What is java

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were “Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic”. Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early ‘90s.

James Gosling – founder of java

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

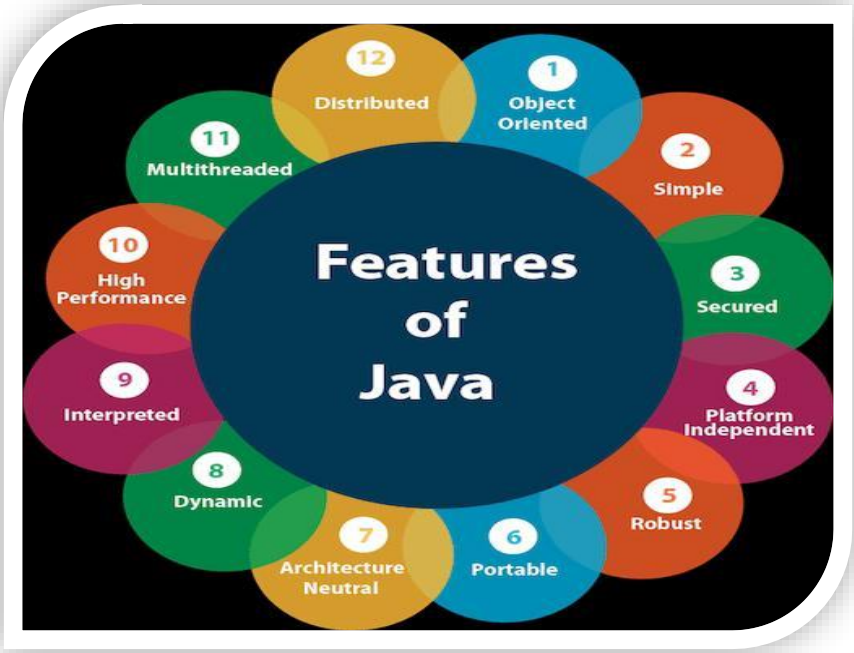
3) Firstly, it was called “Greentalk” by James Gosling, and the file extension was .gt

4) After that, it was called Oak and was developed as a part of the Green project.

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.



Java Features

Simple

Object-Oriented

Portable

Platform independent

Secured

Robust

Architecture neutral

Interpreted

High Performance

Multithreaded

Distributed

Dynamic

Creating Hello World Example

Let's create the hello java program:

```
Class Simple{
    Public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Output:

```
Hello Java
```

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

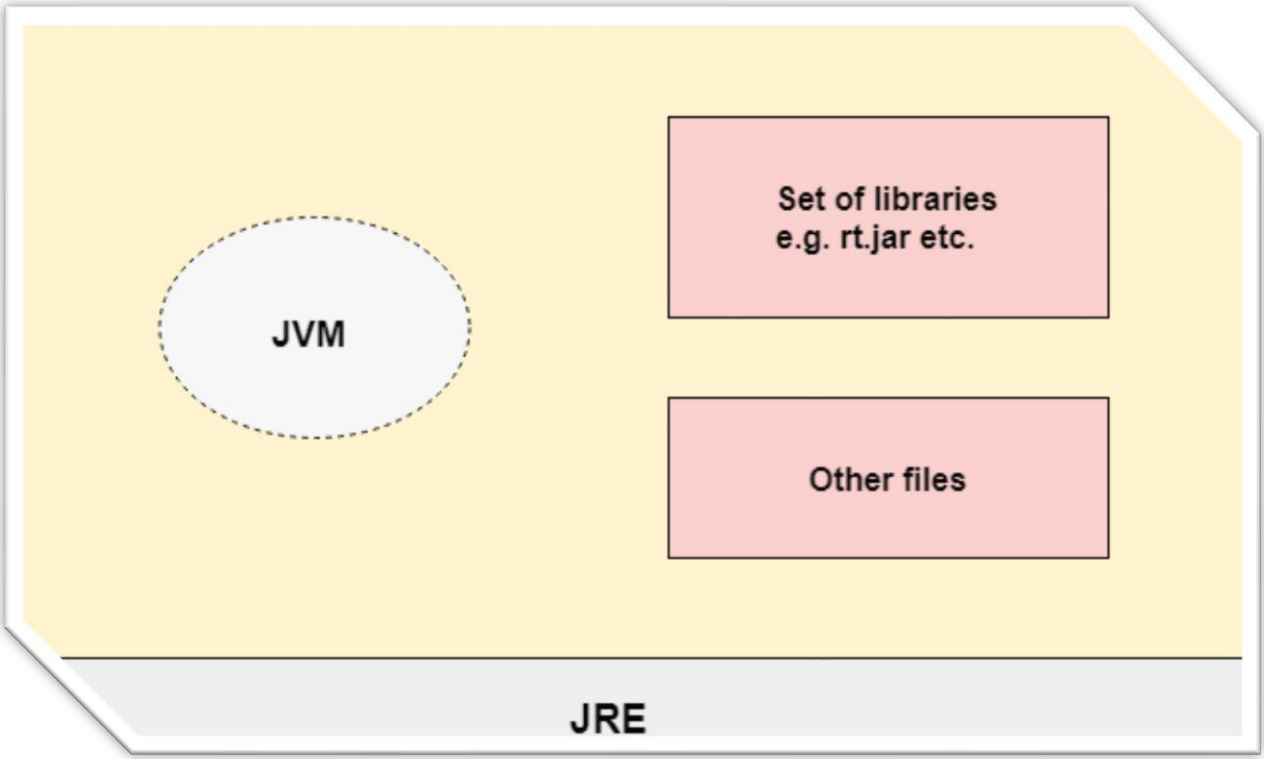
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



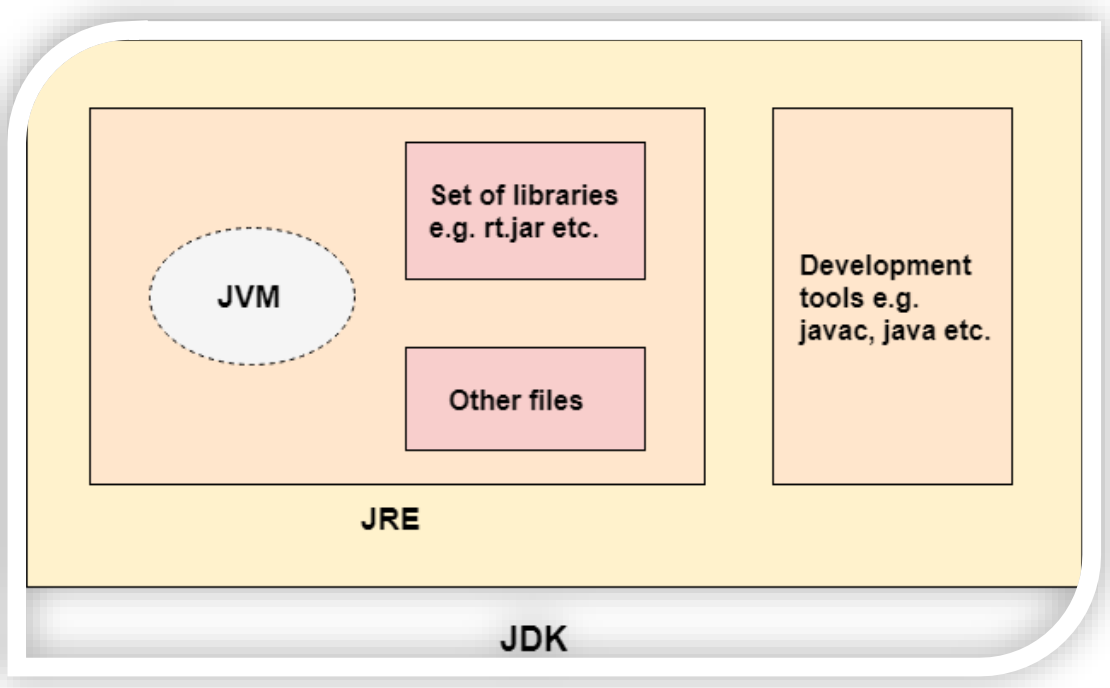
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



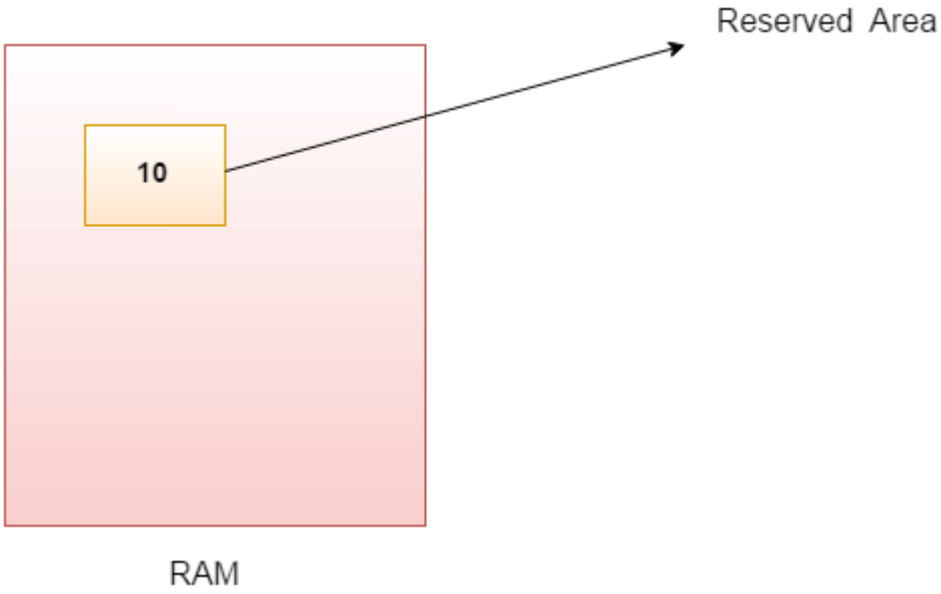
Java variable

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + **able**" which means its value can be changed.




