public static void main(String[] args)

Java main method is the entry point of any java program. Its syntax is always public static void main(String[] args). You can only change the name of String array argument, for example you can change args to myStringArgs. Also String array argument can be written as String... args or String args[]

Public

This is the access modifier of the main method. It has to be public so that java runtime can execute this method. Remember that if you make any method non-public then it's not allowed to be executed by any program, there are some access restrictions applied. So it means that the main method has to be public.

static

When java runtime starts, there is no object of the class present. That's why the main method has to be static so that JVM can load the class into memory and call the main method. If the main method won't be static, JVM would not be able to call it because there is no object of the class is present.

void

Java programming mandates that every method provide the return type. Java main method doesn't return anything, that's why it's return type is void. This has been done to keep things simple because once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM. If we try to return something from the main method, it will give compilation error as an unexpected return value.

main

This is the name of java main method. It's fixed and when we start a java program, it looks for the main method.

String[] args

Java main method accepts a single argument of type String array. This is also called as java command line arguments. Let's have a look at the example of using java command line argument

Packages In Java

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee

- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier

- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

We can reuse existing classes from the packages as many time as we need it in our program.

How packages work?

Package names and directory structure are closely related. For example if a package name is college.staff.cse, then there are three directories, college,

staff and cse such that cse is present in staff and staff is present college. Also, the directory college is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

Package naming conventions : Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.

Subpackages: Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.
Example :

```
import java.util.*;
```

util is a subpackage created inside java package.

Accessing classes inside a package
Consider following two statements :

```
// import the Vector class from util package.
import java.util.vector;
// import all the classes from util package
import java.util.*;
```

- First Statement is used to import Vector class from util package which is contained inside java.
- Second statement imports all the classes from util package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;
```

// Only mentioned class of this package will be accessible.

import package.classname;

// Class name is generally used when two packages have the same

// class name. For example in below code both packages have

// date class so using a fully qualified name to avoid conflict

import java.util.Date;

import my.packag.Date;

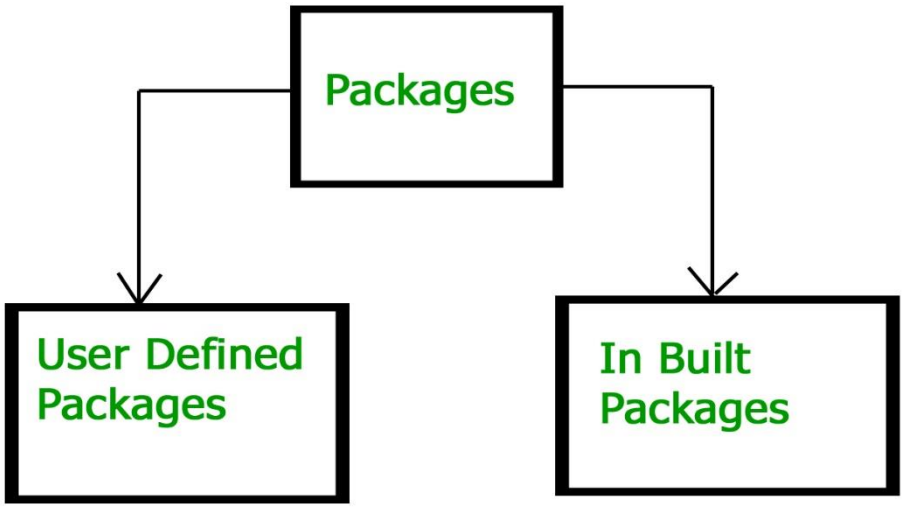
```
// Java program to demonstrate accessing of members when
// corresponding classes are imported and not imported.
import java.util.Vector;

public class ImportDemo
{
   public ImportDemo()
   {
      // java.util.Vector is imported, hence we are
      // able to access directly in our code.
      Vector newVector = new Vector();

      // java.util.ArrayList is not imported, hence
      // we were referring to it using the complete
      // package.
      java.util.ArrayList newList = new java.util.ArrayList();
   }

   public static void main(String arg[])
   {
      new ImportDemo();
   }
}
```

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) java.io: Contains classed for supporting input / output operations.

3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) java.applet: Contains classes for creating Applets.

5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) java.net: Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory myPackage (name should be same as the name of the package). Then create the MyClass inside the directory with the first statement being the package names.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;
```

```
public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;


public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";


      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();


      obj.getNames(name);
   }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

Using Static Import

Static import is a feature introduced in Java programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public static to be used in Java code without specifying the class in which the field is defined.

Following program demonstrates static import :

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo
{
   public static void main(String args[])
   {
         // We don't need to use 'System.out'
         // as imported using static.
         out.println("GeeksforGeeks");
   }
}
```

Output:

```
GeeksforGeeks
```

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;
import java.sql.*;


//And then use Date class, then we will get a compile-time error :


Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class.
For Example:

```
java.util.Date deadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Directory structure

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, the class Circle of

package com.zzz.project1.subproject2 is stored as "$BASE_DIR\com\zzz\project1\subproject2\Circle.class", where $BASE_DIR denotes the base directory of the package. Clearly, the "dot" in the package name corresponds to a sub-directory of the file system.

The base directory ($BASE_DIR) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the $BASE_DIR so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

Setting CLASSPATH:

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows, choose control panel ? System ? Advanced ? Environment Variables ? choose "System Variables" (for all the users) or "User Variables" (only the currently login user) ? choose "Edit" (if CLASSPATH already exists) or "New" ? Enter "CLASSPATH" as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., ".;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar"). Take note that you need to include the current working directory (denoted by '.') in the CLASSPATH.
  To check the current setting of the CLASSPATH, issue the following command:
- > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,
- > java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

Illustration of user-defined packages:

Creating our first package:
File name – ClassOne.java

```
package package_name;

public class ClassOne {
    public void methodClassOne() {
        System.out.println("Hello there its ClassOne");
    }
}
```

Creating our second package:
File name – ClassTwo.java

```
package package_one;

public class ClassTwo {
    public void methodClassTwo(){
        System.out.println("Hello there i am ClassTwo");
    }
}
```

Making use of both the created packages:
File name – Testing.java

```
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
    public static void main(String[] args){
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

Output:

```
Hello there i am ClassTwo

Hello there its ClassOne
```

Important points:

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
5. We can access public classes in another (named) package using: **package-name.class-name**

Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

```
import java.util.ArrayList;
class Autoboxing
{
    public static void main(String[] args)
    {
        char ch = 'a';

        // Autoboxing- primitive to Character object conversion
        Character a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);

        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Output:

```
25
```

Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;

class Unboxing
{
    public static void main(String[] args)
    {
        Character ch = 'a';

        // unboxing - Character object to primitive conversion
        char a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);

        // unboxing because get method returns an Integer object
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Output:

```
24
```

Implementation

```
// Java program to demonstrate Wrapping and UnWrapping
// in Java Classes
class WrappingUnwrapping
{
    public static void main(String args[])
    {
        //  byte data type
        byte a = 1;

        // wrapping around Byte object
        Byte byteobj = new Byte(a);

        // int data type
        int b = 10;

        //wrapping around Integer object
        Integer intobj = new Integer(b);

        // float data type
```

```
        float c = 18.6f;

        // wrapping around Float object
        Float floatobj = new Float(c);

        // double data type
        double d = 250.5;

        // Wrapping around Double object
        Double doubleobj = new Double(d);

        // char data type
        char e='a';

        // wrapping around Character object
        Character charobj=e;

        //  printing the values from objects
        System.out.println("Values of Wrapper objects (printing as objects)");
        System.out.println("Byte object byteobj:  " + byteobj);
        System.out.println("Integer object intobj:  " + intobj);
        System.out.println("Float object floatobj:  " + floatobj);
        System.out.println("Double object doubleobj:  " + doubleobj);
        System.out.println("Character object charobj:  " + charobj);

        // objects to data types (retrieving data types from objects)
        // unwrapping objects to primitive data types
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
        char cv = charobj;

        // printing the values from data types
        System.out.println("Unwrapped values (printing as data types)");
        System.out.println("byte value, bv: " + bv);
        System.out.println("int value, iv: " + iv);
        System.out.println("float value, fv: " + fv);
        System.out.println("double value, dv: " + dv);
        System.out.println("char value, cv: " + cv);
    }
}
```

Output:

```
Values of Wrapper objects (printing as objects)

Byte object byteobj:  1

Integer object intobj:  10

Float object floatobj:  18.6

Double object doubleobj:  250.5

Character object charobj: a

Unwrapped values (printing as data types)

byte value, bv: 1

int value, iv: 10
```

```
float value, fv: 18.6
double value, dv: 250.5
char value, cv: a
```

Overflow and Underflow

Simply put, overflow and underflow happen when we assign a value that is out of range of the declared data type of the variable.

If the (absolute) value is too big, we call it overflow, if the value is too small, we call it underflow.

Let's look at an example where we attempt to assign the value 10^{1000} (a *1* with *1000* zeros) to a variable of type *int* or *double*. The value is too big for an *int* or *double* variable in Java, and there will be an overflow.

As a second example, let's say we attempt to assign the value 10^{-1000} (which is very close to 0) to a variable of type *double*. This value is too small for a *double* variable in Java, and there will be an underflow.

Let's see what happens in Java in these cases in more detail.

3. Integer Data Types

The integer data types in Java are *byte* (8 bits), *short* (16 bits), *int* (32 bits), and *long* (64 bits).

Here, we'll focus on the *int* data type. The same behavior applies to the other data types, except that the minimum and maximum values differ.

An integer of type *int* in Java can be negative or positive, which means with its 32 bits, we can assign values between -2^{31} (*-2147483648*) and $2^{31}-1$ (*2147483647*).

The wrapper class *Integer* defines two constants that hold these values: *Integer.MIN_VALUE* and *Integer.MAX_VALUE*.

3.1. Example

What will happen if we define a variable *m* of type *int* and attempt to assign a value that's too big (e.g., *21474836478 = MAX_VALUE + 1)?*

A possible outcome of this assignment is that the value of *m* will be undefined or that there will be an error.

Both are valid outcomes; however, in Java, the value of *m* will be *-2147483648* (the minimum value). On the other hand, if we attempt to assign a value of -2147483649 (= *MIN_VALUE – 1*), *m* will be *2147483647* (the maximum value). **This behavior is called integer-wraparound.**

Let's consider the following code snippet to illustrate this behavior better:

```
int value = Integer.MAX_VALUE-1;
for(int i = 0; i < 4; i++, value++) {
    System.out.println(value);
}
```

We'll get the following output, which demonstrates the overflow:

```
2147483646
2147483647
-2147483648
-2147483647
```

4. Handling Underflow and Overflow of Integer Data Types

Java does not throw an exception when an overflow occurs; that is why it can be hard to find errors resulting from an overflow. Nor can we directly access the overflow flag, which is available in most CPUs.

However, there are various ways to handle a possible overflow. Let's look at several of these possibilities.

4.1. Use a Different Data Type

If we want to allow values larger than *2147483647* (or smaller than *-2147483648*), we can simply use the *long* data type or a *BigInteger* instead.

Though variables of type *long* can also overflow, the minimum and maximum values are much larger and are probably sufficient in most situations.

The value range of *BigInteger* is not restricted, except by the amount of memory available to the JVM.

Let's see how to rewrite our above example with *BigInteger*

```
BigInteger largeValue = new BigInteger(Integer.MAX_VALUE + "");
for(int i = 0; i < 4; i++) {
    System.out.println(largeValue);
    largeValue = largeValue.add(BigInteger.ONE);
}
```

We'll see the following output:

```
2147483647
2147483648
2147483649
2147483650
```
As we can see in the output, there's no overflow here. Our article *BigDecimal* and *BigInteger* in Java covers *BigInteger* in more detail.

4.2. Throw an Exception

There are situations where we don't want to allow larger values, nor do we want an overflow to occur, and we want to throw an exception instead.

As of Java 8, we can use the methods for exact arithmetic operations. Let's look at an example first:

```
int value = Integer.MAX_VALUE-1;
for(int i = 0; i < 4; i++) {
   System.out.println(value);
   value = Math.addExact(value, 1);
}
```
The static method *addExact()* performs a normal addition, but throws an exception if the operation results in an overflow or underflow:

```
2147483646
2147483647
Exception in thread "main" java.lang.ArithmeticException: integer overflow
        at java.lang.Math.addExact(Math.java:790)
        at
baeldung.underoverflow.OverUnderflow.main(OverUnderflow.java:115)
```
In addition to *addExact()*, the *Math* package in Java 8 provides corresponding exact methods for all arithmetic operations. See the Java documentation for a list of all these methods.

Furthermore, there are exact conversion methods, which throw an exception if there is an overflow during the conversion to another data type.

For the conversion from a *long* to an *int*:

```
public static int toIntExact(long a)
```
And for the conversion from *BigInteger* to an *int* or *long*:

```
BigInteger largeValue = BigInteger.TEN;
long longValue = largeValue.longValueExact();
int intValue = largeValue.intValueExact();
```

4.3. Before Java 8

The exact arithmetic methods were added to Java 8. If we use an earlier version, we can simply create these methods ourselves. One option to do so is to implement the same method as in Java 8:

```
public static int addExact(int x, int y) {
    int r = x + y;
    if (((x ^ r) & (y ^ r)) < 0) {
        throw new ArithmeticException("int overflow");
    }
    return r;
}
```

5. Non-Integer Data Types

The non-integer types *float* and *double* do not behave in the same way as the integer data types when it comes to arithmetic operations

One difference is that arithmetic operations on floating-point numbers can result in a *NaN*. We have a dedicated article on NaN in Java, so we won't look further into that in this article. Furthermore, there are no exact arithmetic methods such as *addExact* or *multiplyExact* for non-integer types in the *Math* package.

Java follows the IEEE Standard for Floating-Point Arithmetic (IEEE 754) for its *float* and *double* data types. This standard is the basis for the way that Java handles over- and underflow of floating-point numbers.

In the below sections, we'll focus on the over- and underflow of the *double* data type and what we can do to handle the situations in which they occur.

5.1. Overflow

As for the integer data types, we might expect that:

```
assertTrue(Double.MAX_VALUE + 1 == Double.MIN_VALUE);
```

However, that is not the case for floating-point variables. The following is true:

```
assertTrue(Double.MAX_VALUE + 1 == Double.MAX_VALUE);
```

This is because a *double* value has only a limited number of significant bits. If we increase the value of a large *double* value by only one, we do not change any of the significant bits. Therefore, the value stays the same.

If we increase the value of our variable such that we increase one of the significant bits of the variable, the variable will have the value *INFINITY*:

```
assertTrue(Double.MAX_VALUE * 2 == Double.POSITIVE_INFINITY);
```
and *NEGATIVE_INFINITY* for negative values:

```
assertTrue(Double.MAX_VALUE * -2 == Double.NEGATIVE_INFINITY);
```
We can see that, unlike for integers, there's no wraparound, but two different possible outcomes of the overflow: the value stays the same, or we get one of the special values, *POSITIVE_INFINITY* or *NEGATIVE_INFINITY*.

5.2. Underflow

There are two constants defined for the minimum values of a *double* value: *MIN_VALUE* (4.9e-324) and *MIN_NORMAL* (2.2250738585072014E-308).

IEEE Standard for Floating-Point Arithmetic (IEEE 754) explains the details for the difference between those in more detail

Let's focus on why we need a minimum value for floating-point numbers at all.

A *double* value cannot be arbitrarily small as we only have a limited number of bits to represent the value.

The chapter about Types, Values, and Variables in the Java SE language specification describes how floating-point types are represented. The minimum exponent for the binary representation of a *double* is given as *-1074*. That means the smallest positive value a double can have is *Math.pow(2, -1074)*, which is equal to *4.9e-324*.

As a consequence, the precision of a *double* in Java does not support values between 0 and *4.9e-324,* or between *-4.9e-324* and *0* for negative values.

So what happens if we attempt to assign a too-small value to a variable of type *double*? Let's look at an example:

```
for(int i = 1073; i <= 1076; i++) {
    System.out.println("2^" + i + " = " + Math.pow(2, -i));
}
```
With output:

```
2^1073 = 1.0E-323
2^1074 = 4.9E-324
2^1075 = 0.0
2^1076 = 0.0
```
We see that if we assign a value that's too small, we get an underflow, and the resulting value is *0.0* (positive zero).
Similarly, for negative values, an underflow will result in a value of *-0.0* (negative zero).

6. Detecting Underflow and Overflow of Floating-Point Data Types

As overflow will result in either positive or negative infinity, and underflow in a positive or negative zero, we do not need exact arithmetic methods like for the integer data types. Instead, we can check for these special constants to detect over- and underflow.

If we want to throw an exception in this situation, we can implement a helper method. Let's look at how that can look for the exponentiation:

```
public static double powExact(double base, double exponent) {
    if(base == 0.0) {
        return 0.0;
    }

    double result = Math.pow(base, exponent);

    if(result == Double.POSITIVE_INFINITY ) {
        throw new ArithmeticException("Double overflow resulting in
POSITIVE_INFINITY");
    } else if(result == Double.NEGATIVE_INFINITY) {
        throw new ArithmeticException("Double overflow resulting in
NEGATIVE_INFINITY");
    } else if(Double.compare(-0.0f, result) == 0) {
        throw new ArithmeticException("Double overflow resulting in
negative zero");
    } else if(Double.compare(+0.0f, result) == 0) {
        throw new ArithmeticException("Double overflow resulting in
positive zero");
    }

    return result;
}
```

In this method, we need to use the method *Double.compare()*. The normal comparison operators (< and >) do not distinguish between positive and negative zero.

7. Positive and Negative *Zero*

Finally, let's look at an example that shows why we need to be careful when working with positive and negative zero and infinity.

Let's define a couple of variables to demonstrate:

```
double a = +0f;
double b = -0f;
```

Because positive and negative *0* are considered equal:

```
assertTrue(a == b);
```

Whereas positive and negative infinity are considered different:

```
assertTrue(1/a == Double.POSITIVE_INFINITY);
assertTrue(1/b == Double.NEGATIVE_INFINITY);
```

However, the following assertion is correct:

```
assertTrue(1/a != 1/b);
```
Which seems to be a contradiction to our first assertion.

Java Type Casting

Type Casting

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.

In Java, there are 13 types of type conversion. However, in this tutorial, we will only focus on the major 2 types.

1. Widening Type Casting

2. Narrowing Type Casting

Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.

Example: Converting int to double

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value: " + num);

    // convert into double type
    double data = num;
    System.out.println("The double value: " + data);
  }
}
```

Output

```
The integer value: 10
The double value: 10.0
```

In the above example, we are assigning the int type variable named num to a double type variable named data.

Here, the Java first converts the int type data into the double type. And then assign it to the double variable.

In the case of **Widening Type Casting**, the lower data type (having smaller size) is converted into the higher data type (having larger size). Hence there is no loss in data. This is why this type of conversion happens automatically.

Note: This is also known as **Implicit Type Casting**.

