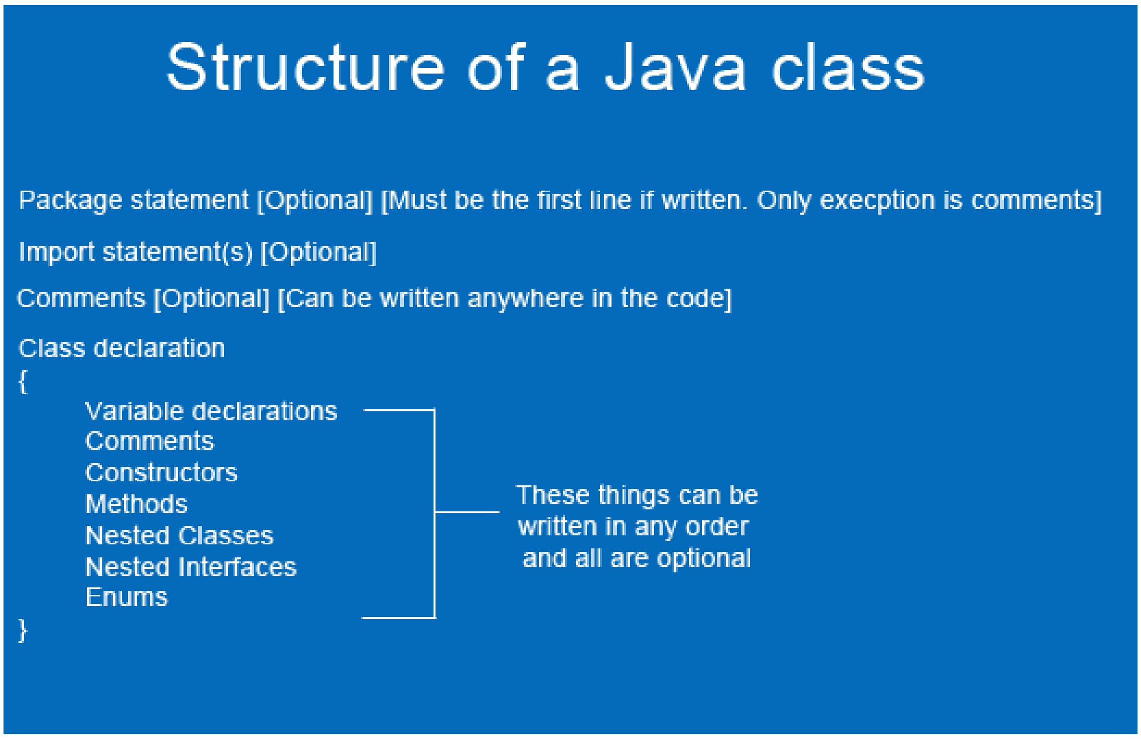
UNIT - 2

CORE JAVA BASICS

Structure of a Java program

Structure of a Java class is as shown below:


Structure of a Java class
Package statement [Optional] [Must be the first line if written. Only execption is comments]
Import statement(s) [Optional]
Comments [Optional] [Can be written anywhere in the code]
Class declaration
{
Variable declarations
Comments
Constructors
Methods
Nested Classes
Nested Interfaces
Enums
}
These things can be written in any order and all are optional

Startertutorials.com

A Java class can contain the following:

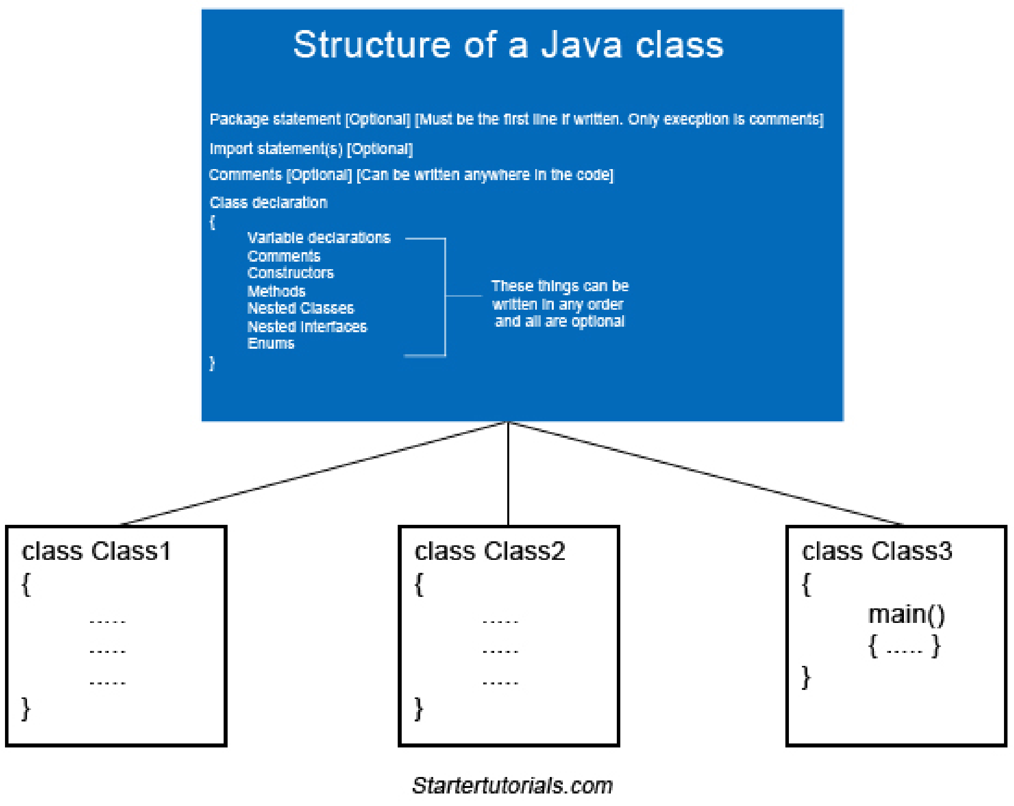
- **Package statement:** A package statement is used to declare a Java class as a part of the specified package. More on declaring packages later. Package statement is optional. When we decide to write a package statement, it should be the first statement in the file. Only exception to this rule is writing comments.

- **Import statement(s):** We can write one or more import statements. These are also optional. An import statement is used to link our class with other classes in the same package or other packages to use their functionality. More on import statement in other articles.

- **Comments:** We can write one or more comments in a Java class to explain the use of certain statements or provide extra information like purpose of the class, author's name, date and time of creation etc. Comments are optional. Comments can be written in the first line or anywhere in the program. More on writing comments later.
- **Class declaration:** A class declaration consists of the class keyword followed by the class name, which is followed by the body of the class represented using braces { }. A class declaration can contain the following:
    - Variable declarations
    - Comments
    - Constructors
    - Methods
    - Nested classes
    - Nested interfaces
    - Enumerations

All the above elements (variable declarations, comments etc...) can occur in any order.

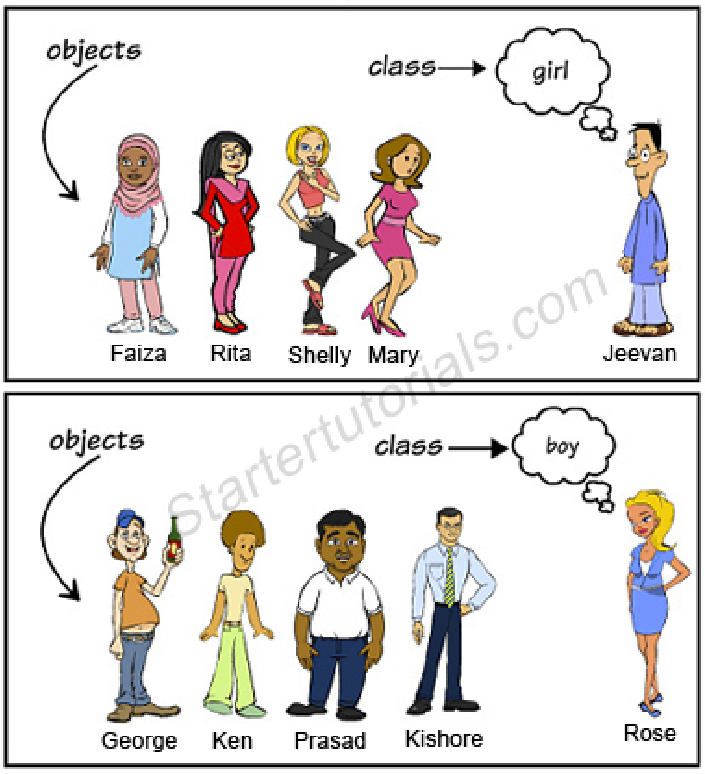
Now, let's look at the structure of a Java program. It will be as shown below:


Structure of a Java class
Package statement [Optional] [Must be the first line if written. Only exception is comments]
Import statement(s) [Optional]
Comments [Optional] [Can be written anywhere in the code]
Class declaration
{
Variable declarations
Comments
Constructors
Methods
Nested Classes
Nested Interfaces
Enums
}
These things can be written in any order and all are optional
class Class1 { .... }
class Class2 { .... }
class Class3 { main() { ..... } }
Startertutorials.com

As I had already mentioned above, a Java program is a set of one or more classes in which each class will be declared following the structure of a Java class and one of the classes will have a main method to start the execution of the Java program.

Classes and Objects

According to object orientation, a class is a template or blueprint for creating objects. An object is an instance of a class. To understand a Java class and object easily let's consider a real world example of boys and girls as shown below:



In the above example, the classes are boy and girl. Objects are Faiza, Rita, Shelly, Mary, Rose, George, Ken, Prasad, Kishore and Jeevan.

If we declare the boy as a class in Java, it will be something as shown below:

```
class Boy
{
        String name;
        String address;
        int age;
        void tellName(){ };
        void tellAddress(){ };
        void tellAge(){ };
}
```

Declaring a class for girl is left to you as an exercise.

Now, let's create an object, say, George. In Java, we create objects as shown below:

```
Boy  george = new  Boy();
```

Again creating other objects is left as an exercise for you.

Class:

To declare a class in Java, we use the Java keyword class. Remember that all the keywords in Java are lowercase letters. The class keyword is followed by class name which is Boy in my example.

According to Java's convention, every first letter in a word must be a uppercase letter. All the predefined Java classes follow this convention. Some of the predefined Java classes are:

Object
Exception
Throwable
System
Print**S**tream
Scanner
String**T**okenizer

In the above example of class Boy, we have three variables: name, address and age. We also have three methods (functions in C and C++) namely, tellName(), tellAddress() and tellAge().

If you are creating a class, it means that you are creating a new user-defined type. That class (type) will be used to create objects later.

General syntax for declaring a class in Java is as shown below:

```
class ClassName
{
        //Class Members
}
```

In the above example, the class members are the three variables and the three methods. So, there are a total of 6 class members in our example of Boy class.

Rules for writing class names or any other identifiers (variable names, method names etc…):

- A name must start with a letter or an underscore (_) or a dollar sign ($). Beginning the name with a letter is recommended. From second character on wards a name can contain digits or numbers also.
- White spaces and special symbols like %, @, * are not allowed in the name.
- Uppercase characters are distinct from lowercase characters. Names are case-sensitive i.e var is different from VAR.
- Keywords or reserved words can't be used for names. For example, goto cannot be used for a name as it is a reserved word in Java.

Some valid class names will be:

Student
Contact
UserDetails
Registration
Marks
Sem2Marks

_Names
$Product

Some invalid examples for class names are:

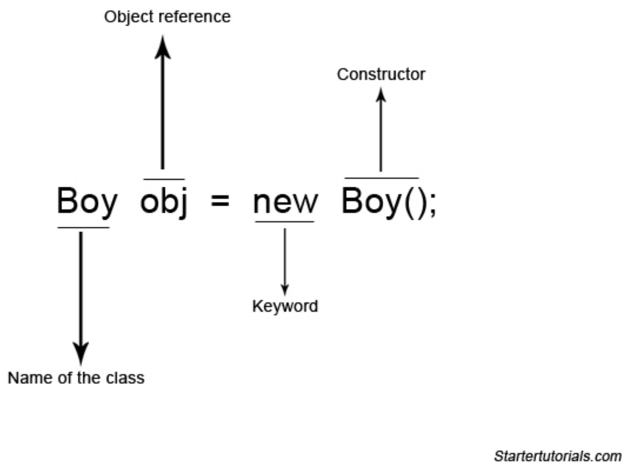
2Inventory (Name cannot start with a number)
User Details (Name cannot contain white spaces)
Cart@Shop (Name cannot contain special symbols)
switch (Keywords cannot be used in names)

Object:

Syntax for creating an object in Java is as shown below:

ClassName object-reference = new ClassName();

An example for creating an object for the Boy class declared above is as shown below:



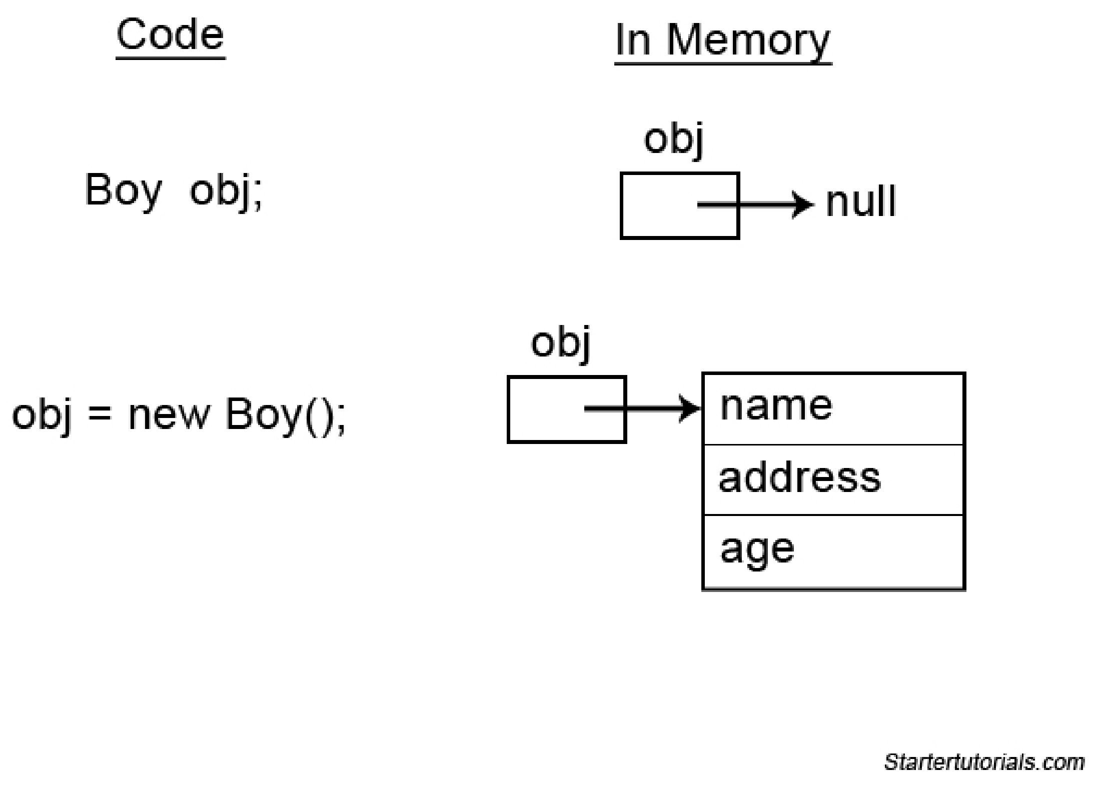
As already denoted in the figure, Boy is the name of the class for which we are creating the object. Remember that objects doesn't exist without a class. obj is the reference to the object. The actual object is created by the new operator which is one of the Java's keywords. The new keyword creates the object dynamically (at runtime) in the RAM (Random Access Memory) and returns the reference (address) of the object created. That reference is stored in obj which will be used to access the object in future.

The last part of the line is Boy() which is a constructor. For now remember that a constructor is a method which will have the same name as class name. Constructors will be explained in detail in a future article.

Above object creation can be separated into two lines as shown below:

```
Boy obj;
obj = new Boy();
```

In the first line, by default, obj refers to null. The memory representation of the above two lines is as shown below:



After creating an object using the above syntax, we can access the class members like variables and methods using the dot (.) operator or also known as dot separator. So, we can access the three variables (fields) name, address and age as shown below:

```
obj.name = “George”;
obj.address = “Los Angels, USA”;
obj.age = 23;
```

The general syntax for accessing a class member is as shown below:

```
object_reference.member_name;
```

By joining all the pieces together, the complete program will be as shown below:

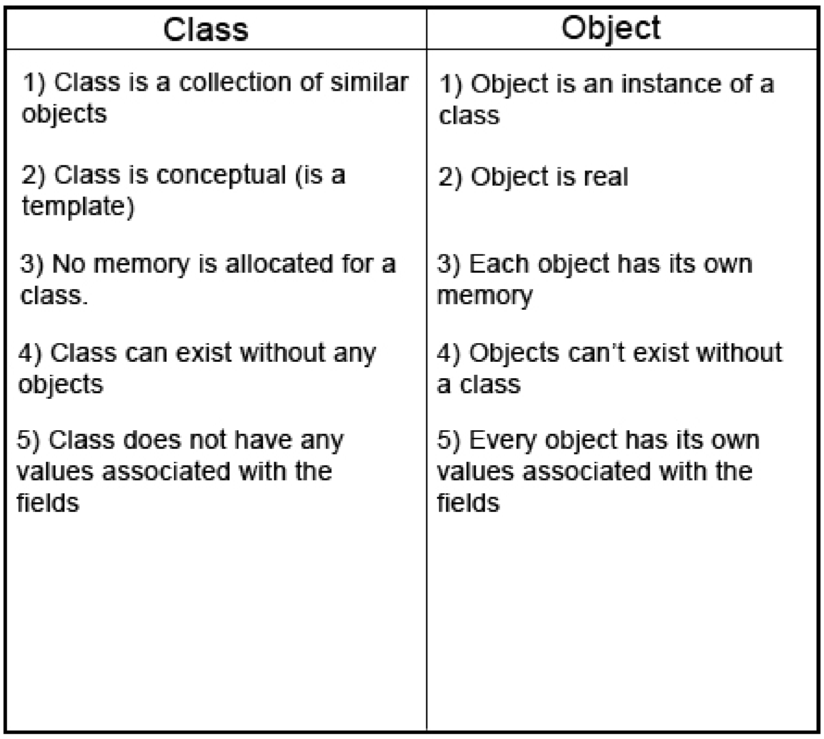
```
class Boy
{
        String name;
        String address;
        int age;
        void tellName(){ };
        void tellAddress(){ };
        void tellAge(){ };
        public static void main(String[] args)
        {
                Boy obj = new Boy();
                obj.name = "George";
                obj.address = "Los Angels, USA";
                obj.age = 23;
                System.out.println("Name of the boy is: "+obj.name);
                System.out.println("Address of the boy is: "+obj.address);
                System.out.println("Age of the boy is: "+obj.age);
        }
}
```

In lines 15,16 and 17 you can see that there is a + operator in the println statement. Here, + is not a addition operator. If one operand of the + operator is a string, then + behaves as a concatenation operator. So, the right side value will be converted into a string and is joined with the left side string.

Output of the above program will be:

```
Name of the boy is: George
Address of the boy is: Los Angels, USA
Age of the boy is: 23
```

Differences between a class and an object:



Class	Object
1) Class is a collection of similar objects	1) Object is an instance of a class
2) Class is conceptual (is a template)	2) Object is real
3) No memory is allocated for a class.	3) Each object has its own memory
4) Class can exist without any objects	4) Objects can't exist without a class
5) Class does not have any values associated with the fields	5) Every object has its own values associated with the fields

Some of the questions related to Java classes and objects:

1) Are object references same as pointers in C and C++?

A: Although references are very similar to pointers (both store address of the memory location), references cannot be manipulated as we can with pointers. In C and C++, one can increment a pointer value or do other operations on it. Such operations are not allowed on references in Java.

2) Where are objects created?

A: Objects are created dynamically in the heap area inside RAM (Random Access Memory).

3) Can classes exist without objects?

A: Yes. But vice-verse is not true.

4) What does an instance actually mean?

A: As already mentioned above, the fields of a class does not contain any values. An object contains its own value for every field. For example, if we consider the object of Boy class (see above), the values for the fields name, address and age are: "George", "Los Angels, USA" and 23 respectively. These values are collectively known as the state of the object. This is why an object is known as an instance of the class as it has its own values for all the fields.

Comments

Writing comments is considered a good programming practice. They will help the client programmers to understand the programs developed by you.

Java comments can be a single line or multiple lines of text which are written by the programmers for documentation purpose. A programmer can write comments to mention the purpose of the program or a part of the program or to provide authorship details.

Need for comments

Following are some of the famous reasons for writing comments in programs:

- Provide the purpose of the program along with sample input and output of the program.
- Provide authorship details.
- Mention date and time of creation of the program and when it was last modified.
- Provide details about several parts of the program.
- For generating documentation which will be helpful for other programmers.