```
1. int data=50;//Here data is variable
```

There are three types of variables in Java:

Local variable

Instance variable

Static variable

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

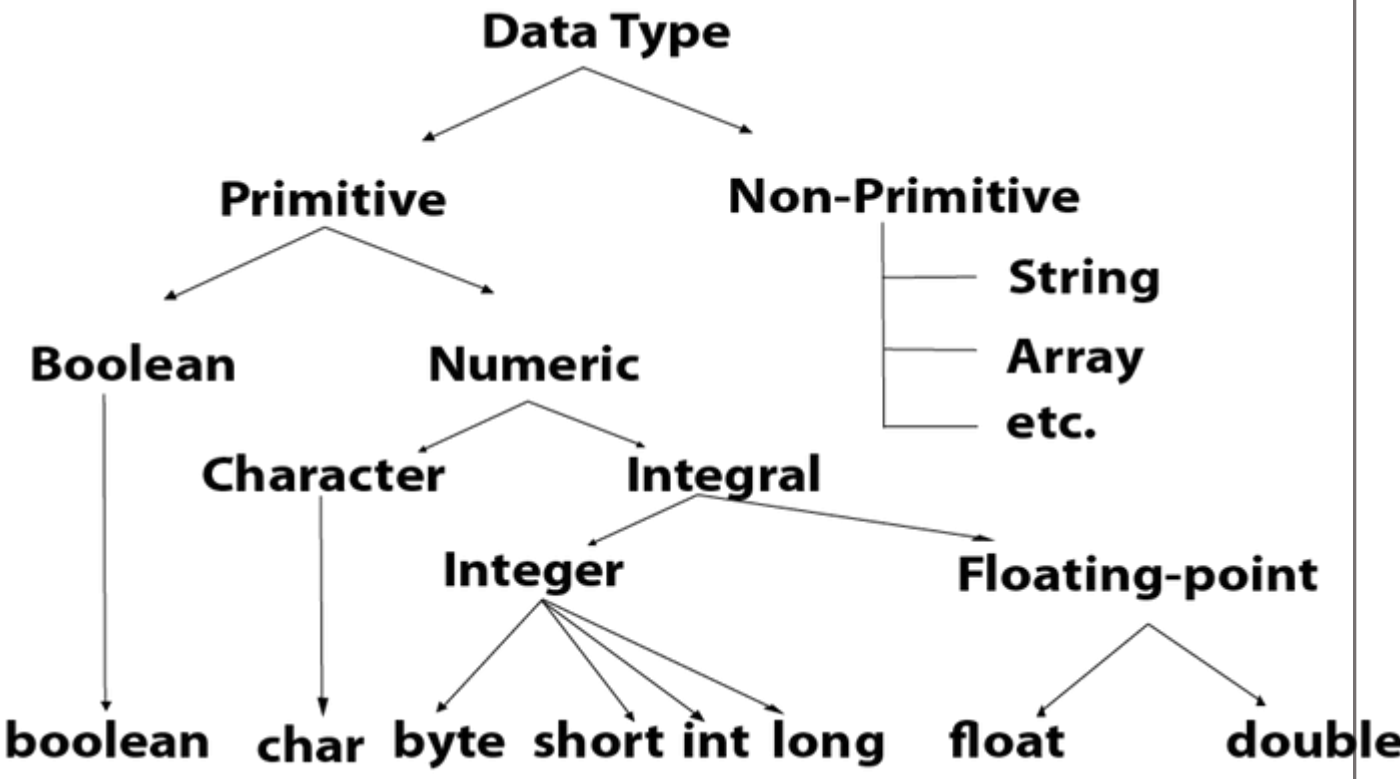
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language

There are 8 types of primitive data types:

1) Boolean data type
2) Byte data type
3) data type
4) Short data type
5) Int data type
6) Long data type

7) Float data type

8) Double data type



Data Type	Default Value	Default size

boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.

- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

Problem

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte.

Solution

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

lowest value:\u0000

highest value:\uFFFF

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>

Relational	comparison	`< > <= >= instanceof`
	equality	`== !=`
Bitwise	bitwise AND	`&`
	bitwise exclusive OR	`^`
	bitwise inclusive OR	`|`
Logical	logical AND	`&&`
	logical OR	`||`
Ternary	ternary	`? :`
Assignment	assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
7.
8. }}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
```

```
4. int b=-10;
5. boolean c=true;
6. boolean d=false;
7. System.out.println(~a);//-
   11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}
```

