<http://www.tutorialspoint.com/java/java_basic_syntax.htm>

Java - Basic Syntax

**Class Names -**For all class names the first letter should be in Upper Case.   
  
If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

**Method Names -**All method names should start with a Lower Case letter.   
  
If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**Program File Name -**Name of the program file should exactly match the class name.

**public static void main(String args[]) -** Java program processing starts from the main() method which is a mandatory part of every Java program.

## Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows:

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).

# Java - Object & Classes

## Constructors:

Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

## Source file declaration rules:

* There can be only one public class per source file.
* A source file can have multiple non public classes.
* The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.
* If the class is defined inside a package, then the package statement should be the first statement in the source file.
* If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
* Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

## Java Package:

In simple, it is a way of categorizing the classes and interfaces.

## Import statements:

In Java if a fully qualified name, which includes the package and the class name, is given then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask compiler to load all the classes available in directory java\_installation/java/io :

import java.io.\*;

# Java - Basic Datatypes

## Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword.

## byte:

* Byte data type is an 8-bit signed two's complement integer.

## short:

* Short data type is a 16-bit signed two's complement integer.
* Minimum value is -32,768 (-2^15)

## int:

* Int data type is a 32-bit signed two's complement integer.

## long:

* Long data type is a 64-bit signed two's complement integer.

## float:

* Float data type is a single-precision 32-bit IEEE 754 floating point.

## double:

* double data type is a double-precision 64-bit IEEE 754 floating point.

## boolean

## char:

* char data type is a single 16-bit Unicode character.
* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).

## Reference Data Types:

* Class objects, and various type of array variables come under reference data type.
* Default value of any reference variable is null.

# Java - Variable Types

## Local variables:

There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

## Instance variables:

Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.

## Class/static variables:

Class variables also known as static variables are declared with the*static* keyword in a class, but outside a method, constructor or a block.

Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Static variables are created when the program starts and destroyed when the program stops.

Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.

Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks

When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

# Java - Modifier Types

## Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

* Visible to the package, the default. No modifiers are needed.
* Visible to the class only (private).
* Visible to the world (public).
* Visible to the package and all subclasses (protected).

## Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

# Java Access Modifiers

## Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

## Private Access Modifier - private:

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

## Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However if the public class we are trying to access is in a different package, then the public class still need to be imported.

## Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

## Access Control and Inheritance:

The following rules for inherited methods are enforced:

* Methods declared public in a superclass also must be public in all subclasses.
* Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
* Methods declared private are not inherited at all, so there is no rule for them.

# Java Non Access Modifiers

## The static Modifier:

## Static Variables:

Static variables are also known as class variables. Local variables cannot be declared static.

## final Classes:

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

## The abstract Modifier:

## abstract Class:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (since a final class cannot be extended).

An abstract class may contain both abstract methods as well normal methods.

## abstract Methods:

An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.

## The synchronized Modifier:

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

## The volatile Modifier:

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

## Example:

public class MyRunnable implements Runnable{

private volatile boolean active;

public void run(){

active = true;

while (active){ // line 1

// some code here

}

}

public void stop(){

active = false; // line 2

}

}

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

## instance of Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as:

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

public class Test {

public static void main(String args[]){

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This would produce the following result:

true

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]){

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This would produce the following result:

true

# Java - Strings Class

The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use [String Buffer & String Builder](http://www.tutorialspoint.com/java/java_string_buffer.htm)Classes.

The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe(not Synchronised).

It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.

# Java - Arrays

## Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way.

## Creating Arrays:

You can create an array by using the new operator with the following syntax:

arrayRefVar = new dataType[arraySize];

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows:

dataType[] arrayRefVar = {value0, value1, ..., valuek};

## The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable

double[] myList = {1.9, 2.9, 3.4, 3.5};

// Print all the array elements

for (double element: myList) {

System.out.println(element);

}

# Java - Methods

In general the keyword *this* is used to :

* Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

class Student{

int age;

Student(int age){

this.age=age;

}

}

* Call one type of constructor( parametrized constructor or default ) from other in a class. It is known as explicit constructor invocation .

class Student{

int age

Student(){

this(20);

}

Student(int age){

this.age=age;

}

}

## Variable Arguments(var-args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

## Example:

public class VarargsDemo {

public static void main(String args[]) {

// Call method with variable args

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {

if (numbers.length == 0) {

System.out.println("No argument passed");

return;

}

double result = numbers[0];

for (int i = 1; i < numbers.length; i++)

if (numbers[i] > result)

result = numbers[i];

System.out.println("The max value is " + result);

}

}

# Java - Exceptions

## The finalize( ) Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

## The try-with-resources

Generally when we use any resources like streams, connections etc.. we have to close them explicitly using finally block. In the program given below we are reading data from a file using**FileReader** and we are closing it using finally block.

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

public class ReadData\_Demo {

public static void main(String args[]){

FileReader fr=null;

try{

File file=new File("file.txt");

fr = new FileReader(file); char [] a = new char[50];

fr.read(a); // reads the content to the array

for(char c : a)

System.out.print(c); //prints the characters one by one

}catch(IOException e){

e.printStackTrace();

}

finally{

try{

fr.close();

}catch(IOException ex){

ex.printStackTrace();

}

}

}

}

**try-with-resources**, also referred as **automatic resource management**. is a new exception handling mechanism that was introduced in Java7, which automatically closes the resources used within the try catch block.

To use this statement you simply need to declare the required resources within the parenthesis, the created resource will be closed automatically at the end of the block, below given is the syntax of try-with-resources statement.

try(FileReader fr=new FileReader("file path"))

{

//use the resource

}catch(){

//body of catch

}

}

Following points are to be kept in mind while working with try-with resources statement.

* To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()**method of it gets invoked automatically at runtime.
* You can declare more than one class in try-with-resources statement.
* while you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
* Except the deceleration of resources within the parenthesis every thing is same as normal try/catch block of a try block.
* The resource declared in try gets instantiated just before the start of the try-block.
* The resource declared at the try block is implicitly declared as final.

# Java - Inner classes

### Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say,**getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, you can instantiate the inner class as shown below.

Outer\_Demo outer=new Outer\_Demo();

Outer\_Demo.Inner\_Demo inner=outer.new Inner\_Demo();

## Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

## Anonymous Inner Class

An inner class declared without a class name is known as an**anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

AnonymousInner an\_inner = new AnonymousInner(){

public void my\_method(){

........

........

}

};

The following program shows how to override the method of a class using anonymous inner class.

abstract class AnonymousInner{

public abstract void mymethod();

}

public class Outer\_class {

public static void main(String args[]){

AnonymousInner inner = new AnonymousInner(){

public void mymethod(){

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

### Anonymous Inner Class as Argument

Generally if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument:

obj.my\_Method(new My\_Class(){

public void Do(){

.....

.....

}

});

The following program shows how to pass an anonymous inner class as a method argument.

//interface

interface Message{

String greet();

}

public class My\_class {

//method which accepts the object of interface Message

public void displayMessage(Message m){

System.out.println(m.greet() +", This is an example of anonymous inner calss as an argument");

}

public static void main(String args[]){

//Instantiating the class

My\_class obj = new My\_class();

//Passing an anonymous inner class as an argument

obj.displayMessage(new Message(){

public String greet(){

return "Hello";

}

});

}

}

# Java - Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:

* Enumeration
* BitSet
* Vector
* Stack
* Dictionary
* Hashtable
* Properties

All these classes are no