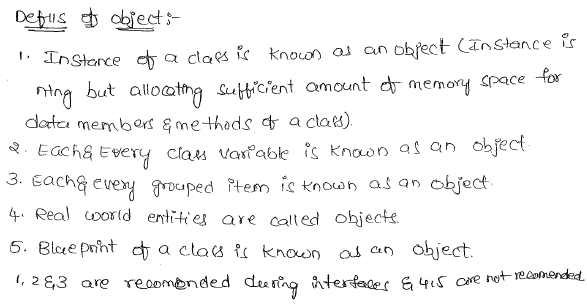
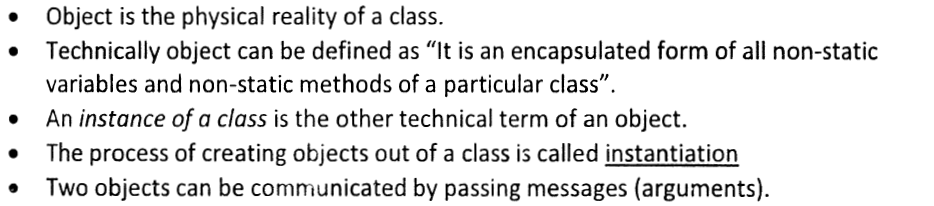
# Objects and Classes

### Object:

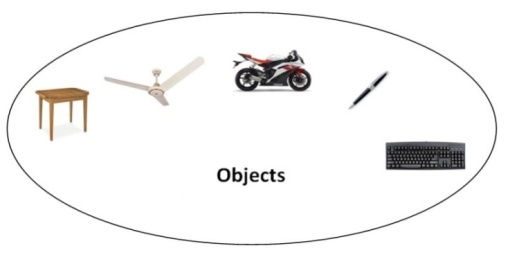
An entity that has state and behavior is known as an **object**.

****

****

**Example**:

Chair, bike, marker, pen, table, car etc.



It can be physical or logical (tangible and intangible).

**Tangible** – Capable of being touch.

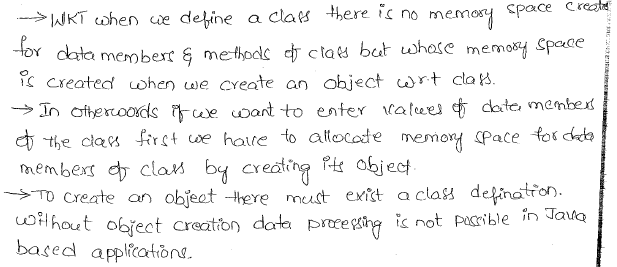
The example of intangible object is banking system.

**What is an Object ?**

Object is an instance of a class.

The memory space which could be allocated for the members of a class dynamically at runtime can be called as an Object.

Objects could be created only for the classes. Thus objects can’t be created/ can’t exist without a class.

****

**What is an Instance?**

Any dynamic allocations can be called as an **Instance** i.e. a memory space allocated for anything (need not for the members of a class) at the run time can be called as **Instance**.

But any dynamic allocation can’t be called as Object.

The memory space allocated (in the RAM) only for the members of a class are called an **Object.**

Thus instance of anything can’t be called as an Object; only the instance of a class is called an Object.

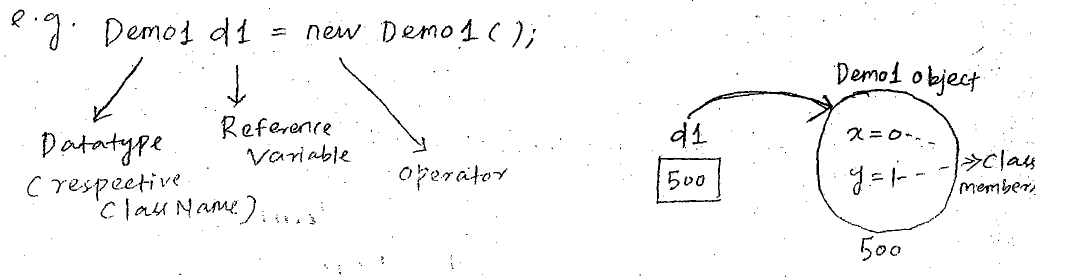
Thus every object could be an instance, but every instance can’t be an object only the instance of class could be called as an Object.

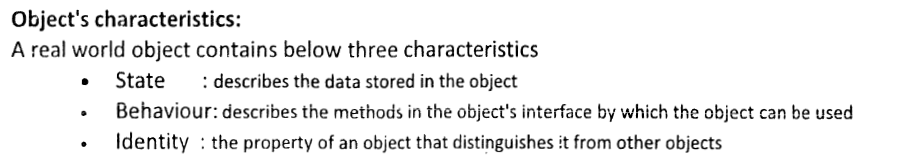
**What is Reference ?**

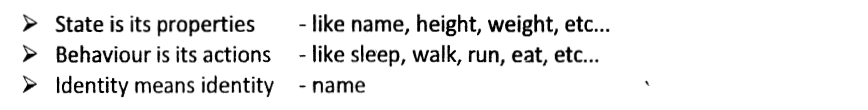
Reference is a variable which would be representing the address of the object.

Thus reference will act as a pointer to the object and also a handler (**Handler**- something using which we can access) to the object.

Since the reference is pointing to an object; in practice people call the reference as an object. But strictly speaking, reference is not an Object, it’s only a handler to the object i.e. using the reference we can access the object.

****

**

**

**What object contains ?**

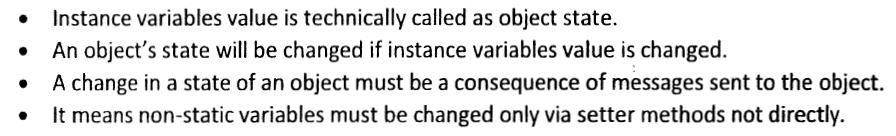
**Object of any class contains only data. So, any object would be representing only data.**

**Under any case objects cannot contains logics and it only represents data.**

**What is the state of the Object ?**

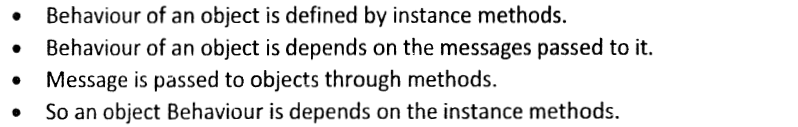
**The data present inside an object of a class at that point of time or at that instance of time is known as state of the object.**

**State of the object changes from time to time depending upon the functionalities, executed on that object.**

****

**Behavior of the object:**

**The functions associated with the objects constitute the behavior of the object.**

****

**Till what period of time any object would be present / persistent (in RAM). When any objected would be deleted from the RAM:**

**Any object would be persistent and any object would be maintained by the JVM as long as the address of that object is present.**

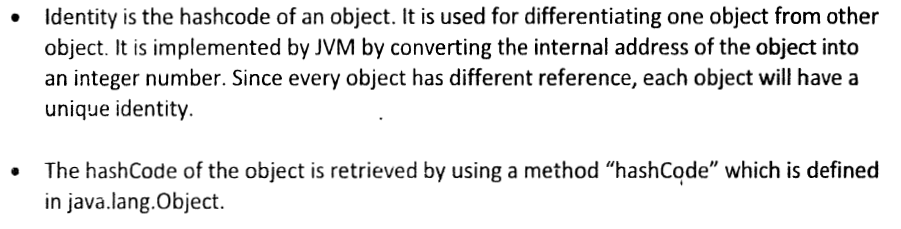
**Once the address of the object is lost then automatically that object would be destroyed / deleted.**