Output:

```
15
```

```
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10*10/5+3-1*4/2);
4. }}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
1. public OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
1. public class OperatorExample{
2. public static void main(String args[]){
3.     //For positive number, >> and >>> works same
4.     System.out.println(20>>2);
5.     System.out.println(20>>>2);
6.     //For negative number, >>> changes parity bit (MSB) to 0
7.     System.out.println(-20>>2);
8.     System.out.println(-20>>>2);
9. }}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10.}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10.System.out.println(a);//10 because second condition is not checked
11.System.out.println(a>b|a++<c);//true | true = true
12.System.out.println(a);//11 because second condition is checked
13.}}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
```

```
6. System.out.println(min);
7. }}
```

Output:

```
2
```

Another Example:

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

```
5
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
1. public class OperatorExample{
2. public static void main(String[] args){
3. int a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a*=2;//9*2
9. System.out.println(a);
10.a/=2;//18/2
11.System.out.println(a);
12.}}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. short a=10;
4. short b=10;
5. //a+=b;//a=a+b internally so fine
6. a=a+b;//Compile time error because 10+10=20 now int
7. System.out.println(a);
8. }}
```

Output:

```
Compile time error
```

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

A list of Java keywords or reserved words are given below:

1. **abstract**: Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break**: Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte**: Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case**: Java case keyword is used with the switch statements to mark blocks of text.
6. **catch**: Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char**: Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class**: Java class keyword is used to declare a class.
9. **continue**: Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.

10. **default**: Java default keyword is used to specify the default block of code in a switch statement.
11. **do**: Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double**: Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else**: Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum**: Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends**: Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final**: Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally**: Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float**: Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for**: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if**: Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements**: Java implements keyword is used to implement an interface.
22. **import**: Java import keyword makes classes and interfaces available and accessible to the current source code.

23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.

36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.

48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

Student.java

```
1. public class Student {
```

```
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
4. else{
5. statement 2; //executes when condition is false
6. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. int x = 10;
4. int y = 12;
5. if(x+y < 10) {
```

```
6. System.out.println("x + y is less than      10");
7. }   else {
8. System.out.println("x + y is greater than 20");
9. }
10.}
11.}
```

Output:

```
x + y is greater than 20
```

if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. else if(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. else {
8. statement 2; //executes when all the conditions are false
9. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. String city = "Delhi";
```

```
4. if(city == "Meerut") {
5. System.out.println("city is meerut");
6. }else if (city == "Noida") {
7. System.out.println("city is noida");
8. }else if(city == "Agra") {
9. System.out.println("city is agra");
10.}else {
11.System.out.println(city);
12.}
13.}
14.}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. if(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. else{
7. statement 2; //executes when condition 2 is false
8. }
9. }
```

Consider the following example.

Student.java

```
1. public class Student {
2. public static void main(String[] args) {
3. String address = "Delhi, India";
4.
5. if(address.endsWith("India")) {
6. if(address.contains("Meerut")) {
7. System.out.println("Your city is Meerut");
8. }else if(address.contains("Noida")) {
9. System.out.println("Your city is Noida");
10.}else {
11.System.out.println(address.split(",")[0]);
12.}
13.}else {
14.System.out.println("You are not living in India");
15.}
16.}
17.}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate

- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
1. switch (expression){
2.     case value1:
3.      statement1;
4.      break;
5.     .
6.     .
7.     .
8.     case valueN:
9.      statementN;
10.     break;
11.    default:
12.     default statement;
13.}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
1. public class Student implements Cloneable {
2. public static void main(String[] args) {
3. int num = 2;
4. switch (num){
5. case 0:
```

```
6. System.out.println("number is 0");
7. break;
8. case 1:
9. System.out.println("number is 1");
10. break;
11. default:
12. System.out.println(num);
13. }
14. }
15. }
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

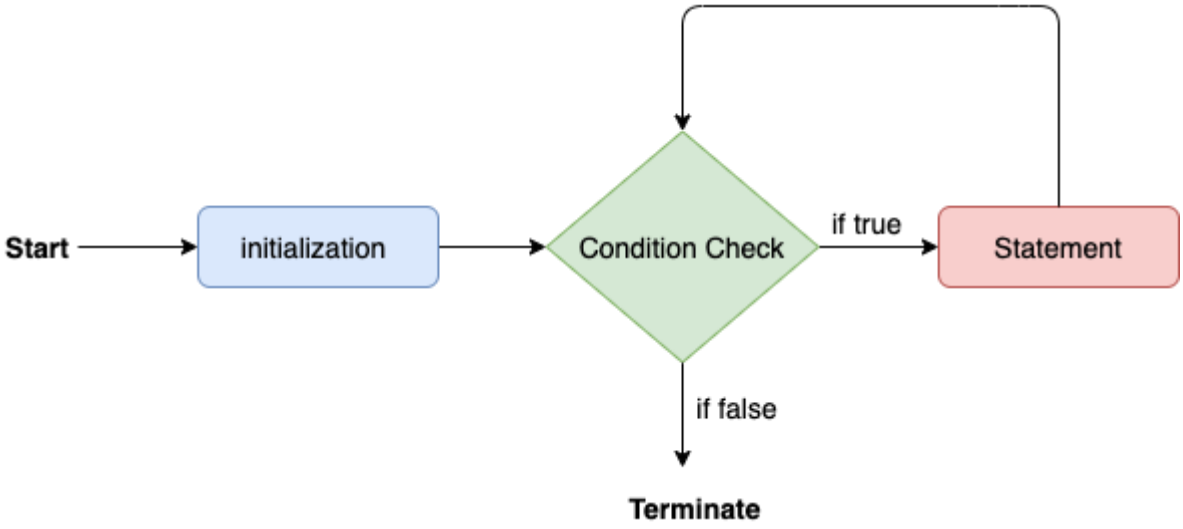
Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of

code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
1. for(initialization, condition, increment/decrement) {
2. //block of statements
3. }
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
1. public class Calculattion {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int sum = 0;
5. for(int j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10.}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
1. for(data_type var : array_name/collection_name){
2. //statements
3. }
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. for(String name:names) {
7. System.out.println(name);
8. }
9. }
10.}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