Types of comments

In Java, there are three types of comments. They are:

- Single line comments
- Multi-line comments
- javadoc comments

A single line comment as the name implies is a comment containing a single line of text. A single line comment starts with //. All the characters after the // to the end of the line is treated as a comment. Syntax for single line comment is as shown below:

```
// This is a single line comment
```

To write a comment containing more than one line, we can use a multi-line comment. A multi-line comment starts with /* and ends with */. Syntax for writing a multi-line comment is as shown below:

```
/*  This is a
multi-line
comment */
```

We can use a multi-line comment for writing a single line comment as shown below:

```
/* This is a single line comment. */
```

Both single line comments and multi-line comments can be written anywhere in the program.

The third type of comments called javadoc comments can be used to generate HTML documentation which can be viewed using a browser. This can help the client programmers to understand the classes that you create. A javadoc comment starts with /** and ends with */. Syntax for writing a javadoc comment is as shown below:

```
/**
* This is a
* javadoc comment
*/
```

Note that * is not necessary before each line of text. It is widely accepted usage of javadoc comments by Java developers. Generally javadoc comments are written at the top of the program. Comments which are written before a class declaration are known as header comments.

We can mention several types of specific information like author name, parameters of a function, return value of a method and many more using a javadoc comment.

One thing to remember about comments is, they cannot be nested. For example, the following is a incorrect way of writing comments:

```
//This is a //bad single line comment
```

Every comment started using a beginning marker must be closed with a ending marker. Following is an example of wrong way of writing comments:

```
/*
This is
a bad
multi-line
comment /
```

Note that in the above example, the ending marker should be */.

Variables

Variable Definition:

A variable is a named location in memory (RAM) to store data.

It is common in programs to store data and process the data to get the required outcome. If the data is predetermined (Case 1), for example, you want to add 2 and 3. Here, you know what to add. But what if the data will be read at runtime (Case 2) or someone else will give the data for your program. You can't predict that data.

For Case 2, you have to use variables for storing the data in memory and then process it to get the output.

Java program for Case 1 will be as shown below:

```
class Sample
{
   public static void main(String[] args)
          {
                      System.out.println("Sum of 2 and 3 is: "+(2+3));
          }
}
```

In the above program you must enclose 2+3 in brackets. Otherwise instead of 5, which is the expected output, you will get 23 (2 and 3 will be treated as strings and gets concatenated with each other).

Before looking at the program for Case 2, let's learn how to declare variables and use them.

Creating or declaring variables in Java:

A variable can be created or declared in Java by using the below syntax:

```
datatype  variable_name;
```

A variable_name can be any valid identifier in Java. Before using any variable in a Java program, you have to declare it first as per the syntax given above. The datatype specifies the type of value you are going to store in the variable. More on Java data types in another article. For now, just think of it as the type of variable.

Example for declaring a variable in Java is shown below:

```
int  x;
```

In the above example, int is a Java's primitive data type and x is the name of the variable.

Initializing variables in Java:

After declaring variables, we can initialize them as shown below:

```
variable_name = value;
```

So, we can initialize the variable x as shown below:

```
x = 10;
```

Remember that x has been declared as an integer. So, we can store integer values in x. For initializing a variable, we use the assignment (=) operator. We can combine variable declaration and initialization into a single line as shown below:

```
int  x = 10;
```

Initializing variables with literal values like 10, 2.5, etc... is known as static initialization. We can also assign an expression to a variable as shown below:

```
int a = 10;
int b = 20;
int c = a+b;
```

The expression a+b will be evaluated at runtime (execution of the program) and then it will be assigned to the variable c. This type of initialization is know as dynamic initialization.

We can declare multiple variables of the same type as shown below:

```
int  x, y, z;
```

Java program for Case 2 shown below will give you an example of how to use variables in Java:

```
import java.util.Scanner;
class Sample
{
   public static void main(String[] args)
          {
                 int x, y;
                 Scanner input = new Scanner(System.in);
                 System.out.println("Enter value of x: ");
                 x = input.nextInt();
                 System.out.println("Enter value of y: ");
                 y = input.nextInt();
                 System.out.println("Sum of x and y is: "+(x+y));
          }
}
```

In the above program Scanner is a utility class available in java.util package to read data from various sources like standard input, files etc... To read input from console (standard input) we provide System.in as a parameter to the Scanner's constructor. nextInt() method available in the Scanner class returns the input entered by the user as an integer. The variables x and y are dynamically initialized. Output of the above program is:

```
Enter value of x:
10
Enter value of y:
20
Sum of x and y is: 30
```

Types of variables:

Based on the location where the variable is declared and how it is declared, variables in Java are divided into three types. They are:

- Instance variables
- Class variables
- Local variables

Instance Variables: A variable which is declared inside a class and outside all the methods, constructors and blocks is known as an instance variable.

Class Variables: A variable which is declared inside a class and outside all the methods, constructors, blocks and is marked as static is known as a class variable. More on static keyword in another article.

Local Variables: Any variable which is declared inside a class and inside a block, method or a constructor is known as a local variable.

Following Java example program demonstrates all the three kinds of variables:

```
class Sample
```

```
{
        int x, y;
        static int result;
        void add(int a, int b)
        {
                x = a;
                y = b;
                int sum = x+y;
                System.out.println("Sum = "+sum);
        }
  public static void main(String[] args)
        {
                Sample obj = new Sample();
                obj.add(10,20);
        }
}
```

In the above program x and y are instance variables, result is a class variable; a, b, sum and args are local variables.

One important point to remember is, every object maintains its own copy of each instance variable and a shared copy of each class variable.

Questions related to variables:

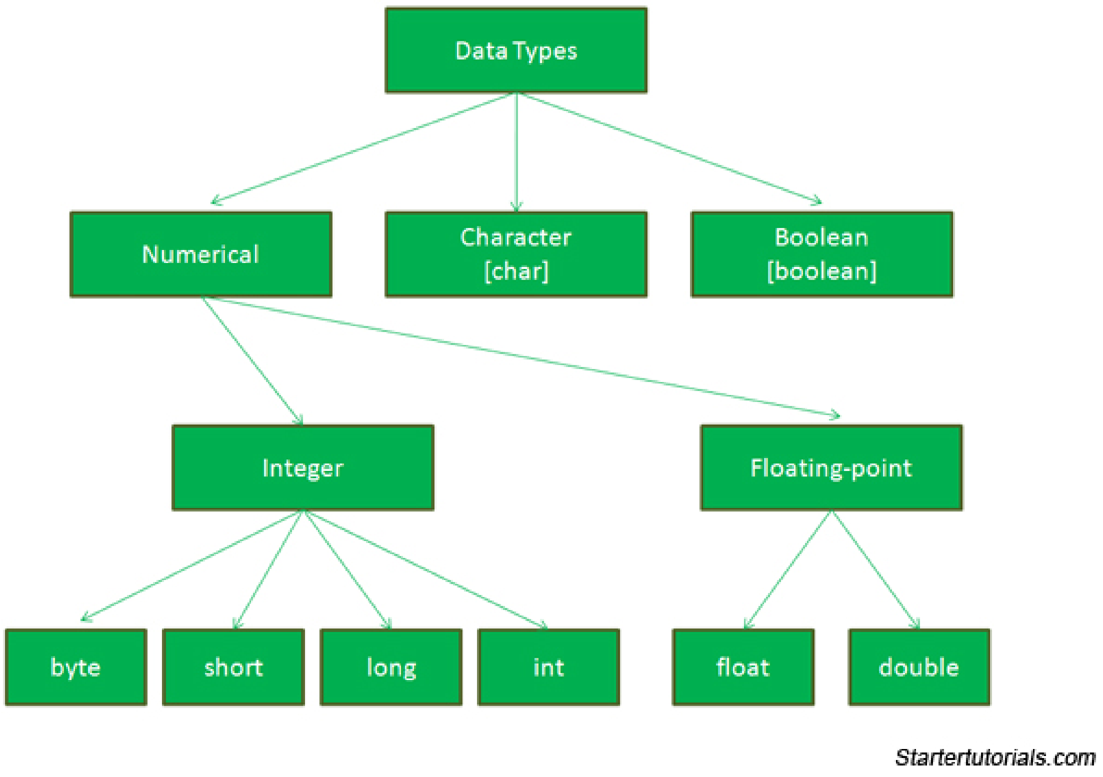
1. What is difference between instantiation and initialization?

A: Instantiation refers to creating an object for the class and initialization refers to assigning some value to a variable.

Data Types

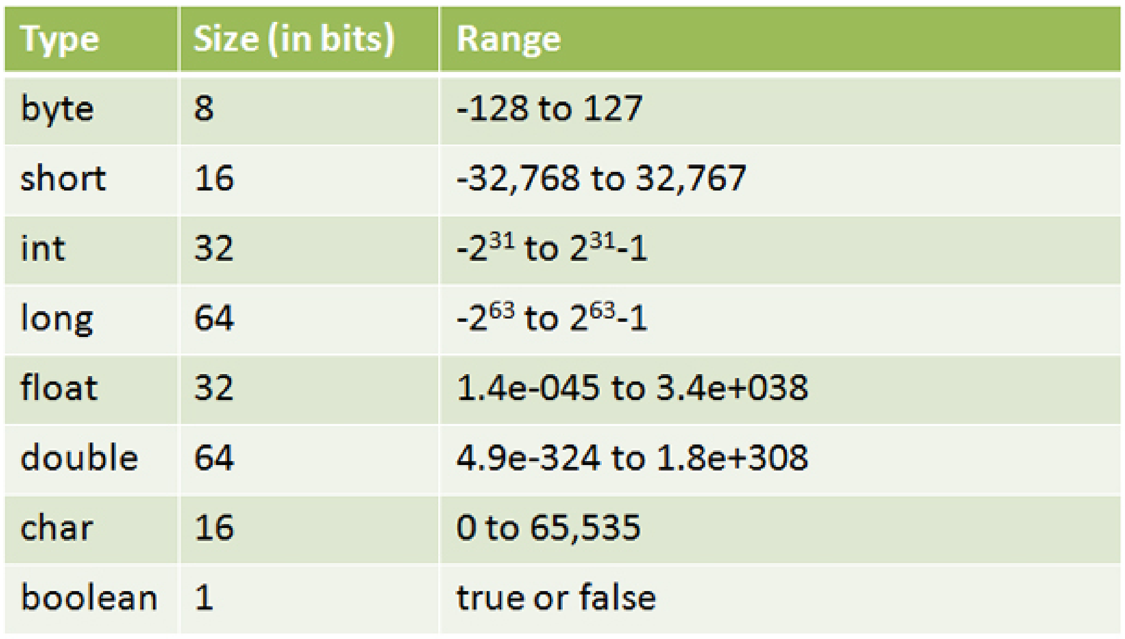
A data type specifies the type of value a variable can store or the type of an expression. Java is a strongly typed language means that you should specify the type of a variable before using it in a Java program.

There are eight primitive types in Java namely: byte, short, long, int, float, double, char and boolean. They can be categorized as shown below:



Also every class and interface existing in Java is also a type (predefined). By creating a class or an interface, you are creating a user defined type. Above eight data types are called as primitive in the sense they are not maintained as objects in memory and they can be used to create user defined types like classes.

Java designers has included these eight primitive data types only due to performance reasons (primitive types are faster). The size and range of values for each primitive data type is specified below:



Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Startertutorials.com

byte: The smallest integer type available in Java is byte. Its size is 8 bits and can store values within the range -128 to 127. byte data type can be useful while working with a stream of data over a network or a file. Declaring variables of the type byte is as shown below:

```
byte a, b;
```

short: Perhaps the least used integer type in Java is short. Its size is 16 bits and it can store values within the range -32,768 to 32,767. Declaring variables of the type short is as shown below:

```
short s;
```

int: *Most widely used type for working with integers in Java programs is int. Its size is 32 bits and it can store values within the range -2 billion to +2 billion. Variables of type int are frequently used in loops and for indexing arrays. Declaring variables of the type int is as shown below:*

```
int i, j;
```

long: *The size of long integer type is 64 bits and can store up to quite a large range of integer values. It is generally used in programs which work with large integer values. Declaring variables of the type long is as shown below:*

```
long a, b;
```

All the four integer types, byte, short, long and int are signed types. In Java, unsigned types are not supported.

float: *The type float is used to specify a single-precision value that uses 32 bits for storage. Single precision is faster on some processors and takes half as much space as double precision. The type float is used to work with fractional values where the precision is not that important. Declaring variables of type float is as shown below:*

```
float f;
```

double: *The type double is used to specify a double-precision value that uses 64 bits of storage. Double precision is faster on most of the modern processors which are optimized for mathematical calculations. The type double is used to work with larger fractional values and when the precision of the fractional value is important. Declaring variables of the type double is as shown below:*

```
double d;
```

char: *The data type which allows us to store characters is the char type. Unlike C and C++, the size of char type in Java is 16 bits. Java uses Unicode to represent characters. Unicode supports most of the international languages, whereas, C and C++ only supports ASCII.*

The char type can also be used to store integer values from 0 to 65,535. Operators allowed on integer types are also allowed on char type. Declaring a char type variable is as shown below:

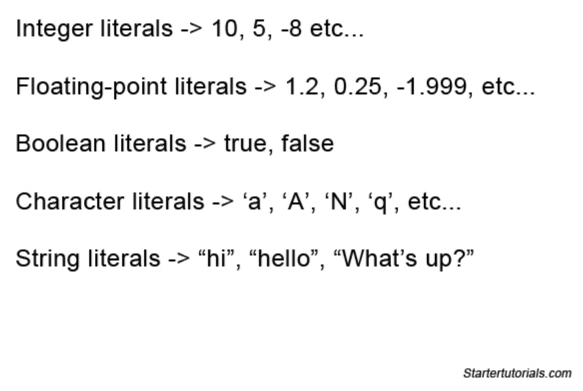
```
char ch;
```

boolean: *The boolean type in Java is used to store a logical value, either true or false. Size of boolean is 1 bit. Variables of the type boolean are used in control statements extensively. Declaring a variable of the type boolean is as shown below:*

```
boolean b;
```

Literals

Every Java primitive data type has its corresponding literals. A literal is a constant value which is used for initializing a variable or directly used in an expression. Below are some general examples of Java literals:



Integer literals -> 10, 5, -8 etc...

Floating-point literals -> 1.2, 0.25, -1.999, etc...

Boolean literals -> true, false

Character literals -> ‘a’, ‘A’, ‘N’, ‘q’, etc...

String literals -> “hi”, “hello”, “What’s up?”

Startertutorials.com

Integer Literals:

Integer literals are the most commonly used literals in a Java program. Any whole number value is an example of an integer literal. For example, 1, 10, 8343 are all decimal values i.e., of base 10. Java also supports other types of integer literals like values of base 8 (octal values) and base 16 (hexadecimal values).

An octal value contains numbers within the range 0 to 7. Octal literals in Java are preceded by a zero (0). So, to represent an octal 6 in Java, we should write 06.

An hexadecimal value contains numbers within the range 0 to 15 in which 10, 11, 12, 13, 14 and 15 will be represented using a, b, c, d, e and f or A, B, C, D, E and F respectively. Hexadecimal literals in Java are preceded by a zero-x (0x or 0X). So, to represent an hexadecimal 99 in Java, we should write 0x99 or 0X99.

Integer literals of type long should be explicitly marked with a lowercase l or an uppercase L at the end of the integer value. For example 9223372036854775807L is the largest long value.

New additions in Java SE 7:

In Java SE 7 binary literals were added. Now, programmers are able to assign numbers in base 2 (binary). A binary literal must be preceded by a zero-b (0b or 0B). For example, to assign the number 12 in binary we will write the literal as 0b1100 or 0B1100.

Along with binary literals, one more modification was made to how the literals can be written. From Java SE 7 onwards, programmers are allowed to specify one or more underscores (_) between the numbers in integer literals. For example, if we write 11_245_346, it will be interpreted as 11,245,346. Remember that an underscore can never occur at the beginning or ending of the literal. Multiple underscores are also supported. For example we can write an integer literal as 23_45 which will be interpreted as 2345.

The underscores are only for semantic (visual) aid. They are ignored by the compiler. Many people have the notion of writing a binary number as 4-bit groups. Underscores can help you to represent a binary literal as 4-bit groups. For example you can write a binary literal using underscores as shown below:

```
0b1010_1100_1011_1001
```

Floating-point Literals:

Floating-point literals are used to represent decimal numbers with a fractional component. Floating-point literals can be represented in either standard notation or in scientific notation.

In standard notation, a literal consists of a whole number followed by a decimal point followed by a fractional component. Examples of floating-point literals represented in standard notation are: 0.234, 435.655, 11.092 etc…

In scientific notation, a literal is represented as a floating-point number followed by an exponent. The exponent consists of e or E followed by a decimal number. Valid examples of floating-point literals represented in scientific notation are: 8.32E4, 0.23e+10, 32.431E-9 etc…

By default a floating-point literal is treated as a double value. To explicitly specify a literal as float you have to write a f or F at the end of the literal. Valid examples are: 1.55f, 0.24312F etc. You can explicitly specify that a literal is of the type double by writing a d or D at the end of the literal. But doing that is unnecessary.

As mentioned above in integer literals, from Java SE 7 onwards, underscores are also supported in floating point literals. Underscore can occur in between the digits before the decimal point or after the decimal point. Valid example is 12_32.2_323_87, which will be interpreted as 1232.232387 by the compiler.

Boolean Literals:

A variable of the type boolean can be initialized with one of the two boolean literals true or false. The boolean literals are not converted to integers i.e., true doesn't mean 1 and false doesn't mean 0. Boolean literals can also be used in expressions with boolean operators.

Character Literals:

A character literal in Java is specified in a pair of single quotes. All the visible ASCII characters can be directly written as 'a', 'x', '@' etc. As per the non-existing characters, you can use the escape sequences. For example, to enter a single quote we will write it as '\''. Here, backslash is used to specify an escape sequence. Similarly to specify a new line, we will write '\n'.

We can also specify integers to be stored as values into char variables. Java converts the integer into corresponding Unicode character. Also, we can specify character literals in octal and hexadecimal format. We specify octal value as a backslash followed by a three digit number. For example, '\141' is equivalent to the letter 'a'. We specify hexadecimal values as a backslash-u (\u) followed by a four digit number. For example, '\u0061' represents the letter 'a'.

String Literals:

Strings cannot be created by using any of the predefined primitive types. Java provides three class String, StringBuffer and StringBuidler for working with strings in Java programs.

A string literal is a string constant which can be created using double quotes as shown below:

```
“hi”
“hello John”
“\”The Game\””
```

The last string literal contains an escape sequence for double quotes.

Scope and Lifetime

Scope of a variable refers to in which areas or sections of a program can the variable be accessed and lifetime of a variable refers to how long the variable stays alive in memory.

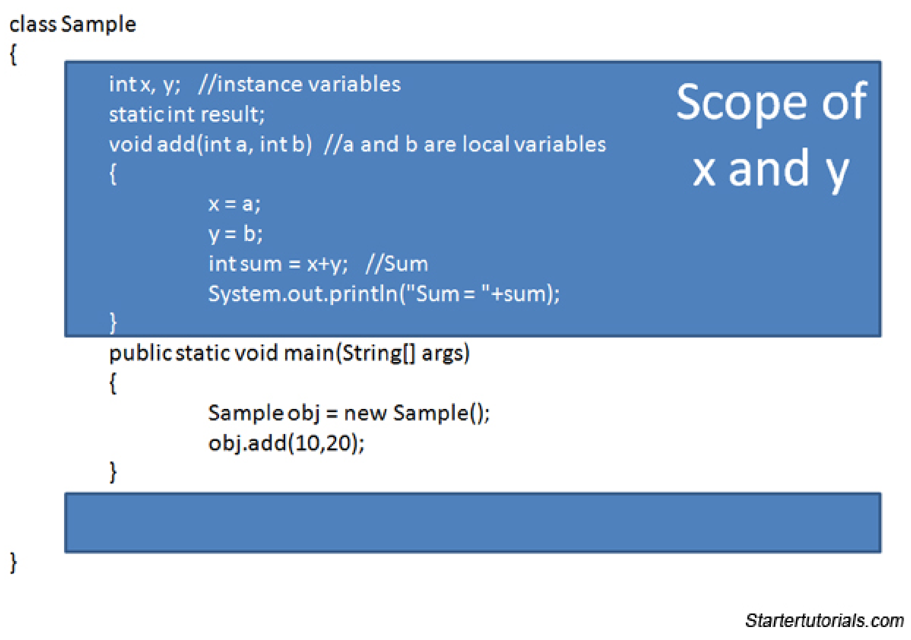
General convention for a variable's scope is, it is accessible only within the block in which it is declared. A block begins with a left curly brace { and ends with a right curly brace }.

As we know there are three types of variables: 1) instance variables, 2) class variables and 3) local variables, we will look at the scope and lifetime of each of them now.

Instance Variables:

A variable which is declared inside a class and outside all the methods and blocks is an instance variable.

General scope of an instance variable is throughout the class except in static methods. Lifetime of an instance variable is until the object stays in memory.

