case 2:
        Statement;
        break;
    default :
        default statement;
}
```

### **Sample program on switch statement**

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

        int month = 8;
        switch (month) {
            case 1: System.out.println("January"); break;
            case 2: System.out.println("February"); break;
            case 3: System.out.println("March"); break;
            case 4: System.out.println("April"); break;
            case 5: System.out.println("May"); break;
            case 6: System.out.println("June"); break;
            case 7: System.out.println("July"); break;
            case 8: System.out.println("August"); break;
            case 9: System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month.");break;
        }
    }
}
```

### **The while and do-while Statements**

## ***Java Reference Manual – Code Archives KnowledgeBase***

The while loop continually executes a block of statements until condition is true. If condition is false control jumps out of the while loop

syntax :

```
while (expression) {  
    statement(s)  
}
```

If condition fails first time, than it returns false, so no chance of execution of statements in while loop,

But when comes to do – while loop, statements will execute at least once than checks the condition is true or false

### **Sample program on while loop print top ten numbers**

```
class TopTenNum  
{  
    public static void main(String[] args)  
    {  
        int i=0;  
  
        while(i<=10)  
        {  
            System.out.println(i);  
            ++i;  
        }  
    }  
}
```

The Java programming language also provides a do-while statement

The difference between **do-while** and **while** is that do-while checks the expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once

Syntax:

```
do {  
    statement(s)
```

```
}
```

```
while (expression);
```

### **Sample program on do-while loop print top ten numbers**

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

### **The for loop :**

Syntax:

```
for(initialization section , conditional section , inc / dec)  
{  
    Statements;  
}
```

**Initialization section:** this part executes only once

There is no way to declare different variable types in initialization section



```
Int i=10,j=20; // compiles fine
```

```
Int i=10,int j=20; // error
```

```
Int i=10,byte j=20; //error
```

**Conditional section:** it can be any valid expression which condition should return boolean type. the condition expression is optional default value is true

**Inc / dec** : in this section we use increment and decrement operators

### **Sample program on for loop print top ten numbers**

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

### **Sample program on for loop**

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

## ***Java Reference Manual – Code Archives KnowledgeBase***

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

```
for(int i=0;i<=3;i++)  
{  
    System.out.println(array[i]);  
}
```

```
}
```

```
}
```

### **For each loop:**

It has introduced in 1.5 version it is most convenient loop for retrieving the elements from arrays or collections.

```
int[] array={10,20,30,40};  
for (int x:array)  
{  
    System.out.println(x);  
}
```

### **Transfer statements:**

Break statement:

We can use break statements in the following three cases

1. Inside the loops to break the loop, based on some condition
2. Inside the switch to stop fall through
3. Inside labeled block, to come out of the block based on some condition

If we use any were else we will get compile time error

### **Sample program on break statement leads to an error**

```
class TransferBreak
{
    public static void main(String[] args)
    {
        int a=10;
        int b=20;
        if(a<b)
        {
            System.out.println("Hello World!");
            break;
        }
    }
}
```

### **Sample program on break statement in for loop**

```
class TransferBreakInForLoop
{
    public static void main(String[] args)
    {
        for(int i=0;i<=10;i++)
        {
            System.out.println(i);
            if(i==9)
                break;
        }
        System.out.println("executes the remaining program");
    }
}
```

### **Continue statement:**

## ***Java Reference Manual – Code Archives KnowledgeBase***

We should use continue statement only in side the loop, to skip the current iteration and continue for next iteration , if we are using continue out side the loop we will get compile time error

### **Sample program**

```
class TransferContinue
{
    public static void main(String[] args)
    {
        for(int i=0;i<3;i++)
        {
            for (int j=0;j<3;j++ )
            {
                if(i==j)
                    continue;

                System.out.println(i+" "+j);
            }
        }
    }
}
```

### **Sample program using lable block**

```
class TransferContinue1
{
    public static void main(String[] args)
    {
        l1:for(int i=0;i<3;i++)
        {
            for (int j=0;j<3;j++ )
            {
                if(i==j)
                    continue l1;
            }
        }
    }
}
```

```
        System.out.println(i+" "+j);
    }

}

}

}
```

## **Arrays :**

An array is also an object in java used to store homogenous (same type) elements in an indexed format.

The first element always indexed at 0 and last element is indexed at n-1.