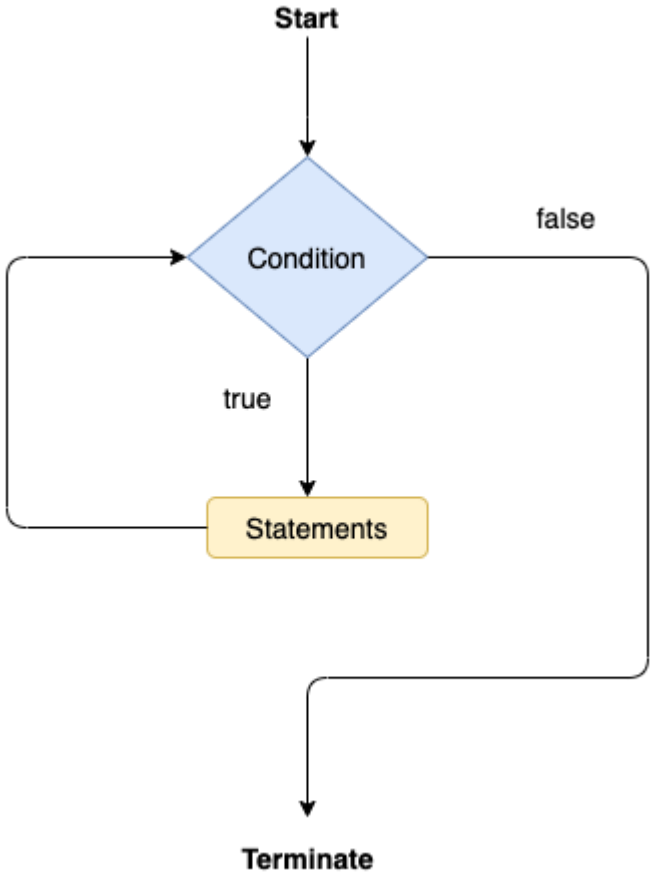
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
1. while(condition){
2. //looping statements
3. }
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. while(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10.}
11.}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

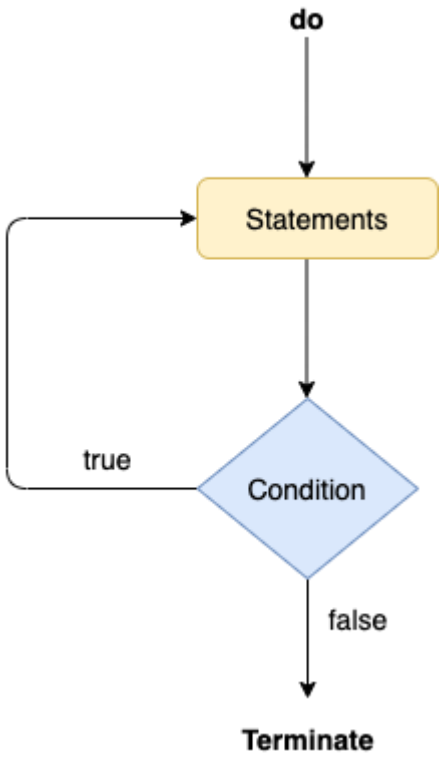
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
1. do
2. {
3. //statements
4. } while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
1. public class Calculation {
2. public static void main(String[] args) {
3. // TODO Auto-generated method stub
4. int i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. do {
7. System.out.println(i);
8. i = i + 2;
9. }while(i<=10);
10.}
11.}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
1. public class BreakExample {
2.
3. public static void main(String[] args) {
4. // TODO Auto-generated method stub
5. for(int i = 0; i<= 10; i++) {
6. System.out.println(i);
7. if(i==6) {
8. break;
9. }
10.}
```

```
11.}
12.}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```
1.  public class Calculation {
2.
3.  public static void main(String[] args) {
4.  // TODO Auto-generated method stub
5.  a:
6.  for(int i = 0; i<= 10; i++) {
7.  b:
8.  for(int j = 0; j<=15;j++) {
9.  c:
10.for (int k = 0; k<=20; k++) {
11.System.out.println(k);
12.if(k==5) {
13.break a;
14.}
15.}
16.}
17.
18.}
19.}
20.
```

```
21.
22.}
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
1.  public class ContinueExample {
2.
3.  public static void main(String[] args) {
4.  // TODO Auto-generated method stub
5.
6.  for(int i = 0; i<= 2; i++) {
7.
8.  for (int j = i; j<=5; j++) {
9.
10.if(j == 4) {
11.continue;
12.}
13.System.out.println(j);
14.}
15.}
16.}
17.
```

```
18.}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

Object class in Java

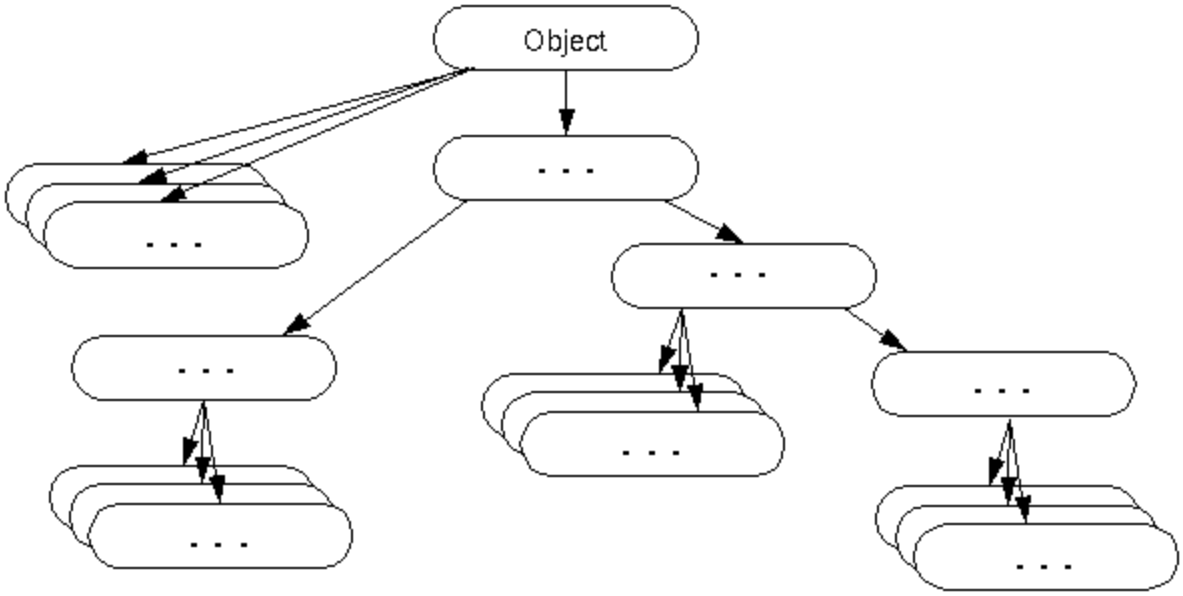
The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

```
1. Object obj=getObject();//we don't know what object will be returned from thi
   s method
```