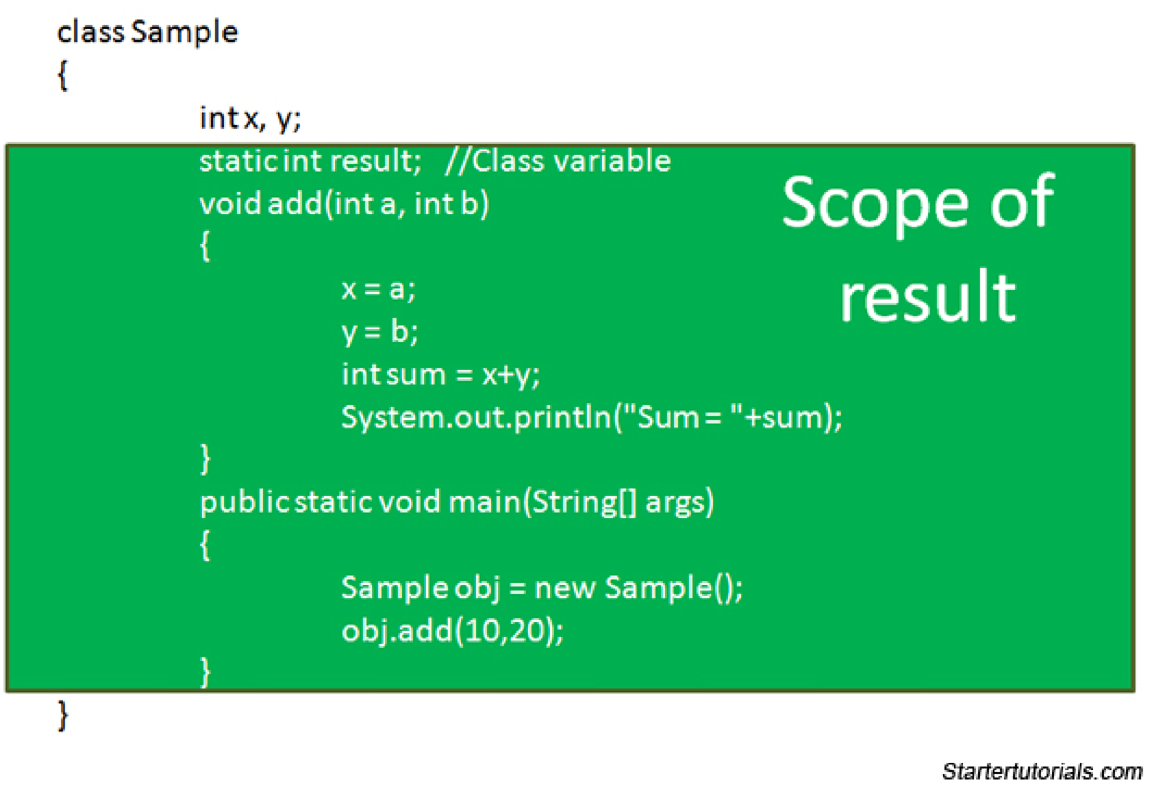


```
class Sample
{
    int x, y;   //instance variables
    static int result;
    void add(int a, int b)  //a and b are local variables
    {
            x = a;
            y = b;
            int sum = x+y;   //Sum
            System.out.println("Sum = "+sum);
    }
    public static void main(String[] args)
    {
            Sample obj = new Sample();
            obj.add(10,20);
    }
}
```

Class Variables:

A variable which is declared inside a class, outside all the blocks and is marked static is known as a class variable.

General scope of a class variable is throughout the class and the lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

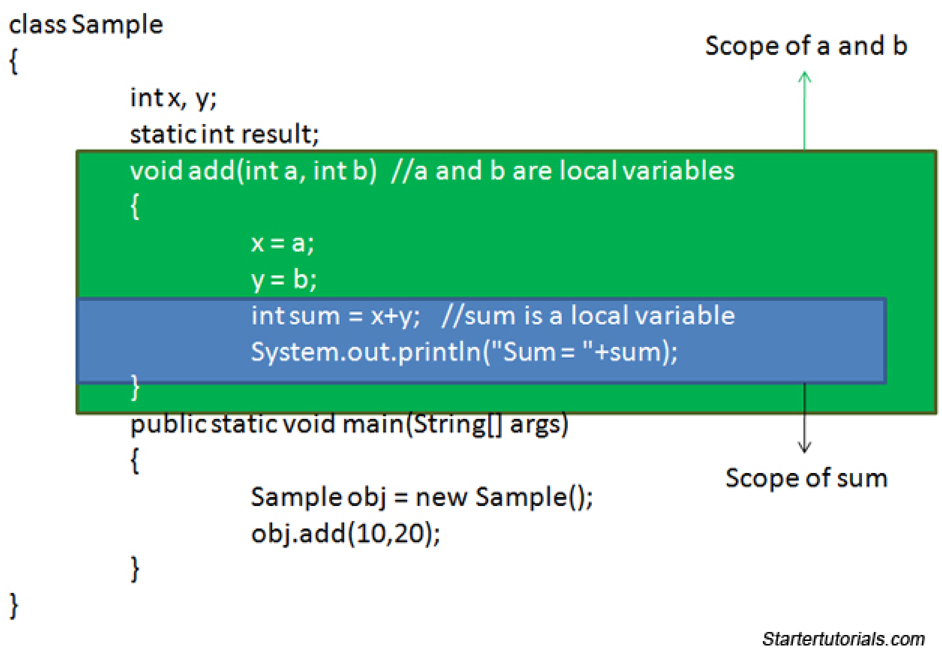


```
class Sample
{
        int x, y;
        static int result;   //Class variable
        void add(int a, int b)
        {
                x = a;
                y = b;
                int sum = x+y;
                System.out.println("Sum = "+sum);
        }
        public static void main(String[] args)
        {
                Sample obj = new Sample();
                obj.add(10,20);
        }
}
```

Local Variables:

All other variables which are not instance and class variables are treated as local variables including the parameters in a method.

Scope of a local variable is within the block in which it is declared and the lifetime of a local variable is until the control leaves the block in which it is declared.



```
class Sample
{
        int x, y;
        static int result;
        void add(int a, int b)  //a and b are local variables
        {
                x = a;
                y = b;
                int sum = x+y;   //sum is a local variable
                System.out.println("Sum = "+sum);
        }
        public static void main(String[] args)
        {
                Sample obj = new Sample();
                obj.add(10,20);
        }
}
```

Nested Scope:

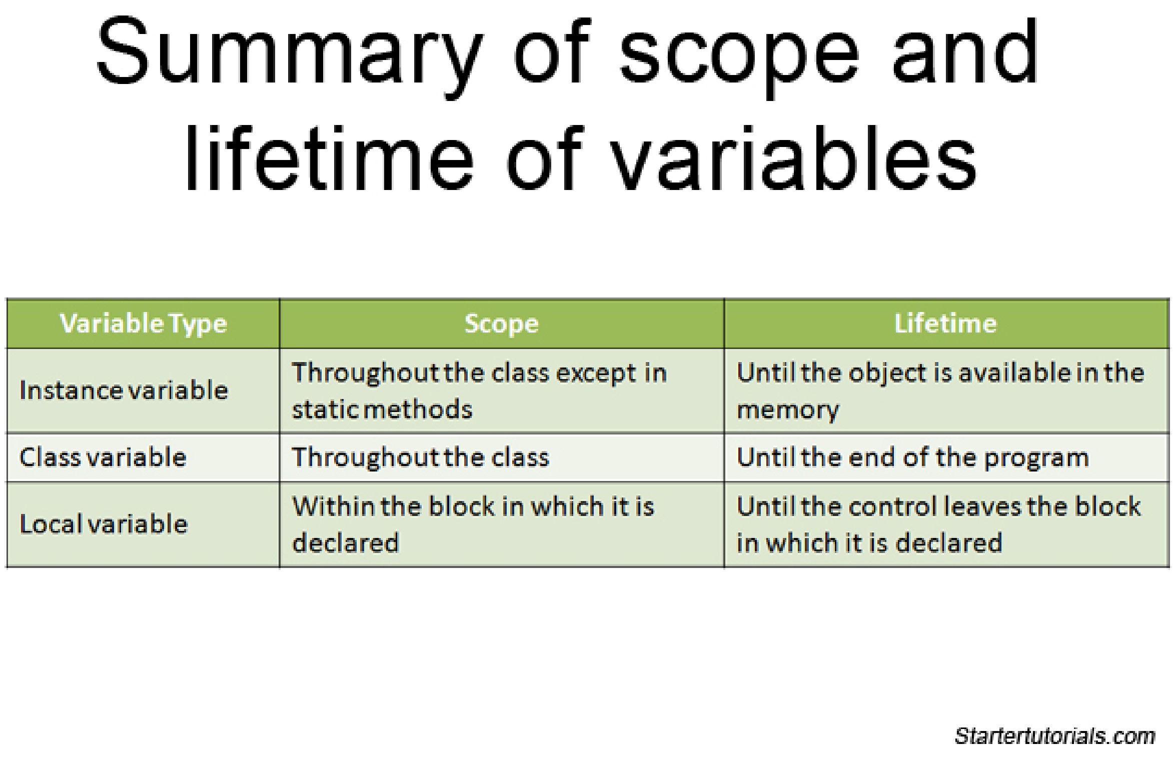
In Java, we can create nested blocks – a block inside another block. In case of nested blocks what is the scope of local variables?

All the local variables in the outer block are accessible within the inner block but vice versa is not true i.e., local variables within the inner block are not accessible in the outer block. Consider the following example:

```
class Sample
{
        public static void main(String[] args)
        {
                int x;
                //Begining of inner block
                {
                        int y = 100;
                        x = 200;
                        System.out.println("x = "+x);
                }
                //End of inner block
                System.out.println("x = "+x);
                y = 200;  //Error as y is not accessible in the outer block
        }
}
```

As you can see in the above program, line 14 generates an error as the variable y is not visible in the outer block and therefore cannot be accessed.

The summary of scope and lifetime of variables is as shown below:



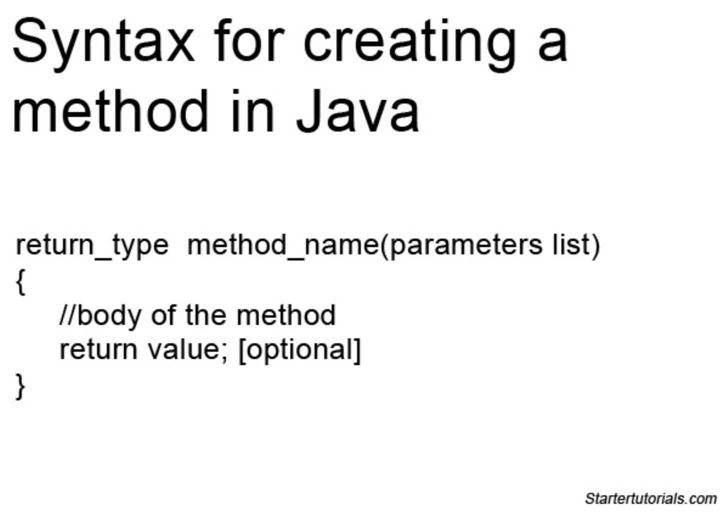
Summary of scope and lifetime of variables

Variable Type	Scope	Lifetime
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Startertutorials.com

Methods

Method: *A method is a piece of code to solve a particular task. A method is analogous to functions in C and C++. Methods are defined inside a class. The syntax for creating a method is as shown below:*



Syntax for creating a method in Java

```
return_type  method_name(parameters list)
{
    //body of the method
    return value; [optional]
}
```

Startertutorials.com

The return_type specifies the type of value that will be returned by the method. The name of the method is specified by method_name. Every parameter in the parameters list follows the below syntax:

```
datatype parameter_name
```

The parameters count can be zero or more based upon your requirements. The body of the method is represented using the braces { and }. Body of the method is also known as the method definition.

If a method does not return any value, its return type must be void. A method can return a single value back by using the return statement. Syntax for using return statement is as shown below:

```
return value;
```

As an example, let's create a method which takes side of a square as parameter and prints out the area of the square. The method is shown below:

```
void area(int s)
{
	System.out.println("Area of the square is: "+(s*s));
}
```

In the above method, s is a parameter and the return of the method is void as the method is not returning back any value.

Let's create another method which returns back the perimeter of the square. The method is shown below:

```
int perimeter()
{
	return 4*side;
}
```

In the above method, there are no parameters. The variable side is an instance variable of the class (see below). The method perimeter returns back an integer value using the return keyword. So the return type of the method is int.

A method in a class can be called by creating an object to that class and then use the dot operator followed by the method name and arguments if any. Syntax for a method call is as shown below:

```
object_reference.method_name(arguments);
```

Now, let's look at the entire Java program which contains the class Sqaure along with the above methods for computing area, perimeter and some other code:

```
class Square
{
        int side;
        void area(int s)
        {
                System.out.println("Area of the square is: "+(s*s));
        }
        int perimeter()
        {
                return 4*side;
        }
        int getSide()
        {
                return side;
        }
        public static void main(String[] args)
        {
                Square s1 = new Square();
                s1.side = 10;
                s1.area(s1.side); //Method call
                System.out.println("Perimeter of the square is: "+s1.perimeter()); //Method
                                                                                  call
                System.out.println("Length of the side is: "+s1.getSide()); //Method call
        }
}
```

Output for the above program is:

```
Area of the square is: 100
Perimeter of the square is: 40
Length of the side is: 10
```

A well written Java program is one which contains a set of classes that hides their fields (instance variables) from direct access and allows them to be accessed only through methods.

The values passed in a method call are known as arguments and the variables declared in the method to receive the values from the method call are known as parameters.

Operators

An operator allows the programmer or the computer to perform an operation on the operands. An operand can be a literal, variable or an expression.

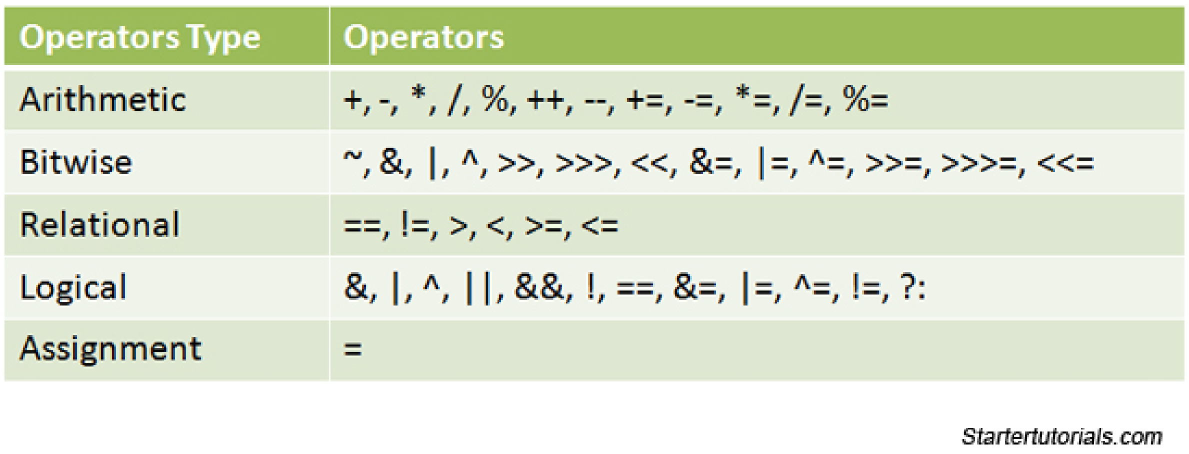
Operators can be divided along two dimensions: 1) number of operands on which the operator works and 2) type of operation the operator performs.

Based on number of operands, operators in Java can be divided into three types:

- Unary operator (works on single operand)
- Binary operator (works on two operands)
- Ternary operator (works on three operands)

Based on the type of operation performed, operators are divided into five categories as shown below:

OPERATORS IN JAVA



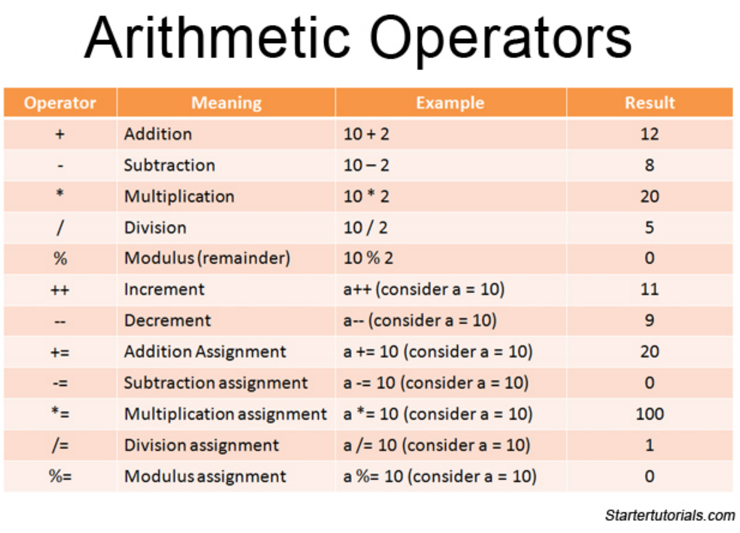
Operators Type	Operators
Arithmetic	+, -, *, /, %, ++, --, +=, -=, *=, /=, %=
Bitwise	~, &, |, ^, >>, >>>, <<, &=, |=, ^=, >>=, >>>=, <<=
Relational	==, !=, >, <, >=, <=
Logical	&, |, ^, ||, &&, !, ==, &=, |=, ^=, !=, ?:
Assignment	=

Startertutorials.com

Arithmetic Operators

Arithmetic operators are frequently used operators in the Java programs. They are used to perform basic mathematical operations like addition, subtraction, multiplication and division.

All the arithmetic operators and an example for each of them is provided below:



Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	10 + 2	12
-	Subtraction	10 – 2	8
*	Multiplication	10 * 2	20
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 2	0
++	Increment	a++ (consider a = 10)	11
--	Decrement	a-- (consider a = 10)	9
+=	Addition Assignment	a += 10 (consider a = 10)	20
-=	Subtraction assignment	a -= 10 (consider a = 10)	0
*=	Multiplication assignment	a *= 10 (consider a = 10)	100
/=	Division assignment	a /= 10 (consider a = 10)	1
%=	Modulus assignment	a %= 10 (consider a = 10)	0

Startertutorials.com

I think that the above table is self explanatory. The modulus (%) operator gives the remainder value of the division.

Let's focus on the increment (++) and decrement (–) operators. They are both unary operators, means, they operate on a single operand. Based on whether the increment or decrement operator is placed before or after the operand, they are divided into two types: 1) Pre increment or Pre decrement and 2) Post increment or Post decrement. They will look like as shown below:

```
++a (Pre increment)
– – a (Pre decrement)
a++ (Post increment)
a – – (Post decrement)
```

Pre and post increment or decrement have different behaviors when used in assignment expressions. Their behavior is explained in the below example:

```
int a,b;
a = 10;
b = a++;   //a value will be assigned to b and then a will be
incremented by 1. So, b value is 10
a = 10;
b = ++a;   //a value is incremented by 1 and then assigned to b. So,
b value is 11
```

The above example only shown post and pre increment. Similar is the behavior for post and pre decrement.

The operators +=, -=, *=, /= and %= are known as shorthand assignment operators. All they do is save you from typing two more extra characters.

Consider the following example which shows you how to use the shorthand assignment operator:

```
int a = 10;
a += 2;   //Same as writing a = a + 2

a += 1;   // Same as writing a++
```

Similar is the behavior of other compound assignment operators. Just experiment with them.

Arithmetic operators work only on numeric types and characters. The char is originally a sub type of int.