Narrowing Type Casting

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

Example: Converting double into an int

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type
    int data = (int)num;
    System.out.println("The integer value: " + data);
  }
}
```

Output

```
The double value: 10.99
The integer value: 10
```

In the above example, we are assigning the double type variable named num to an int type variable named data.

Notice the line,

```
int data = (int)num;
```

Here, the int keyword inside the parenthesis indicates that that the num variable is converted into the int type.

In the case of **Narrowing Type Casting**, the higher data types (having larger size) are converted into lower data types (having smaller size). Hence there is the loss of data. This is why this type of conversion does not happen automatically.

Note: This is also known as **Explicit Type Casting**.

Let's see some of the examples of other type conversions in Java.

Example 1: Type conversion from int to String

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value is: " + num);

    // converts int to string type
    String data = String.valueOf(num);
    System.out.println("The string value is: " + data);
  }
}
```

Output

```
The integer value is: 10
The string value is: 10
```

In the above program, notice the line

```
String data = String.valueOf(num);
```

Here, we have used the valueOf() method of the Java String class to convert the int type variable into a string.

Example 2: Type conversion from String to int

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

```
    // create string type variable
    String data = "10";
    System.out.println("The string value is: " + data);

    // convert string variable to int
    int num = Integer.parseInt(data);
    System.out.println("The integer value is: " + num);
  }
}
```

Output

```
The string value is: 10
The integer value is: 10
```

In the above example, notice the line

```
int num = Integer.parseInt(data);
```

Here, we have used the `parseInt()` method of the Java `Integer` class to convert a string type variable into an `int` variable.

Note: If the string variable cannot be converted into the integer variable then an exception named `NumberFormatException` occurs.

What are Java code blocks and how to use them

Code is nothing more than text and whitespace. Anyone can put code into a computer. Making it readable though is another thing altogether. Let's review some key parts of Java code blocks and how to put it into a readable program. Code blocks are similar to Java packages but have different uses.

Java Code Blocks

Java likes to keep things between the lines, no make that curly braces({}). This how we can start and stop a code block in Java. Braces gives the compiler a heads up of where we want to start({) and end(}).

```
public class JavaCodeBlocks {
        public static void main(String arg[]){
           System.out.println("Inside the Java code block");
         }
}
```

In our first example we have not one but two code blocks. First, for the JavaCodeBlock class with the open brace on line 3. Next on line 4 we open the main method with another open brace. Then we close it up on line 6 for the main method and line 7 for the class. A pro tip, most IDEs will highlight the closing brace when you put your cursor by the open brace and vice versa too.

```
package com.myitcareercoach;

public class IfCodeBlocks {

        public static void main(String arg[]){

           boolean check = true;

                if(check) {

                        System.out.println("Check is true");

                } else {

                        System.out.println("Check is false");

                }



                {

                        System.out.println("Just a random code block");

                }          }

}
```

In this example we have an if else with each having their own code block. We don't have to but helps with readability. Later on we have a random code block too. It looks like it should be wrong if you are new to development but it is perfectly legitimate.

Anonymous

Anonymous code blocks really stand out. They are also called instance initialization blocks. They are run before the constructor is called

```
package com.myitcareercoach;

public class AnonymousCodeBlocks {

	{

		System.out.println("Anonymous Code Blocks");

	}

	public AnonymousCodeBlocks() {

		super();

		System.out.println("Constructor");

	}

	public static void main(String arg[]){

		AnonymousCodeBlocks block = new AnonymousCodeBlocks();

	}

}
```

As you can see in this example we can put the anonymous code block into the class and it will be called prior to the constructor. If you run this example the output would be.

```
Anonymous Code Blocks
Constructor
```

Named Code Blocks

We can also create blocks of code and give them a name. Unlike your pet rock these can be useful.

```
package com.myitcareercoach;

public class NamedCodeBlock {

	public static void main(String[] args) {

		int var1 = 10;

		int var2 = 12;

		codeblock: {

			if (var1 < 0) {

				break codeblock;

			}

			if (var2 < 0) {

				break codeblock;

			}

			System.out.println(var1 + var2);

		}

	}

}
```

Here we see how we can use the named code block to jump out of an if statement. The same thing could be done out of a loop. Many developers consider this akin to using a goto statement. **So you might want to avoid using this.**

Data Types in Java

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

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

Java Primitive Data Types

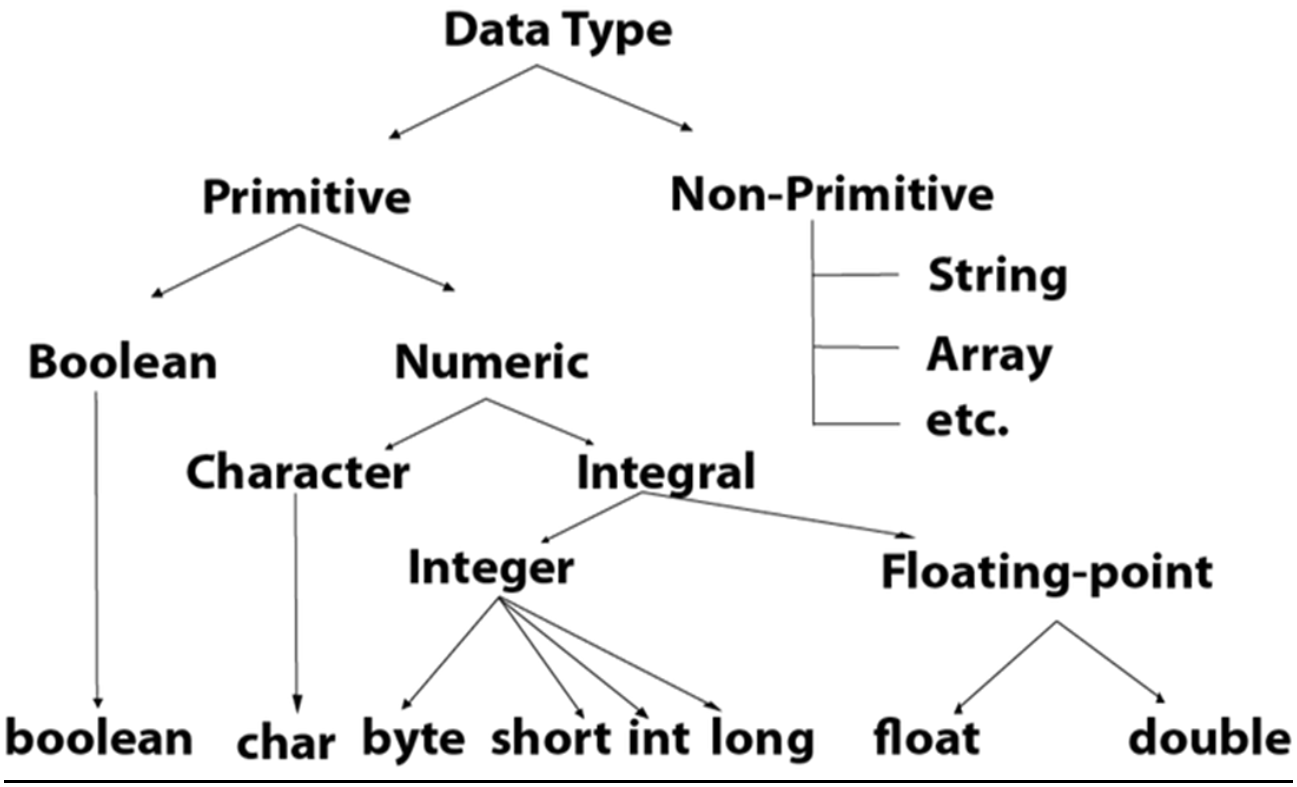
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type

  double data type



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

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is -9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example: char letterA = 'A'

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Unicode System

Unicode is a universal international standard character encoding that is capable of representi
written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- ASCII (American Standard Code for Information Interchange) for the United States.
- ISO 8859-1 for Western European Language.
- KOI-8 for Russian.
- GB18030 and BIG-5 for chinese, and so on.

Problem

This caused two problems:

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

Solution

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

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

lowest value:\u0000

highest value:\uFFFF

Operators in Java

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

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

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,

- Bitwise Operator,
- Logical Operator,
- Ternary Operator and

  Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :

Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >

Java Unary Operator

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

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

Java Unary Operator Example: ++ and —-

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
```

}}

Output:

22

21

Java Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-
11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
 }}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

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

Java Arithmetic Operator Example

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

Java Arithmetic Operator Example

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

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

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

Output:

```
21
```

Java Left Shift Operator

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

Java Left Shift Operator Example

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

Output:

```
40
80
80
240
```

Java Right Shift Operator

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

Java Right Shift Operator Example

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

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

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

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

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

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

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

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

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

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

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

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

Output:

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

Java Ternary Operator

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

Java Ternary Operator Example

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

Output:

```
2
```

Another Example:

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

```
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
5
```

Java Assignment Operator

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

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

Java Assignment Operator Example

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

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

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

Output:

```
Compile time error
```

After type cast:

```
class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

- if statement
- if-else statement
- if-else-if ladder

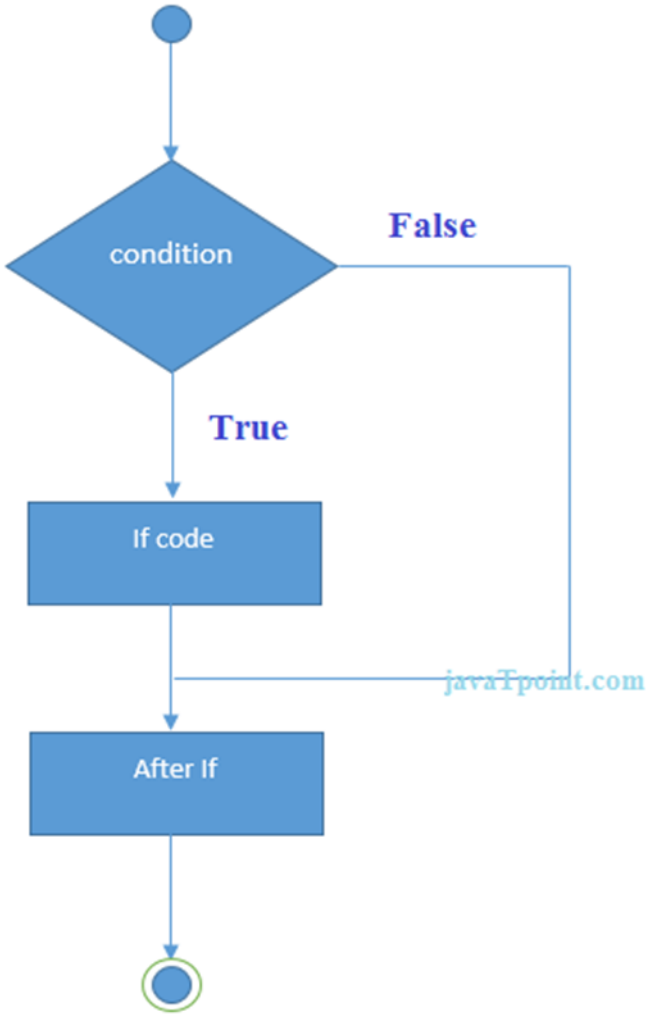
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
if(condition){
//code to be executed
}
```



Example:

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Test it Now

Output:

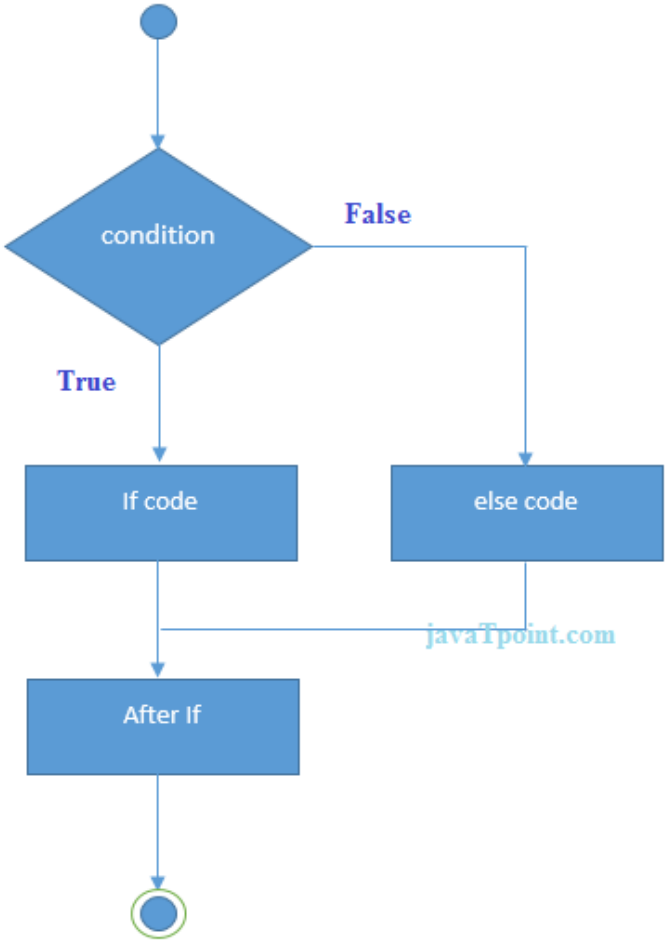
```
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example:

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
        //defining a variable
         int number=13;
         //Check if the number is divisible by 2 or not
         if(number%2==0){
            System.out.println("even number");
         }else{
            System.out.println("odd number");
         }
    }
    }
```

Test it Now

Output:

```
odd number
```

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
 public class LeapYearExample {
 public static void main(String[] args) {
int year=2020;
    if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
       System.out.println("LEAP YEAR");
}
    else{
       System.out.println("COMMON YEAR");
    }
 }
 }
```

Output:

```
LEAP YEAR
```

Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

Example:

```
public class IfElseTernaryExample {
public static void main(String[] args) {
   int number=13;
   //Using ternary operator
   String output=(number%2==0)?"even number":"odd number";
   System.out.println(output);
}
}
```

Output:

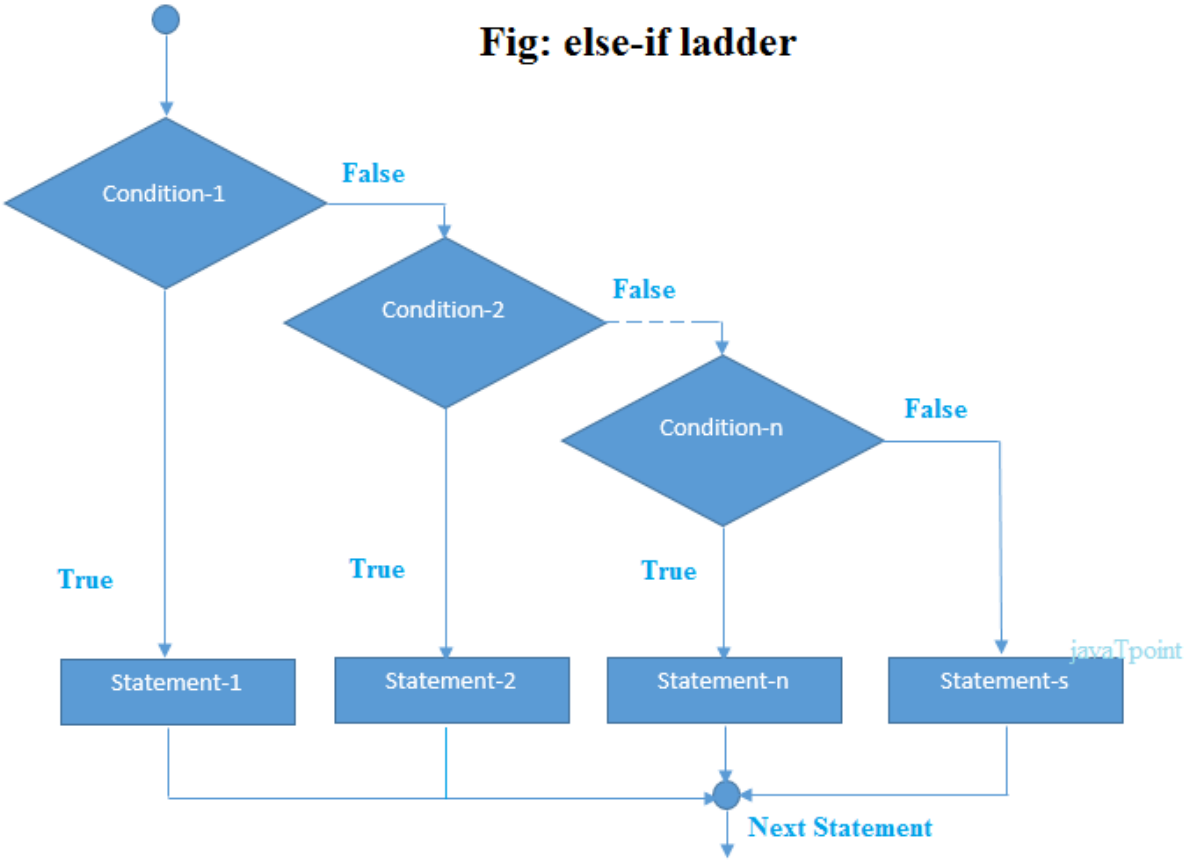
```
odd number
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
    //code to be executed if condition1 is true
    }else if(condition2){
    //code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```



Example:

```
//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.

    public class IfElseIfExample {
public static void main(String[] args) {
   int marks=65;

       if(marks<50){
       System.out.println("fail");
   }
   else if(marks>=50 && marks<60){
       System.out.println("D grade");
   }
   else if(marks>=60 && marks<70){
       System.out.println("C grade");
   }
   else if(marks>=70 && marks<80){
     System.out.println("B grade");
        }
        else if(marks>=80 && marks<90){
           System.out.println("A grade");
        }else if(marks>=90 && marks<100){
           System.out.println("A+ grade");
        }else{
           System.out.println("Invalid!");
   }
    }
}
```

Output:

```
C grade
```

Program to check POSITIVE, NEGATIVE or ZERO:

```
    public class PositiveNegativeExample {
    public static void main(String[] args) {
        int number=-13;
        if(number>0){
   System.out.println("POSITIVE");
   }else if(number<0){
   System.out.println("NEGATIVE");
        }else{
```

```
        System.out.println("ZERO");
       }
    }
    }
```

Output:

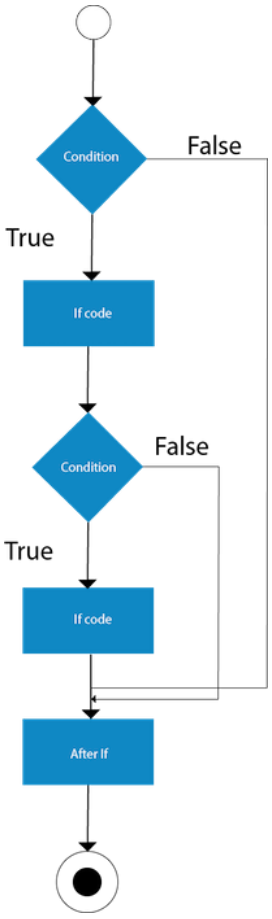
```
NEGATIVE
```

Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed
   }
}
```



Example:

```
//Java Program to demonstrate the use of Nested If Statement.
    public class JavaNestedIfExample {
public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=20;
   int weight=80;
   //applying condition on age and weight
   if(age>=18){
      if(weight>50){
         System.out.println("You are eligible to donate blood");
      }
   }
}}
```

