Data Types Default Values Java

Java does not support garbage values. An unassigned local variable gives compilation error and an instance (global) variable takes a default value. Following program illustrates the default values given by JVM for unassigned instance variables.

String is not a data type. If not assigned, String takes **null**as the default.

public class DataTypeDefaults

{

byte byteValue;

short shortValue;

int intValue;

long longValue;

float floatValue;

double doubleValue;

char charValue;

boolean booleanValue;

String stringValue;

public static void main(String args[])

{

DataTypeDefaults dtd = new DataTypeDefaults ();

System.out.println("byte default value: " + dtd.byteValue);

System.out.println("short default value: " + dtd.shortValue);

System.out.println("int default value: " + dtd.intValue);

System.out.println("long default value: " + dtd.longValue);

System.out.println("float default value: " + dtd.floatValue);

System.out.println("double default value: " + dtd.doubleValue);

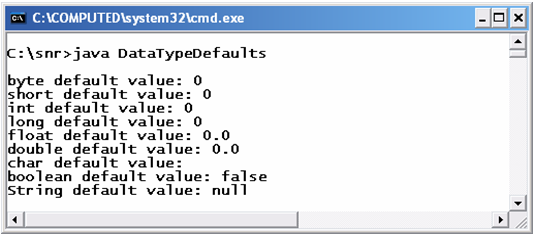
System.out.println("char default value: " + dtd.charValue);

System.out.println("boolean default value: " + dtd.booleanValue);

System.out.println("String default value: " + dtd.stringValue);

}

}



Observe the screenshot on Data Types Default Values Java. The default values for whole numbers is **0**, for floating-point numbers is **0.0** and for [primitive data type](http://way2java.com/java-introduction/primitive-data-types/) boolean it is **false**. String takes null and char does not print any value. Actually, it prints a value equivalent to Unicode value of **/u0000** which prints nothing.

## Unassigned Local and Instance Variables

One of the features of Java is "**Java does not support garbage values**". If a variable is used without a value assigned, it is either compilation error or takes a **default value**. If a local variable is not given a value and still used in the program it is a compilation error. But in case of an instance variable, it takes a default value. For example, an integer variable takes 0 by default. A novice should remember the nature of this Java unassigned variables.

public class Values

{

int price;

double rate;

boolean raining;

public static void main(String args[])

{

int marks;

// System.out.println(marks); // raises compilation error

Values v1 = new Values();

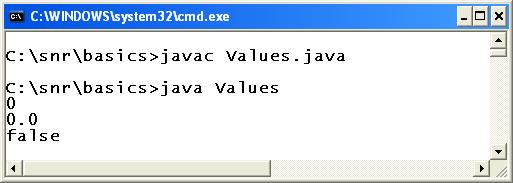
System.out.println(v1.price); // prints 0

System.out.println(v1.rate); // prints 0.0

System.out.println(v1.raining); // prints false

}

}



In the above code, **price**, **rate** and raining are unassigned instance variables and printed default values. But unassigned local variable **marks** is placed in comments. If comments are removed and compiled the program, it raises compilation error.

###### The following table gives the default values for Java unassigned variables (instance variables)

.

|  |  |
| --- | --- |
| DATE TYPE | DEFAULT VALUE |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0 |
| float | 0.0 |
| double | 0.0 |
| char | \u0000 (does not print any value) |
| boolean | false |

On Java unassigned variables, more discussion is available at [Data Types Default Values – No Garbage](http://way2java.com/java-introduction/data-types-default-values-no-garbage/).

## Interface inside Interface

**Note:**It is advised to read the [basics and types of inner classes](http://way2java.com/java-lang/inner-classes/) before proceeding further.

A nested interface is just an interface but declared within another interface or even a class.

A top-level interface is that one which is not nested.

**Nested interfaces are useful to group all the interfaces with some common functionality.** **It is to group the interfaces having some common functionality (purpose).** Following two codes illustrates.

interface Structure

{

interface constructPillars

{

// some abstract methods used for constructing pillars

}

interface constructCeiling

{

// some abstract methods used for constructing ceiling

}

interface constructWalls

{

// some abstract methods used for constructing walls

}

}

To construct a structure, three interfaces are giving the way of constructing pillars, ceiling and walls. Interfaces **Pillars**, **Ceiling** and **Walls** are having a common functionality.

One more practical approach can be seen as in the following code.

public interface Structure

{

public interface CommercialBuilding

{

int numberOfPillars = 12;

int pillarSpan = 20;

}

}

The usage here, you can observe, is no other application other than **Structure** can modify the number and span of pillars to construct the **CommercialBuilding**, even accidentally.

interface Outer1

{

interface Inner1

{

public abstract void display();

}

}

public class Test implements Outer1.Inner1

{

public void display()

{

System.out.println("Hello 1");

}

public static void main(String args[])

{

Test t1 = new Test();

t1.display();

}

}

Let us go little bit deeper.