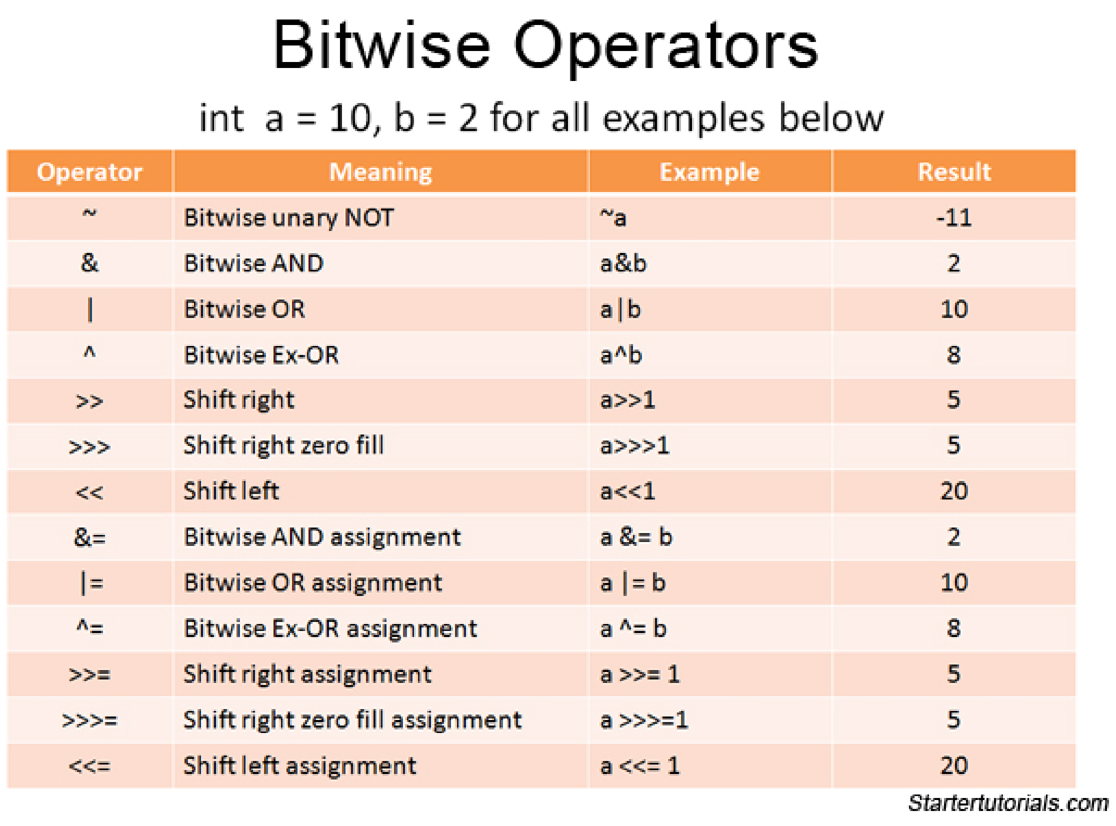
Bitwise Operators

Bitwise operators as the name implies works on bits of the value. These operators can be applied on integer types and character type.

Every integer value is represented as a combination of 0's and 1's (binary) inside the memory. For example the integer value 18 of type byte is represented in memory as shown below:

```
0 0 0 1 0 0 1 0
```

If you want to modify the value at bit-level then you can use the bit-wise operators. All the bit-wise operators and an example for each of them is provided below:



Bitwise Operators

int a = 10, b = 2 for all examples below

Operator	Meaning	Example	Result
~	Bitwise unary NOT	~a	-11
&	Bitwise AND	a&b	2
|	Bitwise OR	a|b	10
^	Bitwise Ex-OR	a^b	8
>>	Shift right	a>>1	5
>>>	Shift right zero fill	a>>>1	5
<<	Shift left	a<<1	20
&=	Bitwise AND assignment	a &= b	2
|=	Bitwise OR assignment	a |= b	10
^=	Bitwise Ex-OR assignment	a ^= b	8
>>=	Shift right assignment	a >>= 1	5
>>>=	Shift right zero fill assignment	a >>>=1	5
<<=	Shift left assignment	a <<= 1	20

Startertutorials.com

Bitwise logical operators ~, &, | and ^ operate on two bits as per the table shown below:

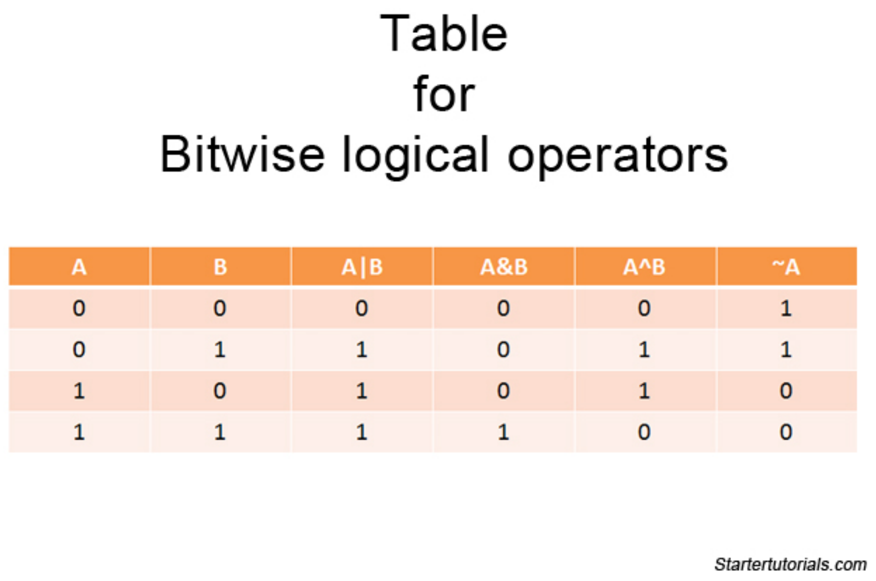


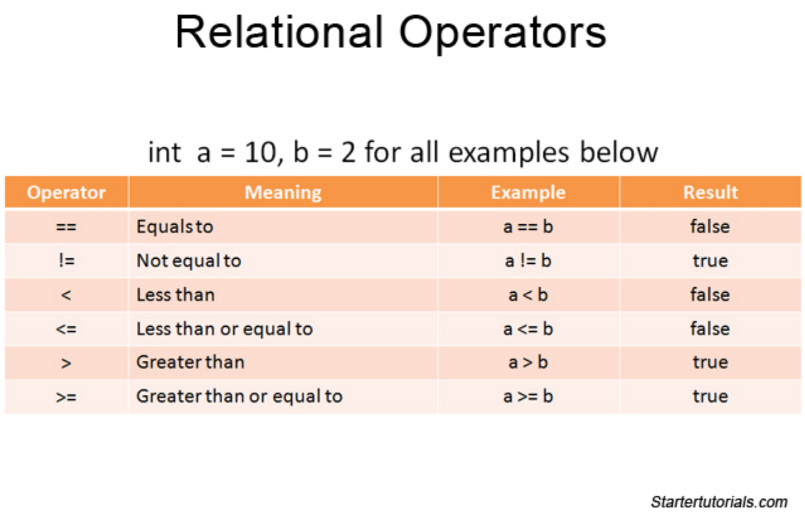
Table for Bitwise logical operators

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Startertutorials.com

Relational Operators

Relational operators are used to compare two values. The result of comparing two values is always a boolean value true or false. The relational operators available in Java and an example for each operator is shown below:



Relational Operators

int a = 10, b = 2 for all examples below

Operator	Meaning	Example	Result
==	Equals to	a == b	false
!=	Not equal to	a != b	true
<	Less than	a < b	false
<=	Less than or equal to	a <= b	false
>	Greater than	a > b	true
>=	Greater than or equal to	a >= b	true

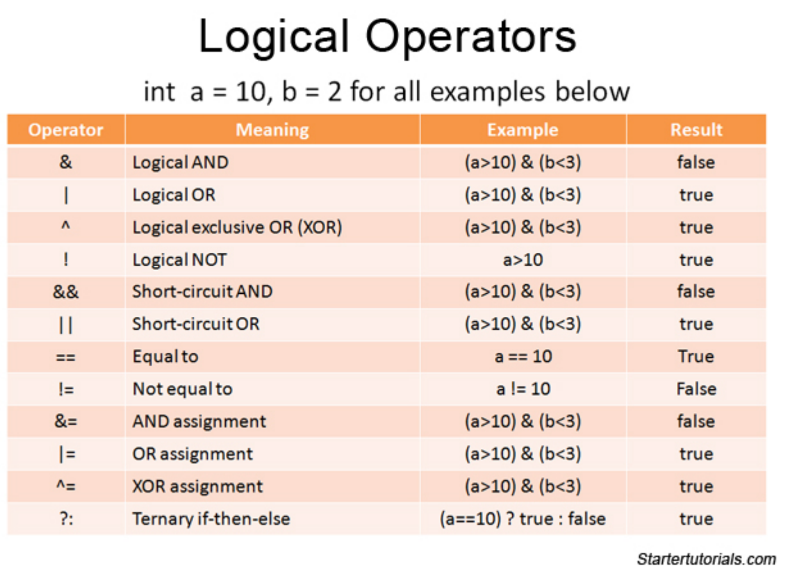
Startertutorials.com

Relational operators are generally used in control statements which will be explained in another article. Unlike C and C++, true doesn't refer any positive value other than zero and false doesn't refer zero. So, writing while(1) to repeat a loop continuously doesn't work in Java even though it works in C and C++.

Logical Operators

Logical operators are used to evaluate two boolean expressions or values and return the resultant boolean (truth) value. All the logical operators and an example for each logical operator is given below:

The logical AND, OR, XOR and NOT work in similar fashion as the bitwise AND, OR, XOR and NOT except that the operands for logical operators are boolean values. The truth table for the logical operators is given below:

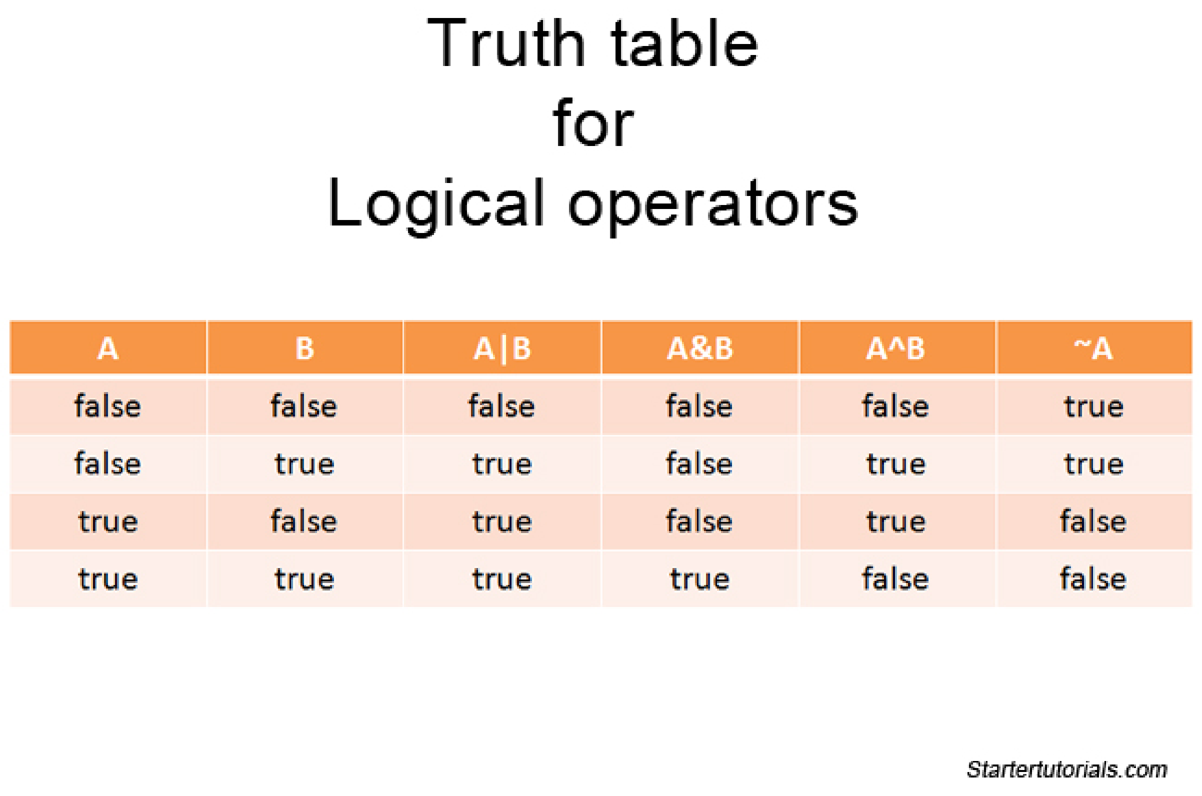


Logical Operators

int a = 10, b = 2 for all examples below

Operator	Meaning	Example	Result
&	Logical AND	(a>10) & (b<3)	false
|	Logical OR	(a>10) & (b<3)	true
^	Logical exclusive OR (XOR)	(a>10) & (b<3)	true
!	Logical NOT	a>10	true
&&	Short-circuit AND	(a>10) & (b<3)	false
||	Short-circuit OR	(a>10) & (b<3)	true
==	Equal to	a == 10	True
!=	Not equal to	a != 10	False
&=	AND assignment	(a>10) & (b<3)	false
|=	OR assignment	(a>10) & (b<3)	true
^=	XOR assignment	(a>10) & (b<3)	true
?:	Ternary if-then-else	(a==10) ? true : false	true

Startertutorials.com



Truth table
for
Logical operators

A	B	A|B	A&B	A^B	~A
false	false	false	false	false	true
false	true	true	false	true	true
true	false	true	false	true	false
true	true	true	true	false	false

Startertutorials.com

The short-circuit AND (&&) and short-circuit OR (||) are special operators in Java. Special in the sense, in the case of short-circuit AND (&&), if the left operand evaluates to false, the right operand is not evaluated and is ignored. Similarly for short-circuit OR (||), if the left operand evaluates to true, the right operand is not evaluated and is ignored.

Below is an example which demonstrates the use of short-circuit AND (&&) to eliminate a runtime exception caused due to dividing a number by zero:

```
if(denom != 0 && num/denom > 10)
{
        //code
}
```

In the above piece of code, if denom !=0 gives false, then the right side expression of && is not evaluated there by eliminating the chance of exception.

Assignment Operator

The assignment operator is represented by =. It is used for assigning a value to a variable as shown below:

```
int a = 10;
```

Conditional Operator

Java provides a special ternary operator that can be used as an replacement for if-then-else selection statement. The conditional operator is represented as ?:

The syntax for using the conditional operator is shown below:

```
expression 1 ? expression 2 : expression 3
```

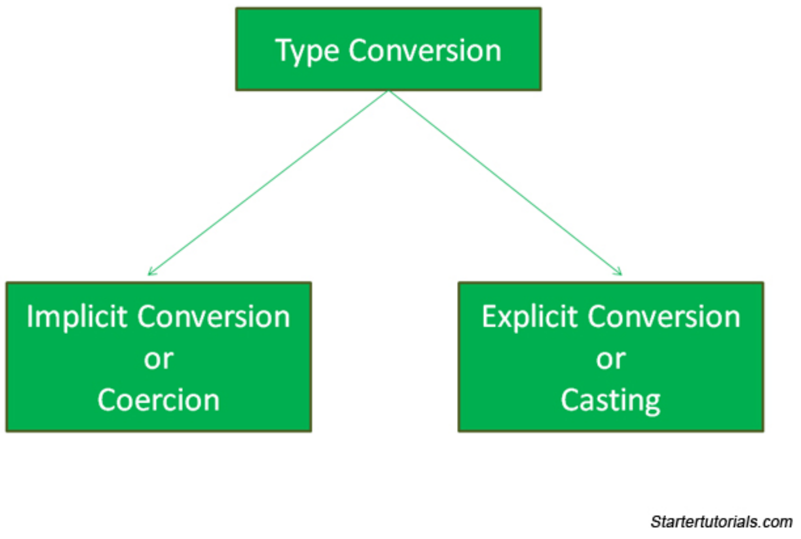
Always expression 1 must evaluate to a boolean value. If the result of expression 1 is true, then expression 2 is evaluated or else if expression 1 is false, expression 3 is evaluated and the resulting value will be returned. Consider the following example which demonstrates the use of conditional or ternary operator:

```
int a = 1, b = 1;
int result = (a==1 && b==1) ? 1 : 0;
```

The value stored in result variable will be 1 as the expression a==1&&b==1 evaluates to true.

Type Conversion and Casting

Type conversion is of types based on how the conversion is performed: 1) Implicit conversion (also known as automatic conversion or coercion), 2) Explicit conversion (also known as type casting).


Type Conversion
Implicit Conversion or Coercion
Explicit Conversion or Casting
Startertutorials.com

Implicit Conversion or Coercion

This type of conversion is performed automatically by Java due to performance reasons. Implicit conversion is not performed at all times. There are two rules to be satisfied for the conversion to take place. They are:

- The source and destination types must be compatible with each other.
- The size of the destination type must be larger than the source type.

For example, Java will automatically convert a value of byte into int type in expressions since they are both compatible and int is larger than byte type. Since a smaller range type is converted into a larger range type this conversion is also known as widening conversion. Characters can never be converted to boolean type. Both are incompatible.

Explicit Conversion or Casting

There may be situations where you want to convert a value having a type of size less than the destination type size. In such cases Java will not help you. You have do it on your own explicitly. That is why this type of conversion is known as explicit conversion or casting as the programmer does this manually.

Syntax for type casting is as shown below:

```
(destination-type) value
```

An example for type casting is shown below:

```
int a = 10;
byte b = (int) a;
```

In the above example, I am forcing an integer value to be converted into a byte type. For type casting to be carried out both the source and destination types must be compatible with each other. For example, you can't convert an integer to boolean even if you force it.

In the above example, size of source type int is 32 bits and size of destination type byte is 8 bits. Since we are converting a source type having larger size into a destination type having less size, such conversion is known as narrowing conversion.

A type cast can have unexpected behavior. For example, if a double is converted into an int, the fraction component will be lost.

Type promotion in expressions

In addition to assignment statements, there is another place where type conversion can occur. It is in expressions. An expression is a collection of variables, values, operators and method calls which evaluate to a single value.

Type promotion rules of Java for expressions are listed below:

- *All char, short and byte values are automatically promoted to int type.*
- *If at least one operand in an expression is a long type, then the entire expression will be promoted to long.*
- *If at least one operand in an expression is a float type, then the entire expression will be promoted to float.*
- *If at least one operand in an expression is a double type, then the entire expression will be promoted to double.*

To understand the above type promotion rules let's consider the following example of expression evaluation:

```
class Sample
{
	public static void main(String[] args)
	{
		int i = 1000000;
		char c = 'z';
		short s = 200;
		byte b = 120;
		float f = 3.45f;
		double d = 1.6789;
		double result = (f * b) + (i / c) - (d * s);
		System.out.println("Result = "+result);
	}
}
```

Output of the above program is: Result = 8274.22

In the above program the expression is (f * b) + (i / c) – (d * s). In the first sub expression (f * b), as one operand is float, the result of the expression will be a float. In the second sub expression (i / c), char type will be promoted to int and the result of the expression will be an int. In the third sub expression (d * s), as one operand is double, the result of the expression is a double.

So the results of the sub expressions are float, int and double. Since one of them is a double, the result of the entire expression is promoted to a double.

Expressions

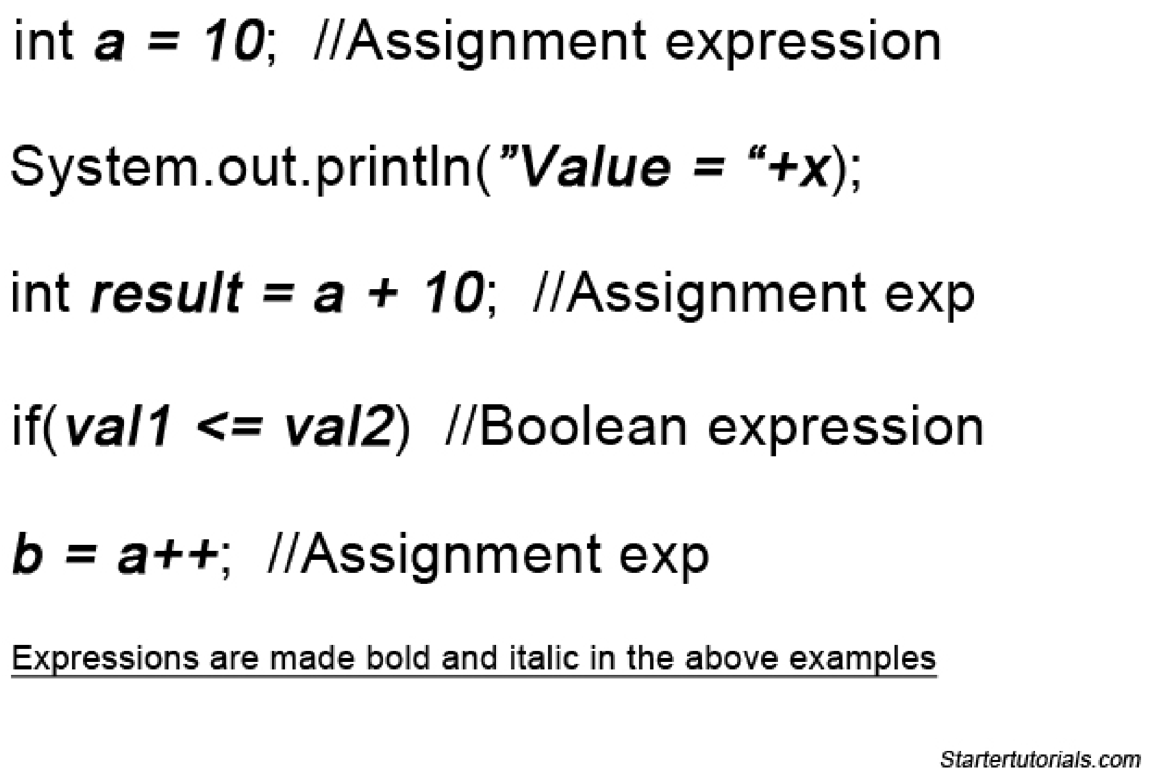
An expression is a construct which is made up of literals, variables, method calls and operators following the syntax of Java. Every expressions consists of at least one operator and an operand. Operand can be either a literal, variable or a method invocation.