Test it Now

Output:

```
You are eligible to donate blood
```

Example 2:

```
//Java Program to demonstrate the use of Nested If Statement.
    public class JavaNestedIfExample2 {
    public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=25;
   int weight=48;
   //applying condition on age and weight
   if(age>=18){
      if(weight>50){
         System.out.println("You are eligible to donate blood");
      } else{
       System.out.println("You are not eligible to donate blood");
          }
   } else{
     System.out.println("Age must be greater than 18");
   }
} }
```

Test it Now

Output:

```
You are not eligible to donate blood
```

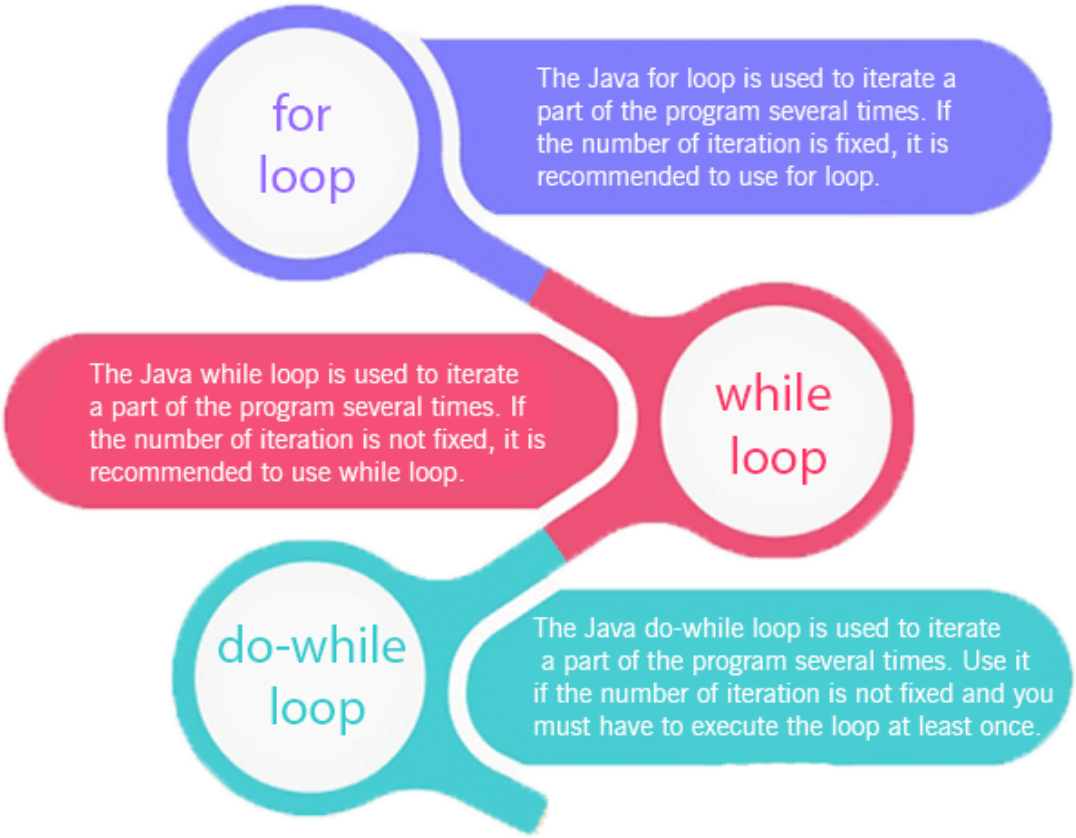
Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

for loop

while loop

do-while loop



Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do w
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	Th co ex pr fu up co
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If no ex it i do

Syntax	`for(init;condition;incr/decr){`
	`// code to be executed`
	`}`
		`while(condition){`
		`//code to be executed`
		`}`
			`do`
			`//`
			`}w`

Example	`//for loop`
	`for(int i=1;i<=10;i++){`
	`System.out.println(i);`
	`}`
		`//while loop`
		`int i=1;`
		`while(i<=10){`
		`System.out.println(i);`
		`i++;`
		`}`
			`//`
			`in`
			`do`
			`Sy`
			`i+`
			`}w`

Syntax for infinitive loop	`for(;;){`
	`//code to be executed`
	`}`
		`while(true){`
		`//code to be executed`
		`}`
			`do`
			`//`
			`}w`

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

Java Simple For Loop

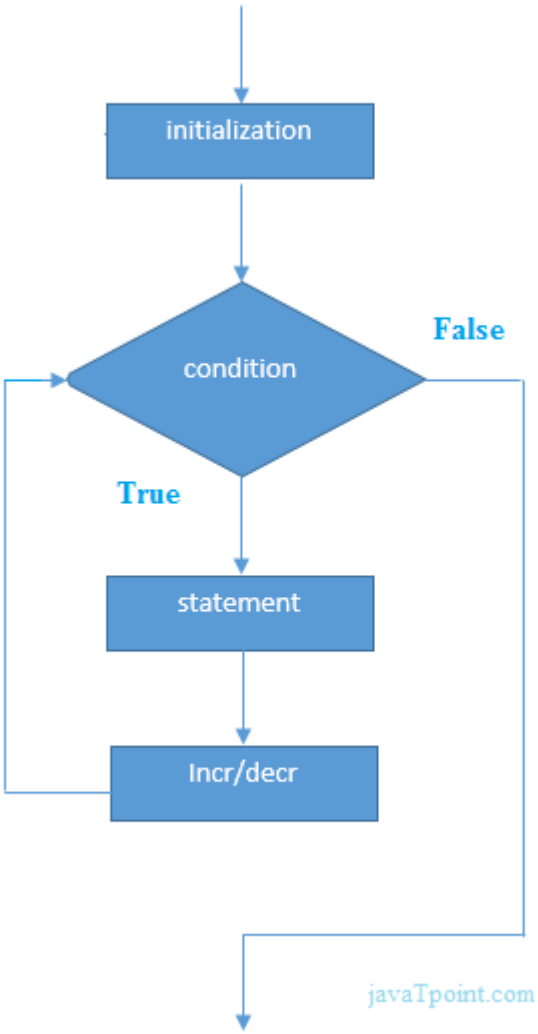
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

Syntax:

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```

FlowChart:



Example:

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Test it Now

Output:

```
1
2
```

```
3
4
5
6
7
8
9
10
```

Java Nested For Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes

Example:

```
    public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
        System.out.println(i+" "+j);
    }//end of i
    }//end of j
    }
    }
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Pyramid Example 1:

```
    public class PyramidExample {
    public static void main(String[] args) {
    for(int i=1;i<=5;i++){
    for(int j=1;j<=i;j++){
        System.out.print("* ");
    }
    System.out.println();//new line
```

```
    }
    }
    }
```

Output:

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

Pyramid Example 2:

```
    public class PyramidExample2 {
    public static void main(String[] args) {
    int term=6;
    for(int i=1;i<=term;i++){
    for(int j=term;j>=i;j--){
      System.out.print("* ");
}
    System.out.println();//new line
    }
    }
    }
```

Output:

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

Java for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```
    for(Type var:array){
```

```
//code to be executed
}
```

Example:

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
   //Declaring an array
   int arr[]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr){
      System.out.println(i);
   }
}
}
```

Test it Now

Output:

```
12
23
44
56
78
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Usually, break and continue keywords breaks/continues the innermost for loop only.

Syntax:

```
labelname:
for(initialization;condition;incr/decr){
//code to be executed
}
```

Example:

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
   //Using Label for outer and for loop
   aa:
      for(int i=1;i<=3;i++){
         bb:
        for(int j=1;j<=3;j++){
              if(i==2&&j==2){
                 break aa;
           }
              System.out.println(i+" "+j);
          }
      }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

```
public class LabeledForExample2 {
public static void main(String[] args) {
   aa:
      for(int i=1;i<=3;i++){
         bb:
        for(int j=1;j<=3;j++){
              if(i==2&&j==2){
                 break bb;
              }
              System.out.println(i+" "+j);
           }
      }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

```
3 1
3 2
3 3
```

Java Infinitive For Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Example:

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
   //Using no condition in for loop
   for(;;){
      System.out.println("infinitive loop");
   }
}
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
```

What is the difference between = (Assignment) and == (Equal to) operators

= operator

The "**=**" is an assignment operator is used to assign the value on the right to the variable on the left.

For example:

```
a = 10;

b = 20;

ch = 'y';
```

Example:

```
// C program to demonstrate
// working of Assignment operators

#include <stdio.h>

int main()
{

    // Assigning value 10 to a
    // using "=" operator
    int a = 10;
    printf("Value of a is %d\n", a);

    return 0;
}
```

Output:

```
Value of a is 10
```

== operator

The '==' operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false.

For example:

```
5==5


This will return true.
```

Example:

```
// C program to demonstrate
// working of relational operators
#include <stdio.h>

int main()
{
    int a = 10, b = 4;

    // equal to
    if (a == b)
        printf("a is equal to b\n");
    else
        printf("a and b are not equal\n");

    return 0;
}
```

Output:

```
a and b are not equal
```

The differences can be shown in tabular form as follows:

=	==
It is an assignment operator.	It is a relational or comparison operator.
It is used for assigning the value to a variable.	It is used for comparing two values. It returns 1 if both the values are equal otherwise returns 0.
Constant term cannot be placed on left hand side. Example: 1=x; is invalid.	Constant term can be placed in the left hand side. Example: 1==1 is valid and returns 1.

Java Ternary Operator

In this tutorial, you will learn about the ternary operator and its use in Java with the help of examples.