2. The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



3.

Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode

	number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.

public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies

	(invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

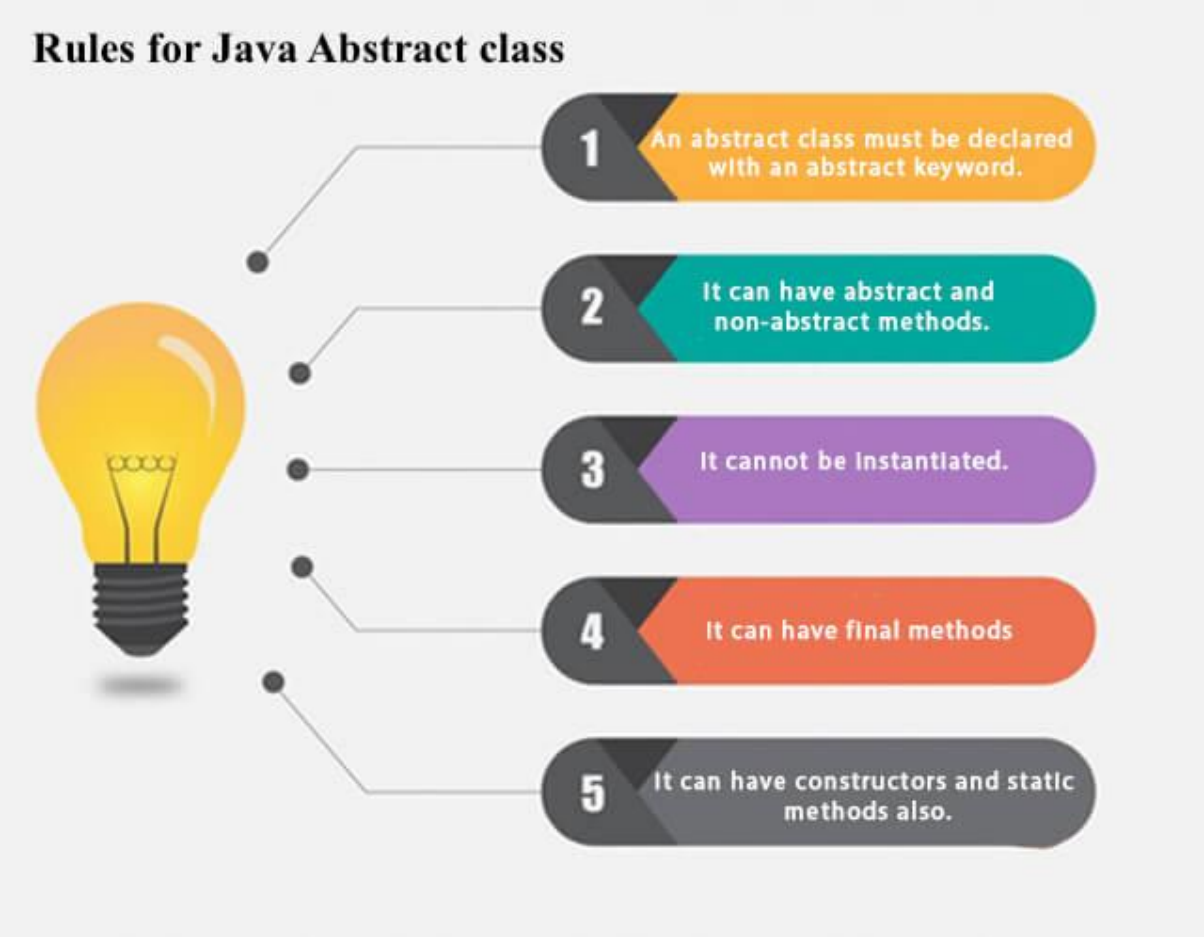
1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.



Example of abstract class

```
1. abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
1. abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7.  Bike obj = new Honda4();
8.  obj.run();
9. }
10.}
```

Test it Now

```
running safely
```

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

Capsule

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```
1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
```

```
3. package com.javatpoint;
4. public class Student{
5. //private data member
6. private String name;
7. //getter method for name
8. public String getName(){
9. return name;
10.}
11.//setter method for name
12.public void setName(String name){
13.this.name=name
14.}
15.}
```

File: Test.java

```
1. //A Java class to test the encapsulated class.
2. package com.javatpoint;
3. class Test{
4. public static void main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=new Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10.System.out.println(s.getName());
11.}
12.}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

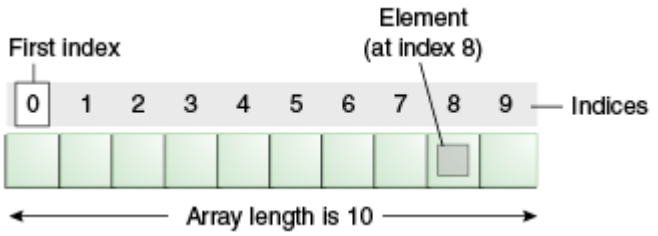
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++,

we can also create single dimentional or multidimentional arrays in Java.

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



Advantages

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

Disadvantages

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

Types of Array in java

There are two types of array.