Following (figure in next page) are some of the examples for expressions in Java:

How expressions are evaluated?

It is common for an expression to have more than one operator. For example, consider the below example:

(20 * 5) + (10 / 2) – (3 * 10)



```
int a = 10;  //Assignment expression

System.out.println("Value = "+x);

int result = a + 10;  //Assignment exp

if(val1 <= val2)  //Boolean expression

b = a++;  //Assignment exp
```

Expressions are made bold and italic in the above examples

Startertutorials.com

So, how is the above expression evaluated? Expression evaluation in Java is based upon the following concepts:

- *Type promotion rules*
- *Operator precedence*
- *Associativity rules*

I have already explained the type promotion rules here. Since all are integer values, there is no need to worry about type promotion rules here. We will now concentrate on operator precedence and associativity rules.

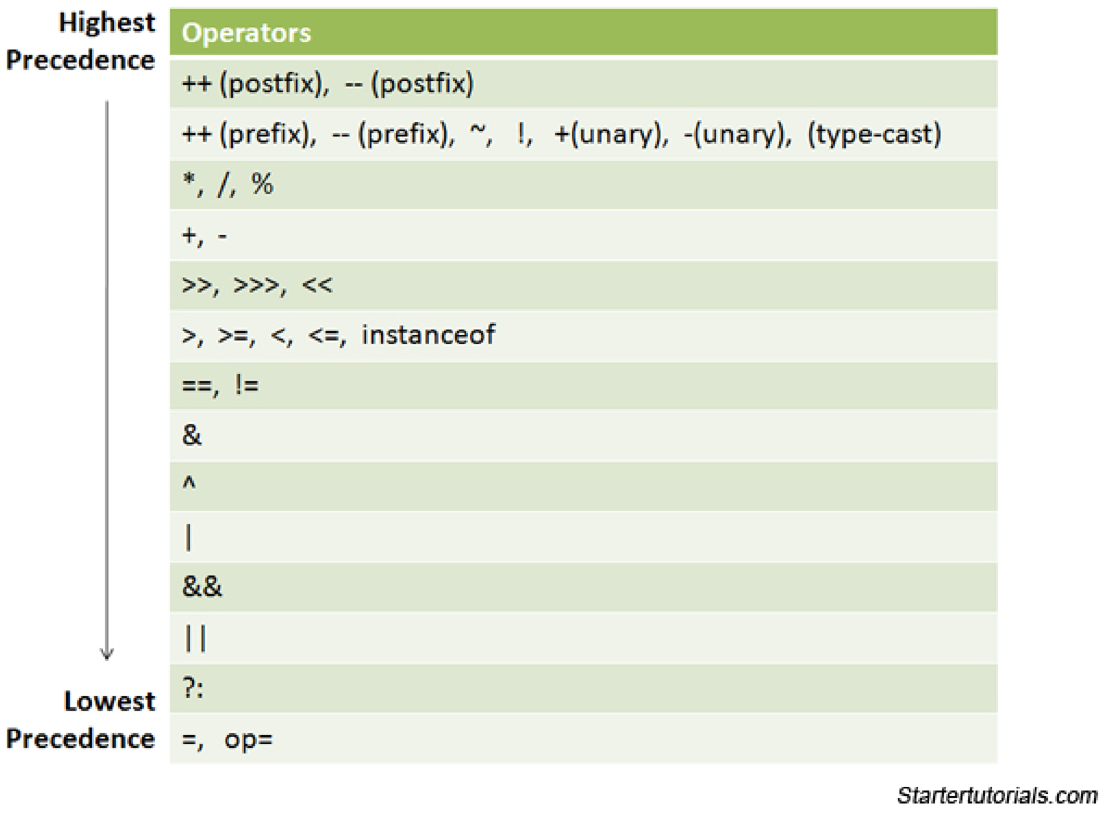
Operator precedence:

All the operators in Java are divided into several groups and are assigned a precedence level. The operator precedence chart for the operators in Java is shown below (figure in next page):

Now let's consider the following expression:

10 – 2 * 5

One will evaluate the above expression normally as, 10-2 gives 8 and then 8*5 gives 40. But Java evaluates the above expression differently. Based on the operator precedence chart shown above, * has higher precedence than +. So, 2* 5 is evaluated first which gives 10 and then 10 – 10 is evaluated which gives 0.

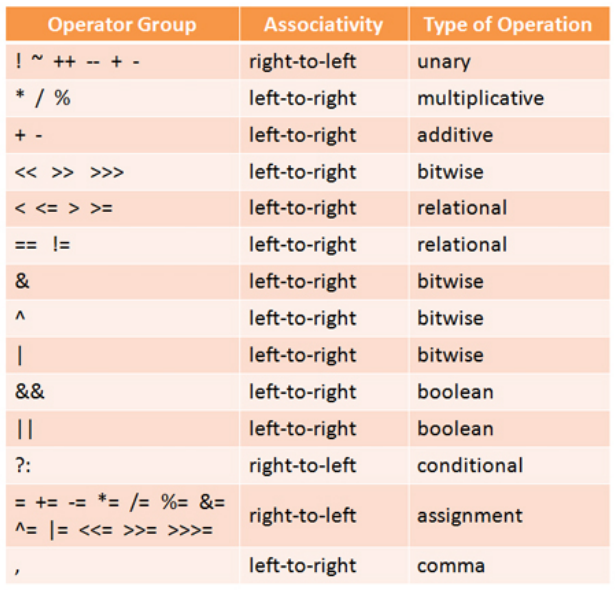

Highest Precedence
Lowest Precedence
Startertutorials.com

Operators
++ (postfix), -- (postfix)
++ (prefix), -- (prefix), ~, !, +(unary), -(unary), (type-cast)
*, /, %
+, -
>>, >>>, <<
>, >=, <, <=, instanceof
==, !=
&
^
|
&&
||
?:
=, op=

What if the expression contains two or more operators from the same group? Such ambiguities are solved using the associativity rules.

Associativity rules:

When an expression contains operators from the same group, associativity rules are applied to determine which operation should be performed first. The associativity rules of Java are shown below:



Operator Group	Associativity	Type of Operation
! ~ ++ -- + -	right-to-left	unary
* / %	left-to-right	multiplicative
+ -	left-to-right	additive
<< >> >>>	left-to-right	bitwise
< <= > >=	left-to-right	relational
== !=	left-to-right	relational
&	left-to-right	bitwise
^	left-to-right	bitwise
|	left-to-right	bitwise
&&	left-to-right	boolean
||	left-to-right	boolean
?:	right-to-left	conditional
= += -= *= /= %= &= ^= |= <<= >>= >>>=	right-to-left	assignment
,	left-to-right	comma

Now, let's consider the following expression:

10-6+2

In the above expression, the operators + and – both belong to the same group in the operator precedence chart. So, we have to check the associativity rules for evaluating the above expression. Associativity rule for + and – group is left-to-right i.e, evaluate the expression from left to right. So, 10-6 is evaluated to 4 and then 4+2 is evaluated to 6.

Use of parenthesis in expressions

Let's look at our original expression example:

(20 * 5) + (10 / 2) – (3 * 10)

You might think that, what is the need of parenthesis (and) in the above expression. The reason I had included them is, parenthesis have the highest priority (precedence) over all the operators in Java.

So, in the above expression, (20*5) is evaluated to 100, (10/2) is evaluated to 5 and (3*10) is evaluated to 30. Now, our intermediate expression looks like:

```
100 + 5 – 30
```

Now, we can apply the associativity rules and evaluate the expression. The final answer for the above expression is 75.

There is another popular use of parenthesis. We will use them in print statements. For example consider the following piece of code:

```
int a=10, b=20;
System.out.println("Sum of a and b is: "+a+b);
```

One might think that the above code will produce the output: "Sum of a and b is: 30". The real output will be:

```
Sum of a and b is: 1020
```

Why? Because, when one of the operand inside a print statement is a string, the + operator acts as a concatenation operator. To make it behave as an arithmetic operator we should enclose a+b is parenthesis as shown below:

```
int a=10, b=20;
System.out.println("Sum of a and b is: "+(a+b));
```

Now (a+b) is evaluated first and then concatenated to the rest.

Control Statements

What are control statements?

In a Java program, we already know that execution starts at main method. The first statement inside the body of main method is executed and then the next statement and so on. In general, the statements execute one after another in a linear fashion.

What if we want the statements to execute in a non-linear fashion i.e., skip some statements or repeat a set of statements or select one set of statements among several alternatives based on the outcome of evaluating an expression or the value of a variable?

To solve the above mentioned problems, every programming language provides control statements which allow the programmers to execute the code in a non-linear fashion.

Types or categories of control statements in Java

In Java, control statements can be categorized into the following categories:

- Selection statements (if, switch)
- Iteration statements (while, do-while, for, for-each)
- Jump statements (break, continue, return)

Selection statements

As the name implies, selection statements in Java executes a set of statements based on the value of an expression or value of a variable. A programmer can write several blocks of code and based on the condition or expression, one block can be executed. Selection statements are also known as conditional statements or branching statements. Selection statements provided by Java are if and switch.

if statement:

The if statement is used to execute a set of statements based on the boolean value returned by an expression. Syntax of if statement is as shown below:

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

If you want to execute only one statement in the if block, you can omit the braces and write it as shown below:

```
if(condition/expression)
        statement;
```

In the above syntax, the set of statements are executed only when the condition or expression evaluates to true. Otherwise, the statement after the if block is executed.

Let's consider the following example which demonstrates the use of if statement:

```
int a = 10;
if(a &lt; 10)
        System.out.println("a is less than 10");
```

The output for the above piece of code will be a blank screen i.e. no output because the condition a < 10 returns false and the print statement will not be executed.

In the above piece of code, instead of displaying nothing when the condition returns false, we can show an appropriate message. This can be done using the else statement. Remember that you can't use else without using if statement. By using the else statement our previous example now becomes as follows:

```
int a = 10;
if(a < 10)
        System.out.println("a is less than 10");
else
        System.out.println("a is greater than or equal to 10");
```

Nested if statement:

If you want to test for another condition after the initial condition available in the if statement, it is common sense to write another if statement. Such nesting of one if statement inside another if statement is called a nested if statement. Syntax of nested if statement is as shown below:

```
if(condition)
{
        if(condition)
        {
                if(condition)
                {
                        ...
                }
        }
}
```

We can also combine multiple conditions into a single expression using the logical operators. As an example for nested if, let's look at a program for finding the largest of the three numbers a, b and c. Below we will look at a code fragment which shows logic for finding whether a is the greatest or not:

```
if(a > b)
{
        if(a > c)
        {
                System.out.println("a is the largest number");
        }
}
```

As mentioned above we can convert a nested if into a simple if by combining multiple condition into a single condition by using the logical operators as shown below:

```
if(a > b && a > c)
{
        System.out.println("a is the largest number");
}
```

if-else-if Ladder:

The if-else-if ladder is a multi-way decision making statement. If you want to execute one code segment among a set of code segments, based on a condition, you can use the if-else-if ladder. The syntax is as shown below:

```
if(condition)
{
        Statements;
}
else if(condition)
{
        Statements;
}
else if(condition)
{
        Statements;
}
…
else
{
        Statements;
}
```

Looking at the above syntax, you can say that only one of the blocks execute based on the condition. When all conditions fail, the else block will be executed. As an example for if-else-if ladder, let's look at a program for finding whether the given character is a vowel (a, e, i, o, u) or not:

```
char ch;
ch = 'e';   //You can also read input from the user
if(ch == 'a')
{
        System.out.println("Entered character is a vowel");
}
else if(ch=='e')
{
        System.out.println("Entered character is a vowel");
}
else if(ch=='i')
{
        System.out.println("Entered character is a vowel");
}
else if(ch=='o')
{
        System.out.println("Entered character is a vowel");
}
else if(ch=='u')
{
        System.out.println("Entered character is a vowel");
}
else
{
        System.out.println("Entered character is not a vowel");
}
```

In the above program, the else block serves as a default block that is to be executed in case if the entered character is not a, e, i, o or u.

switch Statement:

The switch statement is another multi-way decision making statement. You can consider the switch as an alternative for if-else-if ladder. Every code segment written using a switch statement can be converted into an if-else-if ladder equivalent.

Use switch statement only when you want a literal or a variable or an expression to be equal to another value. The switch statement is simple than an equivalent if-else-if ladder construct. You cannot use switch to test multiple conditions using logical operators, which can be done in if-else-if ladder construct. Syntax for switch statement is as shown below:

```
switch(expression)
{
        case label1:
                Statements;
                break;
        case label2:
                Statements;
                break;
        case label3:
                Statements;
                break;
        ...
        default:
                Statements;
                break;
}
```

Some points to remember about switch statement:

- The expression in the above syntax can be a byte, short, int, char or an enumeration. From Java 7 onwards, we can also use Strings in a switch
- case is a keyword use to specify a block of statements to be executed when the value of the expression matches with the corresponding label
- The break statement at the end of each case is optional. If you don't use break at the end of a case, all the subsequent cases will be executed until a break statement is encountered or the end of the switch statement is encountered.
- The default block is also optional. It is used in a switch statement to specify a set of statements that should be executed when none of the labels match with the value of the expression. It is a convention to place the default block at the end of the switch statement but not a rule.

As an example for switch statement, let's consider the vowel's example discussed above:

```
char ch;
ch = ‘u’;  //You can also read input from the user
switch(ch)
{
        case ‘a’:
                System.out.println(“Entered character is a vowel”);
                break;
        case ‘e’:
                System.out.println(“Entered character is a vowel”);
                break;
        case ‘i':
                System.out.println(“Entered character is a vowel”);
                break;
        case ‘o’:
                System.out.println(“Entered character is a vowel”);
                break;
        case ‘u’:
                System.out.println(“Entered character is a vowel”);
                break;
        default:
                System.out.println(“Entered character is not a vowel”);
                break;  //There is no need of this break. You can omit this if you want
}
```

Iteration Statements

While selection statements are used to select a set of statements based on a condition, iteration statements are used to repeat a set of statements again and again based on a condition for finite or infinite number of times.

What is the need for iteration statements or looping statements? To answer this, let's consider an example. Suppose you want to read three numbers from user. For this you might write three statements for reading input. In another program you want to read ten numbers. For this you write ten statements for reading input. Yet in another program you want to read ten thousand numbers as input. What do you do? Even copying and pasting the statements to read input takes time. To solve this problem, Java provides us with iteration statements.

Iteration statements provided by Java are: for, while, do-while and for-each

while Statement:

while is the simplest of all the iteration statements in Java. Syntax of while loop is as shown below:

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

As long as the condition is true, the statements execute again and again. If the statement to be executed in any loop is only one, you can omit the braces. As the while loop checks the condition at the start of the loop, statements may not execute even once if the condition fails. If you want to execute the body of a loop atleast once, use do-while loop.

Let's see an example for reading 1000 numbers from the user:

```
int i;
i = 0;
while(i < 1000)
{
        //Code for reading input...
        i++;
}
```

The above loop works in this way. First i is initialized to zero, then condition i < 1000 is evaluated which is true. So the statements for reading input are executed and finally i is incremented by 1 (i++). Again the condition is evaluated and so on. When i value becomes 1000, condition fails and control exits the loop and goes to the next line after the while statement.

do-while Statement:

do-while statement is similar to while loop except that the condition is checked after executing the body of the loop. So, the do-while loop executes the body of the loop atleast once. Syntax of do-while is as shown below:

```
do
{
        Statements;
}while(condition);
```

This loop is generally used in cases where you want to prompt the user to ask whether he/she would like to continue or not. Then, based on user's input, the body of the loop will be executed again or else the control quits the loop.

Let's consider an example where the body of the loop will be executed based on the user's input:

```
char ch;
do
{
        //Statements;
        System.out.println("Do you want to continue? Enter Y or N);
        //Statement for reading the character from the user
}while(ch == 'Y');
```

In the above code segment, the body of the do-while loop executes again and again as long as the user inputs the character Y when prompted.

for Statement:

The for statement might look busy but provides more control than other looping statements. In a for statement, initialization of the loop control variable, condition and modifying the value of the loop control variable are combined into a single line. The syntax of for statement is as shown below:

```
for(initialization; condition; iteration)
{
        Statements;
}
```

In the above syntax, the initialization is an assignment expression which initializes the loop control variable and/or other variables. The initialization expression evaluates only once. The condition is a boolean expression. If the condition evaluates to true, the body of the loop is executed. Else, the control quits the loop. Every time after the body executes, the iteration expression is evaluated. Generally, this is an increment or decrement expression which modifies the value of the control variable. All the three parts i.e. initialization, condition and iteration are optional.

Let's consider an example code segment which prints the numbers from 1 to 100:

```
for(int i = 1; i <= 100; i++)
{
        System.out.println("i = " + i );
}
```

In the above code segment, int i = 1 is the initialization expression, i <= 100 is the condition and i++ is the iteration expression.

We can create an infinite for loop as shown below:

```
for( ; ; )
{
        Statements;
}
```

for-each Statement:

The for-each statement is a variation of the traditional for statement which has been introduced in JDK 5. The for-each statement is also known as enhanced for statement. It is a simplification over the for statement and is generally used when you want to perform a common operation sequentially over a collection of values like an array etc... The syntax of for-each statement is as shown below:

```
for(type  var : collection)
{
        Statements;
}
```

The data type of var must be same as the data type of the collection. The above syntax is can be read as, for each value in collection, execute the statements. Starting from the first value in the collection, each value is copied into var and the statements are executed. The loop executes until the values in the collection completes.

Let's consider a code segment which declares an array containing 10 values and prints the sum of those 10 values using a for-each statement:

```
int[ ] array = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for(int x : array)
{
        sum += x;
}
System.out.println("Sum is: "+sum);
```

The equivalent code segment for the above code using a for loop is as shown below:

```
int[ ] array = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for(int x = 0; x < 10; x++)
{
        sum += array[x];
}
System.out.println("Sum is: "+sum);
```

Nested Loops:

A loop inside another loop is called as a nested loop. For each iteration of the outer loop the inner loop also iterates. Syntax of a nested loop is shown below:

```
outer loop
{
        inner loop
        {
                Statements;
        }
}
```

Let's consider an example which demonstrates a nested loop:

```
for(int i = 0; i < 5; i++)
{
        for(int j = 0; j <= i; j++)
        {
                System.out.println("*");
        }
}
```

Output for the above code segment is:

```
*
**
***
****
*****
```

Jump Statements

As the name implies, jump statements are used to alter the sequential flow of the program. Jump statements supported by Java are: break, continue and return.

break Statement:

The break statement has three uses in a program. First use is to terminate a case inside a switch statement, second use to terminate a loop and the third use is,break can be used as a sanitized version of goto which is available in other languages. First use is already explained above. Now we will look at the second and third use of break statement.

The break statement is used generally to terminate a loop based on a condition. For example, let's think that we have written a loop which reads data from a heat sensor continuously. When the heat level reaches to a threshold value, the loop must break and the next line after the loop must execute which might be a statement to raise an alarm. In this case the loop will be infinite as we don't know when the value from the sensor will match the threshold value. General syntax of the break statement inside a loop is shown below:

```
Loop
{
        //Statements;
        if(condition)
                break;
        //Statements after break;
}
```

As an example to demonstrate the break statement inside a loop, let's look at the following code segment:

```
for(i = 1; i < 10; i++)
{
        System.out.println(i);
        if(i == 5)
                break;
```

}

Output for the above code segment is:

```
1
2
3
4
5
```

When used inside a nested loop, break statement makes the flow of control to quit the inner loop only and not the outer loop too.

Another use of the break statement is, it can be used as an alternative to goto statement which is available in other programming languages like C and C++. General syntax of break label statement is as shown below:

```
label1:
{
        Statements;
        label2:
        {
                Statements;
                if(condition)
                        break label1;
        }
        Statements;
}
```

In the above syntax, when the condition becomes true, control transfers from that line to the line which is available after the block labeled label1. This form of break is generally used in nested loops in which the level of nesting is high and you want the control to jump from the inner most loop to the outermost loop. This form of break can be used in all loops or in any other block. Syntax for writing a label is, any valid identifier followed by a colon (:).

continue Statement:

Unlike break statement which terminates the loop, continue statement is used to skip rest of the statements after it for the current iteration and continue with the next iteration. Syntax of continue is as shown below:

```
Loop
{
        //Statements;
        if(condition)
                continue;
        //Statements after continue;
}
```

Like break statement, continue can also be used inside all loops in Java.