**An object has three characteristics:**

**State**: Represents data (value) of an object.

**Behavior**: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

**Identity**: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.



|  |  |
| --- | --- |
| **Example:**  **Pen is an object.**   * Its name is Reynolds, * **Color** is white etc. known as its **state**. * It is used to **write**, so writing is its **behavior**. | |
|  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Object is an instance of a class.  **Why do we need Object?**  In order to load the members of the class from the Byte Code to the RAM in the form of executable format and make them available to the program which is under execution (or to the processor), we do need concept of object.  Without using the concept of object we can’t load the members of a class from the Byte code to the RAM at the run time.  The concept of object **would be required** in those technologies which are working based on the concept of **Dynamic loading.**  The concept of object **wouldn’t be required** in those technologies which are working based on the concept of **Static loading**.  Thus the concept of object wouldn’t be required in structured programming language like **C-program** (Because every structured programming language program would be working based on the concept of static loading).  **When an Object would be available to a function and when it wouldn’t be available ?**      **Class:**  Class is a template or blueprint from which objects are created. So, object is the instance (result) of a class.  The process of binding data members and associated methods in a single unit is known as **Class**.  A class is a **collection of data members and methods**.  **A class is a group of objects that has common properties.**  It is a template or blueprint from which objects are created.              **A class in java can contain:**   * **data member** * **method** * **constructor** * **block** * **class and interface**  Simple Example of Object and Class : *class Student1{*  *int id;****//data member (also instance variable)***  *String name;****//data member(also instance variable)***  *public static void main(String args[]){*  *Student1 s1=new Student1();*  ***//creating an object of Student***  *System.out.println(s1.id);*  *System.out.println(s1.name);*  *}*  *}*  **Output**:  *0*  *null*  **Real life example of object and class**  In real world many examples of object and class like dog, cat, and cow are belong to animal's class.  Each object has state and behaviors.  **Example**:  A **dog** has  **State**: color, name, height, age as well as  B**ehaviors**: barking, eating, and sleeping.  Animal class  **Vehicle class**  Car, bike, truck these all are belongs to vehicle class. These Objects have also different states and behaviors.  **Example**  A **Car** has  **State** - color, name, model, speed, Mileage.  **Behaviors** - distance travel  Vehicle class  **Java class concept** Class : Explained with Real Life Example : Consider **Person** as a class.  Now we can have some properties associated with this class **Person** such as: Attributes of Person:  1. *Name of Person* 2. *Gender* 3. *Skin Color* 4. *Hair Color etc.*   Now these are the general properties which forms template of the class Person, and above properties are called as **Attributes of the Class.** Now Person class may have some core functionality such as –  1. *Talking* 2. *Walking* 3. *Eating*  Thus in short Class have –  1. *Class name* 2. *Properties or Attributes* 3. *Common Functions*   Now, we are going to create objects of the class, i.e. actual instance of the class.  Let us say that we have created objects such as **“Ram”,”Sam”**.  Both will have same attributes and functionality, but have different attribute value.  http://img.c4learn.com/2012/03/Clas-Concept-in-Java-Programming-Language.png  Now this is just a template, called as “**Class”**, and **Object** is instance of the class.  Objects in Java Programming  **Syntax of Class:**   * A Class is a blueprint or a template to create objects of identical type. * A Class is core concept of Object [Oriented Programming Language](http://www.c4learn.com/java/java-characteristics-features/).   *class classname {*  *type instance-variable1;*  *type instance-variable2;*  *// ...*  *type instance-variableN;*  *type methodname1(parameter-list) {*  *// body of method*  *}*  *type methodname2(parameter-list) {*  *// body of method*  *}*  *// ...*  *type methodnameN(parameter-list) {*  *// body of method*  *}* ***}***  **Explanation Of Syntax :**  **Class name**  *class classname {*   1. class is Keyword in Java used to create class in java. 2. classname is Name of the User defined Class.       **Class Instance Variable**  *type instance-variable1;*  *type instance-variable2;*  *// ...*  *type instance-variableN;*   1. [Instance Variables](http://www.c4learn.com/java/java-variable-types/) are Class Variables of the class. 2. When a number of objects are created for the same class, the same copy of instance variable is provided to all. 3. **Instance variables have different value for different objects.** 4. **Access Specifiers can be applied to instance variable i.e. public, private.** 5. Instance Variable are also called as “**Fields**“.   **Class Methods**  *type methodname1(parameter-list) {*  *// body of method*  *}*   1. These methods are equivalent to function in [C Programming Language](http://www.c4learn.com/). 2. **Class methods can be declared public or private.** 3. These methods are meant for operating on class data i.e. Class Instance Variables. 4. Methods have return type as well as parameter list.  Rules for Java Class:  1. **A class can have only public or default (no modifier) access specifiers.** 2. It can be either abstract, final or concrete (normal class). 3. It must have the class keyword, and class must be followed by a legal identifier. 4. It may optionally extend one parent class. **By default, it will extend java.lang.Object.** 5. It may optionally implement any number of comma-separated interfaces. 6. The class's variables and methods are declared within a set of curly braces {}. 7. Each .java source file may contain only one public class. A source file may contain any number of default visible classes. 8. Finally, the source file name must match the public class name and it must have a .java suffix.   **Live Example: Class concept**  *public class Rectangle {*  ***// two fields***  *public int breadth;*  *public int length;*  ***// two methods***  *public void setLength(int newValue) {*  *length = newValue;*  *}*  *public void setBreadth(int newValue) {*  *breadth = newValue;*  *}*  *}*  **Explanation** :   | **Rectangle** | Class name | | --- | --- | | **Length** | Instance Variable | | **Breadth** | Instance Variable | | **setLength()** | Method | | **setBredth()** | Method |   As soon as we create this class , Inside memory it would looks like – [Class Structure in Java Programming](http://img.c4learn.com/2012/03/Class-Structure-in-Java-Programming.png) **Java classes basics** Live Example : Simple Class Program [Area of Rectangle] *class Rectangle {*  *double length;*  *double breadth;*  *}*  ***//This class declares an object of type Rectangle.***  *class RectangleDemo {*  *public static void main(String args[]) {*  *Rectangle myrect = new Rectangle();*  *double area;*  ***//assign values to myrect's instance variables***  *myrect.length = 10;*  *myrect.breadth = 20;*  ***// Compute Area of Rectangle***  *area = myrect.length \* myrect.breadth ;*  *System.out.println("Area is " + area);*  *}*  *}* Output : java RectangleDemo  Area is 200.0 Explanation : Class Concept1. Class Creation / Declaration : Consider following class –  *class Rectangle {*  *double length;*  *double breadth;*  *}*   * **Class** defines a **new type of data**. * **Rectangle** is our new data type. * **“Rectangle”** will be used to **declare objects of type Rectangle**. * **Class declaration only creates a template. It does not create an actual object.**   java class image 2. Creation of Object *Rectangle myrect = new Rectangle();*   * Actual Creation of Object. * **Memory is allocated for an object** after executing this statement. * This Statement will create instance of the class “**Rectangle**” and name of instance is nothing but actual object “**myrect**“. * **Fresh copy of Instance variables** gets created for Fresh Instance.   [myrect now have its own instance variables -> length, breadth ] Creating Object of Class in Java Programmming Language3. Accessing Instance Variables Of Object/Instance using DOT Operator *myrect.length = 10;*  *myrect.breadth = 20;*   * Each Instance/Object gets their **own copy of instance variables i.e length and breadth**. * We can access “**myrect’s**” copy of instance variable using “**DOT**” operator.   **Variables – Properties**  A program stores values in variables. As the name indicates, a variable value can be varied by the programmer at any time. An object uses variables to set its properties.  **For example**, a motor car uses the properties like speed and petrol etc. by storing them in variables. The declaration of a variable requires a data type. A data type says what type of value a programmer can store in the variable.  **Java instance variables**  The variables declared within a method body are known as "**local variables**".  The method parameters are also treated as local variables only. As their accessibility (or scope) is limited to the method only, the local variables can be used within the method body only. That is, other methods cannot use them.   * Variables defined within a class are called **instance variables** because each instance of the class contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. * **Instance variable can be declared public or private or default (no modifier).** * When we do not want our variable’s value to be changed outside our class we should declare them private. * public variables can be access and changed from outside of the class.   **Syntax is shown below.**  http://www.w3resource.com/java-tutorial/images/declaration-of-instance-variables.png  **Live Example :**  **Class With Two Objects (Each have its own copy of Instance Variable**  *class Rectangle {*  *double length;*  *double breadth;*  *}*  ***// This class declares an object of type Rectangle.***  *class RectangleDemo {*  *public static void main(String args[]) {*  *Rectangle myrect1 = new Rectangle();*  *Rectangle myrect2 = new Rectangle();*  *double area1,area2;*  ***// assign values to myrect1's instance variables***  *myrect1.length = 10;*  *myrect1.breadth = 20;*  ***// Compute Area of Rectangle***  *area1 = myrect1.length \* myrect1.breadth ;*  *System.out.println("Area of Rectange 1 : " + area1);*  ***// assign values to myrect2's instance variables***  *myrect2.length = 10;*  *myrect2.breadth = 20;*  ***// Compute Area of Rectangle***  *area1 = myrect2.length \* myrect2.breadth ;*  *System.out.println("Area of Rectange 2 : " + area2);*  *}*  *}*  **Explanation** : Class - Object and instance Variables in Java Programming LanguageEach Object has its own set of Instance Variables :  1. In the above program, we have created two objects of the class “**Rectangle**”, i.e myrect1, myrect2.   *Rectangle myrect1 = new Rectangle();*  *Rectangle myrect2 = new Rectangle();*   1. As soon as above two statements gets executed, two objects are created with specimen copy of their instance variables. 2. In short myrect1’s version of length and breadth gets created. Similarly myrect2’s version of length and breadth gets created. 3. Using dot Operator, we can access instance variable of respective object.   If you want to access instance variable of **myrect1** Object –  ***myrect1.length = 10;***  ***myrect1.breadth = 20;***  If you want to access instance variable of **myrect2** Object –  ***myrect2.length = 5;***  ***myrect2.breadth = 10;***   1. **We can assign different values to instance variables of object. Instance variable of different objects though have same name, they have different memory address , different value.** 2. It is important to understand that **changes to the instance variables of one object have no effect on the instance variables of another**.   ***Example:***  *public class MethodVariables{*  *public static void main(String args[]){*  *int mangoes = 10, bananas = 20;*  *System.out.println(mangoes);*  *System.out.println(bananas);*  *System.out.println(mangoes + bananas);*  *System.out.println("No of mangoes: " + mangoes);*  *System.out.println(mangoes + " and " + bananas + " sum is " + (mangoes+bananas));*  *}*  *}*  Java Local Instance Variables  **Note:**  **A class can have any number of objects.**  **The data present inside the objects of the same class may change from one object to another object.**  **What is the difference between pointer variable and a reference variable ?**  **Pointer is a variable which contains the actual address of the RAM.**  **But reference is a variable which contains only the index of the address of the object and a reference variable would not contain the actual address of the RAM.**  **Thus we can’t call reference variable as a pointer variable.**  **JVM maintains certain list of predefined indexes internally.**  **Whenever an object is created and the address of the object is generated, JVM attaches one of the predefined indexes to the actual address and then it assigns only the index of the address to the reference variable.**  **Whenever we use the (.) operator, the functionality of the (.) operator is to fetch the actual address using the index of the address present inside the reference variable.**  **At any given point of time, in Java we can’t see the actual address of the RAM. There is no provision to show the actual address of the RAM. There is no provision to show the actual address of the RAM in Java. Thus Java doesn’t support pointers.**    **Using Instance Variables from static main()**  Java is a [**strongly-typed language**](http://way2java.com/java-introduction/strongly-typed-language/) where variables should be declared and given values before they are used. Java coding uses both local variables and global variables. The global variables are known as "**instance variables**" in Java. Instance variables are to be called with instances (objects) from static methods like main(). It is to be noted that local variables do not require an object. The scope of instance variables is for whole class and local variables are within the method in which they are declared. In the following program local and instance variables are used.   |  |  | | --- | --- | |  | *public class Employee{*  *int salary;****// instance variables***  *String department;*  *public static void main(String args[]){*  *String company = "Lorvent Solutions";* ***// local variable***  *System.out.println(company);*  ***// calling local variable without object***  *Employee emp1 = new Employee();*  *emp1.salary = 8000;*  *emp1.department = "Finance";*  *System.out.println(emp1.salary);*  ***// calling instance variables with object***  *System.out.println(emp1.department);*  *System.out.println(emp1.department + " section exists in " + company);*  ***// System.out.println(salary);***  ***// compilation error***  *}*  *}* | |  |  |   [Java Local Instance Variables](http://way2java.com/wp-content/uploads/2010/11/third2.jpg)  Output screen of Employee.java  **Declaration of Methods**   * A method is a program module that contains a series of statements that carries out a task. * To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method. * Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times.   **Syntax to declare method is given below:**  java methode declaration image  ***Java declaring object in class***  **Declaring Object in Class: Declaring Reference to an Object in Java Programming**   1. When we create class, then we are creating new data type. 2. Newly created data type is used to create an Object of that Class Type. 3. Creating object is two step process.   **Creation of Object involves two steps –**   1. **Declaration** 2. **Allocation and Assigning**   **Rectangle myrect1 = new Rectangle();**  This statement is used to create an object, we are going to break down this statement in two separate statements –  **Rectangle myrect1 ;**  **myrect1 = new Rectangle();**  **Step 1 :**  **Declaration of Variable of Type Class**  Declaring Class Instance in Java Programming Language   1. Above Declaration will just declare a variable of class type. 2. Declared Variable is able to store the reference to an object of Rectangle Type. 3. As we have not created any object of class Rectangle and we haven’t assigned any reference to myrect1 , it will be initialized with null.   **Step 2 :**  **Allocation and Assigning Object to variable of class Type**  Allocation and Assigning Object to Variable   1. Above Statement will create physical copy of an object. 2. This Physical Copy is then assigned to an variable of Type Class i.e myrect1. 3. **Note**: myrect1 holds an instance of an object not actual object.     **Actual Object is created elsewhere and instance is assigned to myrect1.** In short –  1. First statement will just create variable **myrect1,** which will store address of actual object. 2. First Statement will not allocate any physical memory for an object, thus any attempt accessing variable at this stage will **cause compile time error**. 3. Second Statement will create actual object **randomly at any memory address, where it found sufficient memory**. 4. Actual memory address of Object is stored inside **myrect1**.  Difference between object and class There are many differences between object and class.   |  |  |  | | --- | --- | --- | | **No.** | **Object** | **Class** | | 1) | Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. | | 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. | | 3) | Object is a **physical** entity. | Class is a **logical** entity. | | 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} | | 5) | Object is created **many times** as per requirement. | Class is declared **once**. | | 6) | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. | | 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. | | 8) | Whenever we execute the Java program the definition of the class will be loaded in to main memory only once with the help of class loader sub system. | After loading the class definition in the main memory whose objects will be created later. | | |
| **Internal Working:**  Compared to the creation of primitive types, creating an instance (or object) of Student class (or type) is different. **We have to use the new keyword for creating an object of non-primitive data type or user defined data type**. For e.g., in primitive types, the moment a variable is declared, the space required for that variable is also allocated.  *int a = 5;*  This declares the variable a and allocates 4 bytes for that integer. But with classes (or non-primitive data types), the allocation of the space required for that object has to be done using new keyword as shown below. If the class Student is declared like this  *class Student{     String name;     int marks;     char section; }*  We can create object for student Mahesh like this.  *Student mahesh = new Student();*  Similarly, we can also create student objects like ntr, prabhas and arjun as shown below.  *Student ntr = new Student(); Student prabhas = new Student(); Student arjun = new Student();*  Above, the right side of equals shows how to create objects and ntr, prabhas and arjun are called references.  Classes-objects  **Java Object Reference:**  Creating an instance (or object) of Student class (or type) has to be done using the new keyword. This is different when compared to the creation of primitive types.  As discussed in the last topic, we can create an object (or instance) of Student class (or type) as shown below.  *Student raja = new Student();*  Here raja is the *reference* which points to the Student object created using the new keyword.   Class, object and reference can be explained better using the analogy of the house plan, house and a paper slip containing house address.  *class House{     String ownerName;     int noOfBedrooms;     double hallWidth;     double hallLength;     int noOfToilets; }* ***// Create slip, create house and copy the address*** *House yourAddressSlip = new House();*  **We can observe the following in the above code.**   * For designing the House Plan, we need to define the class using class House { ... } * For creating/constructing the houses we use new House() * For creating the address slip, we use House yourAddressSlip   The same word House used in different locations/ways convey different meanings.   Here are some more points we need to understand about House Plan, House and Paper Slip   * The slip is the reference where as the house is the object. So if we want to use the house we need the slip with address. And slip alone is not useful until it contains the address of house.   ***// Slip With Address - Construct Raja's house and using the address slip we can go to the house*** *House rajaAddressSlip = new House();* ***// Slip Without Address - Not useful since there is no address*** *House premAddressSlip;*   * A reference when only created and not assigned to any object will be null and is same as slip which is empty. When a new house is built (or created) using new keyword and assigned to a reference, the address of the newly created house is copied into the slip. That is when the slip is useful and can be used to access a house.   ***// Now useful, since Prem's address slip now points to Raja's house*** *premAddressSlip = rajaAddressSlip;*   * Another reference can be created, which means we create another slip and also copy the address, so both the slips (or references) point to the same house. We can create as many references for a given house as shown below. Here we have created three references rajaAddressSlip, premAddressSlip, and carpenterAddressSlip, all pointing to the sameHouse. i.e. All the slips contain the same house address and hence refer to the same house.   ***// Create slip, create house and copy the address*** *House rajaAddressSlip = new House();* ***// Create the second slip. This will be null.*** *House premAddressSlip;* ***// Copy the address from the first slip to the second slip.*** *premAddressSlip = rajaAddressSlip;* ***// Create a third slip and copy the address from first slip.*** *House carpenterAddressSlip = rajaAddressSlip;*   * If we created a new house, the address on any slip can be changed to that of the new house. So a slip or reference can change address on it but can have only one address at a given time. So slip either points to the old house or the new house.   ***// Create slip, create first house and copy the address*** *House rajaAddressSlip = new House();* ***// Create the second slip and copy the address of first house*** *House premAddressSlip = rajaAddressSlip;* ***// Create second house and copy the address into the first slip*** *rajaAddressSlip = new House();*   * The address on the slip can be erased, so that it does not point to any house. This is also called as nulling the reference (or making a reference as null), so that the reference does not point to any house. Please note that null is a Java keyword.   ***// Create slip, create first house and copy the address*** *House rajaAddressSlip = new House();* ***// Create the second slip and copy the address of first house*** *House premAddressSlip = rajaAddressSlip;* ***// Erase the address on the first slip*** *rajaAddressSlip = null;*  The below table gives the summary on references and objects, in comparison with *House and paper slip* analogy.   |  |  | | --- | --- | | **House Plan, House and Paper Slip** | **Class, Object and Reference** | | Using a House Plan, we can create many houses | Using a class, we can create many objects | | For accessing any house we need an address slip. | For accessing any object we need a reference. | | Multiple address slips can point to the same house. | Multiple references can point to the same object. | | The address on the slip can be changed to point to a new house. | The reference can be changed to point to a new object. | | One slip can contain only one house address at a given time. When a new address is copied, the old address is lost (or overwritten). | A reference can point to only one object at a given time. When a new object is assigned to a reference, the old object is gone. | | The address on the slip can be erased, so that it does not point to any house. | A reference can be nulled (or assigned to null), so that it does not point to any object. |   Classes-objects-references  The variables (or attributes) inside the Student class like name, marks and section can be accessed using the reference as explained in [Member Variable In Java](http://java.meritcampus.com/core-java-topics/member-variable-in-java).  **Member Variable in Java**   |  | | --- | | A *class* comprises of various *variables* (or attributes). For e.g., the Student class is comprised of variables name, marks andsection, the Car class is comprised of variables owner, registrationNumber, engineCC etc., These class variables are also called as *instance variables* or *member variables* or *class attributes*.  *class Student{     String name;     int marks;     char section;     String address;     long mobile; }*  The variables of the class can be used to using the dot (.) operator. The variables in the student class can be accessed as shown below.  *Student mahesh = new Student(); // LINE A mahesh.name = "Mahesh Babu"; mahesh.marks = 87; mahesh.section = 'A'; System.out.println( mahesh.name + " belongs to section " + mahesh.section + " and he got " + mahesh.marks+ " marks.");*  *Here m*ahesh is the reference pointing to the Student object created in LINE AAccess-member-variables  Another Student ntr can be created and the variables in it can be accessed similarly.  *Student ntr = new Student(); // LINE B ntr.name = "N T Rama Rao"; ntr.marks = 85; ntr.section = 'A'; System.out.println( ntr.name + " belongs to section " + ntr.section + " and he got " + ntr.marks + " marks.");*  *As dis*cussed in the [Java Objects References](http://java.meritcampus.com/core-java-topics/java-objects-references), there are now two references mahesh and ntr, pointing to two objects created atLINE A and LINE B respectively.   Also note that when we try to access the variables using a reference which does not point to any object, we get aNullPointerException during run-time.  *Student prabhas = null; // LINE B prabhas.name = "Prabhas"; // Throws NullPointerException* | |

**Note:** A new keyword is used to allocate memory at runtime, new keyword is used for create an object of class,

#### A simple class example

Suppose, Student is a class and student's name, roll number, age will be its property.