Following code also compiles and executes.

interface Outer1

{

public abstract void show();

interface Inner1

{

public abstract void display();

}

}

public class Test implements Outer1.Inner1

{

public void display()

{

System.out.println("Hello 1");

}

public static void main(String args[])

{

Test t1 = new Test();

t1.display();

}

}

In the above code **show()** is not overridden as**Outer1** is not implemented. It is done in the next program.

interface Outer1

{

public abstract void show();

interface Inner1

{

public abstract void display();

}

}

public class Test implements Outer1, Outer1.Inner1

{

public void display()

{

System.out.println("Hello 1");

}

public void show()

{

System.out.println("Hello 2");

}

public static void main(String args[])

{

Test t1 = new Test();

t1.display();

t1.show();

}

}

That is, the Test class can implement **Outer1** and **Inner1**separately or both together.

That is, following statement also executes.

**public class Test implements Outer1**

**Note:** The implementing class can implement either the top-level interface alone or only nested interface or both.

## JDK 1.5 Features

Every version adds new packages and classes. JDK 1.5 version started its work under the code name **Project Tiger** and the version was released on September, 2004. JDK 1.5 version adds the following features to Java language. JDK 1.5 Features are programmatically very important, as often used.

###### Following list gives the JDK 1.5 Features

1. Autoboxing
2. Generics
3. Enhanced for loop
4. Varargs
5. Enums
6. Static imports
7. C-lang printf()
8. StringBuilder
9. Metadata

**Autoboxing – Automatic Conversion**

Upto JDK 1.4, all the data structures of Java stores only objects and when retrieved returns objects. The problem is, even simple data types are to be converted into objects (using wrapper classes) and stored. The retrieved objects are to be converted back to data types to use in arithmetic operations in coding. This is a big nuisance to the programmer. This is overcome in JDK 1.5 with the introduction of **autoboxing** concept. Autoboxing permits to store data types directly in DS and retrieve back data types. Autoboxing is discussed clearly in**data structures** topic.

**Generics – Type-Safe Addition**

A data structure in Java accepts any data type or object as input. If only whole numbers are required to be stored, they must be validated before added to the DS. This requires extra validation code of users input. This extra code is avoided in JDK 1.5 and named as **Generics**. Generics allow adding one type of data only; compiler raises error if other types are added.**Generics** is discussed clearly in DS.

**Enhanced for Loop**

Generally to print the values, we take a **for** loop. The **for** loop includes initialization, test condition and incrementing/decrementing. These are avoided in **enhanced for loop** and this loop works in arrays and DS only. **enhanced for loop** is illustrated in data structures topic.

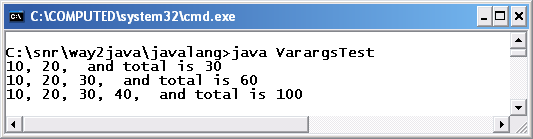
**Varargs – Variable Number of Arguments**

The **Varargs** concept permits the user to pass any number of arguments to a method depending on the requirement at runtime. The arguments are stored as an array internally. Following program illustrates.

Following program illustrates.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class VarargsTest  {    public static void add(int... marks)    {      int total = 0;      for(int x : marks)      {        total += x;        System.out.print(x + ", ");      }      System.out.print(" and total is " + total + "\n");    }    public static void main(String args[])    {      add(10, 20);      add(10, 20, 30);      add(10, 20, 30, 40);    }  } |

[](http://way2java.com/wp-content/uploads/2011/01/ss24.bmp)

*public static void add(int… marks)*

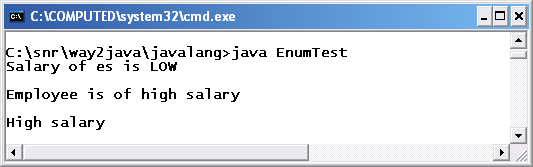
Observe the special syntax of parameter of **add()** method. It takes **three dots**. Internally, the parameters are stored in an **int** array. Enhanced **for** loop is used to print the arguments and their total.

**Enums**

**enum** is a keyword from JDK 1.5. **enum** is a different flavor of a class; enum replaces class prefix. Enums are **type-safe** as they are by default static and final integer values. Generally enum values are written in uppercase, by convention, as they are final.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | enum EmployeeSalaries  {    HIGH, MEDIUM, LOW, POOR  }  public class EnumTest  {    public static void main(String args[])    {      EmployeeSalaries es;      es = EmployeeSalaries.LOW;  // knowing enum value      System.out.println("Salary of es is " + es + "\n");         // one more value can be assigned      es = EmployeeSalaries.HIGH;           // comparing two values      if(es == es.HIGH)      {        System.out.println("Employee is of high salary\n");      }        switch(es)      {        case HIGH:          System.out.println("High salary"); break;        case MEDIUM:          System.out.println("Medium salary");  break;        case LOW:          System.out.println("Low salary");  break;        case POOR:          System.out.println("Poor salary");      }    }  } |

[](http://way2java.com/wp-content/uploads/2011/01/ss25.bmp)

*switch(es)*

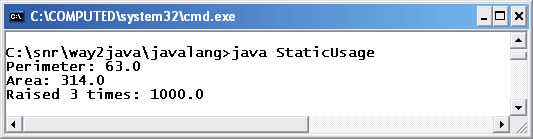
As **enum** represents an integer value always, it can be used with **switch** statement.

