1. **Object class**
2. **Hash code.**
3. **Equals.**
4. **toString()**
5. **Cloning.**
6. **Object Finalization.**

**1.Object class:-** The Object is root (or) super class to every predefined and user defined class. Because of below 2 reasons Object class is developed as super class for every class.

Reason-1: In Java, Every object will have 11 common behaviours. These 11 behaviours must be implemented in every class with the same signature by every class developer. So to reduce burden for implementing these methods SUN developed a class called Object . These common behaviours were implemented as methods in “Object**”. If the logic is not satisfying to subclass requirement then subclass should override these methods.**

Reason-2: If we want define a method to receive and to return any type of object, and further to invoke and execute a from this passed object class, we must have one common super class to all classes. For this purpose Object class is given as super class to all classes.

Common Methods:-

**a.**public native int hashCode()

b.public boolean equals(Object obj)

c.public final native Class getClass()

d.public String toString()

e.protected native Object clone() throws CloneNotSupportedException

f.public void finalize() throws Throwable

g.public final void wait() throws InterruptedException

h.public final native void wait(long mills)throws InterruptedException

i.public final void wait(long mills,int nanosec)throws InterurptedException.

j.public final void notify()

k.public final void notifyAll();

Among the 11 methods, The subclass developer can override following methods only.

a.toString

b.hashCode.

c.clone

d.finalize

e.equals

**2.HashCode:-**  The hashcode is identity of object. Every object has its own unique hashcode. It is used to differentiate one object from another object and also used to differentiate one group of objects from other group of objects.

In Regular programming, The programmer do not use hash code of object. The Set and Map Collections uses Hash Code for storing, removing, and searching objects of class in hashtable data structure.

The hashCode method returns hashcode of Object.

Syntax:

Public native int hashCode();

2.**1.Generating Hash Code:-**The hashcode is generated in two ways

1. Using object Reference.
2. Using Object State.

**2.1.1Using Object Reference:-** It is default implementation of JVM and that hashcode number is returned through hashcode() method.

Example:-

import java.util.\*;

class Employee{

}

public class Sample

{

public static void main(String args[])

{

Employee e1=new Employee();

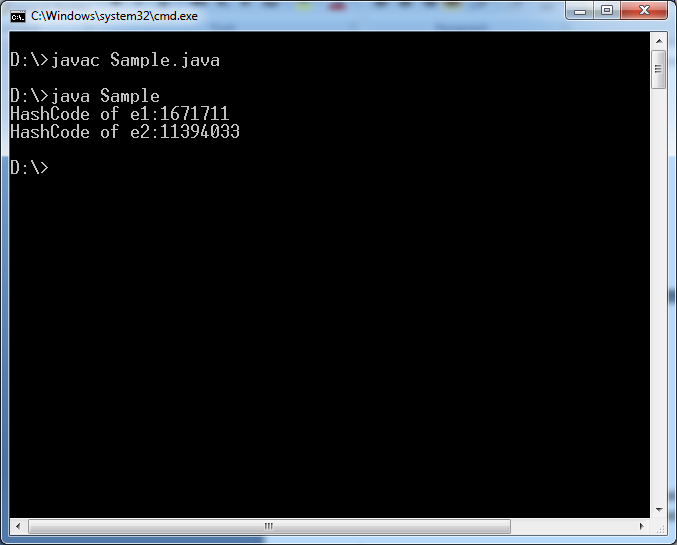
Employee e2=new Employee();

System.out.println("HashCode of e1:"+e1.hashCode());

System.out.println("HashCode of e2:"+e2.hashCode());

}

}



**2.1.2.Using Object State:-** The subclass developer override the hashcode() method in subclass. The overridden hashcode () makes hashcode using object current state .This method returns that.