Let’s see this in Java syntax

***class Student{***

***String name;***

***int rollno;***

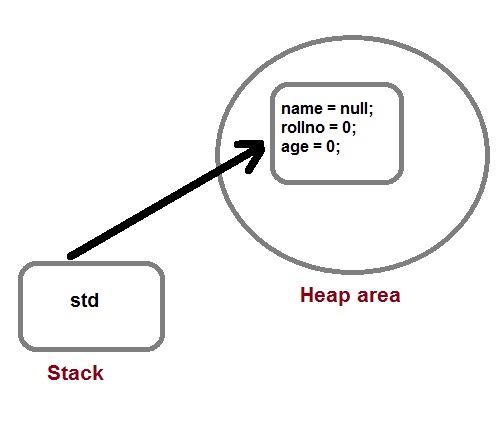
***int age;***

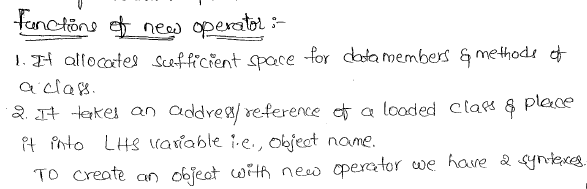
***}***

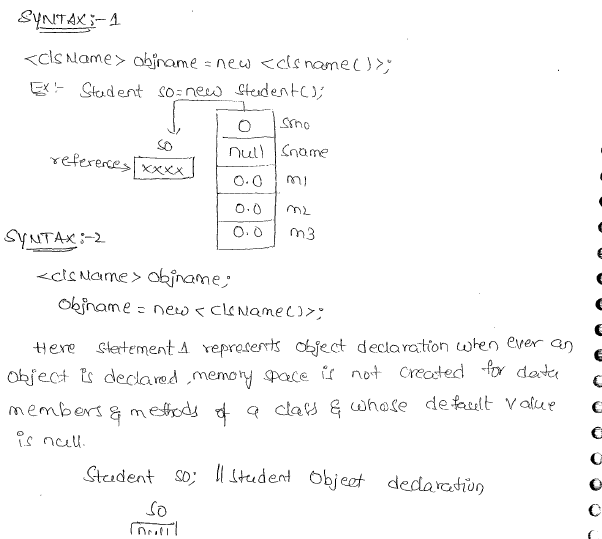
When a reference is made to a particular student with its property then it becomes an object, physical existence of Student class.

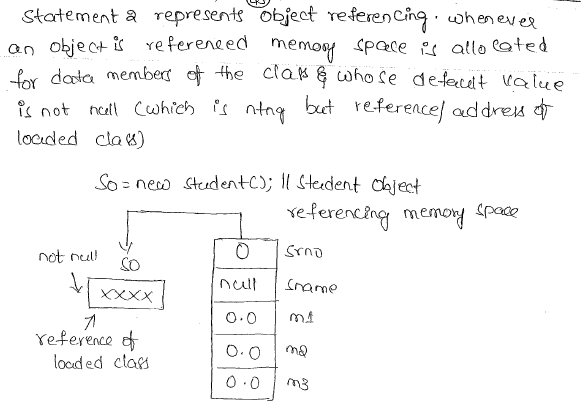
***Student std=new Student();***

After the above statement std is instance/object of Student class. Here the new keyword creates an actual physical copy of the object and assign it to the std variable. It will have physical existence and get memory in heap area. The new operator dynamically allocates memory for an object









#### Q. How a class is initialized in java?

A Class is initialized in Java when an instance of class is created using either new operator or using reflection using class.forName().

A class is also said to be initialized when a static method of Class is invoked or a static field of Class is assigned.

#### Q. How would you make a copy of an entire Java object with its state?

Make that class implement Cloneable interface and call clone() method on its object.

**clone()** method is defined in Object class which is parent of all java class by default.

**Internal structure of Class**

|  |  |
| --- | --- |
|  | *<Access\_Modifier> class <Class\_Name> extends <Super\_Class\_Name> implements <Interface\_Name>{*  *<static initilizar block>*  *<ananymous\_block>*  *<constructor\_declarations>*  *<field\_declarations (Static or Non-Static)>*  *<method\_declarations (Static or Non-Static)>*  *<Inner\_class\_declarations>*  *<nested\_interface\_declarations>*  *variables\_inside\_class(Static or Non Static)*  *}* |

***Class\_Name:****Unique name for the class in a specific package.*

***Super\_Class\_Name:****Name of the class which given class extends.( extends keyword is used for this purpose)*

***Interface\_Name:****Name of an Interface which above class implements.( implementskeyword is used for this purpose).*

**Overall:**

* Class can have only public and default access.
* Public class needs to be in same name java file.
* Single java file can contain more than one non public class but can have only one public class.
* A public class can be seen by all classes from all package
* A class with default access can be seen only by classes within the same package.
* Java file with no public class have no naming restriction.
* Class can also have final and abstract & strictfp non access modifiers.
* An abstract class cannot be instantiated.
* A final class cannot be sub-classed.
* A class cannot be both final and abstract.
* Class visibility can be seen in 3 parameter
  1. If a class can extend another class?
  2. If a class can create instance of another class?
  3. If a class can access methods and variable of another class?

**What is a class in Java- General Terminology?**

##### A class in Java is a structure which contains programming constructs. Programming constructs are small pieces of code with which a program is built; like bricks, cement and sand are constructs to construct a building. The programming constructs are mainly constructors, methods and variables. So finally, to say, a class is made up of or embeds constructors, methods and variables, all doing a predefined task.

To access these constructs, we require a **handle**which points (refer) all these. This handle is known as **object**. So, it is necessary to create an object of the class to access these constructs. A class gives a boundary for the whole code.

**Java Types of Classes**

Java supports three types of classes.

**1. Concrete classes  
2. Abstract classes  
3. Interfaces**

###### **1. Concrete Classes**

Java comes with two types of classes –

**Concrete classes and**

**Abstract classes.**

Generally a class in Java, is known infact as **concrete class**.

Other way, to say, a Java class which is not abstract, is a concrete class.

A concrete class can be **final** also.

**Example:**

*a) public class Employee  
b) public final class Employee*

###### **2. Abstract Classes**

An abstract class differs from concrete class in that a few method of abstract class misses the body part.

Any method without body is known as abstract method.

Another rule is a class contains abstract methods, the class should be declared as abstract class. That is, abstract class can contain abstract methods and also we cannot create objects with abstract class.

###### **3. Interfaces**

An interface is not completely new. It is after all a type of abstract class where all methods should be abstract (know, abstract class can contain a mixture of concrete and abstract methods). It means, interface contains only abstract methods.   
  
Basically Java does not support multiple inheritance. But supports partially through interfaces. To inherit concrete and abstract classes, use extends keyword and with interfaces use implements keyword. Like abstract classes, with interfaces also you cannot create objects and all the abstract methods should be overridden by the subclass.

**Java Create Object**

###### Now imagine there is trunk of wood. The wood as it is waste until and unless converted into articles like door, table and duster etc. No door, no table and no duster, the wood is mere waste. We say, they are instances of wood. Other way, they are objects of wood. Let us dig more through Java Create Object.

Think, in place of wood, the class **Test**. The **Test** class contains the constructs like constructor, method and variables. To make use of the constructs, we require an instance or object. No instance or object, the class is mere waste. The **t1** is object with which we can call all the constructs. For this reason, object creation is known as instantiation or we say object is instantiated.

**Example on Java Create Object:**

*public class Test{*

*int mangoes = 10;                                 //* ***it is variable***

*public Test(){                                 //****it is constructor***

*System.out.println("Constructor is called");*

*}*

*public void show(){                                //****it is method***

*System.out.println("Method is called");*

*}*

*public static void main(String args[]){*

*Test t1 = new Test();*

*//* ***implicitly, constructor is called***

*println("No. of mangoes: " + t1.mangoes);*

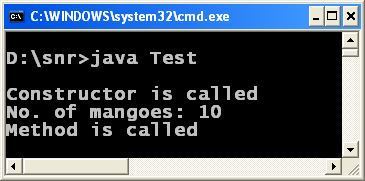
***// variable is called***

*t1.show();*

***// method is called***

*}*

*}*



**Java Object without new Keyword 5 Styles**It is only coding importance of creating Java object without using new keyword. **Five styles are given.**

**1.** [Cloning – Duplicating an Object – Marker Interface](http://way2java.com/java-lang/cloning-%e2%80%93-duplicating-an-object-marker-interface/)  
**2.** [Java forName newInstance](http://way2java.com/java-lang/java-forname-newinstance/)  
**3.** [readObject Using Serialization](http://way2java.com/java-lang/readobject-java/)  
**4.** [Class Loader Java](http://way2java.com/java-lang/class-loader-java/)  
**5.** [Creating Java Object Using Reflection](http://way2java.com/java-lang/creating-java-object-without-new-keyword-using-reflection/)

1. **Cloning Duplicate Object Marker Interface**

Shallow copying is easier and deep copying takes more time, especially, when properties (variables) are many. Here comes another way of copying, **cloning**.

In cloning, object to object is assigned, but at the same time encapsulation is maintained. Both objects occupy two different locations; that is, cloning creates a duplicate object which is no way connected with the original one.

###### **Cloning – interface Cloneable and clone() method**

In cloning, the **interface Cloneable** and method **clone()** of Object class are used. To clone an object, the class should implement Cloneable interface; else, the JVM throws a checked exception, **CloneNotSupportedException**.

**Example on Cloning Duplicate Object Marker Interface:**

*public class Worker implements Cloneable{*

*int salary;*

*public static void main(String args[]) {*

*Worker work1 = new Worker();*

*work1.salary = 3000;*

*try{*

*Worker work2 = (Worker) work1.clone();*

*syso("After cloning, work2 salary: " + work2.salary);*

***// 3000***

*work1.salary = 4000;*

***// change work1 salary***

*syso("Changing work1 salary, work2 salary: "+ work2.salary);* ***// 3000***

*work2.salary = 5000;*

***// now change work2 salary***

*syso("Changing work2 salary, work1 salary: "+ work1.salary);* ***// 4000***

***// let us see the hash codes***

*syso("Hash code of work1: " + work1.hashCode());*

*syso("Hash code of work2: " + work2.hashCode());*

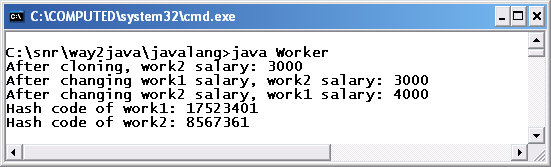
*}catch(CloneNotSupportedException e){*

*syso("Check your class implemented Cloneable interface. " + e);*

*}*

*}*

*}*



Observe, **Worker** class implements **Cloneable** interface.

***Worker work2 = (Worker) work1.clone();***

The above statement can be split into two for more clarity.

***Object obj = work1.clone();  
Worker work2 = (Worker) obj;***

The **clone()**method returns an object of Object class. This is explicitly type casted to Worker. Now the objects **work1** and **work2** occupy different locations and there by encapsulation is maintained. For this reason, when **work1** is changed **work2** salary is not getting affected and similarly with **work2** also.

Observe the screenshot and the program.

As both objects occupy different locations, their **hash codes** are also different.

**Following are the requirements for cloning.**

1. Implement Cloneable interface
2. Use clone() method of Object class
3. After cloning, explicit casting is required

**Marker Interface – Cloneable**

Java API comes with six marker interfaces. An interface that does not contain any methods and variables is known as a **marker Interface**. For more explanation on marker interfaces and their list, refer I/O streams.

**Array Cloning**

Arrays are predefined objects in the Java language itself. So, like any other object, arrays also can be cloned. Following program illustrates.

*public class Demo{*

*public static void main(String args[]){*

*int x1[] = { 10, 29, 30, 40 };*

*int x2[] = x1.clone();*

*System.out.println(x2[0]); // prints 10*

*// now changing x1 value does not affect x2*

*x1[0] = 100;*

*System.out.println(x2[0]); // prints same 10*

*}*

*}*

1. **forName() newInstance() Methods in Java**

##### Let us see another style using **forName()** and **newInstance()** methods of class java.lang.Class.

*public class Demo{*

*int x = 10;*

*public static void main(String args[]) throws ClassNotFoundException, InstantiationException, IllegalAccessException*

*{*

*Class myClass = Class.forName("Demo");*

*Object obj = myClass.newInstance();*

*Demo d1 = (Demo) obj;*

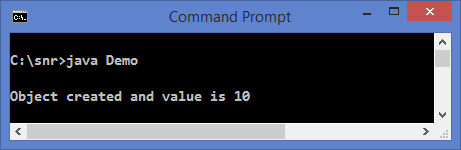
*d1.x = 10;*

*System.out.println("Object created and value is " + d1.x);*

***// prints 10***

*}*

*}*



**forName("Demo")** static method of **java.lang.Class** loads the .class file of Demo from hard disk into the RAM and returns a Class object; in the code it is named as **myClas**s. **myClass**refers to Demo.class. **forName()** method throws [java.lang.ClassNotFoundException](http://way2java.com/exceptions/classnotfoundexception/). **forName()**method, we use mostly in   
JDBC to load the database driver class.