**Static Imports**

Many methods of classes like **Math** and **Character** are static. If the variables and methods of these classes are used very often, all are must be prefixed with Math or Character which is tedious. To overcome this, the JDK 1.5 comes with **static imports**. With **static imports**, the**static** keyword need not be used in coding as in the following program.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import static java.lang.Math.\*;  public class StaticUsage  {    public static void main(String args[])    {      int radius = 10;      System.out.println("Perimeter: " + ceil(2 \* PI \* radius));      System.out.println("Area: " + floor(PI \* pow(radius, 2)));      System.out.println("Raised 3 times: " + pow(radius, 3));    }  } |

[](http://way2java.com/wp-content/uploads/2011/01/ss26.bmp)

*import static java.lang.Math.\*;*

Normal **import** statement imports all classes and interfaces of a package. But **static import**imports only static members of a single class. It avoids usage of the class name multiple times in coding. The above statement avoids **Math** class name to prefix every variable or method used.

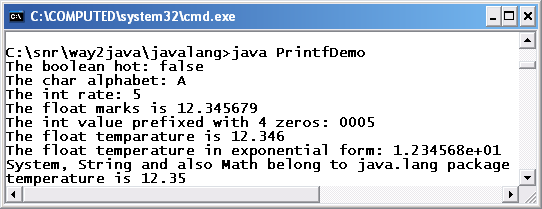
**ceil()**, **floor()**, **pow** are the static methods of Math class and PI is a static variable. All these are used without using prefix Math name.

**Supporting C-lang printf()**

**printf()** is an extra method added to **PrintStream** class from JDK 1.5 version. **printf()** is used to print at command-prompt. The printf() uses **java.util.Formatter**class internally. Following program illustrates a few ways.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class PrintfDemo  {    public static void main(String args[])    {      boolean hot = false;      char alphabet = 'A';      int rate = 5;      float temperature = 12.3456789f;        System.out.printf("The boolean hot: %b", hot);                         // place holder b for boolean      System.out.printf("\nThe char alphabet: %c", alphabet);                // c for char      System.out.printf("\nThe int rate: %d", rate);                         // d for int      System.out.printf("\nThe float marks is %f", temperature);             // f for float      System.out.printf("\nThe int value prefixed with 4 zeros: %04d", rate);// filling with zeros      System.out.printf("\nThe float temparature is %.3f", temperature);     // precision to three decimal values      System.out.printf("\nThe float temperature in exponential form: %e", temperature);        System.out.printf("\n%s, %s and also %s belong to java.lang package", "System","String","Math");      System.out.printf("\ntemperature is %4.2f", temperature); // width is 4 and precision to 2 decimal points    }  } |

[](http://way2java.com/wp-content/uploads/2011/01/ss27.bmp)  
The terminology used to print the values is quiet familiar to C/C++ programmers.

**More on Number Formatting**

Java’s capability of number formatting is far superior to C/C++. The following program illustrates.

import java.text.\*; // for DecimalFormat and NumberFormat classes

public class FormattingDemo

{

public static void main(String args[])

{

DecimalFormat df1 = new DecimalFormat("Rs 0.00");

int rate = 25;

System.out.println(df1.format(rate)); // Rs 25.00

System.out.println(df1.format(34.45682)); // Rs 34.46 (rounded)

NumberFormat nf1 = NumberFormat.getInstance();

nf1.setMinimumFractionDigits(2);

// gives minimum two decimal points; 0 gives no decimal part

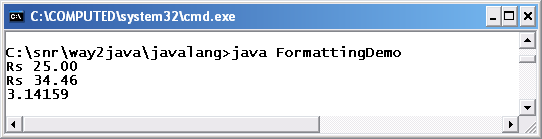
nf1.setMaximumFractionDigits(5);

// gives maximum 5 decimal points

System.out.println(nf1.format(Math.PI)); // prints 3.14159

}

}



The classes **DecimalFormat** and **NumberFormat** from **java.text** package gives more flexibility in number formatting.

*NumberFormat nf1 = NumberFormat.getInstance();*

**NumberFormat** being an abstract class cannot be instantiated directly. The static**getInstance()** method returns an object of NumberFormat.

*nf1.setMinimumFractionDigits(2);  
nf1.setMaximumFractionDigits(5);*

The first statement gives two minimum decimal points and the second statement gives five maximum decimal points. They are given if required only.