Example:-

import java.util.\*;

class Employee{

int a;

int b;

Employee(int a,int b){this.a=a;this.b=b;

}

@Override

public int hashCode()

{

return a+b;

}

}

public class Sample

{

public static void main(String args[])

{

Employee e1=new Employee(10,20);

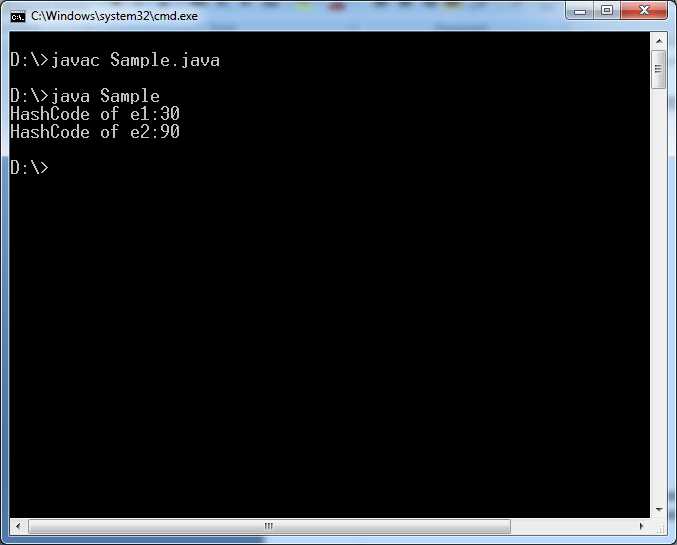
Employee e2=new Employee(40,50);

System.out.println("HashCode of e1:"+e1.hashCode());

System.out.println("HashCode of e2:"+e2.hashCode());

}

}



Even programmer override hashcode() method in subclass, The subclass object has JVM generated hashcode and user assigned hashcode.

Now , we can get JVM generated hash code in two ways:

1.By calling System.identityHashCode(object Reference) method.

2.By calling super.hashCode() method from a new method in subclass.

Example:-

import java.util.\*;

class Employee{

int a;

int b;

Employee(int a,int b){this.a=a;this.b=b;

}

@Override

public int hashCode(){

return a+b;

}

public int JVMHcode(){

return super.hashCode();

}}

public class Sample{

public static void main(String args[]) {

Employee e1=new Employee(10,20);

Employee e2=new Employee(40,50);

System.out.println("HashCode of e1 Based on state:"+e1.hashCode());

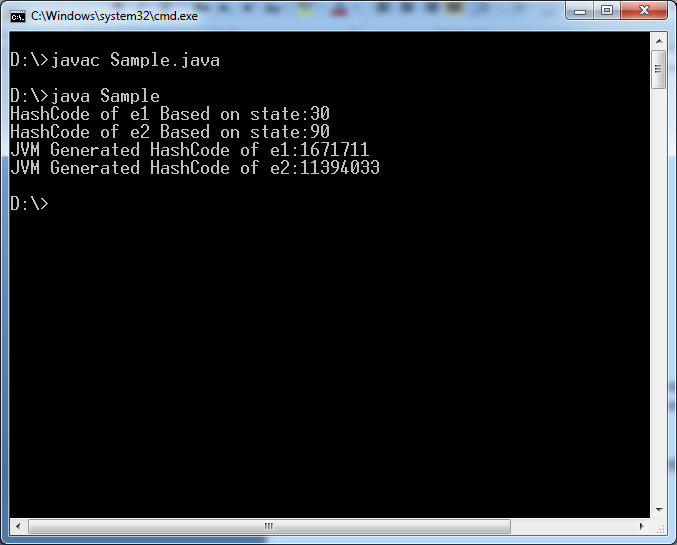
System.out.println("HashCode of e2 Based on state:"+e2.hashCode());

System.out.println("JVM Generated HashCode of e1:"+ System.identityHashCode(e1));

System.out.println("JVM Generated HashCode of e2:"+ e2.JVMHcode());

}

}



**3. equals():-** In java, we can compare objects in two ways either

1. Using objects Reference.

2. Using objects state.

Two objects of same class are said to be equal only if they have same reference or same state.

We compare objects using “==” operator or equals() method.