The size of an array is fixed, it cannot grow dynamically at runtime in java, and to overcome this problem java people introduced collections

### **Why array is needed?**

To store similar **type** of values for large number of data items

### **What is type?**

Here type means either primitive type or reference type.

type [] ref;

Example:

```
Int [] a = {10, 20, 30, 40};
```

```
String [] str = {"hai","hello"};
```

```
Bike [] bk= {new Bike ("Bajaj"), new Bike ("TVs")};
```

**Array declaration**

```
type[] ref;
```

Example:

```
double[] a;
```

Or

```
double a[];
```

**Array construction or allocating memory**

The array can be constructed by new operator at runtime

Syntax:

```
ref = new type[size];
```

example:

```
d = new Double[3];
```

Note : while constructing an array we have to specify the size of an array other wise compile time error

**Initializing an Array**

```
d[0]=20.5;
```

```
d[1]=30.5;
```

**Declaring, Constructing and Initialization of an array in a single line**

```
Type[] ref = {array elements};
```

## ***Java Reference Manual – Code Archives KnowledgeBase***

Example;

```
Int[] I = {10,20,30};//
```

```
Type[] ref = new type[]{10,20,30};
```

### **Arrays are categorized in two parts**

Single dimensional arrays

Multi dimensional arrays

### **sample program to create single-D array and read elements from array and display**

```
class SingleDArray
{
    public static void main(String[] args)
    {
        String[] str={"i","luv","india"};

        for(int i=0;i<=2;i++)
        {
            System.out.println(str[i]);
        }
    }
}
```

### **Sample program on single dimensional Array**

```
class SingleDArray1
{
    public static void main(String[] args)
    {
        String[] a = new String[3];
        a[0]="i";
        a[1]="luv";
        a[2]="india";

        for(String x:a)
```

```
        {
            System.out.println(x);
        }
    }
}
```

### **Multi dimensional arrays :**

Multi dimensional arrays represent 2D, 3D ..... Arrays

#### **Two dimensional arrays:**

Creation of two dimensional arrays

int [][] a={{10,20} {30,40}};//constructing , initializing an 2-D array in single line

```
int[][] a ;//array declaration
a=new int[2][2];//array construction
```

```
a[0][0]=10;//assign values
a[0][1]=20;
a[1][0]=30;
a[1][1]=40;
```

#### **sample program on 2-D Array**

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

        for(int[] subarr:a)
        {
```



```
        for(int x : subarr)
        {
            System.out.println(x);    } } } }
```

### **sample program on 2-D Array**

```
class TwoDArray1
{
    public static void main(String[] args)
    {

        int[][] a = new int[2][2];

        a[0][0]=10;
        a[0][1]=20;
        a[1][0]=30;
        a[1][1]=40;

        for(int[] subarr:a)
        {
            for(int x : subarr)
            {
                System.out.println(x);
            }
        }
    }
}
```

**Note :** two dimensional array handled by 2 loops, similarly , three dimensional array handled by 3 loops

**how to know the size of an array:**

## ***Java Reference Manual – Code Archives KnowledgeBase***

If we want to know the size of an array , we can use property 'length'

Example:

```
Int[] a = new int[5];  
    a[0]=10;  
    a[1]=20;  
    a[2]=30;
```

Now **a.length** gives array size 5, **length** property does not give number of elements in array ,it gives only its size

## **Variables**

---

Variables which are useful to store the values, what type of values variables are going to store

Particular data type values and reference values of objects

***Primitive variables***: variable which are useful to store particular data type values are called primitive variables

**Int i=10;**

***Reference variables***: variable which are useful to store reference value of objects

**Book b = new Book ("head first java");**

Depends up on **scope** and **lifetime** the variables are divided in to 3 types

- Instance variables
- Static variables
- Local variables

Lifetime of a variable means HOW long a variable is available. Scope of variable means from WHERE can we access

### **Instance variables**

If the value of the variable is varied from object to object such variables are called instance variables

For every object separate copy of instance variable will be created

Instance variables are created at time of object creation and will destroy at time of object destruction

For instance variables it is not required to perform explicitly initialization, jvm will always provides default values

```
class Instance
{
    int i;//instance variable
    public static void main(String[] args)
    {
        Test t = new Test();
        t.i++;
        System.out.println(t.i);//ouput 1