**High-performance StringBuilder**

We know String is **immutable** and StringBuffer is **mutable**. When a string is to be manipulated in the program, it is advised to choose StringBuffer for performance reasons. But StringBuffer comes with its own demerits. StringBuffer methods are **synchronized** and thus allowing**thread-safe** operations. To overcome this, designers introduced **StringBuilder** with JDK 1.5 where the methods are not synchronized. Same methods of StringBuffer work with StringBuilder also. For a program and more discussion refer String topic.

**Metadata and Annotations**

**Metadata** replaces the usage of templates. Metadata is declarative programming. **Annotations**are useful for tool developers to generate documentation of a project.

## Java Singleton

###### It may be necessary, sometimes for **performance reasons**, to create a **single object for the whole class**. **Programmer may create a number of objects in the code, but internally only one object is created**. This is transparent (not known) to the Programmer. Java Singleton is a design pattern. A design pattern is a solution given by expert people on the subject matter to a frequently occurring problem. Design patterns are available in every field like Civil, Mechanical and Accounts etc.

Singleton pattern is useful when the Developer would like to create only one object for the whole class; even if created a number of objects, internally the code creates only one. Finally to say, for Singleton class only one object is created. This is the sense and meaning of Singleton.

###### Merits or Uses Java Singleton

1. Java Singleton is useful when the code can manage all the operations with one object. Creating less objects uses less space and less time and thereby performance increases.
2. When the values are required only once in the code to populate the instance variables that can be used number of times. Through Singleton, database access is done only once, values are retrieved and fed to variables or DS elements.
3. To control the number of objects creation in the code and to create only one all through the life.
4. When only one object exists, the fields can be static also to control database access, open socket and to open file handles etc.
5. Java Singleton is useful when one object is required in multithreaded environment like ActionServlet controller in Struts 1.0.
6. In realtme applications, the JRE (Java Runtime Environment) is a Singleton object.

###### Demerits of Java Singleton

1. If not necessary actually, creation of Java Singleton keeps unnecessary restrictions on global runtime code execution environment.

###### Internal mechanism of Java Singleton Object Creation

Generally in usual coding, object is created with a constructor call. But in Java Singleton, the object is created through a method call. The method when first time called creates object and returns. When called second time, the reference of the earlier object is returned but new object is not created. To achieve this, constructor is declared private so that outside classes call the constructor and create the objects (like [public methods and private variables](http://way2java.com/oops-concepts/public-methods-and-private-variables/)concept). Some people make protected constructors so that Testers are allowed to check.

Programmer should take precautions in multithreaded environment not allow the Developers to clone the Single object something by doing synchronization.

###### Steps of creating Java Singleton Object

Steps can be narrowed to four.

1. Declare the constructor private so that it cannot be called directly from outside the code
2. Write a method creating and returning the Java Singleton object
3. Synchronize the method to make better thread-safe
4. To prevent cloning of Singleton object, override the Object class clone() method

###### Learn the technique of Java Singleton object creation in this Employee Example where only one object is created inspite of number of calls.

public class Employee

{ // private variables so that not accessed by outside code

private int salary; // assigned with default 0

private static Employee emp; // assigned with default null

private Employee() // private constructor; 1st step of the above four

{

// not required any code here

}

public static synchronized Employee getPlease() // 2nd and 3rd steps of the above four

{ // public method allowed to access from outside the code

if(emp == null) // this is important statement of all the code

{

emp = new Employee();

}

return emp;

}

public Object clone() throws CloneNotSupportedException // override the clone() method;

{ // 4th step of the above four

throw new CloneNotSupportedException("Cloning is not permitted on Singleton Please");

}

public static void main(String args[])

{

Employee emp1 = getPlease();

Employee emp2 = getPlease();

emp1.salary = 7000;

emp2.salary = 8000;

System.out.println("emp1 Salary when emp2 changes: " + emp1.salary); // 8000

emp1.salary = 9000;

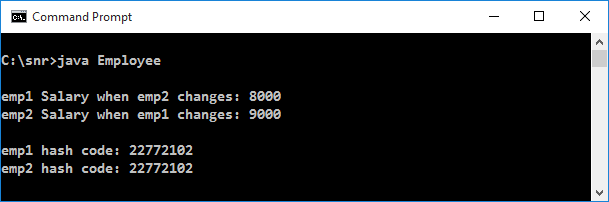
System.out.println("emp2 Salary when emp1 changes: " + emp2.salary); // 9000

System.out.println("\nemp1 hash code: " + emp1.hashCode()); // both hascodes prints the same

System.out.println("emp2 hash code: " + emp2.hashCode());

}

}



Code is almost self-explanatory with all the comments. I discuss which required.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static synchronized Employee getPlease()    {      if(emp == null)      {        emp = new Employee();      }      return emp;    } |

**getPlease()** is made pubic to that the code is permitted to be called by other classes from any package. Made static to allow to be called from main() without object need.an object. First time when called, the method creates an Employee object as object is null (as no object exists). When called second time, the object is not null (as object exists), if statement is not executed, the same emp object created earlier is returned. This is the internal mechanism a Developer should adopt.