**newInstance()** method of class **Class** returns an object of [Object](http://way2java.com/java-lang/class-object/) class and is [type casted](http://way2java.com/casting-operations/object-casting/) in the next step to Demo **d1**. **newInstance()** method throws **java.lang.InstantiationException**,**java.lang.IllegalAccessException**.

1. **read Object Java**

**serialization** is a round about process; I do not like it personally. My aim is to show there exists a way to create an object without [new keyword](http://way2java.com/oops-concepts/reference-variables-anonymous-objects/) using serialization.

It is only coding importance, nobody follows this style to create an object without new keyword. The best way is cloning. Serialization is used to write an object to file or sending the object across network.

*import java.io.\*;*

*class Test implements Serializable{*

*int x;*

*}*

*public class Demo{*

*public static void main(String args[]) throws FileNotFoundException, IOException, ClassNotFoundException{*

*// first, write an object t1 to file abc.txt*

*ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("abc.txt"));*

*Test t1 = new Test();*

*t1.x = 100;*

*oos.writeObject(t1);*

*oos.close();*

*// then, read an object from file abc.txt*

*ObjectInputStream ois = new ObjectInputStream(new FileInputStream("abc.txt"));*

*Object obj = ois.readObject();*

*Test t2 = (Test) obj;*

*ois.close();*

*System.out.println(t1.x); // prints 100*

*}*

*}*

Using **ObjectOutputStream** object **oos**, the object **t1** of **Test** class is written to a file**abc.txt**. Then using **ObjectInputStream** method **readObject()**, the Test object is read from **abc.txt** file. The**readOject()** method returns an object of Object class and is[type casted](http://way2java.com/casting-operations/object-casting/)to Test class. The emphasis is that we are not using **new keyword** to create an object**t2**. But do not say that I have used new keyword while creating t1.

1. **Class Loader Java**

**Class Loader Java:** Here, an object of a class is created without using new keyword using**java.lang.ClassLoader**.

*public class Demo{*

*int x = 10;*

*public static void main(String args[]) throws Exception {*

*Class cls = Class.forName("Demo");*

*ClassLoader cLoader = cls.getClassLoader();*

*Class cl = cLoader.loadClass("Demo");*

*Demo d1 = (Demo) cl.newInstance();*

*System.out.println(d1.x); // prints 10*

*}*

*}*

*Class cls = Class.forName(“Demo”);*

The [forName()](http://way2java.com/java-lang/java-forname-newinstance/) method of class **Class** returns an object of Class. Now **cls** represents Demo class.

*ClassLoader cLoader = cls.getClassLoader();  
Class cl = cLoader.loadClass(“Demo”);*

**getClassLoader()** method of Class returns an object of ClassLoader (it is abstract class).**loadClass()** returns an object Class.

*Demo d1 = (Demo) cl.newInstance();*

The **newInstance()** method returns an [Object](http://way2java.com/java-lang/class-object/) and is type casted to Demo. Now **Demo** object**d1**is created without using new keyword. It is a very round about process. The best one is using [cloning](http://way2java.com/java-lang/cloning-%e2%80%93-duplicating-an-object-marker-interface/).

1. **Creating Java Object without new Keyword using Reflection**

Object without new Keyword: We have seen four styles of creating a Java object without using new keyword. Now let us see with [Reflection API](http://way2java.com/reflection-api/reflection-api/) [Constructor](http://way2java.com/reflection-api/read-the-constructors/) class.

***import java.lang.reflect.\*;***

***public class Demo***

***{***

***int x = 10;***

***public Demo() { } // default constructor. Even without it also program words as default is created implicitly***

***public static void main(String args[]) throws Exception***

***{***

***Class cl = Class.forName("Demo");***

***Constructor con[] = cl.getDeclaredConstructors();***

***Demo d1 = (Demo) con[0].newInstance();***

***System.out.println(d1.x); // prints 10***

***}***

***}***

*Class cl = Class.forName(“Demo”);  
Constructor con[] = cl.getDeclaredConstructors();*

[forName()](http://way2java.com/java-lang/java-forname-newinstance/) method of class Class loads the class Demo into the RAM and returns a **Class**object. The **Class** object **cl** refers Demo.class. **getDeclaredConstructors()** method returns the list of all constructors in Demo class as an array of [java.lang.reflect.Constructor](http://way2java.com/reflection-api/read-the-constructors/) class. The first element of **con[]**array contains the default constructor.

*Demo d1 = (Demo) con[0].newInstance();*

**con[0]** refers the default constructor using which Demo object **d1**is created. Now**d1** object is created without using new keyword.

**Java getClass**

##### Java getClass () method is used to know a class’s super classes and interfaces, its name etc. Following code explains.

Following is the class signature of class Class.

**public final class Class extends Object**

Each class, like**Demo**, **Test**, **Integer**, **String** etc., executed at runtime will be represented by a **Class** object by JVM. class **Class** is used extensively by [Reflection API](http://way2java.com/reflection-api/reflection-api/).

###### Let us write one program to show how to get an object of **Class** and to use its methods Java getClass ().

***interface*** *A { }*

***interface*** *B { }*

***class*** *C { }*

***public******class*** *Demo* ***extends*** *C* ***implements*** *A, B{*

***public******static******void*** *main(String args[]){*

*Integer i1 =* ***new*** *Integer(10);*

*System.out.println("i1.getClass(): " + i1.getClass());*

*java.util.Date today =* ***new*** *java.util.Date();*

*System.out.println("today.getClass(): " + today.getClass());*

*Demo d1 =* ***new*** *Demo();*

*Class cls = d1.getClass();*

*System.out.println("cls: " + cls);*

*System.out.println("cls.getName(): " + cls.getName());*

*System.out.println("Demo super class: " + cls.getSuperclass());*

*Class cArray[] = cls.getInterfaces();*

***for****(Class c : cArray){*

*System.out.println("Demo super interface: " + c);*

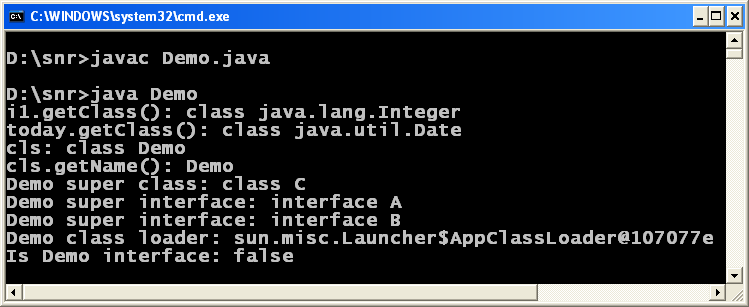
*}*

*System.out.println("Demo class loader: " + cls.getClassLoader());*

*System.out.println("Is Demo interface: " + cls.isInterface());*

*}*

*}*



***Class cls = d1.getClass();***

The class **Class** does not have a constructor and for this reason we cannot create an object of class **Class**directly. The **getClass()**method of Object class returns an object of **Class**.

*System.out.println(“cls: ” + cls);  
System.out.println(“cls.getName(): ” + cls.getName());*

The **cls** object **Class** represents the Demo class at runtime and prints the name of the class along with the word "class" like **class Demo**. The **getName()** method of **Class** returns String, just name of the class represented by**cls**; it prints simply **Demo** (not along with class word).

*System.out.println(“Demo super class: ” + cls.getSuperclass());*

**getSuperclass()** method of **Class** returns the super class of **Demo**. The above statement can be written as follows also.

*Class c1 = cls.getSuperclass();  
System.out.println(c1);*

We can know the super interfaces of class **Demo**by calling **getInterfaces()** method of **Class**. The method returns an array of Class objects.

*Class cArray[] = cls.getInterfaces();  
for(Class c : cArray)  
{  
System.out.println(“Demo super interface: ” + c);  
}*

Using the [new for loop](http://way2java.com/collections/arrays-enhanced-for-loop-foreach/), the interfaces implemented by Demo are printed.

*System.out.println(“Demo class loader: ” + cls.getClassLoader());  
System.out.println(“Is Demo interface: ” + cls.isInterface());*

The **getClassLoader()** returns the class loader which loads the class Demo.

**cls**can be known whether it represents an interface using **isInterface()**.

**Java newInstance**

##### With Java newInstacnce() of class Class, we can create an object as you will do with [new](http://way2java.com/oops-concepts/reference-variables-anonymous-objects/) keyword.

**Example on Java newInstance**

***public******class*** *Student{*

***int*** *marks;*

***public******void*** *display(){*

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

*}*

***public******static******void*** *main(String args[])* ***throws*** *InstantiationException, IllegalAccessException, ClassNotFoundException{*

*Student std1 =* ***new*** *Student();*

*std1.marks = 50;*

*Class cls1 = std1.getClass();*

*Object obj1 = cls1.newInstance();*

***// Observe, usage of Java newInstance***

*Student std2 = (Student) obj1;*

*System.out.println("\nstd1 marks: " + std1.marks);*

***// prints 50***

*System.out.println("std2 marks: " + std2.marks);*

***// prints 0***

*std2.display();*

***// Hello 1***

***// the other way of getting Class object***

*Class cls2 = Class.forName("Student");*

*Object obj2 = cls2.newInstance();*

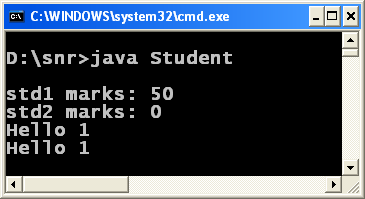
*Student std3 = (Student) obj2;*

*std3.display();*

***// Hello 1***

*}*

*}*



The **newInstance()** method of **java.lang.Class**returns an object of class Object represented by a **Class**object. It is very confusing. Let us see what I mean.

*Student std1 = new Student();  
Class cls1 = std1.getClass();  
Object obj1 = cls1.newInstance();  
Student std2 = (Student) obj1;*

The **getClass()** method of Object class returns an object of Class, **cls1**. Here, the Class object**cls1**represents a Student class object. The **newInstance()**method of java.lang.Class returns an object Object class, **obj1**. The**obj1**represents an object of **Student**. If required, we can cast this **obj1**to Student class object, as I did in the above code. These objects can be used to call the methods of **Student**.

*System.out.println(“\nstd1 marks: ” + std1.marks); // prints 50  
System.out.println(“std2 marks: ” + std2.marks); // prints 0*

The Student objects **std1** and **std2** are very different. They maintain [encapsulation](http://way2java.com/oops-concepts/three-great-principles-%e2%80%93-data-binding-data-hiding-encapsulation/). For this reason, **std2.marks** prints **0** (the default value of marks) and not 50 of **std1** marks.

The **newInstance()** method throws two [checked exceptions](http://way2java.com/exceptions/hierarchy-of-exceptions-checkedunchecked-exceptions/) – [InstantiationException](http://way2java.com/exceptions/java-instantiationexception/) and IllegalAccessException.

*Class cls2 = Class.forName(“Student”);  
Object obj2 = cls2.newInstance();*

There is another way of getting Class object. It is by using **forName()** method of Class as in the above code.

The **forName()** method throws [ClassNotFoundException](http://way2java.com/exceptions/classnotfoundexception/).

**Basic Constructs – Objects**

Every real time entity can be treated as an object. For example, a motor car can be treated as an object. Variables are used to set the properties for an object. To manipulate the variables, methods are used. Methods are called with **objects**. So variables, methods and objects constitute a program. An object is a handle with which all the code of a class can be accessed or manipulated.

A number of objects can be created for a single class. Each object can be given properties through variables. OOP permits to have a separate set of values for each object. Methods increase the reusability of code. Methods of Java are called as procedures or functions in other languages.

Let us define what an object is. A C++ programmer says, an instance of a class is known as object. Of course, he is right but the definition is somewhat confusing. Let us put it more explanatory. Suppose, when we cut the tree, we get the wood. Wood, as it is, is useless until and unless it is converted into objects like door, table etc. Without the articles like door or table, wood is waste. These articles are termed as objects in an OOP language. A class comprises of methods, constructors and variables etc. To use the class we require an object. So, the object represents the whole class. We can treat object as a handle with which the whole class can be handled. Without object, the class is useless as it cannot be used. In our first simple program, the name of the class is “Wishes” and includes only one method, main() method. We did not create any objects and we do it in the next program.

**Shallow Copying Vs Deep Copying Copying an Object**

###### Copying an object involves getting another object with the same properties of the original. Here, there exists two ways – two objects having their own set of properties (instance variables) or both objects referring the same location of properties.

**Shallow Copying**

Shallow copying is the easier of the two styles; here, one object is assigned with another. When assigned, both objects refer the same location of variables. When both objects refer or share the same location, the minus point is if one object changes the value, the other object also gets affected. That is, to say straight is **no encapsulation** exists. One small advantage is memory is less used.

Following is the example on shallow copying.

*public class Student{*

*int marks;*

*public static void main(String args[]){*

*Student std1 = new Student();*

*Student std2 = new Student();*

*std1.marks = 75;*

*std2 = std1;*  ***// object to object assignment***

*System.out.println("std1 marks before change: " + td1.marks);* ***// 75***

*System.out.println("std2 marks before change: " + td2.marks);* ***// 75***

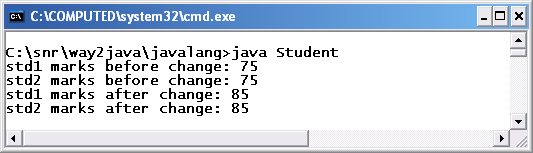
*std1.marks = 85;*

*System.out.println("std1 marks after change: " + std1.marks);* ***// 85***

*System.out.println("std2 marks after change: " + std2.marks);* ***// 85***

*}*

*}*



**std1** and **std2** are the two objects of **Student** class. Student object **std1** binds with an instance variable **marks** with a value of 75 (we call marks is a property of Student).