        Test t1 = new Test();
        System.out.println(t1.i);// out put 0
    }
}
```

### **Static variables**

If the value of variable is common for all objects , than it is not to recommended to declare such variable at object level, we have to declare such type of variable at class level by using static keyword

### ***Java Reference Manual – Code Archives KnowledgeBase***

In the instance variable for every object a separate copy will be created, but in the case of static variable a single copy will create at class level and shared by all objects of that class

Static variable we can declare with in the class directly by using static keyword

Static variable will create at time of class loading and destroy at the time of class unloading

For static variables it is not required to perform explicitly initialization, jvm will always provides default values

```
class Static
{
    int i;//instance variable
    static int j; // static variable
    public static void main(String[] args)
    {
        Static s = new Static();
        //j++;
        System.out.println(Static.j);

        Static s1 = new Static();
        System.out.println(Static.j);

    }
}
```

### **Local variables**

The variables which are declared in side a block are called local variables; the different blocks are method, constructor, if, else, switch, for, while, do-while, try, catch, finally and etc, the local variables are also known as automatic, temporary and stack variables

For local variables jvm wont provide any default values, the programmer is responsible to provide initialization

### ***Java Reference Manual – Code Archives KnowledgeBase***

Before using local variables, it is compulsory we should perform initialization, other wise compile time error

```
class Local
{
    public static void main(String[] args)
    {
        int i;
        System.out.println(i);//compile time error
    }
}
```

### **Command line args**

The arguments which are passing from command prompt are called command line arguments and these are arguments to main method

args[0] ----- first command line argument

args[1] ----- second command line argument

args.length no of command line arguments

Java test A B C

args.length // 3

```
class CMD
{
    public static void main(String[] args)
    {
        System.out.println(args.length);

        for(int i =0; i<args.length;i++)
```

```
        {  
            System.out.println(args[i]);  
        }  
    }  
}
```

### **String, StringBuilder and StringBuffer**

String, StringBuilder and stringBuffer are classes in a java.lang package

#### **String class**

Strings are designed to represent text

```
String s="text";
```

There are three ways to create strings in java

1. We can create string just by assigning a group of characters to string type variable

```
String str="hello";
```

2. We can create by using new operator

```
String str = new String("hello");
```

3. The third way to create strings is by converting character array into strings

```
Char[] ch={'h','e','l','l','o'};
```

**String str = new String(ch);**

*Sample program*

```
class StringSample
{
    public static void main(String[] args)
    {
        String str="hello";
        System.out.println(str);

        String str1= new String("hello");
        System.out.println(str1);

        char[] ch ={'h','e','l','l','o'};
        String str2 = new String(ch);
        System.out.println(str1);

    }
}
```

**String Comparison**

**String comparison using == operator**

```
class StringSample
{
    public static void main(String[ ] args)
    {
        String str="hello";
        System.out.println(str);//hello

        String str1= new String("hello");
        System.out.println(str1);//hello
    }
}
```

```
        System.out.println(str==str1);//false
    }
}
```

In the above program both contents are same but == operator returns false, because when we are using == operator it will not compare the contents, it will compare the memory address of the object as hexadecimal which is called object reference

### **String comparison using equals () method**

```
If(str.equals(str1))
System.out.println("both are same");
else
System.out.println("not same");
```

Note: In String class equals () method is overridden for content comparison

### **Immutability of strings**

Once we create a string object we are not allowed to perform any modifications on the existing object, if we are trying to perform any changes with those changes new string object will be created, this behavior is called immutability

```
String s1="hai";
String s2="hello";
S1=s1+s2;
System.out.println(s1);//haihello
```



**concat() method:**

```
String str = new String("hai");
```

```
Str.concat("hello");
```

```
System.out.println(str);//hai
```

**Methods in String class:**

**1) Public char charAt(int index)**

```
Ex: String str="aspire";
```

```
System.out.println(str.charAt(2));//p
```

**2) Public String concat(String s)**

```
Ex: String s="aspire";
```

```
s.concat("tech");//new object created aspiretech
```

```
System.out.println(s); //aspire
```

**3) Public boolean equals(object obj)**

For content comparison including case sensitive

**4) Public boolean equalsIgnoreCase(String s)**

```
Ex:
```

```
String s="shiva";
```

```
System.out.println(s.equals("sHiva"));//false
```

```
System.out.println(s.equalsIgnoreCase("sHiva"));//true
```

### **5) Public int length()**