As an example to demonstrate the continue statement inside a loop, let's look at the following code segment:

```
for(i = 1; i < 5; i++)
{
        if(i == 3)
                continue;
        System.out.println(i);
}
```

Output for the above code segment is:

```
1
2
```

```
4
5
```

When the value of i becomes 3, continueis executed and the print statement gets skipped.

return Statement:

The return statement can only be used inside methods which can return control or a value back to the calling method. The return statement can be written at any line inside the body of the method. Common convention is to write the return statement at the end of the method's body. Let's consider a small example to demonstrate the use of return statement:

```
void fact(int n)
{
        if(n == 0) return 1;
        else if(n == 1) return 1;
        else return fact(n)*fact(n-1);
}
```

Arrays

An array is a set of homogeneous variables sharing the same name. Homogeneous in the sense, all the elements in the array must be of the same data type. All the elements of an array are stored side-by-side in the memory. Arrays are generally used when there is a need to manipulate a set of logically related data elements. Individual elements of an array are referred using an index value or also known as subscript. The index value of an array always starts from zero.

One-Dimensional Arrays

An array can be one-dimensional, two-dimensional or multi-dimensional. Creation of a one-dimensional array is a two step process. First, we have to create an array variable and second; we have to use the new operator to allocate memory for the array elements.

Syntax for declaring an array variable is as shown below:

```
data-type array-name[ ];
```

Example for declaring an array is shown below:

```
int a[ ];
```

In the above example, int is the data type and a is the array name. One pair of square brackets ([ ]) indicates that the array is one-dimensional.

Syntax for allocating memory for the array elements is as shown below:

```
array-name = new data-type[size];
```

Example for allocating memory for the array elements is shown below:

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

In the above example, new is a keyword which allocates memory for 10 elements of the type int. In Java, all the arrays are allocated memory dynamically at runtime.

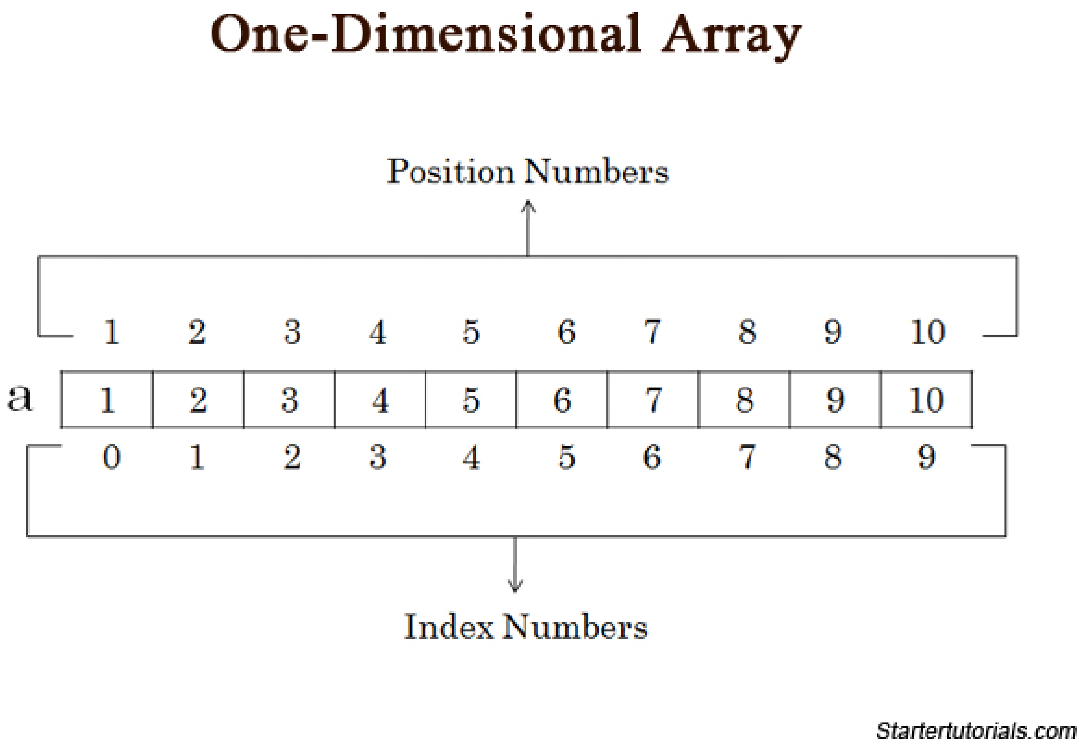
Two steps for creating an array can be combined into a single step as shown below:

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

Array Initialization: Assigning values into array locations is known as array initialization. Array initialization can be static or dynamic. Static initialization involves specifying values at the time of declaring an array as shown below:

```
int a[ ] = {1,2,3,4,5,6,7,8,9,10};
```

When the above line is executed, an array of size 10 is created by JVM as shown in the below figure:



Note that, position of 1 in the array is 1, whereas index or subscript of element 1 is 0. In general, the index or subscript of Nth element is N-1.

Dynamic initialization involves assigning values into an array at runtime. Following code segment demonstrates dynamic initialization of arrays:

```
int  a[] = new int[10];
Scanner  input = new Scanner(System.in);
for(int i = 0; i < 10; i++)
{
        System.out.println("Enter element number "+(i+1)+" : ");
        a[i] = input.nextInt();
}
```

Note: By default, when an array is created, all the elements of numeric type are initialized to zero, all elements of boolean type are initialized to false, all elements of type char are initialized to '\u0000' and all reference types are initialized to null.

Accessing Array Elements:

Individual elements of an array can be accessed using the index or subscript. In Java, the subscript always begins at zero. So, the first element of an array a can be accessed as a[0] and second element with a[1] and so on Nth element can be accessed with a[N-1].

Two-Dimensional Arrays

A two-dimensional array can be indicated by specifying two pairs of square brackets ([ ]) while declaring an array. A two-dimensional array can be created as shown below:

```
int a[ ][ ] = new int[3][3];
```

Above line creates an array with 3 rows and 3 columns (matrix) with a size of 3*3 = 9. So, the above array a is capable of storing 9 integer elements.

We can also declare a two-dimensional array by specifying only the number of rows as shown below:

```
int a[ ][ ] = new int[3][ ];
```

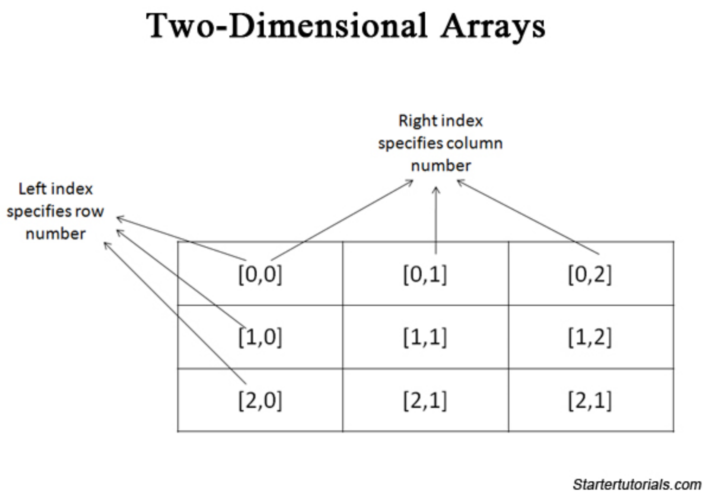
We can specify the number of columns for each row as shown below:

```
a[0] = new int[3];
a[1] = new int[3];
a[2] = new int[3];
```

So, by looking at the above example, we can say that a multi-dimensional array (2-D, 3-D, ..., N-D) is an array of arrays.

The above way of declaration can be used to create uneven arrays which contain unequal number of columns for each row. Such kind of arrays is also known as jagged arrays.

Individual elements of a two-dimensional array can be accessed by specifying both row-index and column-index as shown in the below figure:



Alternative Array Declaration Syntax

Java supports another way to declare arrays as shown below:

```
data-type[ ] array-name = new data-type[size];
```

Example for declaring arrays using the above syntax is shown below:

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

You might think what is the need for this alternative way of declaring an array? For example, you might want to declare three one-dimensional arrays. You can do it using the conventional syntax like:

```
int a[ ], b[ ], c[ ];
```

Instead, you can use the alternative syntax like:

```
int[ ] a, b, c;
```

Constructors

A constructor is a special method which has the same name as class name and that is used to initialize the objects (fields of an object) of a class. A constructor has the following characteristics:

- Constructor has the same name as the class in which it is defined.
- Constructor doesn't have a return type, not even void. Implicit return type for a constructor is the class name.
- Constructor is generally used to initialize the objects of a class.

Syntax for creating a constructor is as shown below:

```
ClassName( [Parameters List] )
{
        //Code of the constructor
}
```

Use of Constructors:

A constructor is used for initializing (assigning values to fields) the objects of a class. Can't we use normal methods for initializing the fields of an object? Why do we require constructors? To make it clear what is the need for constructors, let's consider an example of where we want to develop a Tic-Tac-Toe game as shown below:

To make it simple, let's not consider how to display graphics. We will consider only the general details. If you look at the above picture, we have to display a canvas or board (for example) which contains 3×3 = 9 squares. To display the canvas with 9 squares, let's consider the following class definition:

```
class TicTacToe
{
        Square block1, block2, ... , block9;
        //Other fields
        void display()
        {
                block1 = new Square();
                block1.createBlock();
                //Code for rest of the 8 blocks, block2, block3, ... , block9
                //Code to arrange the blocks as a 3x3 matrix
        }
        //Other code
}
```

The Square class will be something as shown below:

```
class Square
{
	int side;
	void createBlock()
	{
		side = 4;
		//Code for creating square
	}
}
```

The driver program (which executes our logic classes) will be as shown below:

```
class Game
{
	public static void main(String[] args)
	{
		TicTacToe board = new TicTacToe();
		board.display();
		//Other code
	}
}
```

If you look at the above program, the task of display() method is to create 9 square blocks by internally calling createBlock() method of Square class. This whole process can be seen as initializing the board. Every time, when someone wants to play the game, the board has to be initialized (square blocks have to be created and displayed to the user). To make the initialization process simple, constructors have been introduced whose sole purpose is to initialize the object. Now, let's modify our previous classes to include constructors. The code is shown below:

```
class TicTacToe
```

```
{
        Square block1, block2, ... , block9;
        //Other fields
        TicTacToe()
        {
                block1 = new Square();
                //Code for rest of the 8 blocks, block2, block3, ... , block9
                //Code to arrange the blocks as a 3x3 matrix
        }
        //Other code
}
```

The Square class will be as shown below:

```
class Square
{
        int side;
        Square()
        {
                side = 4;
                //Code for creating square
        }
}
```

The driver program (which executes our logic classes) will be as shown below:

```
class Game
{
        public static void main(String[] args)
        {
                TicTacToe board = new TicTacToe();
                //Other code
```

```
	}
}
```

If you look at the above code, the methods display() and createBlock() are gone. Remember that initialization might not be only simply assigning values to the fields of an object. The purpose of a constructor is to initialize the objects such that they will be ready to be used in our program like the board object in above program.

Constructor Invocation:

A constructor is invoked (called) automatically whenever an object is created using the new keyword. For example, in the above example, new TicTacToe() calls the constructor of TicTacToe class. If no constructor has been defined, Java automatically invokes the default constructor which initializes all the fields to their default values.

Types of Constructor:

Based on the number of parameters and type of parameters, constructors are of three types:

- Parameter less constructor or zero parameter constructor
- Parameterized constructor
- Copy constructor

Parameter less constructor: As the name implies a zero parameter constructor or parameter less constructor doesn't have any parameters in its signature. These are the most frequently found constructors in a Java program. Let's consider an example for zero parameter constructor:

```
class Square
{
	int side;
	Square()
	{
		side = 4;
	}
}
```

In the above example, Sqaure() is a zero parameter or parameter less constructor which initializes the side of a square to 4.

Parameterized constructor: This type of constructor contains one or more parameters in its signature. The parameters receive their values when the constructor is called. Let's consider an example for parameterized constructor:

```
class Sqaure
{
	int side;
	Square(int s)
	{
		side = s;
	}
}
```

In the above example, the Square() constructor accepts a single parameter s, which is used to initialize the side of a square.

Copy constructor: A copy constructor contains atleast one parameter of reference type. This type of constructor is generally used to create copies of the existing objects. Let's consider an example for copy constructor:

```
class Sqaure
{
	int side;
	Square()
	{
		side = 4;
	}
	Square(Square original) //This is a copy constructor
	{
		side = original.side;
	}
}
```

Now let's an example which covers all three types of constructors:

```
class Square
{
	int side;
	Square()
	{
		side = 4;
	}
	Square(int s)
	{
		side = s;
	}
	Square(Square original)
	{
		side = original.side;
	}
}
```

The driver program is shown below:

```
class SquareDemo
{
	public static void main(String[] args)
	{
		Sqaure normal = new Sqaure(); //Invokes zero parameter constructor
		Square original = new Sqaure(8); //Invokes single parameter constructor
		Sqaure duplicate = new Square(original); //Invokes copy constructor
		System.out.println("Side of normal square is: "+normal.side);
		System.out.println("Side of original square is: "+original.side);
		System.out.println("Side of duplicate square is: "+duplicate.side);
	}
}
```

Run the above program and observe the output.

Overloading

One of the way through which Java supports polymorphism is overloading. It can be defined as creating two or more methods in the same class sharing a common name but different number of parameters or different types of parameters. You should remember that overloading doesn't depend upon the return type of the method. Since method binding is resolved at compile-time based on the number of parameters or type of parameters, overloading is also called as compile-time polymorphism or static binding or early binding.

Why overloading?

In Java, overloading provides the ability to define two or more methods with the same name. What is the use of that? For example, you want to define two methods, in which, one method adds two integers and the second method adds two floating point numbers. If there is no overloading, we have to create two different methods. One for adding two integers and the other for adding two floating point numbers. As the underlying purpose of the methods is same, why create methods with different name? Instead of creating two different methods, overloading allows us to define two methods with the same name.

Method Overloading:

Creating two or more methods in the same class with same name but different number of parameters or different types of parameters is known as method overloading. Let's consider the following code segment which demonstrates method overloading:

```
class Addition
{
        void sum(int a, int b)
        {
                System.out.println("Sum of two integers is: "+(a+b));
        }
        void sum(float a, float b)
        {
                System.out.println("Sum of two floats is: "+(a+b));
        }
}
```

In the above code segment, the method sum is overloaded. Java compiler decides which method to call based on the type of the parameters in the method call. For example, if the method is called as shown below:

```
Addition obj = new Addition();

obj.sum(10, 20);
```

The sum method with two integer parameters will be invoked and the output will be Sum of two integers is: 30.

If the method is called as shown below:

```
Addition obj = new Addition();

obj.sum(1.5, 1.2);
```

The sum method with two float parameters will be invoked and the output will be Sum of two floats is: 2.7.

Note: It should be remembered that Java automatically performs type conversion from one type to another type. So, proper care should be taken while defining overloaded methods.

Constructor Overloading:

As constructor is a special type of method, constructor can also be overloaded. Several constructors can be defined in the same class given that the parameters vary in each constructor. As an example for constructor overloading, let's consider the following code segment:

```
class Square
```

```
{
        int side;
        Square(int s)
        {
                side = s;
                //Code to create a square and return it
        }
        Sqaure(int s, int n)
        {
                side = s;
                //Code to create n number of squares
        }
}
```

In the above code segment we can see that the constructor Sqaure() is overloaded. One constructor accepts a single integer parameter and returns a single square with side s. Another constructor accepts two integer parameters and returns n number of squares each with side s.

Note: In the above examples I have defined only two overloaded methods or constructors. You can create any number of overloaded methods or constructors based on your requirements.

Parameter Passing Techniques

If you have any previous programming experience you might know that most of the popular programming languages support two parameter passing techniques namely: pass-by-value and pass-by-reference.

In pass-by-value technique, the actual parameters in the method call are copied to the dummy parameters in the method definition. So, whatever changes are performed on the dummy parameters, they are not reflected on the actual parameters as the changes you make are done to the copies and to the originals.

In pass-by-reference technique, reference (address) of the actual parameters are passed to the dummy parameters in the method definition. So, whatever changes are performed on the dummy parameters, they are reflected on the actual parameters too as both references point to same memory locations containing the original values.

To make the concept more simple, let's consider the following code segment which demonstrates pass-by-value. This is a program for exchanging values in two variables:

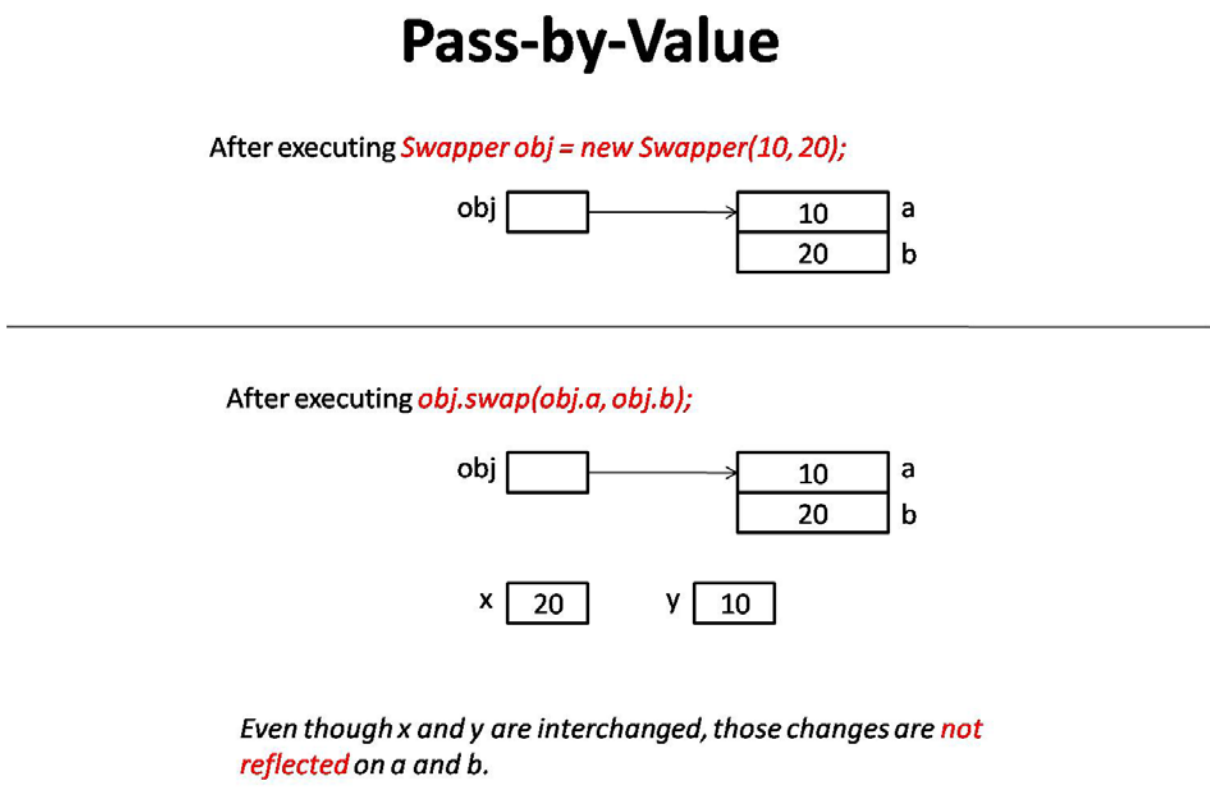
```
class Swapper
{
        int a;
        int b;
        Swapper(int x, int y)
        {
                a = x;
                b = y;
        }
        void swap(int x, int y)
        {
                int temp;
                temp = x;
                x = y;
                y = temp;
        }
}
class SwapDemo
{
        public static void main(String[] args)
        {
                Swapper obj = new Swapper(10, 20);
                System.out.println("Before swapping value of a is "+obj.a+" value of b is "+obj.b);
                obj.swap(obj.a, obj.b);
                System.out.println("After swapping value of a is "+obj.a+" value of b is "+obj.b);
        }
}
```

Output of the above programming will be:

```
Before swapping value of a is 10 value of b is 20
After swapping value of a is 10 value of b is 20
```

Although values of x and y are interchanged, those changes are not reflected on a and b.

Memory representation of variables is shown in below figure:


Pass-by-Value

After executing *Swapper obj = new Swapper(10, 20);*

After executing *obj.swap(obj.a, obj.b);*

Even though x and y are interchanged, those changes are not reflected on a and b.