- Single Dimensional Array

- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];
```

Instantiation of an Array in Java

```
1. arrayRefVar=new datatype[size];
```

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10.a[4]=50;
11.//traversing array
12.for(int i=0;i<a.length;i++)//length is the property of array
13.System.out.println(a[i]);
14.}}
```

Test it Now

Output:

```
10
20
70
40
50
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

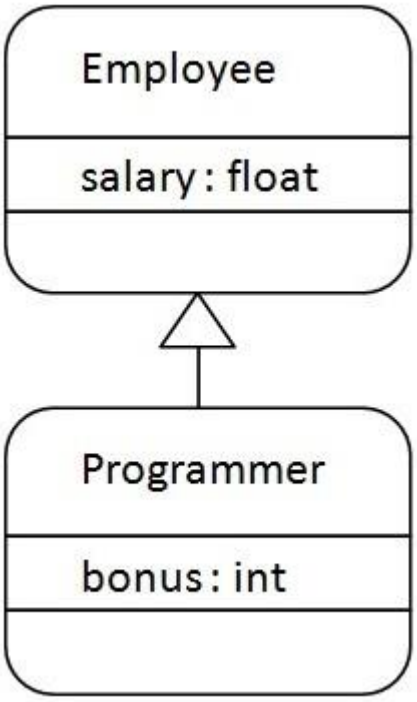
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.    //methods and fields
4. }
```

5. The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
6. In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.
7.

8. Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
4. class Programmer extends Employee{
```

```
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10.}
11.}
```

Test it Now

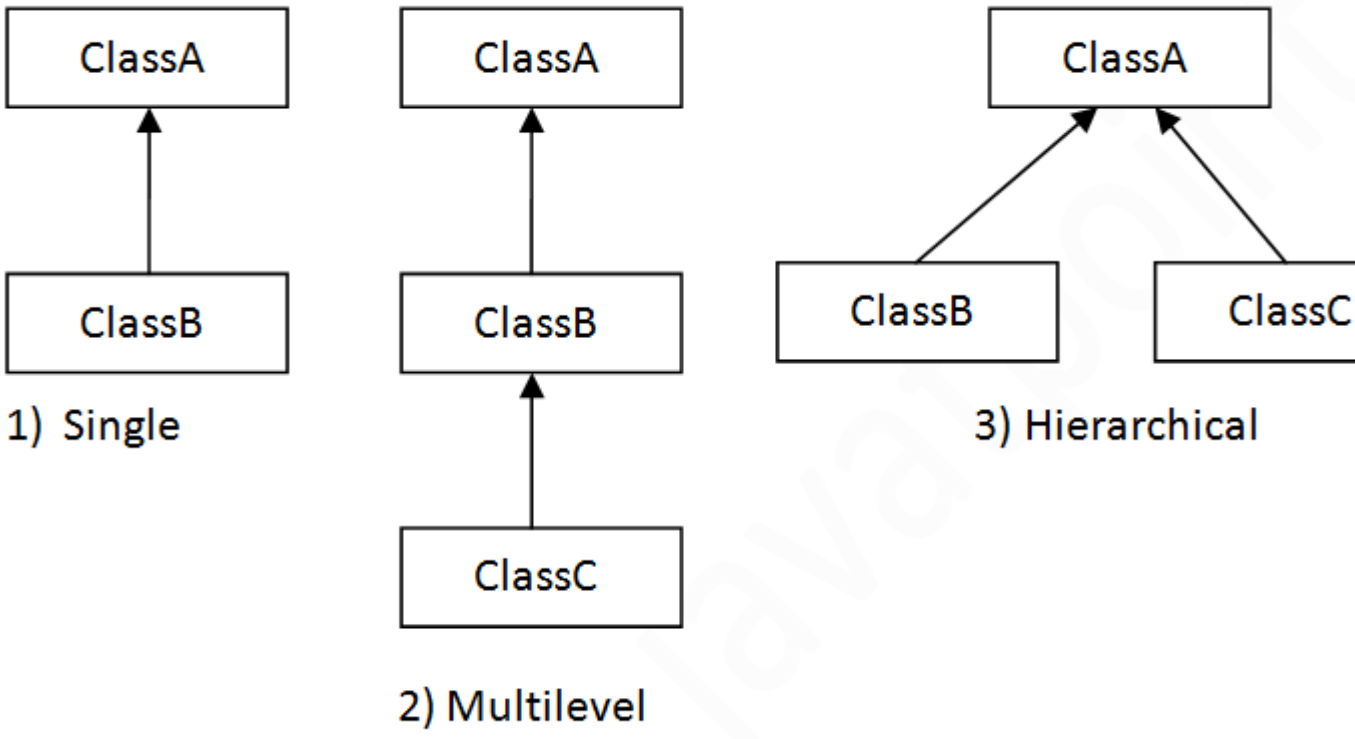
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

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

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance.

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Method Overloading in Java

If a class has multiple methods having same name but different in 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.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

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

> ***In Java, Method Overloading is not possible by changing the return type of the method only.***

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

Test it Now

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

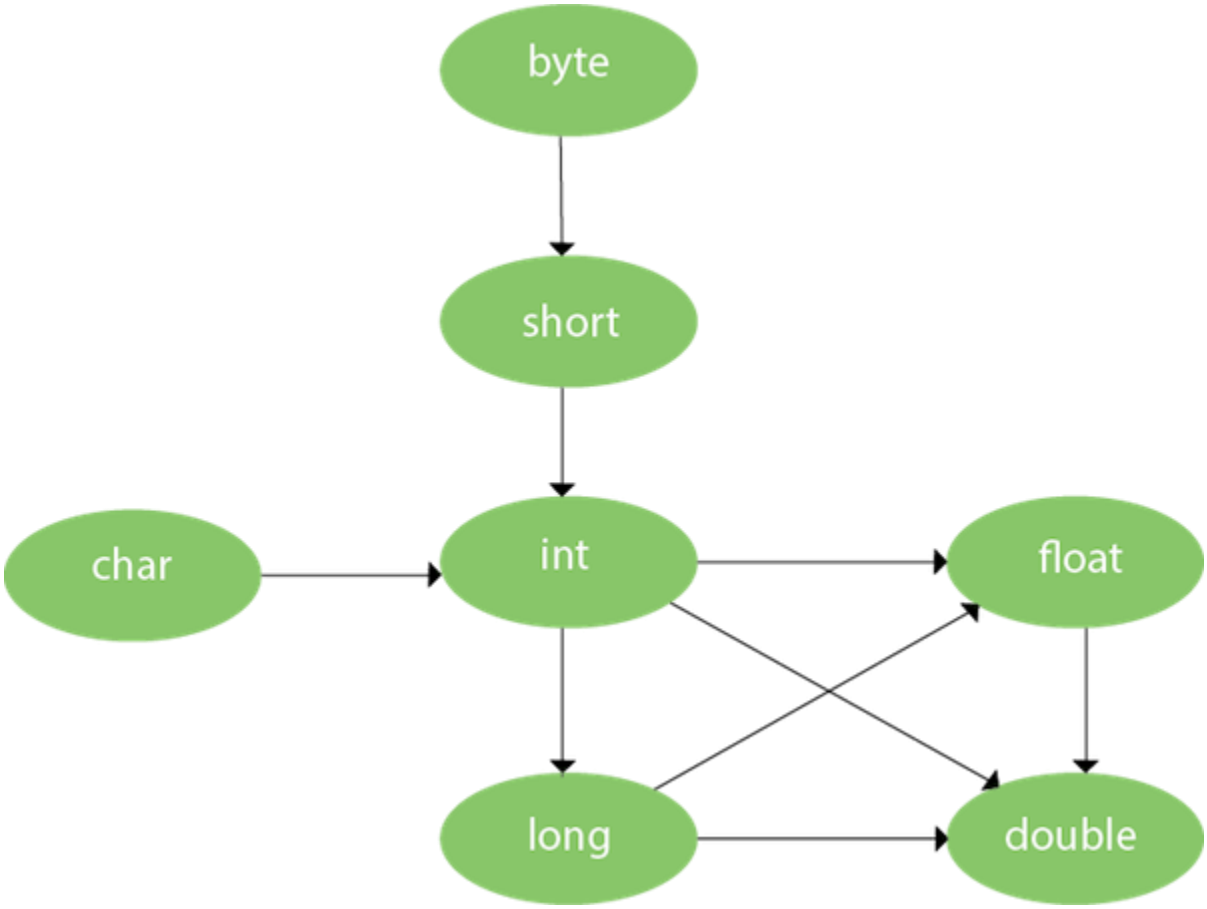
Test it Now

Output:

```
22
24.9
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation1 obj=new OverloadingCalculation1();
7.   obj.sum(20,20);//now second int literal will be promoted to long
8.   obj.sum(20,20,20);
9.
10.  }
```

```
11.}
```

Test it Now

```
Output:40
       60
```

Method Overriding in Java

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

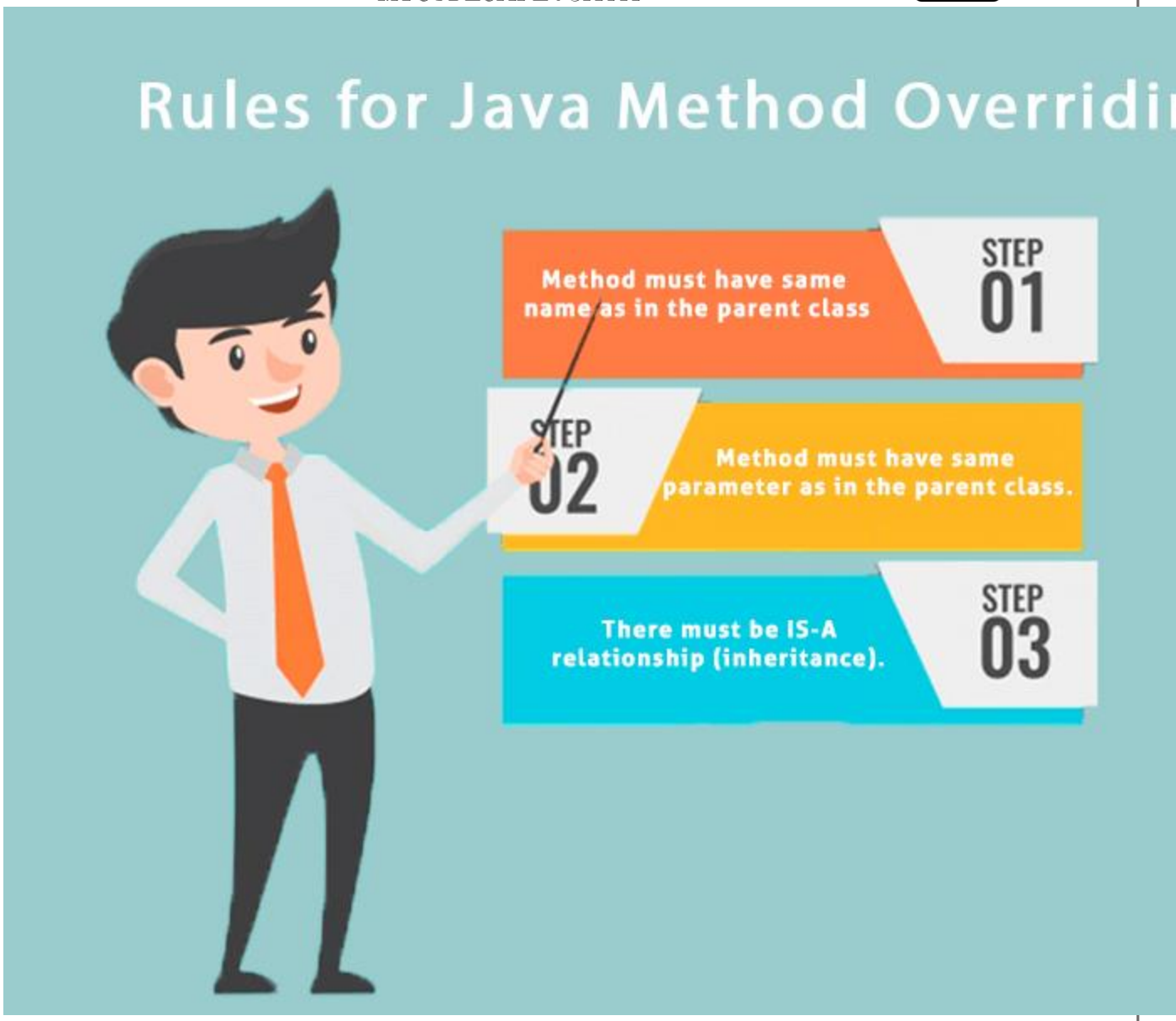
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

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

Rules for Java Method Overriding

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



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
```

```
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Test it Now

Output:

```
Vehicle is running
```

Example of method overriding

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

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

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

```
14. obj.run();//calling method
15. }
16.}
```

Test it Now

Output:

```
Bike is running safely
```

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

FileName: B1.java

```
1. class A{
2. A get(){return this;}
3. }
4.
5. class B1 extends A{
6. @Override
7. B1 get(){return this;}
8. void message(){System.out.println("welcome to covariant return type");}
9.
10.public static void main(String args[]){
11.new B1().get().message();
12.}
13.}
```

Output:

```
welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

Advantages of Covariant Return Type

Following are the advantages of the covariant return type.

1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.

2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.

3) Covariant return type helps in preventing the run-time *ClassCastExceptions* on returns.

Let's take an example to understand the advantages of the covariant return type.