```
String s="shiva";
```

```
s.o.p(s.length());//5
```

```
s.o.p(s.length());//error
```

### **6) Public String replace(char old , char new)**

ex:

```
String s="ababa";
```

```
s.o.p(s.replace('a','b'));//bbbbbb
```

### **7) Public String substring(int begin)**

Ex;

```
String s="howrU";
```

```
s.o.p(s.substring(2));//wrU
```

### **8) Public String substring(int begin , int end)**

Returns begin to end-1

```
String str="howrUfriend";
```

```
System.out.println(str.substring(2,6));//wrUf
```

### **9) Public String toLowerCase()**

**10)        Public string toUpperCase()**

**11)        Public String trim()**

To remove the blank space present at beginning and at end of the string, but not blank spaces present in the middle

**12)        Public int indexOf(char ch)**

Returns index of first occurrence of specified character

```
String str="howrUfriend";
```

```
System.out.println(str.indexOf('r'));//3
```

**13)        Public int lastIndexOf(char ch)**

Returns index of last occurrence of specified character

```
String str="howrUfriend";
```

```
System.out.println(str.lastIndexOf('r'));//6
```

**StringBuffer class**

The main difference between String class and StringBuffer class is, String Class is immutable and StringBuffer class is mutable

If we perform any modifications on existing StringBuffer object , those changes are applicable on the existing object without creating a new StringBuffer object

There are two ways to create the StringBuffer Objects

**StringBuffer sb = new StringBuffer ("hai");//creating stringBuffer object and storing Characters**

**System.out.println(sb);//hai**

**System.out.println(sb.length());//3**

**StringBuffer sb = new StringBuffer();**

Here we are creating StringBuffer object as an empty object and not passing any String to it, in this case, a StringBuffer object will be created with a default capacity 16 characters.

**StringBuffer sb = new stringBuffer(50);**

Here we are creating StringBuffer object as an empty object with capacity of 50 characters,

Of course even we declare 50 characters we can add more than 50 characters because StringBuffer is immutable and grows dynamically

To add characters we use append() method as

**sb.append("hai");**

**System.out.println(sb.length());//3**

**sb.append("abcdefghijklmn");**

**System.out.println(sb.length());//17**

Note: append() method is not available in String class

### **Some important methods in StringBuffer class**

### **1) Public int capacity()**

This method returns capacity if StringBuffer object

```
StringBuffer sb = new StringBuffer();
```

```
System.out.println(sb.capacity());//16
```

### **2) Public StringBuffer append(type var)**

### **3) Public StringBuffer insert(int index , String str)**

Ex:

```
StringBuffer sb = new StringBuffer("abcdefgh");
```

```
Sb.insert(3,"0123");
```

```
System.out.println(sb);//abc0123defgh
```

### **4) Public StringBuffer delete(int start , int end)**

For deleting characters from start index to (end-1) index

```
StringBuffer sb = new StringBuffer();//initial capacity is 16
```

```
sb.append("howrufriend");
```

```
System.out.println(sb.delete(2,5));
```

### **5) Public StringBuffer deleteCharAt(int index)**

For deleting a character at specified index

### **6) Public StringBuffer reverse ()**

```
StringBuffer sb = new StringBuffer("aspire");
```

```
System.out.println(sb.reverse());//eripsa
```

## **7) Public StringBuffer replace(int start , int end , String str)**

```
StringBuffer sb1 = new StringBuffer("haihowru");
```

```
System.out.println(sb1.replace(3,7,"hello")); //haihellou
```

The main problem with StringBuffer is all of its methods are synchronized, hence at time only one thread is allowed to perform any operation on the StringBuffer object, hence performance of the system is going down but we will get thread safety

If the performance is important instead of thread safety we should go for StringBuider

## **Object-Oriented Programming**

---

### **What is a class?**

A Class is a user defined data type which contains the variables, properties and methods in it

A Class is a blueprint or template from which an object is derived

Note: class wont exists physically in RAM, just occupies a place in hard disk

### **What is an Object?**

Object is also named as instance; an instance is an executable copy of a class which can be created at run time by help of new keyword

Note: object exists physically in heap memory in JVM which runs in RAM, object creation is very costly unnecessary creation of objects leads to wastage of system resources

### **Concepts of OOP:**

- ***Encapsulation (Data hiding & Method abstraction)***
- ***Inheritance (IS-A Relationship)***
- ***HAS-A Relationship***
- ***Polymorphism***
- ***Method Signature***
- ***Overloading***
- ***Overriding***
- ***Member Hiding (Both Data and Method)***
- ***Keywords: this or super***
- ***Constructors***
- ***Abstract class***
- ***Interfaces***

**Encapsulation:** it is a protective mechanism in java

**Data hiding:** One of the important object-oriented techniques is **hiding the data**

What is the scope of the hiding data?

Hiding the data within the class, it means we can't access the data outside the class directly

How we are hiding the data?

We are hiding the data by declaring variables with private modifier, so we can't access those variables outside the class directly, but we can access those variables outside the class by help of public methods

**Example:**