*Employee emp1 = getPlease();  
Employee emp2 = getPlease();*

**emp1** and **emp2** objects are created by calling getPlease() method. But both refer the same**emp** object. For outsiders, they look different. But actually only one exists. This is transparent (not known) to outsiders.

The proof is when **emp2** changes **emp1** gets affected and vice versa.

Observe the Screenshot, their [hash codes](http://way2java.com/java-lang/hashcode-equals-methods/) are same.

**clone()** method is more discussed at[Cloning Duplicate Object Marker Interface](http://way2java.com/java-lang/cloning-%e2%80%93-duplicating-an-object-marker-interface/).

###### JDK 1.5 gives simple way to create Java Singleton.

1. Write[Enum object of JDK 1.5](http://way2java.com/java-versions-2/jdk-1-5-java-se-5-version/)which implicitly thread-safe (no necessary for synchronization)
2. Enum Singleton is capable to do Serialization implicitly

###### Uses of Java Singleton

1. In a class if no variables exist to maintain state between objects.
2. If all fields or variables are final. That is, to have read-only state.( all attributes are final)

# [IIB – Instance Initialization Block](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

IIB stands for **Instance Initialization Block**. As the name suggest this block is used to initialize **state of an object**. State of an object is indicated by instance variables or non-static variables. So, IIB is used to initialize instance variables or non-static variables.

This is a block with no name and enclosed within {}. The syntax for IIB is,

[?](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

|  |  |
| --- | --- |
| 1  2  3 | {        //Set Of Statements, mostly initialization statements  } |

Consider this example,

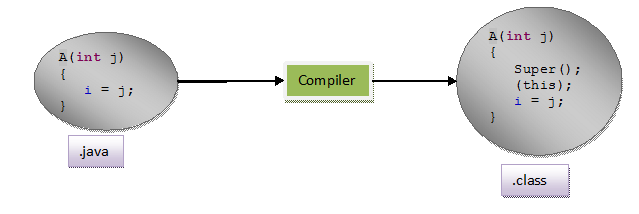
[?](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A  {       int i;         {           i = 10;       }         A(int j)       {           i = j;       }  }    class MainClass  {       public static void main(String[] args)       {            A a = new A(50);            System.out.println(a.i);       }  } |

In the above example, **Class A**has one instance variable (int i), one IIB block (From Line 5 to Line 7) and one constructor.

We all know that [first statement of constructor](http://javaconceptoftheday.com/constructors-in-java/) is super() or this(). After executing first statement, IIB blocks are called. After executing IIB blocks, remaining statements are executed.

So, when the constructor is called while creating an object (Line 19), compiler will treat constructor code like this,



where (this) is a calling statement to IIB block.

You can keep any number of IIB blocks in a class. All blocks are called after super() in the constructor in the order they appear.

**Important Note :** IIB blocks will not be called from the constructor in which **this()** statement is written as a first statement. For example,

[?](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class A  {       int i;         {            System.out.println("First IIB Block");       }         {            System.out.println("Second IIB Block");       }         A(int j)       {            this();            System.out.println("First Constructor");       }         A()       {            System.out.println("Second Constructor");       }  }    class MainClass  {       public static void main(String[] args)       {            A a = new A(50);       }  } |

For the above code, both the IIBs are executed only once. You are creating an object through First constructor. It has this() statement as first statement. It is nothing but the calling statement to second constructor. IIBs will not be executed in first constructor. They will be executed only in second constructor. If you execute above program, output will be,

First IIB Block  
Second IIB Block  
Second Constructor  
First Constructor,

IIBs can also be written as,

[?](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4 | class A  {       int i = 10;  } |

This is same as,

[?](http://javaconceptoftheday.com/instance-initialization-block-in-java/)

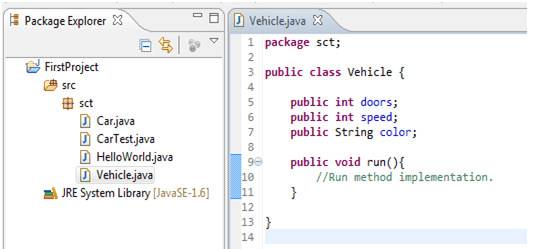
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class A  {       int i;         {            i = 10;       }  } |

**Access modifiers:**

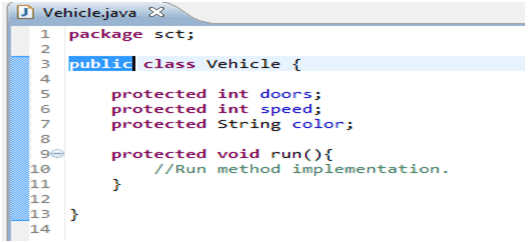
Each object has members (members can be variable and methods) which can be declared to have specific access. Java has 4 access level and 3 access modifiers.