*std2 = std1;*

In the above statement, object **std2** is assigned with **std1**. Java rule is, when two objects are assigned, they refer the same location of variables (other way, this is how Java achieves pointer concept). The two objects **std1** and **std2** refer the same location of **marks**. The bad affect is if one object changes the value, the other is also gets affected. For this reason, when**std1.marks** is assigned with a new value of 85, **std2.marks** is also gets changed; observe the screenshot.

Assigning object to object is known as **shallow copying**. Even though it is easier, **no encapsulation** exists.

**Deep Copying**

The minus of point of **shallow copying** is over come in **deep copying**. In deep copying, we assign variable by variable of one object to the other. Even though, it is tedious,**encapsulation exists**.

**Following program explains.**

*public class Officer{*

*int salary;*

*public static void main(String args[]){*

*Officer o1 = new Officer();*

*Officer o2 = new Officer();*

*o1.salary = 5000;*

*o2.salary = o1.salary;*

*System.out.println("o1 salary before change: " + o1.salary); // 5000*

*System.out.println("o2 salary before change: " + o2.salary); // 5000*

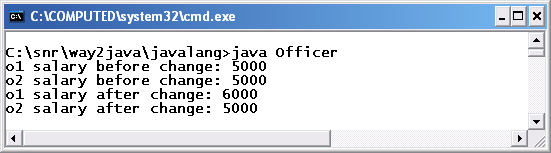
*o1.salary = 6000;*

*System.out.println("o1 salary after change: " + o1.salary); // 6000*

*System.out.println("o2 salary after change: " + o2.salary); // 5000*

*}*

*}*



To make the program simple, only one property is given to **Officer**, **salary**. Two objects **o1** and**o2** are created.

*o2.salary = o1.salary;*

In the above statement, salary of **o1** is assigned to the salary of **o2**. Here, variable to variable is assigned (in the earlier program, object to object is assigned). In this case, encapsulation is maintained as both objects have their separate locations for salary variable. One object does not have any relation (connection) with the other. For this reason, as you can observe in the above screenshot, when **o1** salary is changed to 6000, **o2.salary** did not get affect. You can also try with **o2** salary and observe **o1**salary does not change.

**Java Cloning**

Another copying exists in Java called [**cloning**](http://way2java.com/java-lang/cloning-%e2%80%93-duplicating-an-object-marker-interface/). Cloning gives the affect of deep copying with shallow copying.

**Cloning Duplicate Object Marker Interface**

We have seen earlier, shallow copying is easier and deep copying takes more time, especially, when properties (variables) are many. Here comes another way of copying, **cloning**.

In cloning, object to object is assigned, but at the same time encapsulation is maintained. Both objects occupy two different locations; that is, cloning creates a duplicate object which is no way connected with the original one.

###### **Cloning – interface Cloneable and clone() method**

In cloning, the **interface Cloneable** and method **clone()** of Object class are used. To clone an object, the class should implement Cloneable interface; else, the JVM throws a checked exception, **CloneNotSupportedException**.

###### **Example on Cloning Duplicate Object Marker Interface**

*public class Worker implements Cloneable{*

*int salary;*

*public static void main(String args[]) {*

*Worker work1 = new Worker();*

*work1.salary = 3000;*

*try{*

*Worker work2 = (Worker) work1.clone();*

*System.out.println("After cloning, work2 salary: " + work2.salary);* ***// 3000***

*work1.salary = 4000;*  ***// change work1 salary***

*System.out.println("After changing work1 salary, work2 salary: "+ work2.salary);*

***// 3000***

*work2.salary = 5000;*  ***// now change work2 salary***

*System.out.println("After changing work2 salary, work1 salary: "+ work1.salary);* ***// 4000***

***// let us see the hash codes***

*System.out.println("Hash code of work1: " + work1.hashCode());*

*System.out.println("Hash code of work2: " + work2.hashCode());*

*}*

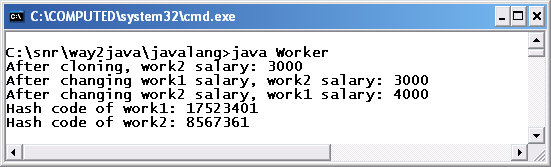
*catch(CloneNotSupportedException e){*

*System.out.println("Check your class implemented Cloneable interface. " + e);*

*}*

*}*

*}*



Observe, **Worker** class implements **Cloneable** interface.

*Worker work2 = (Worker) work1.clone();*

The above statement can be split into two for more clarity.

*Object obj = work1.clone();  
Worker work2 = (Worker) obj;*

The **clone()**method returns an object of Object class. This is explicitly type casted to Worker. Now the objects **work1** and **work2** occupy different locations and there by encapsulation is maintained. For this reason, when **work1** is changed **work2** salary is not getting affected and similarly with **work2** also. Observe the screenshot and the program.

As both objects occupy different locations, their **hash codes** are also different. Following are the requirements for cloning.

1. **Implement Cloneable interface**
2. **Use clone() method of Object class**
3. **After cloning, explicit casting is required**

**Marker Interface – Cloneable**

Java API comes with six marker interfaces. An interface that does not contain any methods and variables is known as a **marker Interface**. For more explanation on marker interfaces and their list, refer I/O streams.

**Array Cloning**

Arrays are predefined objects in the Java language itself. So, like any other object, arrays also can be cloned. Following program illustrates.

*public class Demo{*

*public static void main(String args[]) {*

*int x1[] = { 10, 29, 30, 40 };*

*int x2[] = x1.clone();*

*System.out.println(x2[0]); // prints 10*

*// now changing x1 value does not affect x2*

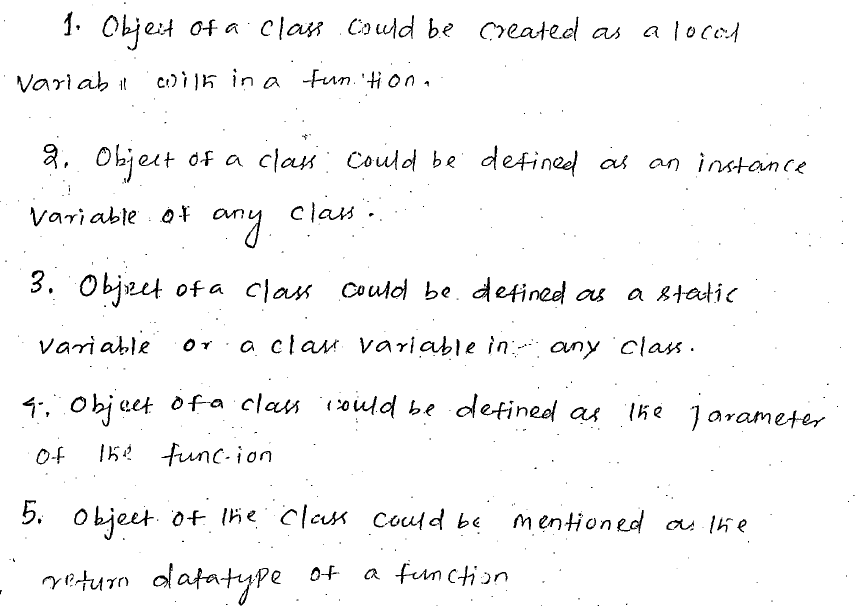
*x1[0] = 100;*

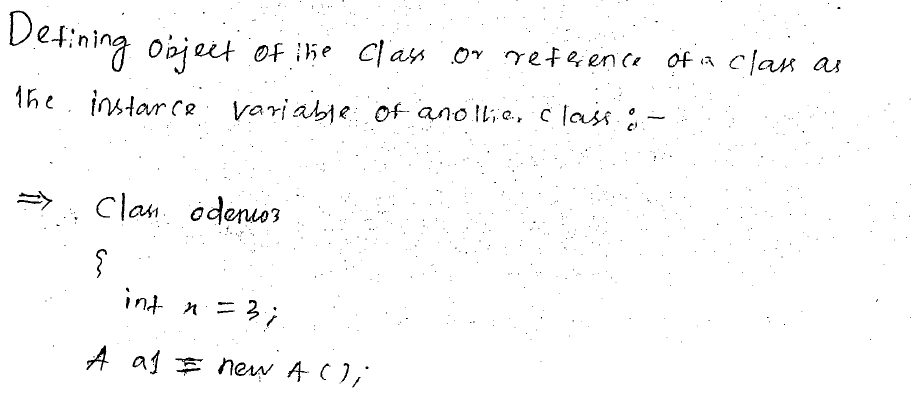
*System.out.println(x2[0]); // printgs same 10*

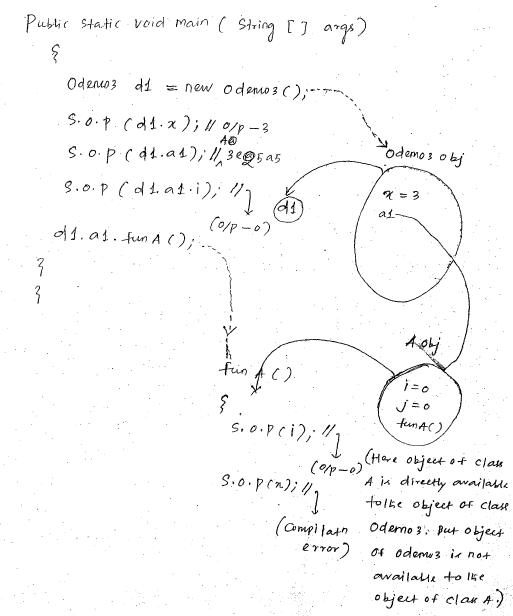
*}*

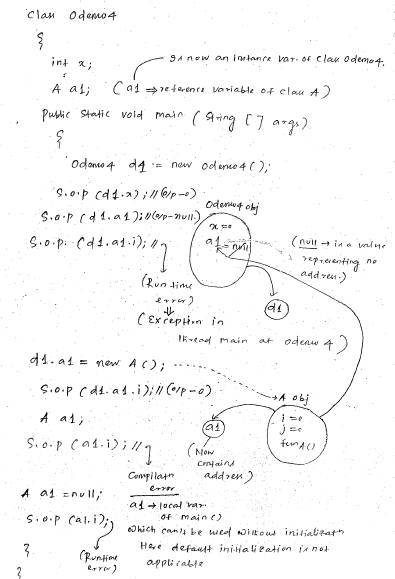
*}*

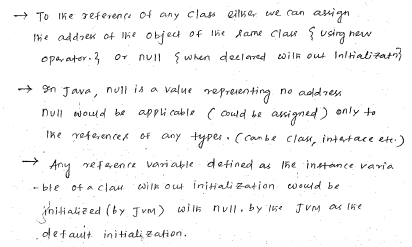
***The 5 different locations where we can see objects in Java:***