```
class Test
{
    //data hiding
    private int i=10;
```



```
public int get()
```

```
{
```

```
    return i;
```

```
}
```

```
public void set(int j)
```

```
{
```

```
    i=j;
```

```
}
```

```
}
```

```
class Encap1
```

```
{
```

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

```
    {
```

```
        Test t = new Test();
```

```
        System.out.println(t.i);//compile time error

        t.set(12);

        System.out.println(t.get());//accessing thru public methods

    }

}
```

## ***Method Abstraction***

Hiding the internal implementation details is called as an abstraction.

### **Example:**

```
Public class DriverManager{
    //abstraction
    public static Connection getConnection(String url, String name, String
password){

    }//method
}
```

***Encapsulation = Data hiding + Method Abstraction;***

Inheritance is a concept defines common methods in base class and specific methods in subclass.

**Sub class** has the capability to use the properties and methods of **super class** while adding its own functionality.

The base class is also called as super class, super type, or parent class. The subclass is also called as subtype, derived class, or child class. In java, **extends** keyword is used to inherit a super class

**Example:**

```
class Figure{  
    private int width;  
    private int height;  
    public Figure(){}  
    public Figure(int w, int h){  
        width = w;
```

```
        height = h;

    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

}

class Rectangle extends Figure{
    public Rectangle(int w, int h){
        super(w,h);
    }
    //subclass specific method.
    void area(){
        int a = getHeight()* getWidth();
        System.out.println("Area of the rectangle is:"+a);
    }
}
```

```
class Triangle extends Figure{  
    public Triangle(int w, int h){  
        super(w,h);  
    }  
    //subclass specific method.  
    void area(){  
        double a = 0.5 * getHeight()* getWidth();  
        System.out.println("Area of the triangle is:"+a);  
    }  
}
```

**Example:**

```
class Rectangle extends Figure{}
```

In java, a class can inherit at most one super class. This is called single level inheritance

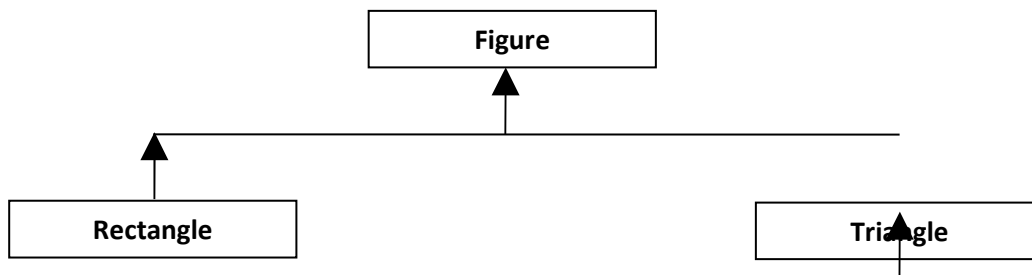


Figure 1: Inheritance hierarchy

Subclass specific methods cannot be invoked using superclass reference variable because the compiler only knows about the reference variable type.

**Example:**

```
Figure fRef = new Figure();  
fRef.area();           //Compilation error. Undefined method.
```

**Example:**