**Access levels are listed below in least to most restrictive order.**

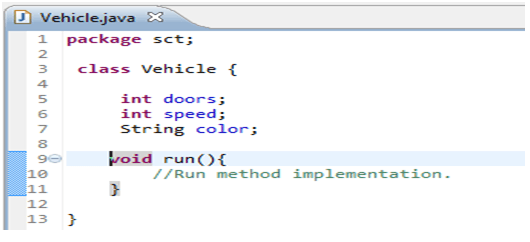
**Public:**Members (variables, methods and constructors) declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package. Below screen shot shows eclipse view of public class with public members.



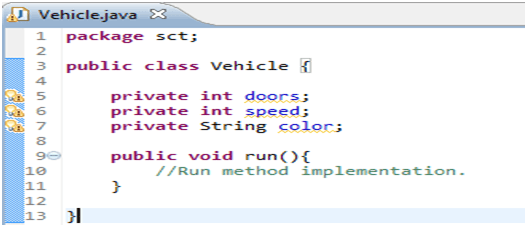
**Protected:** The protected fields or methods cannot be used for classes and Interfaces. Fields, methods and constructors declared protected in a super class can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member’s class.



**Default (no value):** The default access level is declared by not writing any access modifier at all. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package.

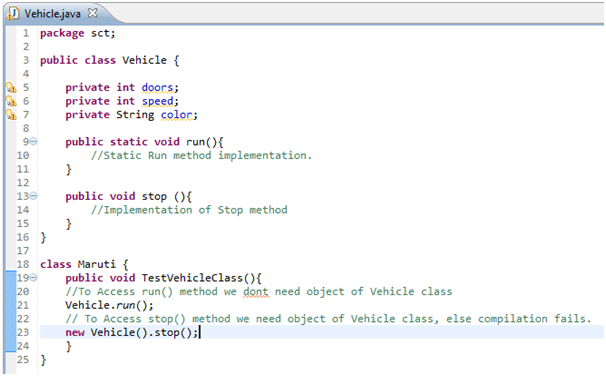


**Private:** The private (most restrictive) modifiers can be used for members but cannot be used for classes and Interfaces. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class.

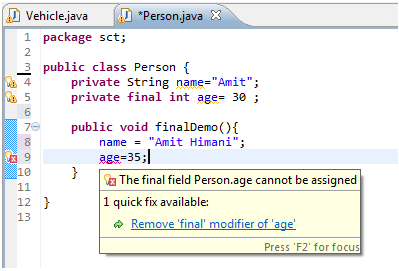


**Java has modifiers other than access modifiers listed below:**

**static:** static can be used for members of class. The static members of class can be access without creating object of class. Lets take an example of Vehicle class which has run () as static method and stop () as non-static method. In Maruti class we can see how to access static method run () and non-static method stop ().



**final:** This modifier applicable to class, method and variables. This modifier tells compiler not to change value of variable once assigned. If applied to class, it cannot be sub-classed. If applied to method, the method cannot be over-ridden in sub-class. In below sample we can see compiler errors while trying to change value of filed age because it is defined as final while we can change value of name field.



**abstract:** There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. This modifier is applicable to class and methods only.

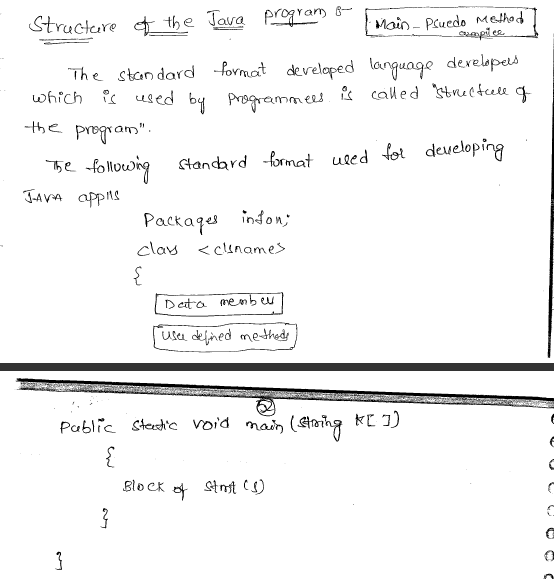
**Below Table summarizes the access modifiers**