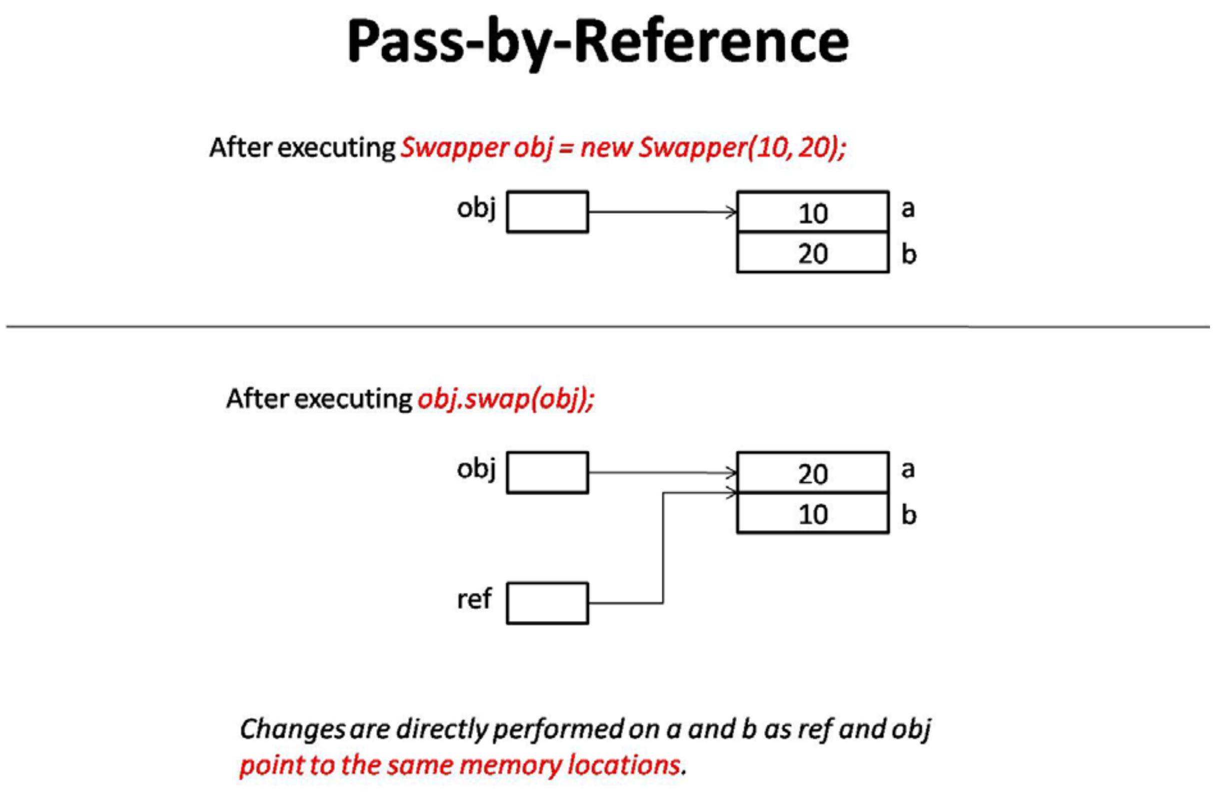
Let's consider the following code segment which demonstrates pass-by-reference. This is a program for exchanging values in two variables:

```
class Swapper
{
        int a;
        int b;
        Swapper(int x, int y)
        {
                a = x;
                b = y;
        }
        void swap(Swapper ref)
        {
                int temp;
                temp = ref.a;
                ref.a = ref.b;
                ref.b = temp;
        }
}
class SwapDemo
{
        public static void main(String[] args)
        {
                Swapper obj = new Swapper(10, 20);
                System.out.println("Before swapping value of a is "+obj.a+" value of b is "+obj.b);
                obj.swap(obj);
                System.out.println("After swapping value of a is "+obj.a+" value of b is "+obj.b);
        }
}
```

Output of the above programming will be:

```
Before swapping value of a is 10 value of b is 20
After swapping value of a is 20 value of b is 10
```

The changes performed inside the method swap are reflected on a and b as we have passed the reference obj into ref which also points to the same memory locations as obj. Memory representation of variables is shown in below figure:



After executing *Swapper obj = new Swapper(10, 20);*

After executing *obj.swap(obj);*

Changes are directly performed on a and b as ref and obj point to the same memory locations.

Note: In Java, parameters of primitive types are passed by value which is same as pass-by-value and parameters of reference types are also passed by value (the reference is copied) which is same as pass-by-reference. So, in Java all parameters are passed by value only.

Access Control

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the access modifiers. There are four access modifiers in Java. They are:

- public
- protected
- default

- private

If the member (variable or method) is not marked as either public or protected or private, the access modifier for that member will be default. We can apply access modifiers to classes also. Among the four access modifiers, private is the most restrictive access modifier and public is the least restrictive access modifier. Syntax for declaring a access modifier is shown below:

```
access-modifier  data-type  variable-name;
```

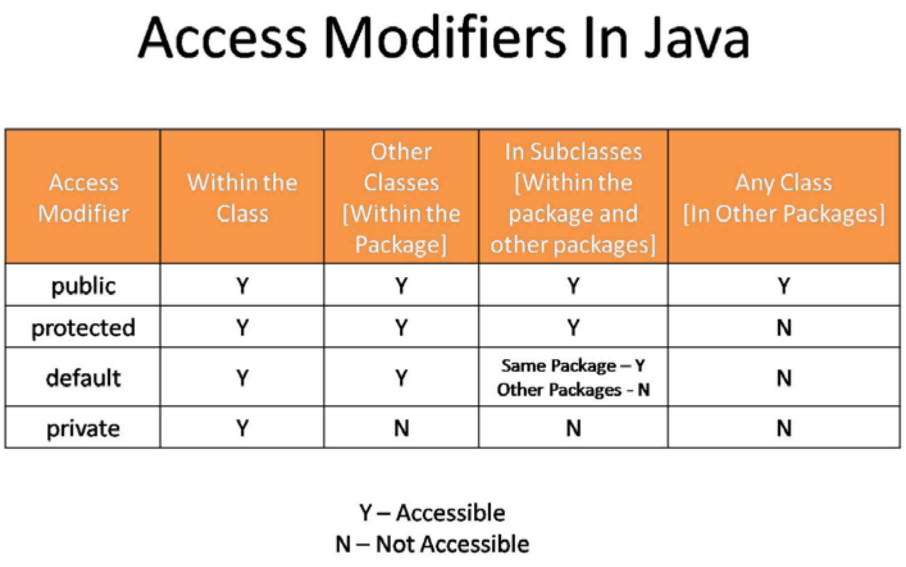
Example for declaring a private integer variable is shown below:

```
private int side;
```

In a similar way we can apply access modifiers to methods or classes although private classes are less common.

Note: Packages and inheritance will be discussed in another article in future. So, I will defer the explanation of protected to packages article as protected is useful only when there is inheritance.

Accessibility restrictions of the four access modifiers is as shown below:



Access Modifiers In Java

Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

What is the use of access modifiers?

As I had said above, access modifiers are used to restrict the access of members of a class, in particular data members (fields). Let me explain this through Java code. Let's consider an employee class as shown below:

```
class Employee
{
        int  empid;
        String  empname;
        float  salary;
        //Methods which operate on above data members
}
```

By looking at the above code we can say that the access modifier for all the three data members is default. As members with default (also known as Package Private or no modifier) access modifier are accessible throughout the package (in other classes), a programmer might, by mistake, try to make an employee's salary negative as shown below:

```
Employee e1 = new Employee();

e1.salary = -1000.00;
```

Although above code is syntactically correct, it is logically incorrect. To prevent such things to happen, in general, all the data members are declared private and are accessible only through public methods. So, we can modify our Employee class as shown below:

```
class Employee
{
        private int empid;
        private String emp;
        private float salary;
        public void setSalary(float sal)
        {
                if(sal &lt; 0)
                {
                        System.out.println("Salary cannot be negative");
                }
                else
                {
                        salary = sal;
                }
        }
        //Other methods
}
```

By looking at the above code, we can say that one can access the salary field only through setSalary() method. Now, we can set the salary of an employee as shown below:

```
Employee e1 = new Employee();
e1.setSalary(-1000.00);  //Gives error as salary cannot be negative
e1.setSalary(2000.00);  //salary of e1 will be assigned 2000.00
```

The modified Employee class is the best way of writing programs in Java.

this Keyword

this keyword in Java is used to refer current object on which a method is invoked. Using such property, we can refer the fields and other methods inside the class of that object. The this keyword has two main uses which are listed below:

1. It is used to eliminate ambiguity between fields and method parameters having the same name.
2. It is used for chaining constructors.

To explain the first use, let's consider the following program, which creates a square and displays it:

```
class Square
{
        int side;
        Square(int  s)
        {
                side = s;
                //Code to display a square with side s
        }
}
class SquareDemo
{
        Sqaure s1 = new Sqaure(4);
}
```

In the above code segment, there are no errors as the field name side and parameter name s are different. Now, let's modify our Square class as shown below:

```
class Sqaure
{
        int side;
        Square(int side)
        {
                side = side;
                //Code to display a square with side side
        }
}
class SquareDemo
{
        Sqaure s1 = new Sqaure(4);
}
```

In the above code segment, observe line no: 6. At this line, JVM will be in a ambiguous situation to decide whether to initialize the field side or the parameter side. To eliminate such ambiguity, the left side variables can be preceded with this keyword as shown below:

```
class Sqaure
{
        int side;
        Square(int side)
        {
                this.side = side;
                //Code to display a square with side side
        }
}
class SquareDemo
{
        Sqaure s1 = new Sqaure(4);
```

}

Now, JVM will be able to decide that the left hand side variable is the field side, and the right hand side variable is the parameter side. Some programmers prefer to declare fields and parameters with different names and other programmers prefer the same names for both fields and parameters. Which one to practice, is left to you.

The second use of this keyword is constructor chaining, which refers to the invocation of one constructor from another constructor. We can call a constructor from another constructor using this keyword as shown below:

```
this(param1, param2, …, paramn);
```

While using the above syntax, the constructor whose signature (number of and type of parameters) matches, will be invoked.

Note: While invoking another constructor using the this keyword, it should be the first line inside the constructor's definition. For clarity regarding this, look at the example given below:

Example on constructor chaining:

```
class Square
{
        int side;
        Sqaure( )
        {
                this(4);  //demonstrates constructor chaining
                //Code to display a square
        }
        Square(int  s)
        {
                side = s;
        }
}
class SquareDemo
{
        Sqaure s1 = new Sqaure(4);
}
```

In the above program, at line number 6, the second constructor is invoked as it is having a single integer parameter and 4 is passed to the parameter s which will be assigned to the field, side.

static Keyword

In Java programs, static keyword can be used to create the following:

1. Class variables
2. Class methods
3. Static blocks

Class Variables:

The static keyword is used to create one of the three types of variables called as class variables. A class variable is a variable declared inside a class and outside all the methods and is marked as static. Syntax for declaring a class variable or a static variable is shown below:

```
static data-type variable-name;
```

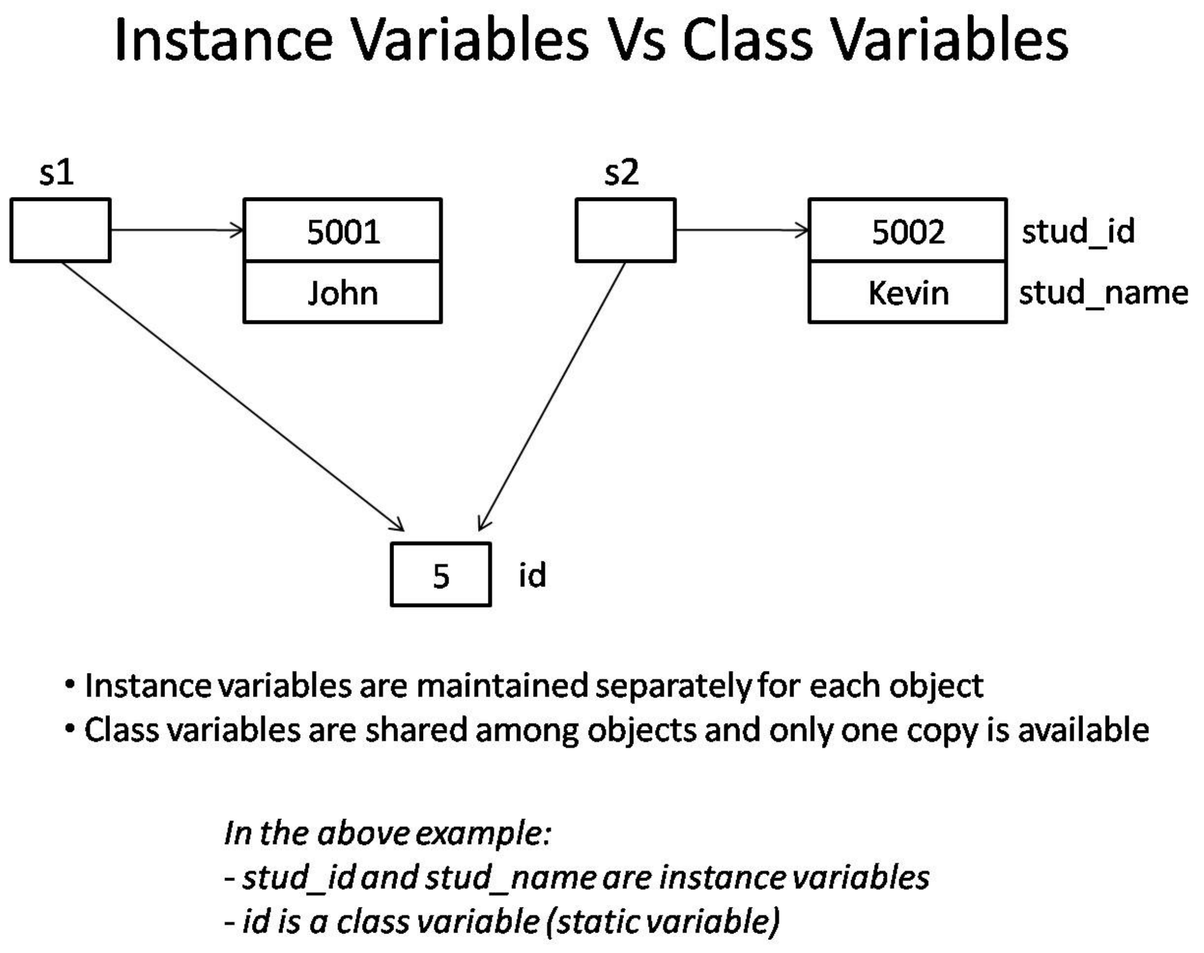
Example for declaring a class variable is shown below:

```
static int id;
```

What is special about a class variable? As you already know, an instance variable is created separately for every object of the class. But a class variable is created only once inside the memory and the same is shared among all the objects of a class. Consider the following code segment to demonstrate the difference between instance variables and class variables.

```
class  CSEStudent
{
        static  int  id = 5;
        int  stud_id;
        String  stud_name;
}
class  StudentDemo
{
        public static void main(String[] args)
        {
                CSEStudent  s1 = new CSEStudent();
                s1.stud_id = 5001;
                s1.stud_name = "John";
                CSEStudent  s2 = new CSEStudent();
                s1.stud_id = 5002;
                s1.stud_name = "Kevin";
                System.out.println("Branch id is: "+s1.id);
                System.out.println("Branch id is: "+s2.id);
        }
}
```

Below figure illustrates the difference between an instance variable and a class variable in the above code segment:



Instance Variables Vs Class Variables

- Instance variables are maintained separately for each object
- Class variables are shared among objects and only one copy is available

In the above example:
- stud_id and stud_name are instance variables
- id is a class variable (static variable)

Class Methods:

All the non-static methods inside a class are known as instance methods and all the static methods inside a class are known as class methods. A class method is generally used to process class variables. A class method has the following limitations:

1. *A class method (static method) can access only other class methods.*
2. *A class method can access only class variables (static variables).*

3. this and super cannot be used in class methods.

A class method can be created using the following syntax:

```
static  return-type  method-name(parameters-list)
{
            //statements
}
```

An example for creating a class method is shown below:

```
static  void  changeID(int  newid)
{
            id = newid;
}
```

Following code segment illustrates the use of class methods in Java:

```
class  CSEStudent
{
            static  int  id = 5;
            int  stud_id;
            String  stud_name;
            static  void  changeID(int  newid)
            {
                        id = newid;
            }
}
class  StudentDemo
{
            public static void main(String[] args)
            {
                        CSEStudent  s1 = new CSEStudent();
```

```
                    s1.stud_id = 5001;
                    s1.stud_name = "John";
                    CSEStudent  s2 = new CSEStudent();
                    s1.stud_id = 5002;
                    s1.stud_name = "Kevin";
                    System.out.println("Branch id is: "+s1.id);
                    System.out.println("Branch id is: "+s2.id);
                    s1.changeID(2);
                    System.out.println("Branch id is: "+s1.id);   //Guess the output
                    System.out.println("Branch id is: "+s2.id);  //Guess the output
          }
}
```

Static Blocks:

A static block is a block of statements prefixed with static keyword. The syntax for creating a static block is shown below:

```
static
{
          //Statements
}
```

An important property of a static block is, the statements in a static block are executed as soon as the class is loaded into memory even before the main method starts its execution. A typical use of static blocks is initializing the class variables.

Following program demonstrates the use of static blocks:

```
class  CSEStudent
{
	static int id;
	int stud_id;
	String stud_name;
	static
	{
		id = 5;
		//other statements
	}
}
class StudentDemo
{
	public static void main(String[] args)
	{
		CSEStudent s1 = new CSEStudent();
		s1.stud_id = 5001;
		s1.stud_name = "John";
		CSEStudent s2 = new CSEStudent();
		s1.stud_id = 5002;
		s1.stud_name = "Kevin";
		System.out.println("Branch id is: "+s1.id);
		System.out.println("Branch id is: "+s2.id);
	}
}
```

Note: *Another important property of static keyword is, any class variable (static variable) or a class method (static method) can be accessed directly without creating an object for the class using the following syntax:*

ClassName.variablename;

or

ClassName.methodname( );

This is the reason why the main method is declared as static in every program so that the JVM can access it directly without creating an object for the class. Following program demonstrates the alternative syntax for accessing class variables and class methods:

```
class CSEStudent
{
        static int id = 5;
        int stud_id;
        String stud_name;
        static void changeID(int newid)
        {
                id = newid;
        }
}
class StudentDemo
{
        public static void main(String[] args)
        {
                CSEStudent s1 = new CSEStudent();
                s1.stud_id = 5001;
                s1.stud_name = "John";
                CSEStudent s2 = new CSEStudent();
                s1.stud_id = 5002;
                s1.stud_name = "Kevin";
                System.out.println("Branch id is: "+ CSEStudent.id);
                s1.changeID(2);
                System.out.println("Branch id is: "+ CSEStudent.id);   //Guess the output
        }
}
```

final Keyword

Primary use of the final keyword is to declare constants in Java programs. A constant is like a variable but, whose value cannot be changed once initialized. Syntax for declaring a constant using final keyword is shown below:

```
final data-type identifier = value;
```

You should remember that a constant should be initialized at the time of declaration itself. Once a constant has been initialized, it cannot be modified later. For example you can declare a PI constant as shown below:

```
final float PI = 3.14;
```

Java's convention for constant names is that they should be written in uppercase. Below code segment demonstrates the use of constants (created using final keyword) in a Java program:

```
class Circle
{
        float  radius;
        final  float  PI = 3.14;
        void displayArea()
        {
                float area;
                radius = 2.3;
                area = PI * radius * radius;
                System.out.println("Area of the circle is: "+area);
        }
}
```

There are two more uses of final keyword which will be discussed in inheritance section.

Garbage Collection

In most of the object oriented programming languages, memory management (allocating and deallocating memory) is left to the user and it is generally error prone (user might forget to free the memory occupied by an object). Objects are allocated memory in the heap memory which grows and shrinks dynamically. Java provides automatic memory management through a mechanism known as garbage collection.

Unused objects are collected for garbage collection by the program known as garbage collector. Java has its own set of algorithms which decides whether an object is eligible for garbage collection or not. Generally, an object is garbage collected when it goes out of scope or when the object is no longer referenced.

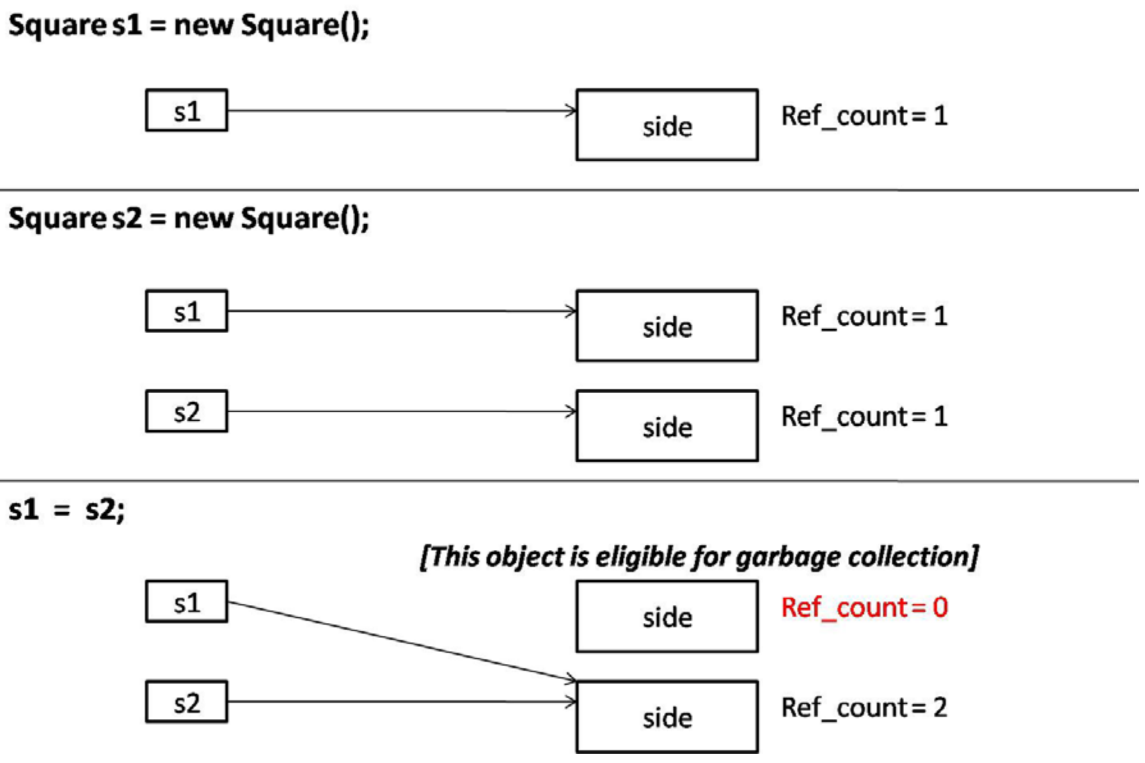
Garbage Collector

The garbage collector frequently scans the heap memory for detecting unused objects. Two general methods used by garbage collectors for finding unused objects are: reference counting and tracing.