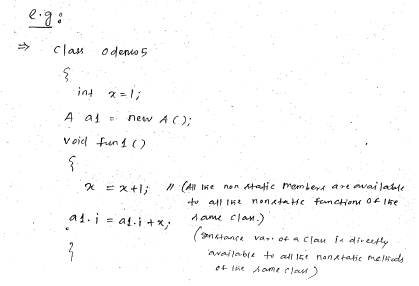
******

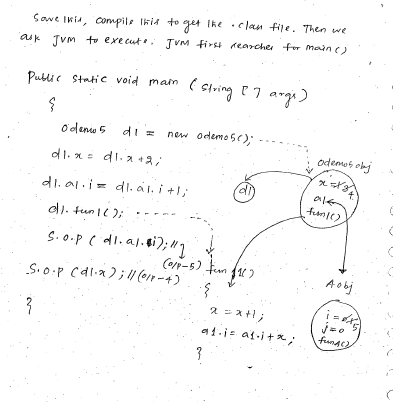
******

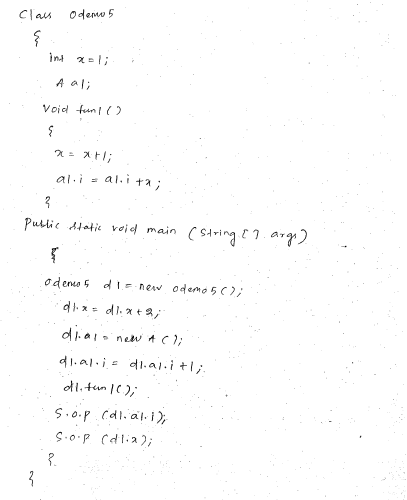
******

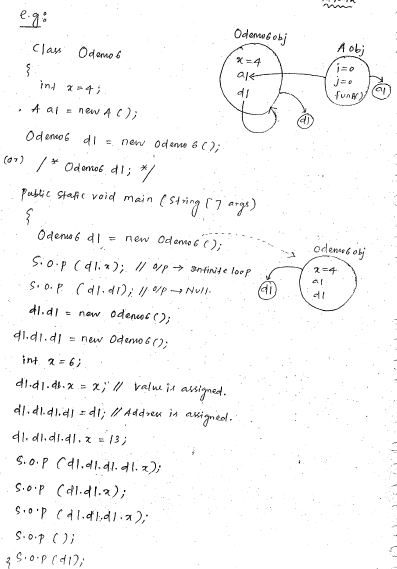
******

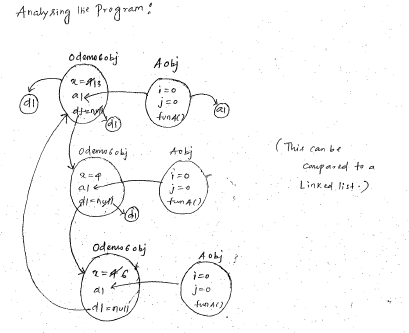
******

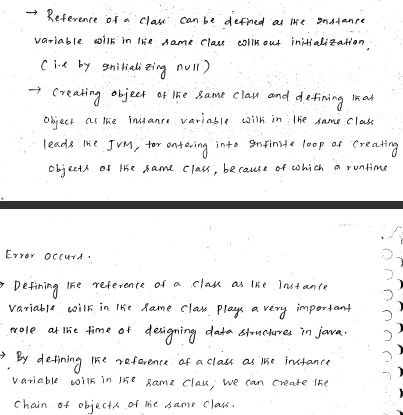
******

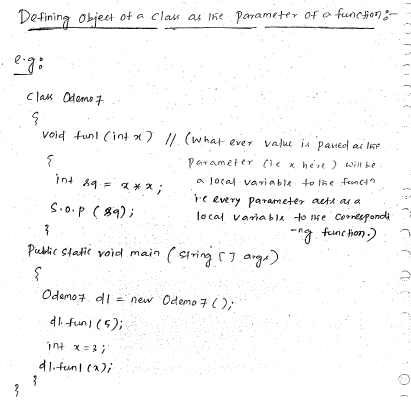
******

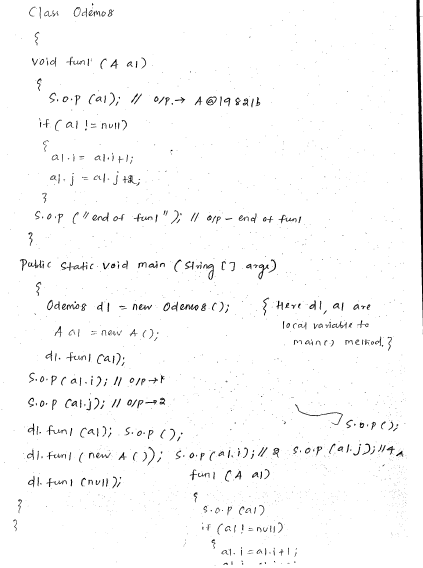
******

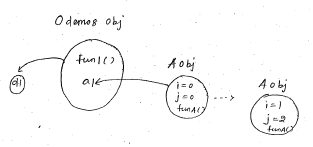
******

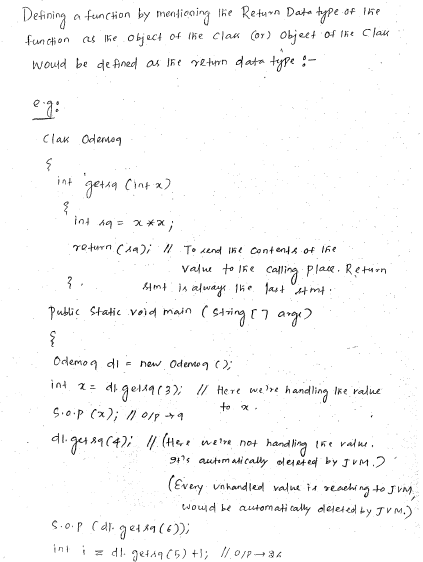
******

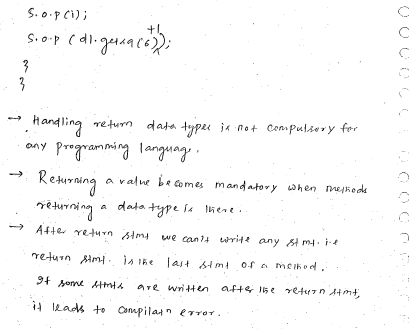
******

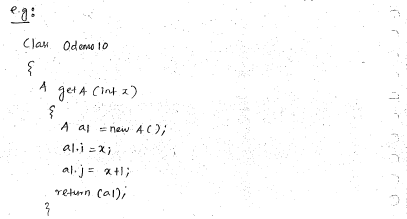
******

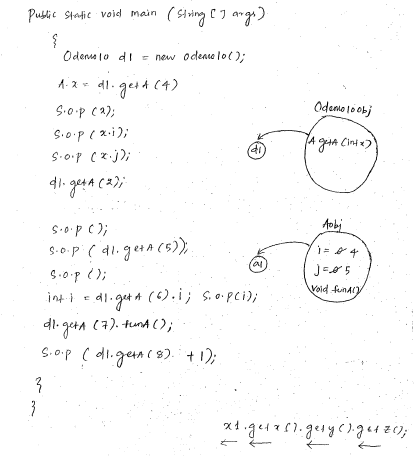
******

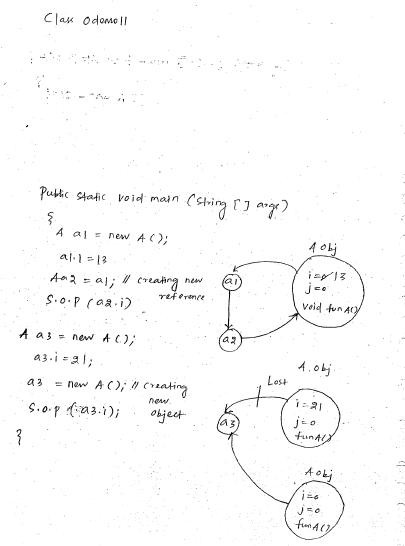
******

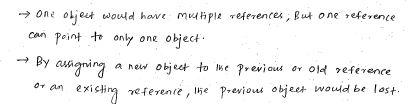
******

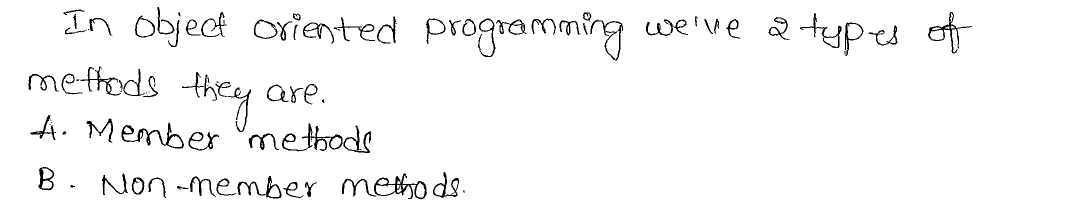
******

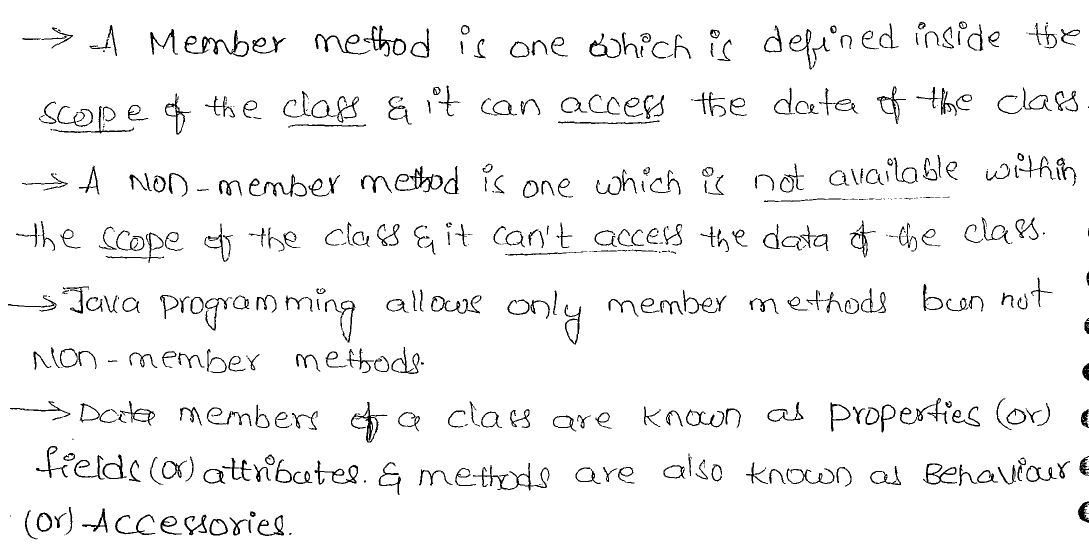
******

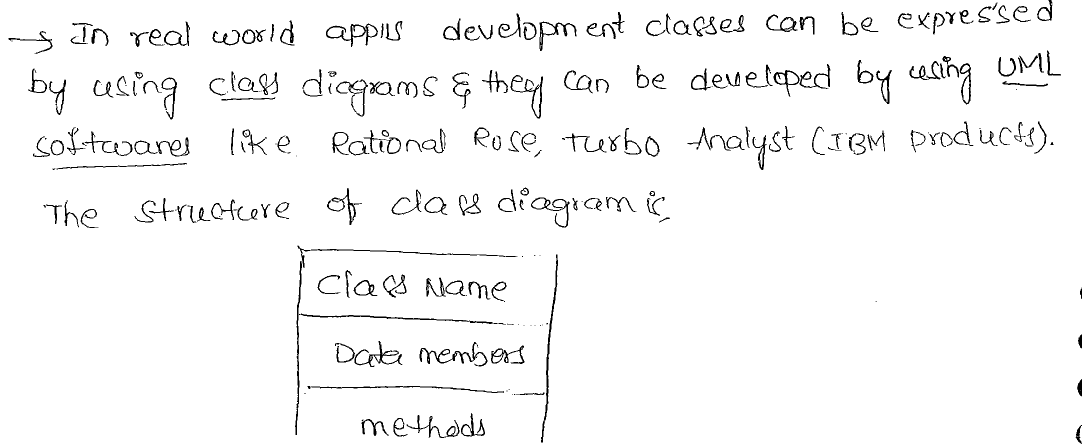
******

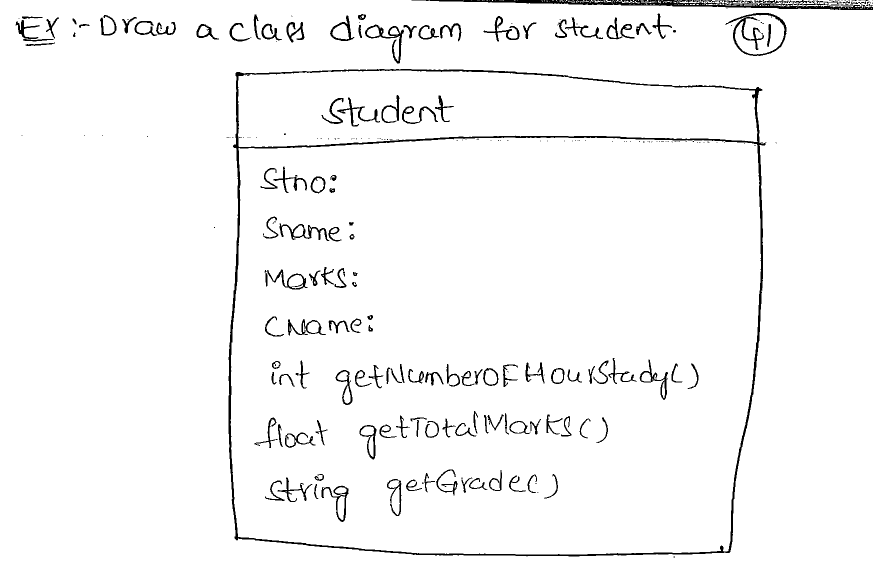
******

******

******

******

******

******

***Object of any class could be created at any place:***

***A.java:*** *Class A{*

*int i,j;*

*void funA(){*

*System.out.println(“funA() of class A”);*

*}*

*}*

*Compile 🡪 A.class*

*Class Odemo1{*

*int x,y;*

*void fun1(){*

*x=x+1;*

*i=x+1;//Not Possible*

*A a1=new A();//Creation object of class A*

*Odemo1 obj*

*a1.i=9;*

*}*

X=0;🡪 1  
y=0;  
fun1()

*public static void main(String[] args){*

*Odemo1 d1=new Odemo1();*

*d1.fun1();  
 System.out.println(a1.i);//****Compilation Error***

*A a1=new A();*

fun1(){  
x=x+1;  
A a1=new A();  
a1.i=9;  
}  
when fun1(); execution is finished, its deleted from RAM. So its local variable a1 is also deleted from RAM.

Here the control comes out of the function fun1().  
The object created here is also deleted.

*System.out.println(a1.i);// Output: 0*

*}*

A Obj

*}*

A Obj

*Class Odemo2{*

int x,y;

*void fun1(){*

*x=x+1;*

*Odemo2 d1=new Odemo2();*

*//Object of a class can be created as a local variable within the non-static method of the same class.*

*d1.x=x+2;*

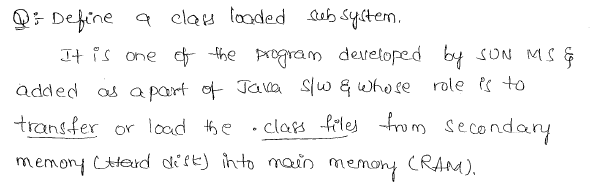
*}*

*public static void main(String[] args){*

Odemo2 d1=new Odemo2();

d1.fun1();  
System.out.println(d1.x);//Output: 1  
}

*}*

******

***Note***

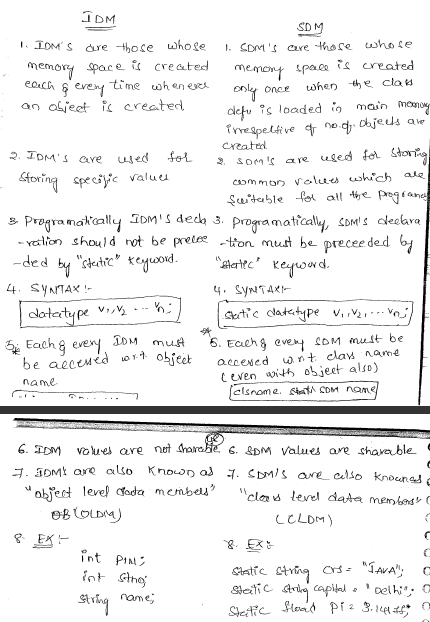
*All the objects of Java programming resides in heap memory.*

*All the content/ figurative values resides in Associated Memory.*

***Types of data members:***

*In Java there are two types of data members in Java. They are*

* 1. *Instance/ Non-static data members.*
  2. *Static data members.*

**

***Example***

*class Student{*

*int rno;*

*String name;*

*double fee;*

*void display(){*

*int a=10;*

*System.out.println("RNO:"+rno);*

*System.out.println("NAME:"+name);*

*System.out.println("FEE:"+fee);*

*}*

*public static void main(String args[]){*

*/\**

*Student s = new Student();*

*s.display()*

***//accessing variables using reference***

*s.rno=7;*

*s.name="sachin";*

*s.fee=1200.00;*

*System.out.println("rno="+s.rno);*

*System.out.println("name="+s.name);*

*System.out.println("fee="+s.fee);*

***//local variables are not part of object***

*//System.out.println("a="+s.a); // X-invalid*

*s.display();*

*\*/*

*Student s;*

*s = new Student();*

*s.display();*

***//anonymous objects***

***/\* new Student(); //valid but no output\*/***

***/\*(new Student()).display();//valid & student data displayed\*/***

***/\****

***RTE:NullPointerException***

*Student s=null;*

*s.display();*

*\*/*

*}*

*}*

**Instance variables**

The variables declared in side class and outside the methods then these variables are called as instance variables. it means they are part of the object.