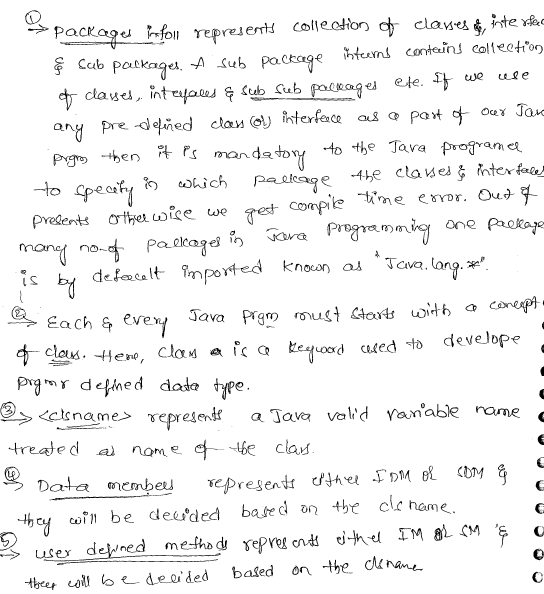
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | **class** | **constructor** | **method** | **Data/variables** |
| **public** | Yes | Yes | Yes | Yes |
| **protected** |  | Yes | Yes | Yes |
| **default** | Yes | Yes | Yes | Yes |
| **private** |  | Yes | Yes | Yes |
| **Static** |  |  | Yes |  |
| **Final** | Yes |  | Yes |  |

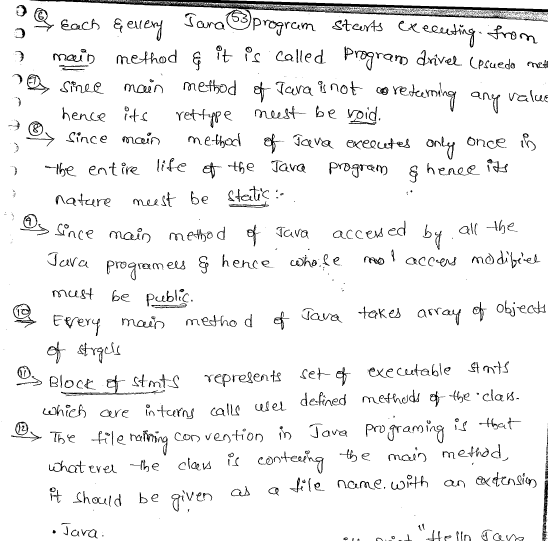
Let’s take first column example to interpret. A “class” can have public, default, final and abstract access modifiers.

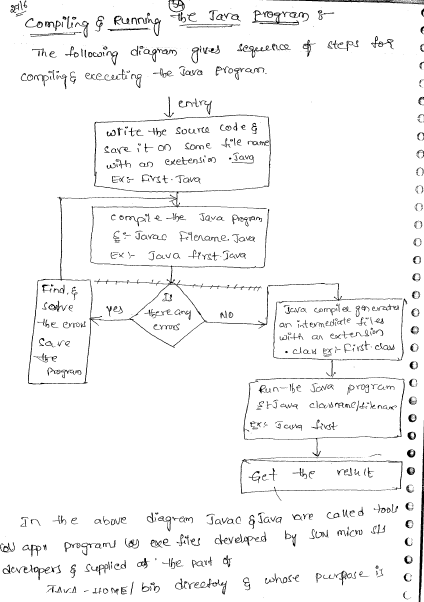
**Summary**

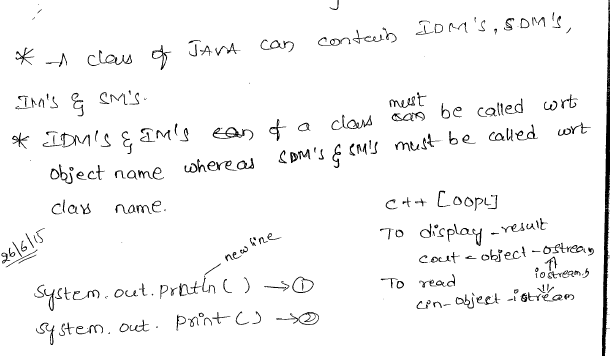
* Access modifiers helps to implement encapsulation principle of object orientation programming.
* Java has 4 access modifiers public, protected, default, private.
* Java has other modifiers like static, final and abstract.