```
Rectangle rRef = new Rectangle(10,20);  
rRef.area();           //Compiles and runs without errors.
```

**Example:**

Figure fRef = new Rectangle(); //Reference type widening. Subtype object is assigned to supertype

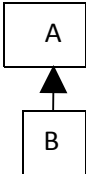
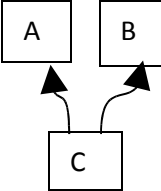
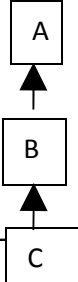
```
fRef.area();           // Compilation error. Undefined method.
```

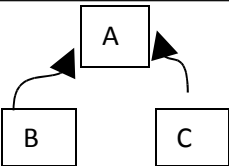
**Example:**

Figure fref = new Rectangle();

fref.getWidth(); //compiles fine

## Types of inheritance

<u>level</u>	<u>representatio</u> <u>n</u>	<u>In c+</u> <u>±</u>	<u>In java</u>	<u>Java statements</u>
<b>single</b>	 <pre>graph BT; B --&gt; A</pre>	yes	yes	Class A { } Class B extends_A { }
<b>Multiple</b>	 <pre>graph BT; C --&gt; A; C --&gt; B</pre>	yes	no	Class A { } Class B { } Class C extends A, extends B { }
<b>Multi level</b>	 <pre>graph BT; C --&gt; B; B --&gt; A</pre>	yes	yes	Class A { } Class B extends A{ } Class C extends B{ }

<b>hierarchal</b>		yes	yes	Class A{ } Class B extends A{ } Class C extends A{ }
<b>hybrid</b>		yes	no	

**this keyword**

In java **this** keyword represents the current object



class Student

```
{  
  
    int i;  
  
    Student(int i)  
    {  
  
        this.i=i;  
  
    }  
  
  
    public static void main(String[] args)  
    {  
  
  
        Student t = new Student(10);  
        System.out.println(t.i);  
  
        Student t1 = new Student(20);  
        System.out.println(t1.i);  
  
    }  
}
```

```
}
```

## Method Signature

It is the combination of the method name and its parameters list. Method signature does not include return type and its modifiers.

### Example:

```
public void add(int, int){  
    }  
public float add(float, float){  
    }
```

**Note: Two methods with same signature will not be allowed with in the class.**

### Example:

```
public void add(int, int) {  
    }  
public int add(int, int) { } //Compilation error. Duplicate method add ()
```

## Method overloading

Two or more methods are said to be overloaded if and only if methods with same name but differs in the number of parameters, parameter type, or parameter order.

```
public void m1() { }
```

```
public void m1(int i) { }
```

```
public void m1(int i, float f) { }
```

**Example program:**

```
class Sample
{
    public void m1()
    {
        System.out.println("no arg");
    }

    public void m1(int i)
    {
        System.out.println("int arg");
    }
}
```

```
    }

    public void m1(int i, float f)
    {
System.out.println("int and float arg");
    }
}

class Oload
{
    public static void main(String[] args)
    {
        Sample t = new Sample();
        t.m1();
        t.m1(10);
        t.m1(10,10.5f);
    }
}
```

### ***Overload methods with varargs Resolution***

### ***Java Reference Manual – Code Archives KnowledgeBase***

The overloaded methods with varargs parameter is always gets least priority.

Example:

```
public class OverloadMethodVarargsResl {  
    static void meth(int i){  
        System.out.println("non-vararg method...");  
    }  
    static void meth(int... vararg){  
        System.out.println("vararg method...");  
    }  
    public static void main(String[] args) {  
        //The below method maps with meth(int)  
        meth(1);  
        //The below method maps with meth(int...)  
        meth(1,2);  
    }  
}
```

O/P:

non-vararg method...

vararg method...

**Note:** overloading method resolution is always take care by compiler based on **ref types** and **arguments** overloading is also called as '**Static Polymorphism**' or '**Early Binding**' or **compile time polymorphism**

## Method Overriding

What ever the parent have default available to the child thru inheritance , if child not satisfied with parent class implementation than child is allowed to override the method with its own implementation this concept is called method overriding

```
class Parent
{
    public void marry()
    {
        System.out.println("iileyana");
    }
}

class Child extends Parent
{

```

```
    public void marry()
    {
System.out.println("anushka");
    }
}

class Oride
{

    public static void main(String[] args)
    {
        Parent p = new Parent();
        p.marry();//illeyana

        Child c = new Child();
        c.marry();//anushka

        Parent p1 = new Child();
        p.marry();//illeyana
    }
}
```

```
}
```

## **Rules for overriding**

- Method name and arguments must be same i.e. signature of methods must be same
- Method Return types must be same

```
Class P {  
    Public void m1()  
{  
}  
}
```

```
Class C extends P {  
    Private int m1()  
{  
}  
}
```



***Java Reference Manual – Code Archives KnowledgeBase***

- While overriding we are not allowed to decrease the scope of access modifier , but we can increase