When we declare instance variables and if we don't provide any value then they automatically initialized with default values.

|  |  |
| --- | --- |
| **data type** | **default value** |
| Byte | 0 |
| Short | 0 |
| Int | 0 |
| Long | 0 |
| Float | 0 |
| Double | 0 |
| Char | space |
| Boolean | FALSE |
| String | null |
| StringBuffer | null |
| Student | null |
| any reference | null |

Instance variables are used in the class to store the data which are also called as properties.

**Instance methods:**  
  
- In a class we can declare methods to perform some operations.

* Methods are nothing but actions that are done by the particular object.
* In a class we can declare any number of methods including main() method but JVM always invoke only main() method
* But if we want to invoke other methods which are defined by the programmer then the programmer is only responsible to invoke these methods explicitly.

**Note:**

-In Java declaring the variables is dynamic it means we can declare the variables any where inside the program

-Java language is called as strongly type language because it always check for the variables whether they declared or not and type of the value that we are storing is correct or not

# Java Class Instance

The following is the general syntax to create an instance of a class:

**new** <Class Constructor>;

The new operator is followed by a call to the constructor.

The new operator creates an instance of a class by allocating the memory on heap. The following statement creates an instance of the Dog class:

**new** Dog();

Dog() is a call to the constructor of the Dog class.

When we do not add a constructor to a class, the Java compiler adds one for us.

The constructor added by the Java compiler is called a default constructor. The default constructor accepts no arguments.

The name of the constructor of a class is the same as the class name.

The new operator allocates memory for each instance field of the class. Class static variables are not allocated memory when an instance of the class is created.

To access instance variables of an instance of a class, we must have its reference.

The name of a class defines a new reference type in Java. A variable of a specific reference type can store the reference of an instance of the same reference type.

To declare a reference variable, which will store a reference of an instance of the Dog class.

Dog anInstance;

Dog is the class name, which is also a reference type, and anInstance is a variable of that type.

anInstance is a reference variable of Dog type. The anInstance variable can be used to store a reference of an instance of the Dog class.

The new operator allocates the memory for a new instance of a class and returns the reference to that instance.

We need to store the reference returned by the new operator in a reference variable.

anInstance = **new** Dog();

**null Reference Type**

We can assign a null value to a variable of any reference type. A null value means that the reference variable is referring to no object.

Dog obj = null; // obj is not referring to any object

obj = **new** Dog(); // Now, obj is referring to a valid Dog object

You can use a null literal with comparison operators to check for equality and inequality.

**if** (obj == null) {

//obj is null

}

**if** (obj != null) {

//obj is not null

}

Java does not mix reference types and primitive types. We cannot assign null to a primitive type variable.

**Dot Notation to Access Fields of a Class**

Dot notation is used to refer to instance variables.

The general form of the dot notation syntax is

<Reference Variable Name>.<Instance Variable Name>

obj.name to refer to the name instance variable of the instance to which the obj reference variable is referring.

To assign a value to the name instance variable, use

obj.name = **"Rectangle"**;

The following statement assigns the value of the name instance variable to a String variable aName:

String aName = obj.name;

To reference class variables, use the name of the class.

ClassName.ClassVariableName

For example, we can use Dog.count to refer to the count class variable of the Dog class.

To assign a new value to the count class variable

Dog.count = 1;

To read the value of the count class variable into a variable

long count = Dog.count;

The following code shows how to use class fields

***class*** *Dog {*

***static******int*** *count = 0;*

*String name;*

*String gender;*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*Dog obj =* ***new*** *Dog();*

***// Increase count by one***

*Dog.count++;*

*obj.name =* ***"Java"****;*

*obj.gender =* ***"Male"****;*

*obj.name =* ***"XML"****;*

*String changedName = obj.name;*

*}*

*}*

**Default Initialization of Fields**

All fields of a class, static as well as non-static, are initialized to a default value.

The default value of a field depends on its data type.

A numeric field (byte, short, char, int, long, float, and double) is initialized to zero. A boolean field is initialized to false. A reference type field is initialized to null.

The following code demonstrates the default initialization of fields.

***public******class*** *Main {*

***byte*** *b;* ***short*** *s;*

***int*** *i;*

***long*** *l;*

***float*** *f;*

***double*** *d;*

***boolean*** *bool;*

*String str;*

***public******static******void*** *main(String[] args) {*

*Main obj =* ***new*** *Main();*

*System.out.println(****"byte is initialized to "*** *+ obj.l);*

*System.out.println(****"short is initialized to "*** *+ obj.s);*

*System.out.println(****"int is initialized to "*** *+ obj.i);*

*System.out.println(****"long is initialized to "*** *+ obj.l);*

*System.out.println(****"float is initialized to "*** *+ obj.f);*

*System.out.println(****"double is initialized to "*** *+ obj.d);*

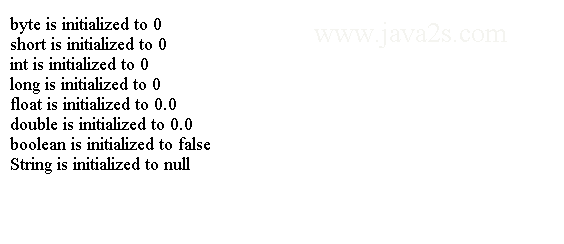
*System.out.println(****"boolean is initialized to "*** *+ obj.bool);*

*System.out.println(****"String is initialized to "*** *+ obj.str);*

*}*

*}*

The code above generates the following result.



# Java Access Level

Class simple name is the name between class keyword and {.

When we refer to a class by its simple name, the compiler looks for that class declaration in the same package where the referring class is.

We can use full name to reference a class as follows.

com.java2s.Dog aDog;

The general syntax specifying access-level for a class is

*<access level modifier>****class*** *<****class*** *name> {*

*// Body of the* ***class***

*}*

There are only two valid values for <access level modifier> in a class declaration:

* no value
* public

No value is known as package-level access. A class with package-level access can be accessed only within the package in which it has been declared.

Class with public access level modifier can be accessed from any package in the application.

*package com.java2s;*

***public******class*** *Dog {*

*}*

# Java Object Class

Java has an Object class in the java.lang package.

All Java classes extend the Object class directly or indirectly.

All Java classes are a subclass of the Object class and the Object class is the superclass of all classes.

The Object class itself does not have a superclass.

A reference variable of the Object class can hold a reference of an object of any class.

The following code declares a reference variable obj of the Object type:

Object obj;

**Methods**

The Object class has nine methods, which are available to be used in all classes in Java.

* **public String toString()**

It is Implemented in Object class and we can customize it.  
It returns a string representation of an object.   
Typically, it is used for debugging purpose.

* **public boolean equals(Object obj)**

It is implemented in Object class and we can customize it.  
It is used to compare two objects for equality.

* **public int hashCode()**  
  It is implemented in Object class and we can customize it.  
  It returns a hash code (an integer) value of an object.
* **protected Object clone() throws CloneNotSupportedException**  
  It is not implemented in Object class and we can customize it by overriding the clone method.  
  It is used to make a copy of an object.
* **protected void finalize() throws Throwable**  
  It is not implemeneted in Object class and we can customize it.  
  It is called by the garbage collector before an object is destroyed.
* **public final Class getClass()**  
  It is implemented in Object class and we cannot customize it.  
  It returns a reference to the Class object of the object.
* **public final void notify()**  
  It is implemeneted in Object class and we cannot customize it.  
  This method notifies one thread in the wait queue of the object.
* **public final void notifyAll()**  
  It is implemeneted in Object class and we cannot customize it.  
  This method notifies all threads in the wait queue of the object.
* **public final void wait() throws InterruptedException  
  public final void wait(long timeout) throws InterruptedException   
  public final void wait (long timeout, int nanos) throws InterruptedException**  
  It is implemeneted in Object class and we cannot customize it.  
  Makes a thread wait in the wait queue of the object with or without a timeout.

**Example**

The following code shows how to reimplement the toString() method of the Object class.

***public******class*** *Test {*

***public*** *String toString() {*

*return* ***"Here is a string"****;*

*}*

*}*

**What Is the Class of an Object?**

Every object in Java belongs to a class.

The getClass() method of the Object class returns the reference of the Class object.

The following code shows how to get the reference of the Class object for a Cat object:

*Cat c =* ***new*** *Cat();*

*Class catClass = c.getClass();*

The Class class is generic and its formal type parameter is the name of the class that is represented by its object.

We can rewrite the above statement using generics.

*Class<Cat> catClass = c.getClass();*

# Java Hash Code

**Hash Code of an Object**

A hash code is an integer value. The algorithm to compute an integer is called a hash function.

Java uses hash codes to efficiently retrieve data from hash based collections.

The Object class has a hashCode() method that returns an int, which is the hash code of the object.

The default implementation of this method computes the hash code of an object by converting the memory address of the object into an integer.

Here are the rules that we must follow when we override the hashCode() method in your class.

**Suppose there are two object references, x and y.**

If x.equals(y) returns true, x.hashCode() must return an integer, which is equal to y.hashCode().

If two objects are equal using the equals() method, they must have the same hash codes.

If x.hashCode() is equal to y.hashCode(), it is not necessary that x.equals(y) returns true.

If the hashCode() method is called on the same object multiple times, the method must return the same integer value.

If a class overrides any of these two methods, it must override both for the objects of the class to work correctly in hash-based collections.

Java 7 added a utility class java.lang.Objects. It contains a hash() method that computes the hash code for any number of values.

From java 7, use the Objects.hash() method to compute the hash code of an object.

**Example**

The following code shows how to calculate a hash value.

***class*** *Book {*

***private*** *String title;*

***private*** *String author;*

***public******int*** *hashCode() {*

***int*** *hash = 37;*

***int*** *code = 0;*

***// Use title***

*code = (title == null ? 0 : title.hashCode());*

*hash = hash \* 59 + code;*

***// Use author***

*code = (author == null ? 0 : author.hashCode());*

*hash = hash \* 59 + code;*

***return*** *hash;*

*}*

*}*

# Java Object Equals

The following code shows how to implement equals() and hashCode() Methods

***class*** *Point {* ***private******int*** *x;*

***private******int*** *y;*

***public*** *Point(****int*** *x,* ***int*** *y) {*

*this.x = x;*

*this.y = y;*

*}*

***/\* implement the equals() method \*/***

***public******boolean*** *equals(Object otherObject) {*

***// Are the same?***

***if*** *(****this*** *== otherObject) {*

***return*** *true;*

*}*

***// Is otherObject a null reference?***

***if*** *(otherObject == null) {*

***return*** *false;*

*}*

***// Do they belong to the same class?***

***if*** *(this.getClass() != otherObject.getClass()) {*

***return*** *false;*

*}*

***// Get the reference of otherObject in a SmartPoint variable***

*Point otherPoint = (Point) otherObject;*

***// Do they have the same x and y co-ordinates***

***boolean*** *isSamePoint = (this.x == otherPoint.x && this.y == otherPoint.y);*

***return*** *isSamePoint;*

*}*

***/\****

***\* implement hashCode() method of the Object class, which is a requirement***

***\* when you implement equals() method***

***\*/***

***public******int*** *hashCode() {*

***return*** *(this.x + this.y);*

*}*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*Point pt1 =* ***new*** *Point(10, 10);*

*Point pt2 =* ***new*** *Point(10, 10);*

*Point pt3 =* ***new*** *Point(12, 19);*

*Point pt4 = pt1;*

*System.out.println(****"pt1 == pt1: "*** *+ (pt1 == pt1));*

*System.out.println(****"pt1.equals(pt1): "*** *+ pt1.equals(pt1));*

*System.out.println(****"pt1 == pt2: "*** *+ (pt1 == pt2));*

*System.out.println(****"pt1.equals(pt2): "*** *+ pt1.equals(pt2));*

*System.out.println(****"pt1 == pt3: "*** *+ (pt1 == pt3));*

*System.out.println(****"pt1.equals(pt3): "*** *+ pt1.equals(pt3));*

*System.out.println(****"pt1 == pt4: "*** *+ (pt1 == pt4));*

*System.out.println(****"pt1.equals(pt4): "*** *+ pt1.equals(pt4));*

*}*

*}*

The code above generates the following result.



**Note**

Here are specifications for the equals() method's implementation. Assume that x, y, and z are non-null references of three objects.

* **Reflexivity**. The expression x.equals(x) should return true.
* **Symmetry**. If x.equals(y) returns true, y.equals(x) must return true.
* **Transitivity**. If x.equals(y) returns true and y.equals(z) returns true, x.equals(z) must return true.
* **Consistency**. If x.equals(y) returns true, it should keep returning true until the state of x or y is modified. If x.equals(y) returns false, it should keep returning false until the state of x or y is modified.
* **Comparison with null reference**: An object of any class should not be equal to a null reference. The expression x.equals(null) should always return false.
* **Relationship with hashCode() method**: If x.equals(y) returns true, x.hashCode() must return the same value as y.hashCode().

# Java Object toString

The string representation of an object should contain enough information about the state of the object in a readable format.

The toString() method of the Object class represents the object of a class in a string.

The Object class provides a default implementation of the toString() method. It returns a string in the following format:

*<fully qualified* ***class*** *name>@<hash code of object in hexadecimal>*

**Example**

Consider the following code and its output. You may get a different output.