FileName: CovariantExample.java

```
1. class A1
2. {
3.     A1 foo()
4.     {
5.         return this;
6.     }
7.
8.     void print()
9.     {
10.        System.out.println("Inside the class A1");
11.    }
```

```
12.}
13.
14.
15.// A2 is the child class of A1
16.class A2 extends A1
17.{
18.    @Override
19.    A1 foo()
20.    {
21.        return this;
22.    }
23.
24.    void print()
25.    {
26.        System.out.println("Inside the class A2");
27.    }
28.}
29.
30.// A3 is the child class of A2
31.class A3 extends A2
32.{
33.    @Override
34.    A1 foo()
35.    {
36.        return this;
37. }
38.
39.    @Override
40.    void print()
41.    {
42.        System.out.println("Inside the class A3");
```

```
43.    }
44.}
45.
46.public class CovariantExample
47.{
48.    // main method
49.    public static void main(String argvs[])
50.    {
51.        A1 a1 = new A1();
52.
53.        // this is ok
54.        a1.foo().print();
55.
56.        A2 a2 = new A2();
57.
58.        // we need to do the type casting to make it
59.        // more clear to reader about the kind of object created
60.        ((A2)a2.foo()).print();
61.
62.        A3 a3 = new A3();
63.
64.        // doing the type casting
65.        ((A3)a3.foo()).print();
66.
67.    }
68.}
```

Output:

```
Inside the class A1
Inside the class A2
Inside the class A3
```

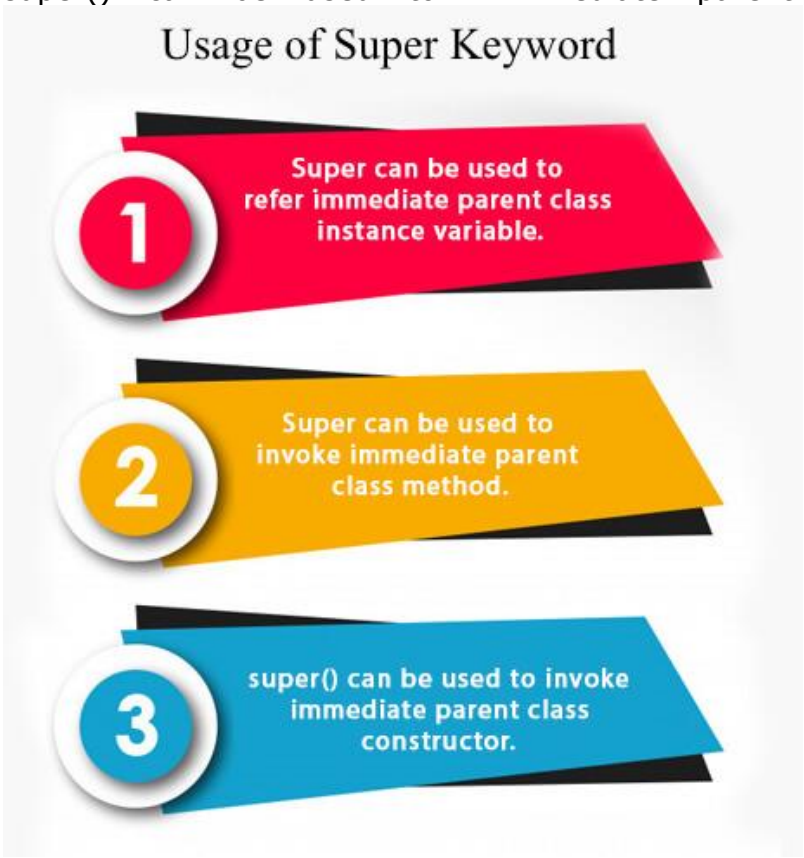
Super Keyword in Java

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

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

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



o

Instance initializer block

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Example of instance initializer block

Let's see the simple example of instance initializer block that performs initialization.

```
1. class Bike7{
2.     int speed;
3.
4.     Bike7(){System.out.println("speed is "+speed);}
5.
6.     {speed=100;}
7.
8.     public static void main(String args[]){
9.     Bike7 b1=new Bike7();
10.    Bike7 b2=new Bike7();
11.    }
12.}
```

Test it Now

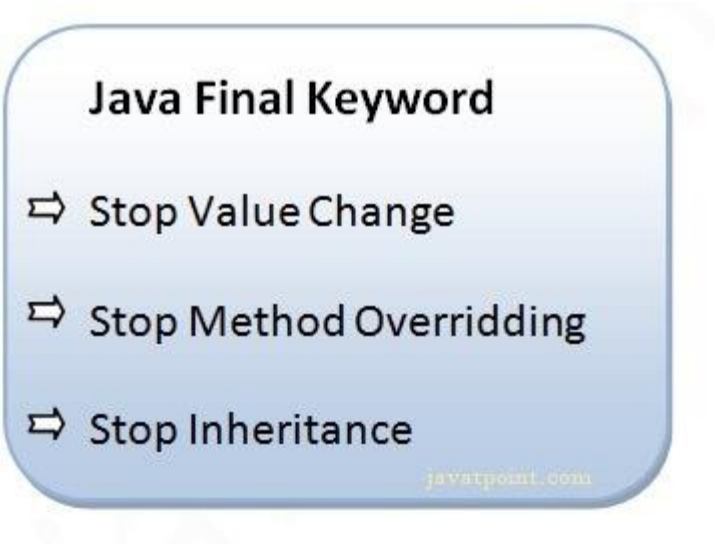
```
Output:speed is 100
       speed is 100
```

Final Keyword In Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

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

Example of final variable

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

```
1. class Bike9{
```

```
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  }
10.}//end of class
```

Test it Now

```
Output:Compile Time Error
```

Runtime Polymorphism in Java

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example of Java Runtime Polymorphism

```
1.  class Bike{
2.    void run(){System.out.println("running");}
3.  }
4.  class Splendor extends Bike{
5.    void run(){System.out.println("running safely with 60km");}
6.
7.    public static void main(String args[]){
8.      Bike b = new Splendor();//upcasting
9.      b.run();
10.   }
11.}
```

Test it Now

Output:

```
running safely with 60km.
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
1. class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6.  void eat(){System.out.println("dog is eating...");}
7.
8.  public static void main(String args[]){
9.   Animal a=new Dog();
10.  a.eat();
11. }
12.}
```

Test it Now

```
Output:dog is eating...
```

Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple1{
2.  public static void main(String args[]){
3.  Simple1 s=new Simple1();
4.  System.out.println(s instanceof Simple1);//true
5.  }
6. }
```

Test it Now

```
Output:true
```

java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple1{
2.  public static void main(String args[]){
3.  Simple1 s=new Simple1();
4.  System.out.println(s instanceof Simple1);//true
5.  }
6. }
```

Test it Now

```
Output:true
```