```
Class P {  
    Public void m1(){}  
}
```

```
Class C extends P {  
    Private void m1(){} //compile time error  
}
```

```
Class P {  
    protected void m1(){}  
}
```

```
Class C extends P {  
    public void m1(){} //legal  
}
```

## Difference between overloading and overriding

Property	Overloading	Overriding
Method names	Must be same	Must be same
Arguments	Must be different at least order	Must be same including order
Method signature	Must be different	Must be same
Return types	Same or diff no restrictions	Must be same

### Abstract modifier

**'Abstract'** modifier applicable for classes and methods, but not for variables

#### Abstract class:

- If a class is prefixed with abstract key word such classes are called abstract classes
- In abstract classes we can write abstract methods and concrete methods

- Abstract methods are with out body and ends with semicolon
- Concrete methods contains the method body
- Child class is the responsible for provide implementation for abstract methods of parent class

### **Example program:**

```
abstract class Parent
```

```
{
```

```
    public abstract void m1();//abstract method
```

```
    public void m2();//concrete method
```

```
{
```

```
        System.out.println("concrete method");
```

```
}
```

```
}
```

```
class Child extends Parent
```

```
{  
  
    public void m1()//concrete method  
  
    {  
  
        System.out.println(" absract to non abstract method");  
  
    }  
  
}  
  
class Abs  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        Child c = new Child();  
  
  
        c.m1();  
  
    }  
}
```

```
c.m2();
```

```
}
```

```
}
```

## **Interface :**

We can declare an interface by using interface key word

## **Example :**

```
interface Service
```

```
{
```

```
int i=10;
```

```
void m1();
```

```
void m2();
```

```
}
```

- By default all methods in interface are **public** and **abstract**
- We can provide implementation for an interface by using implements key word
- When we are providing implementation for interface methods they should declared as the **public** other wise compile time error
- When we are providing implementation for interface we should provide implementation for **all methods** in interface, other wise compile time error
- Variables which we defined in interface are by default public, static and final if we allowed to change any variable value in implementation class leads to compile time error

**Example :**

```
class SubService implements Service
```

```
{
```

```
    public void m1()
```

```
    {
```

```
    }
```

```
void m2()//compile time error
```

```
{  
  
}  
  
    Public static void main(String[] args)  
  
    {  
  
        i=20; // compile time error  
  
    }  
  
}
```

### **Program on interface:**

```
interface Service  
  
{  
  
    int i=10;//by default public, static , final  
  
  
    void m1();//by default public and abstract
```

```
}
```

```
class ServiceProvider implements Service
```

```
{
```

```
    public void m1()
```

```
    {
```

```
        System.out.println("i luv u maheshbabu");
```

```
    }
```

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

```
    {
```

```
        //i=20;//compile time erro
```

```
        ServiceProvider s = new ServiceProvider();
```

```
        s.m1();
```



```
}
```

```
}
```

**Output: I luv u maheshbabu**

### ***Difference between abstract class and interface:***

<b>Abstract class</b>	<b>Interface</b>
If we are taking about implementation but not complete	If we never talking about implementation
An abstract class contains concrete methods	Interface should contains only abstract methods
An abstract class contain instance and static variable	Only static variables
An abstract class can contain constructor	Doesn't contain

### Input and Output in Java

To display the content on the screen we are using two statements in java

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

```
System.out.print();
```

The difference between two statements is print () method keeps the cursor on the same line after displaying the output and println() method throws cursor to the next line after displaying the output

```
Int a=10;
```

```
Int b=20;
```

```
Int c=a + b;
```

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

In this example, the values of **a** and **b** are taken as **10** and **20**, respectively and the value of **c** is calculated and displayed. Even if this code is executed several times we get the value of **c** always **30** it means that this code is use ful to perform addition of the two numbers **10** and **15** only

So we want extend our code to perform addition on any two numbers, we have to accept data from keyboard and use it in program

### Accepting a data from keyboard

To accept a data from keyboard and to display on the screen we need streams concept in java

There are two types of Streams: Input streams and Out put streams. Input streams are responsible to read data

and out put streams to send or write the data

So java people provided predefined stream classes in **java.io** package which are capable to perform read and write operations.

For More Information Please Visit:  
**<http://code-archives.appspot.com>**