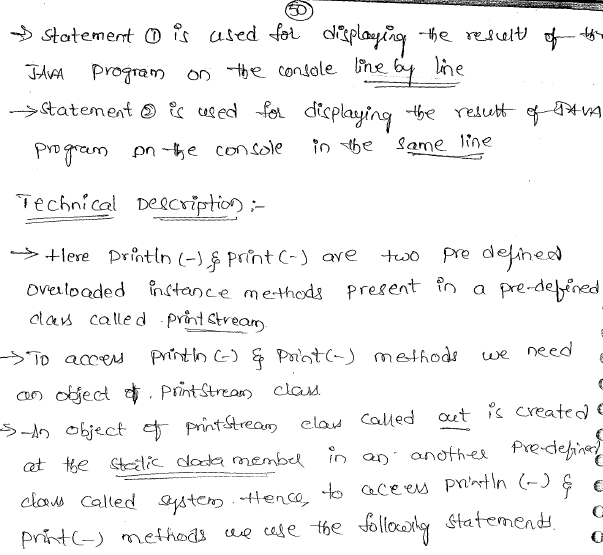
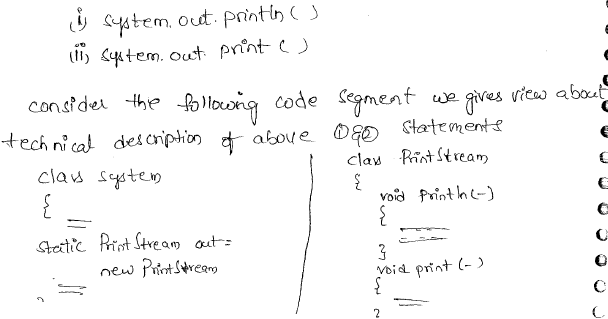
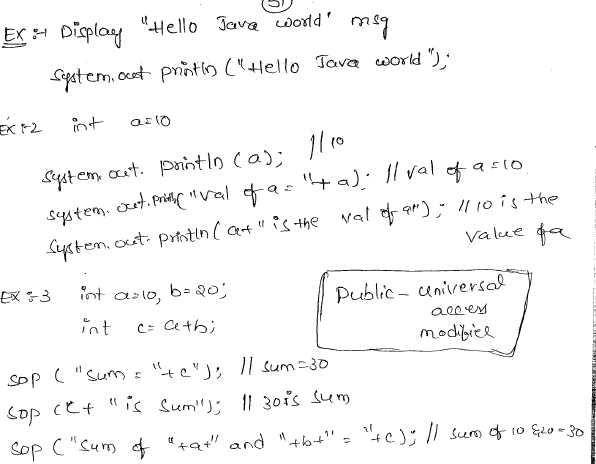








****

**  **

Call Constructor from Constructor Java

###### Calling Constructor from Constructor requires precautions on usage. Read and understand carefully.

**What is a constructor and what is its role in programming?**

A constructor is that one which is called implicitly when an object is created in Java. **A constructor gives properties to an object while it is being created itself**. Else, separate methods are required to give properties to the object after it is created. Both styles are given hereunder.

Observe the code where properties for a Student object is given using a constructor at the time of object creation.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Student  {    int marks;    String name;    public Student(int marks, String name)    {      this.marks = marks;      this.name = name;    }    public static void main(String args[])    {      Student std1 = new Student(50, "Jyostna");  // while std1 is being created, marks and name are given      System.out.println(std1.marks + " : " + std1.name);    }  } |

Through constructor, **std1** object is given properties of **marks**and**name** (in Spring, it is known as constructor injection; injecting the properties to an object through constructor).

Let us repeat the same code with methods, say **setter methods** (in Spring, it is known as setter injection; injecting the properties to an object through setter methods).



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class Student  {    int marks;    String name;    public void setMarks(int marks)    {      this.marks = marks;    }    public void setName(String name)    {      this.name = name;    }    public static void main(String args[])    {      Student std1 = new Student();   // first object is created      std1.setMarks(50);         // then properties are assgined      std1.setName("Jyostna");      System.out.println(std1.marks + " : " + std1.name);    }  } |

See how much code increases with methods. This is the importance of constructor in Java. In Java, String constructor is overloaded many folds and is discussed in [**Java String Constructors**](http://way2java.com/string-and-stringbuffer/java-string-constructors/).

Constructors is good concept in Java and must be studied elaborately; should be known how to write a constructor, constructor overloading, calling same class constructor and super class constructor etc. and all are discussed very clearly with code examples in [**Constructors and Constructor overloading**](http://way2java.com/oops-concepts/constructors-and-constructor-overloading/).

**instance block**

if we write a group of characters in between a pair of curly brackets {} inside the class and outside the methods without any static word then it is called as instance blocks

syntax:

{

//group of statments

}

-instance blocks are mainly used to initialize the instance variables

-when instance variables are declared they automatically initialized with default values but if we want to initialize

instance variables with our own values then we can initialize in following locations.

1. at the time of declaration 2. inside the contructor 3. inside the instance blocks

- we can write any number of instance blocks and any where in the class outside the methods.

**procedure how jvm execute the instance blocks**

-when ever an object is created then all the instance blocks will be executed top to bottom and next corresponding

Constructor will be executed.

- all instance blocks are executed top to bottom for every object 1 time.

**static blocks**

if we write a group of characters in between a pair of curly brackets {} inside the class and outside the methods with static keyword then it is called as staic blocks

syntax:

static {

//group of statements

}

- static blocks are mainly used to initialize the static variables

- when static variables are declared they automatically initialized with default values but if we want to initialize instance

variables with our own values then we can initialize in following 2 locations.

1. at the time of declaration 2. inside the static blocks

- we can write n number of static blocks any where in the class outside the methods.

**procedure how jvm execute the stactic blocks**

-at the time of class loading all the static blocks will be executed top to bottom and next jvm will search for main()

method and start executing main() method.

-static blocks will be executed top to bottom only for 1 time.

//wap to demo on instance blocks and static blocks



//writing a program without main() method

class BlocksDemo1{

static{

int a,b,c;

a=10;

b=20;

c=a+b;

System.out.println("Addition="+c);

}

}