In Java, a ternary operator can be used to replace the if...else statement in certain situations. Before you learn about the ternary operator, make sure you visit Java if...else statement.

Ternary Operator in Java

A ternary operator evaluates the test condition and executes a block of code based on the result of the condition.

It's syntax is:

```
condition ? expression1 : expression2;
```

Here, `condition` is evaluated and

- if condition is true, expression1 is executed.
- And, if condition is false, expression2 is executed.

The ternary operator takes **3 operands** (condition, expression1, and expression2). Hence, the name **ternary operator**.

Example: Java Ternary Operator

```
import java.util.Scanner;

class Main {
  public static void main(String[] args)
    // take input from users
    Scanner input = new Scanner(System.in);
    System.out.println("Enter your marks: ");
    double marks = input.nextDouble();

    // ternary operator checks if
    // marks is greater than 40
    String result = (marks > 40) ? "pass" : "fail";

    System.out.println("You " + result + " the exam.");
    input.close();
  }
}
```

Output 1

```
Enter your marks:
75
You pass the exam.
```

Suppose the user enters 75. Then, the condition marks > 40 evaluates to true. Hence, the first expression pass is assigned to result.

Output 2

Enter your marks:

24

You fail the exam.

Now, suppose the user enters 24. Then, the condition marks > 40 evaluates to false. Hence, the second expression fail is assigned to result.

When to use the Ternary Operator?

In Java, the ternary operator can be used to replace certain types of if...else statements. For example,

You can replace this code

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

    // create a variable
    int number = 24;

    if(number > 0) {
      System.out.println("Positive Number");
    }
    else {
      System.out.println("Negative Number");
    }
  }
}
```

with

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

    // create a variable
    int number = 24;

    String result = (number > 0) ? "Positive Number" : "Negative Number";
    System.out.println(result);
  }
}
```

Output

```
Positive Number
```

Here, both programs give the same output. However, the use of the ternary operator makes our code more readable and clean.

Note: You should only use the ternary operator if the resulting statement is short.

Nested Ternary Operators

It is also possible to use one ternary operator inside another ternary operator. It is called the nested ternary operator in Java.

Here's a program to find the largest of 3 numbers using the nested ternary operator.

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

    // create a variable
```

```
    int n1 = 2, n2 = 9, n3 = -11;

    // nested ternary operator
    // to find the largest number
    int largest = (n1 >= n2) ? ((n1 >= n3) ? n1 : n3) : ((n2 >= n3) ? n2 : n3);
    System.out.println("Largest Number: " + largest);
  }
}
```

Output

```
Largest Number: 9
```

In the above example, notice the use of the ternary operator,

```
(n1 >= n2) ? ((n1 >=n3) ? n1 : n3) : ((n2 >= n3) ? n2 : n3);
```

Here,

- (n1 >= n2) - first test condition that checks if n1 is greater than n2
- (n1 >= n3) - second test condition that is executed if the first condition is true
- (n2 >= n3) - third test condition that is executed if the first condition is false

Note: It is not recommended to use nested ternary operators. This is because it makes our code more complex.

Method in Java

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**. In this section, we will learn **what is a method in Java, types of methods, method declaration,** and **how to call a method in Java**.

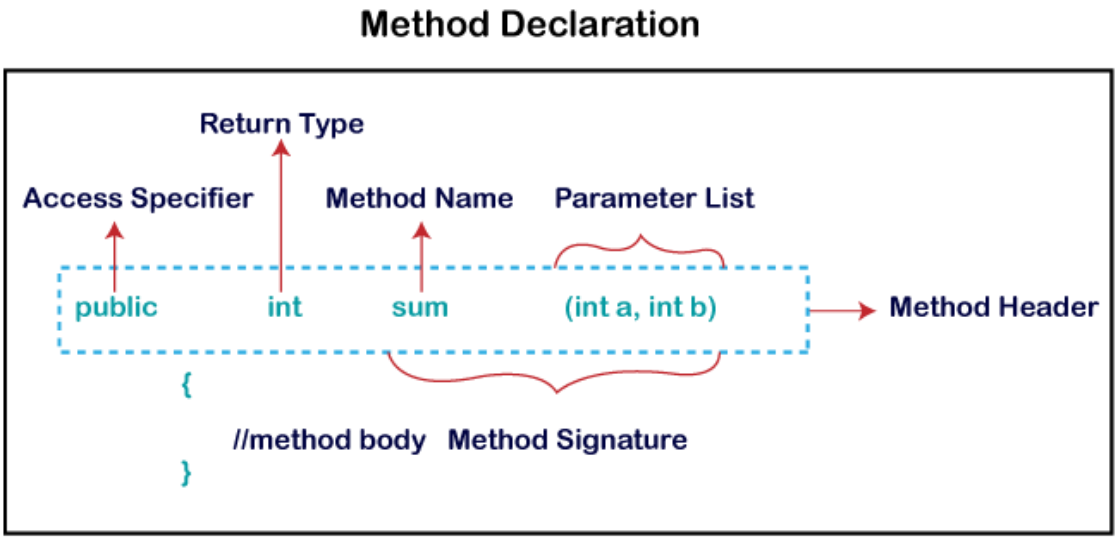
What is a method in Java?

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.

- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

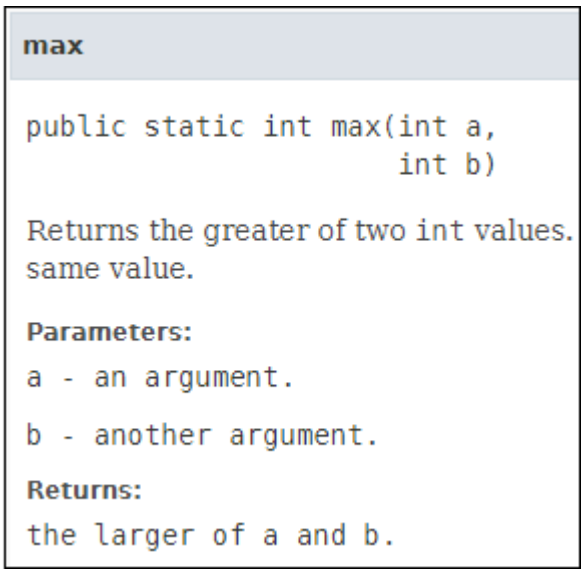
```
public class Demo
    {
    public static void main(String[] args)
    {
    // using the max() method of Math class
    System.out.print("The maximum number is: " + Math.max(9,7));
    }
    }
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods **main(), print(),** and **max()**. We have used these methods directly without declaration because they are predefined. The print() method is a method of **PrintStream** class that prints the result on the console. The max() method is a method of the **Math** class that returns the greater of two numbers.

We can also see the method signature of any predefined method by using the link https://docs.oracle.com/. When we go through the link and see the max() method signature, we find the following:



max

```
public static int max(int a,
                      int b)
```

Returns the greater of two int values. same value.

Parameters:

a - an argument.

b - another argument.

Returns:

the larger of a and b.

In the above method signature, we see that the method signature has access specifier **public**, non-access modifier **static**, return type **int**, method name **max(),** parameter list **(int a, int b).** In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

Similarly, we can also see the method signature of the print() method.

User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
    public static void findEvenOdd(int num)
    {
    //method body
    if(num%2==0)
System.out.println(num+" is even");
    else
    System.out.println(num+" is odd");
    }
```

We have defined the above method named findevenodd(). It has a parameter **num** of type int. The method does not return any value that's why we have used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
```

In the above code snippet, as soon as the compiler reaches at line **findEvenOdd(num),** the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

EvenOdd.java

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
```

```
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
}
```

Output 1:

```
Enter the number: 12
12 is even
```

Output 2:

```
Enter the number: 99
99 is odd
```

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named **add()** that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Addition.java

```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b);   //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2)   //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
```

Output:

```
The sum of a and b is= 24
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

Display.java

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output:

```
It is an example of a static method.
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
public class InstanceMethodExample
{
public static void main(String [] args)
{
//Creating an object of the class
InstanceMethodExample obj = new InstanceMethodExample();
//invoking instance method
```

```
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
}
}
```

Output:

```
The sum is: 25
```

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

Example

```
public int getId()
{
return Id;
}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)
{
this.roll = roll;
}
```

Example of accessor and mutator method

Student.java

```
public class Student
{
private int roll;
private String name;
public int getRoll()    //accessor method
{
return roll;
}
public void setRoll(int roll) //mutator method
{
this.roll = roll;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name = name;
}
public void display()
{
System.out.println("Roll no.: "+roll);
System.out.println("Student name: "+name);
}
}
```

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Syntax

```
abstract void method_name();
```

Example of abstract method

Demo.java

```
abstract class Demo //abstract class
{
//abstract method declaration
abstract void display();
}
public class MyClass extends Demo
{
//method impelmentation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Demo obj = new MyClass();
//invoking abstract method
obj.display();
}
}
```

Output:

```
Abstract method...
```

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

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

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

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

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

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

1) Method Overloading: changing no. of arguments

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

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

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
```

```
}}
```

Test it Now

Output:

```
22
33
```

---

2) Method Overloading: changing data type of arguments

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

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Test it Now

Output:

```
22
24.9
```

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Test it Now

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

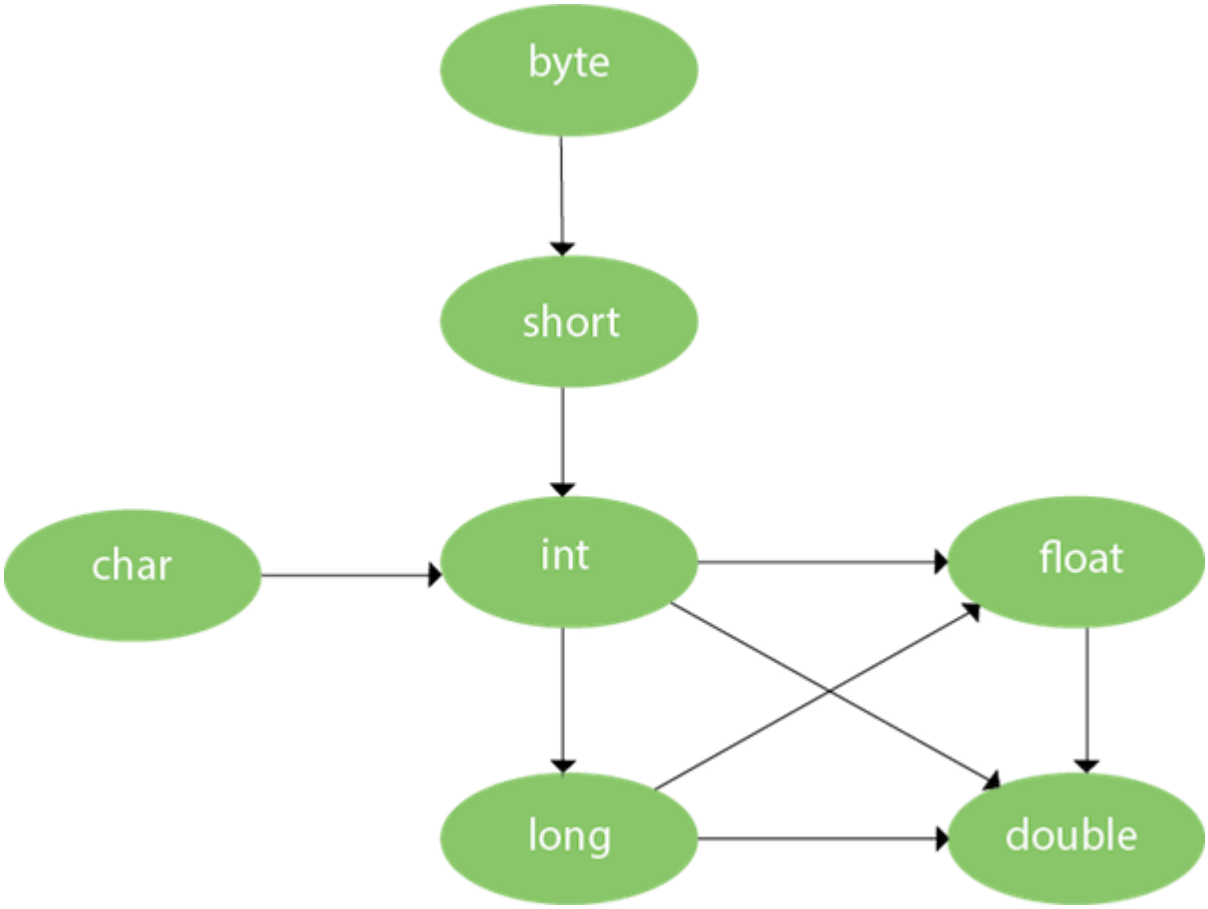
Test it Now

Output:

```
main with String[]
```

Method Overloading and Type Promotion

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



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

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```

Test it Now

```
Output:40
       60
```

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

Test it Now

```
Output:int arg method invoked
```

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
   class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```

Test it Now

```
Output:Compile Time Error
```

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

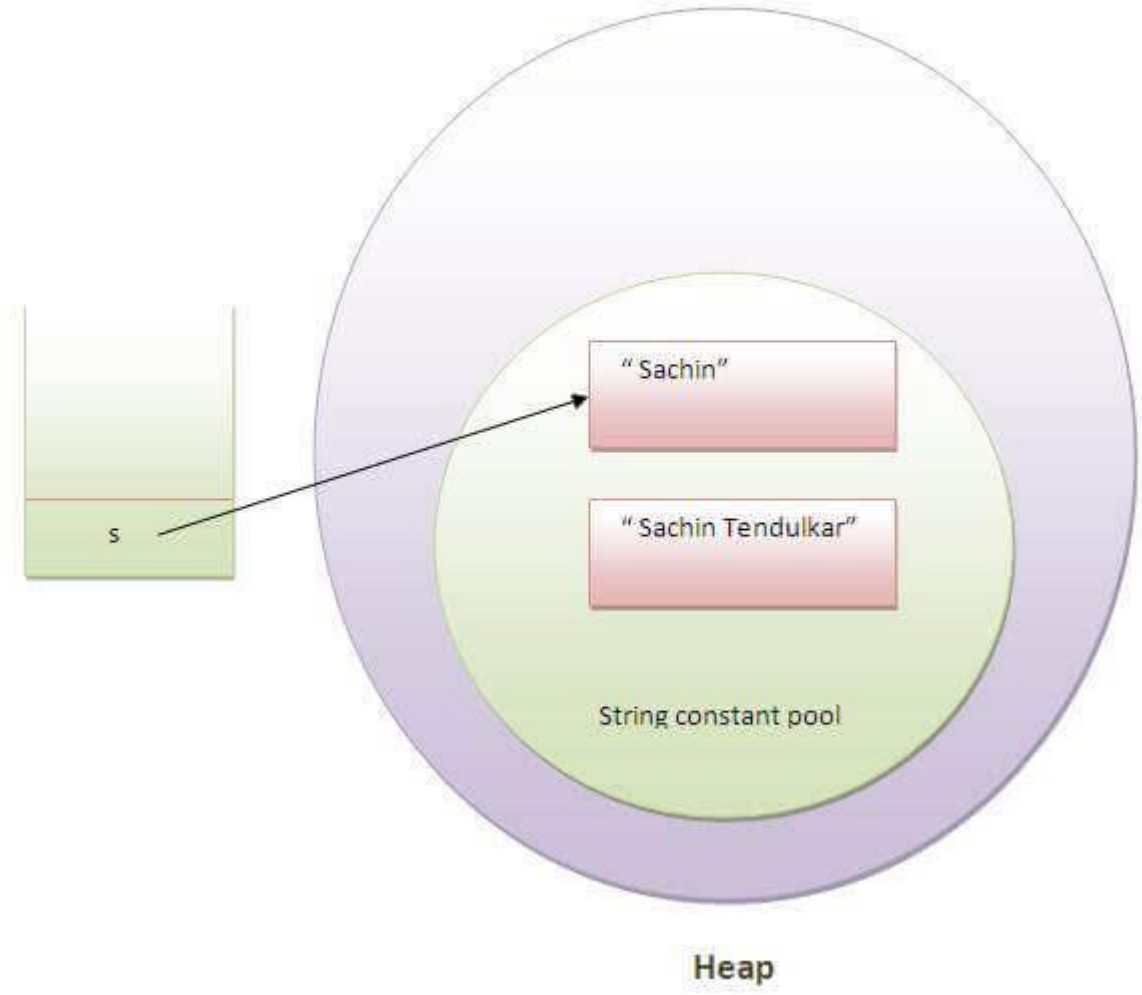
Let's try to understand the immutability concept by the example given below:

```
1. class Testimmutablestring{
2.  public static void main(String args[]){
3.    String s="Sachin";
4.    s.concat(" Tendulkar");//concat() method appends the string at the end
5.    System.out.println(s);//will print Sachin because strings are immutable objects
6.  }
7. }
```

Test it Now

```
Output:Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
1. class Testimmutablestring1{
2.  public static void main(String args[]){
3.    String s="Sachin";
4.    s=s.concat(" Tendulkar");
5.    System.out.println(s);
6.  }
7. }
```

Test it Now

```
Output:Sachin Tendulkar
```

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

Why string objects are immutable in java?

Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one reference variable changes the value of the object, it will be affected to all the reference variable objects are immutable in java.

Java - parseInt() Method

Description

This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.

Syntax

Following are all the variants of this method −

```
static int parseInt(String s)
static int parseInt(String s, int radix)
```

Parameters

Here is the detail of parameters −

- **s** − This is a string representation of decimal.
- **radix** − This would be used to convert String s into integer.

Return Value

- **parseInt(String s)** − This returns an integer (decimal only).
- **parseInt(int i)** − This returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.

Example

Live Demo

```
public class Test {

   public static void main(String args[]) {
      int x =Integer.parseInt("9");
      double c = Double.parseDouble("5");
      int b = Integer.parseInt("444",16);

      System.out.println(x);
      System.out.println(c);
      System.out.println(b);
   }
```

```
}
```

This will produce the following result −

Output

```
9
5.0
1092
```

Java Scanner

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Java Scanner Class Declaration

1. **public final class** Scanner
2. **extends** Object
3. **implements** Iterator<String>

How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

```
1. Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

```
1. Scanner in = new Scanner("Hello Javatpoint");
```

Java Scanner Class Constructors

SN	Constructor	Descriptio
1)	Scanner(File source)	It constructs a new Scanner tha the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner tha the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner tha the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It constructs a new Scanner tha the specified input stream.
5)	Scanner(Readable source)	It constructs a new Scanner tha the specified source.
6)	Scanner(String source)	It constructs a new Scanner tha the specified string.
7)	Scanner(ReadableByteChannel source)	It constructs a new Scanner tha the specified channel.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs a new Scanner tha the specified channel.
9)	Scanner(Path source)	It constructs a new Scanner tha the specified file.
10)	Scanner(Path source, String charsetName)	It constructs a new Scanner tha the specified file.