In **reference counting method**, whenever an object is created, its reference count will be 1. As the object is referenced by other references, the reference count will be incremented by 1 for reference. As the references decrease, the reference count will be decremented. When the reference count is finally 0, the object will be garbage collected. Let's consider the following code segment as an example:

```
class Square
{
        int side;
}
class SquareDemo
{
        public static void main(String[] args)
        {
                Square s1 = new Square();
                Square s2 = new Square();
                s1 = s2;
        }
}
```

Following memory representation illustrates the reference count for each object:



*In **tracing method**, also known as mark and sweep method, the garbage collector scans the dynamic memory areas for all the objects. The mark phase marks all the objects which are being referenced. After the marking is completed, the sweep phase frees the memory allocated to all other unmarked objects.*

The garbage collectors using mark and sweep method uses compaction and copying (related to operating system memory management). During the sweep phase, the memory might get fragmented. The garbage collector moves all the fragmented memory towards one end (compaction) so that the other end contains a free block of memory which can be used by other code. Any other objects on the other side of the used memory are copied to this side (copying).

The garbage collector can run synchronously when the program runs out of memory or asynchronously when the system is idle. The garbage collector is an example of daemon thread (a thread which runs in the background).

Garbage collector can be explicitly invoked by using System.gc() or Runtime.gc(). Explicit invocation doesn't guarantee that unused objects will be garbage collected. It is up to Java Virtual Machine when the unused objects will be actually freed.

Finalization

It is common in Java programs to access files and databases (will be covered in future articles). It is also common for programmers to open the files and other resources and forget to close the connections to those resources. The code which is used to close the connections to resources like files, databases etc… is known as cleanup code or housekeeping code.

The cleanup code is generally written in the **finalize()** method which belongs to the Object class, as the finalize() method is guaranteed to be executed when the object gets cleaned up from the memory. This process is known as finalization.

Recursion

Recursion is a famous way to solve problems with less lines of code. Many programmers prefer recursion over iteration as less amount of code is required to solve problems using recursion.

A method calling itself in its definition (body) is known as recursion. For implementing recursion, we need to follow the below requirements:

- There should be a base case where the recursion terminates and returns a value.
- The value of the parameter(s) should change in the recursive call. Otherwise, this leads to infinite loop.

Let's consider the code for finding factorial of a number using iteration as shown below:

```
int fact(int n)
{
	int r = 1;
	for(int i = n; i > 1 ; i--)
	 {
		r = i * r;
	}
	 return r;
}
```

Now, let's consider the recursive solution for finding the factorial of a given number which is shown below:

```
int rfact(int n)
{
  if(n == 0 || n == 1)
    return 1;
  else
    return n * rfact(n-1);  //recursive call
}
```

The complete Java program which demonstrates recursion is given below:

```
class RecursionDemo
{
	public static void main(String[] args)
	{
		int n = 6;
		System.out.println("Factorial of " + n + " without recursion is: " + fact(n));
		System.out.println("Factorial of " + n + " using recursion is: " + rfact(n));
	}
	static int fact(int n)
	{
		int r = 1;
		for(int i = n; i > 1 ; i--)
		{
			r = i * r;
		}
		return r;
	}
	static int rfact(int n)
```

```
        {
                if(n == 0 || n == 1)
                        return 1;
                else
                        return n * rfact(n-1);  //recursive call
        }
}
```

Command Line Arguments

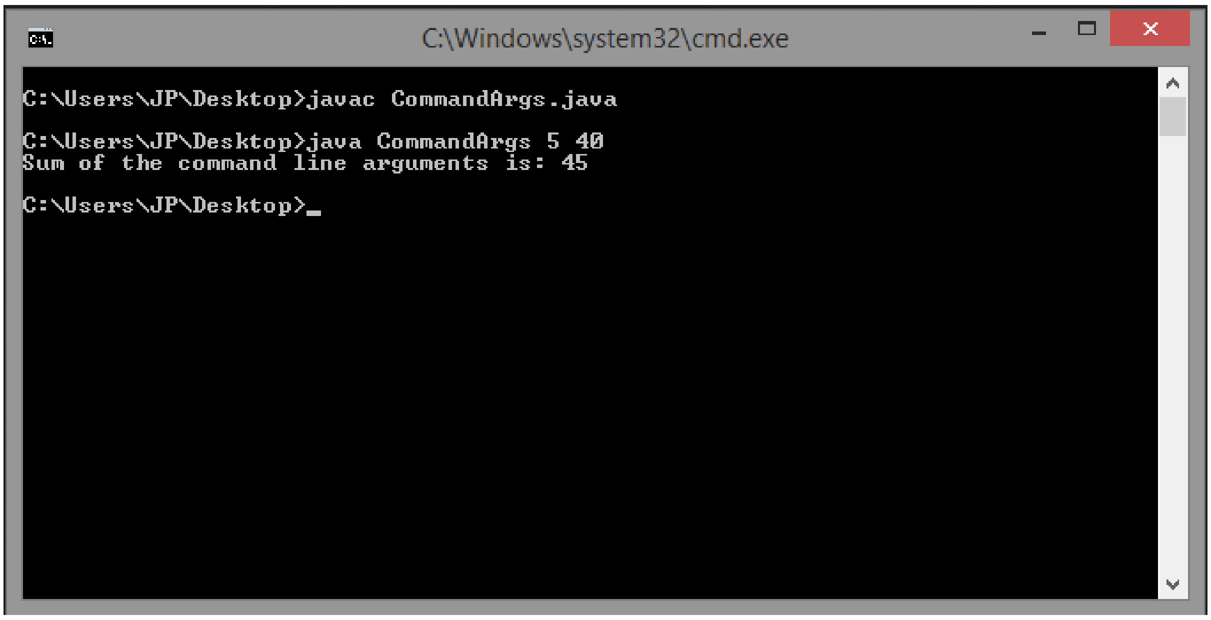
Sometimes we might want to pass extra information while running a Java program. This extra information passed along with the program name are known as command line arguments. These command line arguments are separated by white spaces.

Command line arguments can be accessed using the string array specified in the main function's signature. For example, if the array name is args, then the first command line argument can be accessed as args[0] and the second command line argument can be accessed as args[1] and so on.

Let's look at the following Java program which access two command line arguments, adds them and displays the result to the user:

```
class CommandArgs
{
  public static void main(String[] args)
  {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    int sum = x + y;
    System.out.println("Sum of the command line arguments is: " + sum);
  }
}
```

Command line arguments are specified as shown in the below screenshot:



```
C:\Users\JP\Desktop>javac CommandArgs.java

C:\Users\JP\Desktop>java CommandArgs 5 40
Sum of the command line arguments is: 45

C:\Users\JP\Desktop>_
```

Variable Length Arguments

There might be some situations while creating methods, where the programmer might not know how many arguments are needed in the method definition or how many arguments are going to be passed at run-time by the user. Prior to Java 5, there were a couple of workarounds for this problem but was often complex or error prone.

With Java 5, a new feature called varargs was introduced which allows programmers to specify variable length arguments in the method definition. A method with variable length arguments is specified as shown below:

return-type method-name(type … variable-name) { }

In the above method declaration, ellipses (...) represents variable length arguments. We can also specify normal variables along with variable length arguments in the method definition. There are two rules that should be followed while working with variable length arguments. They are as follows:

- When normal variables (arguments) are specified along with variable length arguments, the variable length argument should be the last in the list of arguments.
- One cannot specify more than one variable length argument in a method definition.

Below example Java program demonstrates the use of variable length arguments (varargs):

```
class VarArgsDemo
{
  public static void main(String[] args)
  {
    VarArgsDemo v = new VarArgsDemo();
    v.sum(1);
    v.sum(1, 2);
    v.sum(1, 2, 3);
  }
  public void sum(int ... var)
  {
    int s = 0;
    for(int x : var)
      s += x;
    System.out.println("Sum is: " + s);
  }
}
```

Nested Classes

Java 1.1 added support for nested classes. A class defined as a member of another class definition is known as a nested class. A nested class can be either static or non-static. The non-static nested class is known as an inner class.

The enclosing class of an inner class can be called as an outer class. An inner class can access all the members of an outer class. But the outer class cannot access the inner class members. Inner classes and anonymous classes (inner classes without a name) are useful in event handling.

Below Java program demonstrates inner class (non-static nested class):

```
class Person
{
  int age;
  String name;
  class Details
  {
    public void showDetails()
    {
      System.out.println("Age: " + age);
      System.out.println("Name: " + name);
    }
  }
  public void show()
  {
    Details d = new Details();
    d.showDetails();
  }
  public static void main(String[] args)
  {
    Person p = new Person();
    p.age = 25;
    p.name = "surya";
    p.show();
  }
}
```

In the above example Details is the inner class and Person is the outer class.

Strings

A string is a collection of characters. In Java, strings can be created using three predefined classes namely String, StringBuffer and StringBuilder which are available in java.lang package. Why did Java designers provide three classes for creating strings?

Each of the above three string classes has their own advantages and disadvantages. If you want to create an immutable (whose content cannot be changed once created) string, String class is the best choice. Strings created using StringBuffer and StringBuilder classes are mutable (content can be changed after they are created).

Among StringBuffer and StringBuilder classes, strings created using StringBuffer are thread safe whereas strings created using StringBuilder are non-thread safe. I will explain about threads in future articles.

Differences between all string classes: String, StringBuffer and StringBuilder are summarized in the below table:

Factor / Class	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread Safety	Not thread safe	Thread safe	Not thread safe
Performance	Very high	Moderate	Very high

Now let's look at each class in detail...

String Class

A string can be created in Java using the following alternatives:

```
String s1 = "This is string s1.";
String s2 = new String("This is string s2.");
```

In line 2, we are using a String constructor along with the new operator to create the string explicitly. In line 1, the whole process (creation of string) is done implicitly by JVM.

If the content of the newly created string matches with the existing string (already available in memory), then a reference to the existing string in the memory is returned instead of allocating new memory. Since strings created using String are shared, hence they are immutable as changing the string content using one reference might effect the others.

String Manipulation

As we already know that strings created using String class are immutable, how to manipulate them? Let us concatenate (join) two strings as shown below:

```
String str = "hello";
str = str + "world";
```

From the above two lines of code, content in str is "helloworld". How is this possible if strings created using String class are immutable. Actually what JVM has done implicitly (automatically) is this:

```
str = new StringBuffer().append(str).append("world").toString();
```

Since concatenation is a manipulation, a new StringBuffer object is created and the content of str which is "hello" is appended to the buffer by using append method and then "world" is concatenated to "hello" making it "helloworld". Finally the StringBuffer object is converted to a String object using the toString. All this process is done automatically by JVM whenever we try to concatenate two or more String objects.

Now we will look at various methods supported by String class for working with strings.

String Methods

The length method can be used to find the length of given string. Syntax of this method is given below:

int length()

An example of using length method is given below:

```
String str = "hello";
System.out.println(str.length());
```

The above code prints 5.

The equals method can be used to compare whether two strings are having the same content or not. If both the strings are same, this method returns true, otherwise, false. Syntax of this method is given below:

boolean equals(String str)

An example of using equals method is given below:

```
String str1 = "hello";
String str2 = new String("hello");
System.out.println(str1.equals(str2));
```

The above code prints true.

The compareTo method can be used to compare whether two strings are having the same content or not. Syntax of this method is given below:

int compareTo(String str)

This method returns 0 when both strings are same, returns -ve number when the invoking string is less than the argument string, and +ve number when the invoking string is greater than the argument string.

An example of using compareTo method is given below:

```
String str1 = "hello";
String str2 = new String("hello");
System.out.println(str1.compareTo(str2));
```

Above example prints 0 as both strings are equal (same content).

Another example of compareTo method is given below:

```
String str1 = "hallo";
String str2 = new String("hello");
System.out.println(str1.compareTo(str2));
```

Above example prints -4 as the difference between 'a' in "hallo" and 'e' in "hello" is 4 and "hallo" is the invoking string.

The indexOf method is used to find the position (number) of a specified character in the invoking string. Syntax of this method is given below:

int indexOf(char c)

An example of using indexOf method is given below:

```
String str1 = "hello world";
System.out.println(str1.indexOf('o'));
```

Above example prints 4 which the index of first 'o' in the string "hello world". Index of a string always starts from zero.

The overloaded indexOf method is used to find the position (number) of a specified string in the invoking string. Syntax of this method is given below:

int indexOf(String str)

An example of using indexOf method is given below:

```
String str1 = "hello world";
System.out.println(str1.indexOf("wor"));
```

Above example prints 6.

The lastIndexOf method is used to find the position (number) of a specified character from the end of the invoking string. Syntax of this method is given below:

int lastIndexOf(char c)

An example of using lastIndexOf method is given below:

```
String str1 = "hello world";
System.out.println(str1.lastIndexOf('o'));
```

Above example prints 7 which is the position of 'o' from the end of the invoking string "hello world".

The overloaded lastIndexOf method is used to find the position (number) of a specified string from the end of the invoking string. Syntax of this method is given below:

int lastIndexOf(String str)

An example of using lastIndexOf method is given below:

```
String str1 = "I am a good and I am bad";
System.out.println(str1.lastIndexOf("am"));
```

Above example prints 18.

The substring method can be used to extract a sub string from the invoking string from a specified position to the end of the string. Syntax of this method is given below:

String substring(int index)

An example of substring method is given below:

```
String str1 = "hello world";
System.out.println(str1.substring(2));
```

Above example prints "llo world". The starting index specified in the above example is 2.

The overloaded substring method can be used to extract a sub string from the invoking string from a specified start position to end position in the string. Syntax of this method is given below:

String substring(int startindex, int endindex)

An example of substring method is given below:

```
String str1 = "hello world";
System.out.println(str1.substring(6, 9));
```

Above example prints "wor". The starting index is 6 and the end index is 9. All the characters from starting index to excluding end index are returned.

The toLowerCase method can be used to convert all characters in a string to lower case characters. Syntax is as given below:

String toLowerCase()

An example of toLowerCase method is given below:

```
String str1 = "Hello World";
System.out.println(str1.toLowerCase());
```

Above example prints “hello world”.

The toUpperCase method can be used to convert all characters in a string to upper case characters. Syntax is as given below:

String toUpperCase()

An example of toUpperCase method is given below:

```
String str1 = "Hello World";
System.out.println(str1.toUpperCase());
```

Above example prints “HELLO WORLD”.

The startsWith method can be used to check whether the invoking string starts with a specified string or not. Syntax of this method is given below:

boolean startsWith(String str)

An example of startsWith method is given below:

```
String str1 = "hello world";
System.out.println(str1.startsWith("hello"));
```

Above example prints true since the invoking string “hello world” starts with the specified string “hello”.

Similarly, endsWith method can be used to find out whether a string ends with a specified string or not.

The overloaded valueOf method can be used to convert primitive type values like int, float, double etc to a String. Syntax of valueOf method that converts a given int value to a String value is given below:

static String valueOf(int value)

An example of valueOf method is given below:

```
int x = 10;
String str = String.valueOf(x);
System.out.println(str);
```

Above example prints 10.

Above mentioned methods are the most frequently used methods of String class. You can have a look at other methods by clicking the link which is provided above.

Now we will look at the StringBuffer class and some of the methods available in that class.

StringBuffer

As already mentioned, strings created using StringBuffer class are mutable i.e., after creating a string, we can change the content of the string. Also strings created with StringBuffer class are thread safe (I will cover this aspect in future article on MultiThreading).

Creating Strings using StringBuffer Class

We can use the overloaded StringBuffer constructors as shown below:

```
StringBuffer sb1 = new StringBuffer();
StringBuffer sb2 = new StringBuffer(30);
StringBuffer sb3 = new StringBuffer("hello");
```

The first line in the above example creates a empty StringBuffer object with a default capacity of 16 characters. The second line creates a StringBuffer object with a capacity of 30 characters and the last line creates a StringBuffer object from the supplied string "hello" and reserves an additional space for accommodating 16 characters to reduce reallocation.

Now let's look at most of the frequently used methods available in the StringBuffer class.

StringBuffer Methods

The length method can be used to find the size of a StringBuffer object. Syntax of this method is given below:

int length()

An example of using length method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.length());
```

Above example prints 5 which is the number of characters (size) in the buffer.

The capacity method can be used to find the capacity of a StringBuffer object. Syntax of this method is given below:

int capacity()

An example of using capacity method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.capacity());
```

Above example prints 21 which is the capacity (size + 16) of the buffer.

You can use ensureCapacity and setLength methods to set the capacity and size of the buffer respectively. Syntax of each of these methods is given below:

void ensureCapacity(int minCapacity)

void setLength(int length)

The charAt method can be used to retrieve a single character from a specified position. Syntax of this method is given below:

char charAt(int pos)

An example of using charAt method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.charAt(1));
```

Above example prints the character 'e' which is available at the specified position 1.

The setCharAt method can be used to set (change) a character at a specific location in the buffer. Syntax of this method is given below:

void setCharAt(int pos, char ch)

An example of using setCharAt method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
sb1.setCharAt(1, 'a');
System.out.println(sb1);
```

Above example prints "hallo".

The getChars method can be used to extract a sub string from the buffer to a character array. Syntax of this method is given below:

void getChars(int startIndex, int endIndex, char target[], int targetStart)

In the above syntax, startIndex is the starting position in the invoking string and endIndex is the position up to which the characters are extracted plus one. target is the array in to which the sub string is extracted and targetStart specifies the index in the target array from which the sub string must be stored.

An example of getChars method is given below:

```
char[] target = new char[10];
StringBuffer sb1 = new StringBuffer("hello");
sb1.getChars(0,  3, target, 0);
System.out.println(target);
```

Above example prints "hel".

The append method can be used to concatenate the string representation of different types of data to the end of a buffer. This method is overloaded. Syntax of this method which accepts a String argument is shown below:

StringBuffer append(String str)

An example of this method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.append("world"));
```

Above example prints "helloworld".

The insert method can be used to insert string representation of different types of data into a buffer at a specified position. This method is overloaded like append method. Syntax of this method for String type is given below:

StringBuffer insert(int index, String str)

An example of using insert method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.insert(5,"world"));
```

Above example prints "helloworld". Although the example inserts the string "world" at the end of the buffer, it can be inserted at any position you want.

The reverse method can be used to reverse the content of a buffer. Syntax of this method is given below:

StringBuffer reverse()

An example of using reverse method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.reverse());
```

Above example prints "olleh".

The deleteCharAt method can be used to delete a character at specified location in the buffer. Syntax of this method is given below:

StringBuffer deleteCharAt(int pos)

An example of using deleteCharAt method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.deleteCharAt(1));
```

Above example prints "hllo".

The delete method can be used delete a set of characters from the buffer. The starting index and ending index specifies the set of characters to be deleted. Syntax of this method is given below:

StringBuffer delete(int startIndex, int endIndex)

An example of using delete method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.delete(1, 4));
```

Above example prints "ho". Characters from index 1 to 3 are deleted.

The replace method can be used to replace a set of characters with a specified string. The sub string is specified with starting index and ending index plus one. Syntax of this method is given below:

StringBuffer replace(int startIndex, int endIndex, String str)

An example of using replace method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.replace(1, 3, "a"));
```

Above example prints "halo".

The substring method can be used to extract a substring from the buffer. This is method is overloaded. Syntax of this method is given below:

String substring(int startIndex)

String substring(int startIndex, int endIndex)

An example of using substring method is given below:

```
StringBuffer sb1 = new StringBuffer("hello");
System.out.println(sb1.substring(1));
System.out.println(sb1.substring(1, 4));
```

In the above example, line 2 prints "ello" and line 3 prints "ell".

StringBuilder

The StringBuilder class is similar to StringBuffer class in the sense that both classes contain same methods. The difference between them is in the case of performance. StringBuilder is faster than StringBuffer.