***public******class*** *Main{*

***public******static******void*** *main(String[] argv){*

*Object obj =* ***new*** *Object();*

*String objStr = obj.toString();*

*System.out.println(objStr);*

*}*

*}*

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/E/EXAMPLE__EF64B6F4D24BFF114675.PNG

**Example**

The following code shows how to create your own toString method.

***public******class*** *Main{*

***public******static******void*** *main(String[] argv){*

*MyClass obj =* ***new*** *MyClass(123);*

*String objStr = obj.toString();*

*System.out.println(objStr);*

*}*

*}*

***class*** *MyClass {*

***private******int*** *value;*

***public*** *MyClass(****int*** *value) {*

*this.value = value;*

*}*

***public******void*** *setValue(****int*** *value) {*

*this.value = value;*

*}*

***public******int*** *getValue() {*

***return*** *value;*

*}*

***/\* override toString() method of the Object class \*/***

***public*** *String toString() {*

***// Return the stored value as a string***

*String str = String.valueOf(this.value);*

***return*** *str;*

*}*

*}*

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/E/EXAMPLE_2__DDFD3FD151F61B265C8D.PNG

**Note**

You need to make sure it is declared public, its return type is String, and it does not take any parameters.

The toString() method of a class is very important. Java calls the toString() method automatically when it needs a string representation of the object.

**Two such situations that are worth mentioning:**

When you concatenate a string and an object like

*String str =* ***"Hello"*** *+* ***new*** *Point(10, 20);*

Java calls the toString() method on the Point object and concatenate the returned value to the "Hello" string.

**The above statement is the same as the following one:**

*String str =* ***"Hello"*** *+* ***new*** *Point(10, 20).toString();*

# Java Object Clone

Java does not provide an automatic mechanism to clone (copy) an object.

Cloning an object means copying the content of the object bit by bit.

To support clone operation, implement the clone() method in the class.

The declaration of the clone() method in the Object class is as follows:

***protected*** *Object* ***clone****() throws CloneNotSupportedException*

clone() method is declared protected. Therefore, we cannot call it from the client code. The following code is not valid:

*Object obj =* ***new*** *Object();*

*Object* ***clone*** *= obj.clone();*

***// Error. Cannot access protected clone() method***

We need to declare the clone() method public in a class to clone objects of the class.

Its return type is Object. It means you will need to cast the returned value of the clone() method.

Suppose MyClass is cloneable. The cloning code will look as

*MyClass mc =* ***new*** *MyClass();*

*MyClass* ***clone*** *= (MyClass)mc.clone(); // Need to* ***use*** *a cast*

The clone() method in the Object class throws a CloneNotSupportedException.

To call the clone() method, we need to place the call in a try-catch block, or rethrow the exception.

**Example**

The following code shows how to implement a clone method.

***class*** *MyClass* ***implements*** *Cloneable {*

***private******double*** *value;*

***public*** *MyClass(****double*** *value) {*

*this.value = value;*

*}*

***public******void*** *setValue(****double*** *value) {*

*this.value = value;*

*}*

***public******double*** *getValue() {*

***return*** *this.value;*

*}*

***public*** *Object clone() {*

*MyClass copy = null;*

***try*** *{*

*copy = (MyClass) super.clone();*

*}* ***catch*** *(CloneNotSupportedException e) {*

*e.printStackTrace();*

*}*

***return*** *copy;*

*}*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*MyClass dh =* ***new*** *MyClass(100.00);*

*MyClass dhClone = (MyClass) dh.clone();*

*System.out.println(****"Original:"*** *+ dh.getValue());*

*System.out.println(****"Clone :"*** *+ dhClone.getValue());*

*dh.setValue(200.00);*

*dhClone.setValue(400.00);*

*System.out.println(****"Original:"*** *+ dh.getValue());*

*System.out.println(****"Clone :"*** *+ dhClone.getValue());*

*}*

*}*

The code above generates the following result.



**Example**

The following code is not returning Object type from clone method, which is compiled only in Java 5 or later.

***class*** *MyClass* ***implements*** *Cloneable {*

***public*** *MyClass* ***clone****() {*

*Object copy = null;*

*return (MyClass)copy;*

*}*

*}*

The following code shows how to do shallow clone.

***class*** *MyClass* ***implements*** *Cloneable {*

***private******double*** *value;*

***public*** *MyClass(****double*** *value) {*

*this.value = value;*

*}*

***public******void*** *setValue(****double*** *value) {*

*this.value = value;*

*}*

***public******double*** *getValue() {*

***return*** *this.value;*

*}*

***public*** *Object clone() {*

*MyClass copy = null;*

***try*** *{*

*copy = (MyClass) super.clone();*

*}* ***catch*** *(CloneNotSupportedException e) {*

*e.printStackTrace();*

*}*

***return*** *copy;*

*}*

*}*

***class*** *ShallowClone* ***implements*** *Cloneable {*

***private*** *MyClass holder =* ***new*** *MyClass(0.0);*

***public*** *ShallowClone(****double*** *value) {*

*this.holder.setValue(value);*

*}*

***public******void*** *setValue(****double*** *value) {*

*this.holder.setValue(value);*

*}*

***public******double*** *getValue() {*

***return*** *this.holder.getValue();*

*}*

***public*** *Object clone() {*

*ShallowClone copy = null;*

***try*** *{*

*copy = (ShallowClone) super.clone();*

*}* ***catch*** *(CloneNotSupportedException e) {*

*e.printStackTrace();*

*}*

***return*** *copy;*

*}*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*ShallowClone sc =* ***new*** *ShallowClone(100.00);*

*ShallowClone scClone = (ShallowClone) sc.clone();*

*System.out.println(****"Original:"*** *+ sc.getValue());*

*System.out.println(****"Clone :"*** *+ scClone.getValue());*

*sc.setValue(200.00);*

*System.out.println(****"Original:"*** *+ sc.getValue());*

*System.out.println(****"Clone :"*** *+ scClone.getValue());*

*}*

*}*

The code above generates the following result.



**Example**

The code in the clone() method of the ShallowClone class is the same as for the clone() method of the MyClass class.

When the ShallowClone class calls the clone() method of the Object class using super.clone(), it receives a shallow copy of itself. That is, it shares the DoubleHolder object used in its instance variable with its clone.

In a deep cloning, you need to clone all objects referenced by all reference instance variables of an object.

***class*** *MyClass* ***implements*** *Cloneable {*

***private******double*** *value;*

***public*** *MyClass(****double*** *value) {*

*this.value = value;*

*}*

***public******void*** *setValue(****double*** *value) {*

*this.value = value;*

*}*

***public******double*** *getValue() {*

***return*** *this.value;*

*}*

***public*** *Object clone() {*

*MyClass copy = null;*

***try*** *{*

*copy = (MyClass) super.clone();*

*}* ***catch*** *(CloneNotSupportedException e) {*

*e.printStackTrace();*

*}*

***return*** *copy;*

*}*

*}*

***class*** *DeepClone* ***implements*** *Cloneable {*

***private*** *MyClass holder =* ***new*** *MyClass(0.0);*

***public*** *DeepClone(****double*** *value) {*

*this.holder.setValue(value);*

*}*

***public******void*** *setValue(****double*** *value) {*

*this.holder.setValue(value);*

*}*

***public******double*** *getValue() {*

***return*** *this.holder.getValue();*

*}*

***public*** *Object clone() {*

*DeepClone copy = null;*

***try*** *{*

*copy = (DeepClone) super.clone();*

*copy.holder = (MyClass) this.holder.clone();*

*}* ***catch*** *(CloneNotSupportedException e) {*

*e.printStackTrace();*

*}*

***return*** *copy;*

*}*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*DeepClone sc =* ***new*** *DeepClone(100.00);*

*DeepClone scClone = (DeepClone) sc.clone();*

*System.out.println(****"Original:"*** *+ sc.getValue());*

*System.out.println(****"Clone :"*** *+ scClone.getValue());*

*sc.setValue(200.00);*

*System.out.println(****"Original:"*** *+ sc.getValue());*

*System.out.println(****"Clone :"*** *+ scClone.getValue());*

*}*

*}*

The code above generates the following result.



# Java Object Finalize

Java provides a way to perform resource release, when an object is about to be destroyed.

In Java, we create objects, but we cannot destroy objects.

The JVM runs a low priority special task called garbage collector to destroy all objects that are no longer referenced.

The garbage collector gives us a chance to execute the cleanup code before an object is destroyed.

The Object class has a finalize() method, which is declared as follows:

***protected*** *void finalize() throws Throwable { }*

The finalize() method in the Object class does not do anything.

You need to override the method in your class.

The finalize() method of your class will be called by the garbage collector before an object of your class is destroyed.

**Example**

The following code shows how to create a Finalize Class that Overrides the finalize() Method of the Object Class.

***class*** *Finalize {* ***private******int*** *x;*

***public*** *Finalize(****int*** *x) {*

*this.x = x;*

*}*

***public******void*** *finalize() {*

*System.out.println(****"Finalizing "*** *+ this.x);*

*}*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

***for*** *(****int*** *i = 0; i < 20000; i++) {*

***new*** *Finalize(i);*

*}*

*}*

*}*

The code above generates the following result.



# Java Immutable Objects

An object whose state cannot be changed after it is created is called an immutable object.

A class whose objects are immutable is called an immutable class.

An immutable object can be shared by different areas of a program without worrying about its state changes.

An immutable object is inherently thread-safe.

**Example**

The following code creates an Example of an Immutable Class.

***public******class*** *IntWrapper {*

***private******final******int*** *value;*

***public*** *IntWrapper(****int*** *value) {*

*this.value = value;*

*}*

***public******int*** *getValue() {*

***return*** *value;*

*}*

*}*

**Note**

This is how you create an object of the IntWrapper class:

*IntWrapper wrapper =* ***new*** *IntWrapper(101);*

At this point, the wrapper object holds 101 and there is no way to change it.

Therefore, the IntWrapper class is an immutable class and its objects are immutable objects.

It is good practice to declare all instance variables final so the Java compiler will enforce the immutability during compile time

# Java Objects Class

Java has a utility class Objects in the java.util package for working with objects.

It consists of all static methods. Most of the methods of the Objects class deal with null values gracefully.

The following is the list of methods in the class. Their descriptions follow the list.

* **int compare(T a, T b, Comparator c)**  
  Returns 0 if the arguments are identical and c.compare(a, b) otherwise. Consequently, if both arguments are null 0 is returned.
* **boolean deepEquals(Object a, Object b)**check if two objects are deeply equal. It returns true if both arguments are deeply equal. Otherwise, it returns false. It returns true if both arguments are null.
* **boolean equals(Object a, Object b)**compares two objects for equality. It returns true if both arguments are equal. Otherwise, it returns false. It returns true if both arguments are null.
* **int hash(Object... values)**generates a hash code for all specified objects. It can be used to compute the hash code for an object, which is based on the multiple instance fields.
* **int hashCode(Object o)**returns the hash code value of the specified object. If the argument is null, it returns 0.
* **boolean isNull(Object obj)**The isNull() method returns true if the specified object is null. Otherwise, it returns false. You can also check whether an object is null using the comparison operator ==, for example, obj == null returns true of obj is null.
* **boolean nonNull(Object obj)**performs the check opposite of what the isNull() method does.
* **T requireNonNull(T obj)  
  T requireNonNull(T obj, String message)  
  T requireNonNull(T obj, Supplier messageSupplier)**  
  checks if the argument is not null. If the argument is null, it throws a NullPointerException. This method is designed for validating parameters of methods and constructors.   
  The second version can specify the message for the NullPointerException that is thrown when the argument is null.   
  The third version of the method takes a Supplier as the second argument.
* **String toString(Object o)  
  String toString(Object o, String nullDefault)**  
    
  The toString() method returns a "null" string if the argument is null. For a non-null argument, it returns the value returned by calling the toString() method on the argument.

**hash code**

The following code demonstrates how to use the method from the Objects class to calculate the hash code.

***import*** *java.util.Objects;*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

***// Compute hash code for two integers, a char, and a string***

***int*** *hash = Objects.hash(10, 800,* ***'\u20b9'****,* ***"Hello"****);*

*System.out.println(****"Hash Code is "*** *+ hash);*

*}*

*}*

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/H/HASH_CODE__0F29CB915CB6C470981A.PNG

**equals**

The following code shows how to use equals method from Objects class to compare two objects.

***import*** *java.util.Objects;*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

***// Test for equality***

***boolean*** *isEqual = Objects.equals(null, null);*

*System.out.println(****"null is equal to null: "*** *+ isEqual);*

*isEqual = Objects.equals(null,* ***"XYZ"****);*

*System.out.println(****"null is equal to XYZ: "*** *+ isEqual);*

*}*

*}*

The code above generates the following result.



**toString**

The following code shows how to use toString method from Objects to convert object to a String.

***import*** *java.util.Objects;*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

***// toString() method test***

*System.out.println(****"toString(null) is "*** *+ Objects.toString(null));*

*System.out.println(****"toString(null, \"XXX\") is "***

*+ Objects.toString(null,* ***"XXX"****));*

*}*

*}*

The code above generates the following result.



**requireNonNull**

The following code shows how to use requireNonNull from Objects class.

***import*** *java.time.Instant;*

***import*** *java.util.Objects;*

***import*** *java.util.function.Supplier;*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

***try*** *{*

*printName(****"A"****);*

*printName(null);*

*}* ***catch*** *(NullPointerException e) {*

*System.out.println(e.getMessage());*

*}*

***try*** *{*

*Supplier<String> messageSupplier = () ->* ***"Name is required. Error generated on "***

*+ Instant.now();*

*printNameWithSuplier(****"asdf"****, messageSupplier);*

*printNameWithSuplier(null, messageSupplier);*

*}* ***catch*** *(NullPointerException e) {*

*System.out.println(e.getMessage());*

*}*

*}*

***public******static******void*** *printName(String name) {*

*Objects.requireNonNull(name,* ***"Name is required."****);*

*System.out.println(****"Name is "*** *+ name);*

*}*

***public******static******void*** *printNameWithSuplier(String name,*

*Supplier<String> messageSupplier) {*

*Objects.requireNonNull(name, messageSupplier);*

*}*

*}*

The code above generates the following result.