Java Scanner Class Methods

The following are the list of Scanner methods:

SN	Modifier & Type	Method	Description
1)	void	close()	It is used to close this scanner.
2)	pattern	delimiter()	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
3)	Stream<MatchResult>	findAll()	It is used to find a stream of match results that match the provided pattern string.
4)	String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	hasNext()	It returns true if this scanner has another token in its input.
7)	boolean	hasNextBigDecimal()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method o not.
8)	boolean	hasNextBigInteger()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method o not.
9)	boolean	hasNextBoolean()	It is used to check if the next token in this scanner's input can be interpreted as a Boolean using the nextBoolean() method or not.
10)	boolean	hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte usin the nextBigDecimal() method or not.

11)	boolean	hasNextDouble()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean	hasNextFloat()	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean	hasNextInt()	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean	hasNextLine()	It is used to check if there is another line in the input of this scanner or not.
15)	boolean	hasNextLong()	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
16)	boolean	hasNextShort()	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.
17)	IOException	ioException()	It is used to get the IOException last thrown by this Scanner's readable.
18)	Locale	locale()	It is used to get a Locale of the Scanner class.
19)	MatchResult	match()	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String	next()	It is used to get the next complete token from th scanner which is in use.
21)	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.

22)	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolea value and returns that value.
24)	byte	nextByte()	It scans the next token of the input as a byte.
25)	double	nextDouble()	It scans the next token of the input as a double.
26)	float	nextFloat()	It scans the next token of the input as a float.
27)	int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skippe of the Scanner object.
29)	long	nextLong()	It scans the next token of the input as a long.
30)	short	nextShort()	It scans the next token of the input as a short.
31)	int	radix()	It is used to get the default radix of the Scanner use.
32)	void	remove()	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner	reset()	It is used to reset the Scanner which is in use.
34)	Scanner	skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>	tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.

36)	String	toString()	It is used to get the string representation of Scanner using.
37)	Scanner	useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner	useLocale()	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner	useRadix()	It is used to set the default radix of the Scanner which is in use to the specified radix.

Example 1

Let's see a simple example of Java Scanner where we are getting a single input from the user. Here, we are asking for a string through in.nextLine() method.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
        }
}
```

Output:

```
Enter your name: sonoo jaiswal
Name is: sonoo jaiswal
```

Example 2

```
import java.util.*;
public class ScannerClassExample1 {
    public static void main(String args[]){
        String s = "Hello, This is JavaTpoint.";
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " +scan.nextLine());
```

```
        scan.close();
        System.out.println("--------Enter Your Details-------- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name: " + name);
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
        in.close();
        }
}
```

Output:

```
Boolean Result: true
String: Hello, This is JavaTpoint.
-------Enter Your Details---------
Enter your name: Abhishek
Name: Abhishek
Enter your age: 23
Age: 23
Enter your salary: 25000
Salary: 25000.0
```

Example 3

```
import java.util.*;
public class ScannerClassExample2 {
    public static void main(String args[]){
        String str = "Hello/This is JavaTpoint/My name is Abhishek.";
        //Create scanner with the specified String Object
        Scanner scanner = new Scanner(str);
        System.out.println("Boolean Result: "+scanner.hasNextBoolean());
        //Change the delimiter of this scanner
        scanner.useDelimiter("/");
        //Printing the tokenized Strings
        System.out.println("---Tokenizes String---");
       while(scanner.hasNext()){
          System.out.println(scanner.next());
       }
        //Display the new delimiter
        System.out.println("Delimiter used: " +scanner.delimiter());
        scanner.close();
        }
```

```
}
```

Output:

```
Boolean Result: false
---Tokenizes String---
Hello
This is JavaTpoint
My name is Abhishek.
Delimiter used: /
```

Java Math class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Example 1

```
public class JavaMathExample1
{
   public static void main(String[] args)
   {
      double x = 28;
     double y = 4;

      // return the maximum of two numbers
      System.out.println("Maximum number of x and y is: " +Math.max(x, y));

      // return the square root of y
      System.out.println("Square root of y is: " + Math.sqrt(y));

      //returns 28 power of 4 i.e. 28*28*28*28
    System.out.println("Power of x and y is: " + Math.pow(x, y));

      // return the logarithm of given value
```

```
        System.out.println("Logarithm of x is: " + Math.log(x));
        System.out.println("Logarithm of y is: " + Math.log(y));

        // return the logarithm of given value when base is 10
        System.out.println("log10 of x is: " + Math.log10(x));
        System.out.println("log10 of y is: " + Math.log10(y));

        // return the log of x + 1
        System.out.println("log1p of x is: " +Math.log1p(x));

        // return a power of 2
        System.out.println("exp of a is: " +Math.exp(x));

        // return (a power of 2)-1
        System.out.println("expm1 of a is: " +Math.expm1(x));
    }
}
```

Test it Now

Output:

```
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906
log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624
log1p of x is: 3.367295829986474
exp of a is: 1.446257064291475E12
expm1 of a is: 1.446257064290475E12
```

Example 2

```
public class JavaMathExample2
{
    public static void main(String[] args)
    {
        double a = 30;

        // converting values to radian
        double b = Math.toRadians(a);

        // return the trigonometric sine of a
        System.out.println("Sine value of a is: " +Math.sin(a));

        // return the trigonometric cosine value of a
        System.out.println("Cosine value of a is: " +Math.cos(a));
```

```
        // return the trigonometric tangent value of a
            System.out.println("Tangent value of a is: " +Math.tan(a));

            // return the trigonometric arc sine of a
            System.out.println("Sine value of a is: " +Math.asin(a));

        // return the trigonometric arc cosine value of a
            System.out.println("Cosine value of a is: " +Math.acos(a));

            // return the trigonometric arc tangent value of a
            System.out.println("Tangent value of a is: " +Math.atan(a));

            // return the hyperbolic sine of a
        System.out.println("Sine value of a is: " +Math.sinh(a));

            // return the hyperbolic cosine value of a
            System.out.println("Cosine value of a is: " +Math.cosh(a));

            // return the hyperbolic tangent value of a
            System.out.println("Tangent value of a is: " +Math.tanh(a));
         }
    }
```

Test it Now

Output:

```
Sine value of a is: -0.9880316240928618
Cosine value of a is: 0.15425144988758405
Tangent value of a is: -6.405331196646276
Sine value of a is: NaN
Cosine value of a is: NaN
Tangent value of a is: 1.5374753309166493
Sine value of a is: 5.343237290762231E12
Cosine value of a is: 5.343237290762231E12
Tangent value of a is: 1.0
```

Java Math Methods

The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:

Basic Math methods

Method	Description
Math.abs()	It will return the Absolute value of the given value.

Math.max()	It returns the Largest of two values.
Math.min()	It is used to return the Smallest of two values.
Math.round()	It is used to round of the decimal numbers to the nearest value.
Math.sqrt()	It is used to return the square root of a number.
Math.cbrt()	It is used to return the cube root of a number.
Math.pow()	It returns the value of first argument raised to the power to second arg
Math.signum()	It is used to find the sign of a given value.
Math.ceil()	It is used to find the smallest integer value that is greater than or equa mathematical integer.
Math.copySign()	It is used to find the Absolute value of first argument along with sign s
Math.nextAfter()	It is used to return the floating-point number adjacent to the first argu second argument.
Math.nextUp()	It returns the floating-point value adjacent to d in the direction of posit
Math.nextDown()	It returns the floating-point value adjacent to d in the direction of nega
Math.floor()	It is used to find the largest integer value which is less than or equal to to the mathematical integer of a double value.
Math.floorDiv()	It is used to find the largest integer value that is less than or equal to
Math.random()	It returns a double value with a positive sign, greater than or equal to
Math.rint()	It returns the double value that is closest to the given argument and e integer.

Math.hypot()	It returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
Math.ulp()	It returns the size of an ulp of the argument.
Math.getExponent()	It is used to return the unbiased exponent used in the representation o
Math.IEEEremainder()	It is used to calculate the remainder operation on two arguments as pr standard and returns value.
Math.addExact()	It is used to return the sum of its arguments, throwing an exception if or long.
Math.subtractExact()	It returns the difference of the arguments, throwing an exception if the
Math.multiplyExact()	It is used to return the product of the arguments, throwing an exceptio an int or long.
Math.incrementExact()	It returns the argument incremented by one, throwing an exception if
Math.decrementExact()	It is used to return the argument decremented by one, throwing an ex overflows an int or long.
Math.negateExact()	It is used to return the negation of the argument, throwing an exceptio an int or long.
Math.toIntExact()	It returns the value of the long argument, throwing an exception if the

Logarithmic Math Methods

Method	Description
Math.log()	It returns the natural logarithm of a double value.
Math.log10()	It is used to return the base 10 logarithm of a double value.
Math.log1p()	It returns the natural logarithm of the sum of the argument

Math.exp()	It returns E raised to the power of a double value, where E i is approximately equal to 2.71828.
Math.expm1()	It is used to calculate the power of E and subtract one from

Trigonometric Math Methods

Method	Description
Math.sin()	It is used to return the trigonometric Sine value of a Given double value.
Math.cos()	It is used to return the trigonometric Cosine value of a Given double value.
Math.tan()	It is used to return the trigonometric Tangent value of a Given double value.
Math.asin()	It is used to return the trigonometric Arc Sine value of a Given double value
Math.acos()	It is used to return the trigonometric Arc Cosine value of a Given double value.
Math.atan()	It is used to return the trigonometric Arc Tangent value of a Given double value.

Hyperbolic Math Methods

Method	Description
Math.sinh()	It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.
Math.cosh()	It is used to return the trigonometric Hyperbolic Sine value of a Given double value.
Math.tanh()	It is used to return the trigonometric Hyperbolic Tangent value of a Given double value.

Angular Math Methods

Method	Description
Math.toDegrees	It is used to convert the specified Radians angle to equivalent angle measured in Degrees.
Math.toRadians	It is used to convert the specified Degrees angle to equivalent angle measured in Radians.