**3.1. using” ==” operator:- It always compare objects with their reference.**

Example:-

import java.util.\*;

class Employee{

int a;

int b;

Employee(int a,int b)

{

this.a=a;this.b=b;

}

}

public class Sample

{

public static void main(String args[])

{

Employee e1=new Employee(10,20);

Employee e2=new Employee(10,20);

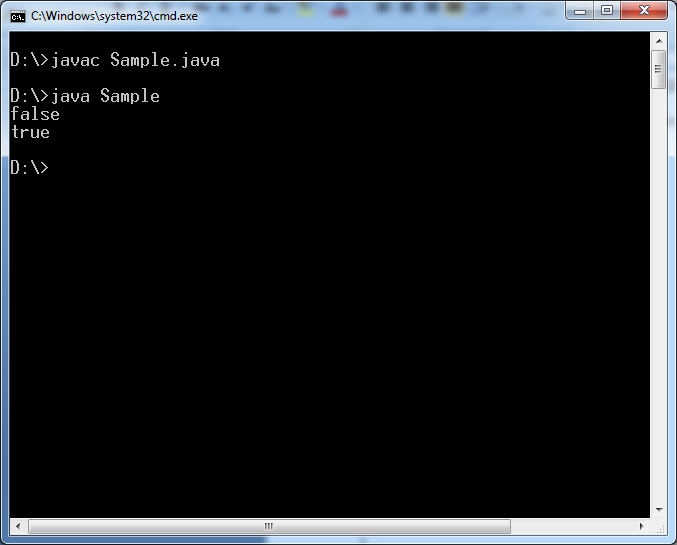
Employee e3=e2;

System.out.println(e1==e2);

System.out.println(e2==e3);

}

}



**3.2.using equals() method:-** generally people say,equals() method always compare two objects with their data, this statement is 100% rong. By default, equals() method will compare two objects with their reference.

Example:-

import java.util.\*;

class Employee{

int a;

int b;

Employee(int a,int b)

{

this.a=a;this.b=b;

}

}

public class Sample

{

public static void main(String args[])

{

Employee e1=new Employee(10,20);

Employee e2=new Employee(10,20);

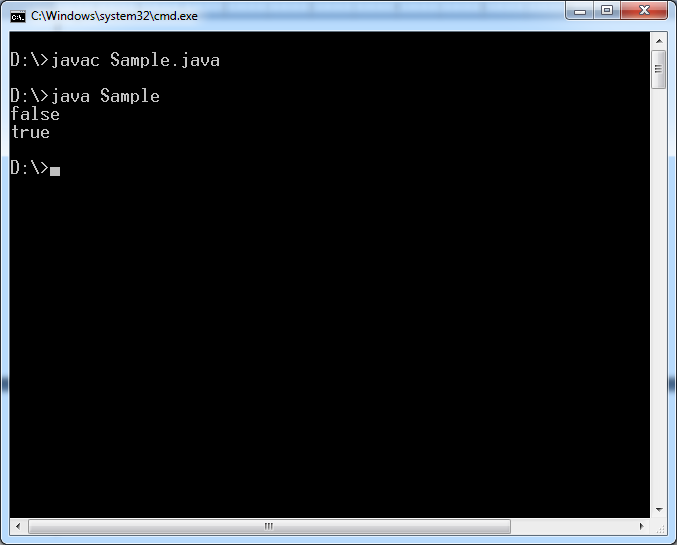
Employee e3=e2;

System.out.println(e1.equals(e2));

System.out.println(e2.equals(e3));

}

}



**If we want to compare two objects with state then we must over ride equals() method in sub class.**

**Example:**

**import java.util.\*;**

**import java.lang.Object;**

**class Employee{**

**int a;**

**int b;**

**Employee(int a,int b){**

**this.a=a;this.b=b;**

**}**

**public boolean equals(Object obj1){**

**Employee obj=(Employee)obj1;**

**if((this.a==obj.a) && (this.b==obj.b)) {**

**return true;**

**}**

**else {**

**return false;**

**}**

**}}**

**public class Sample{**

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

**Employee e1=new Employee(10,20);**

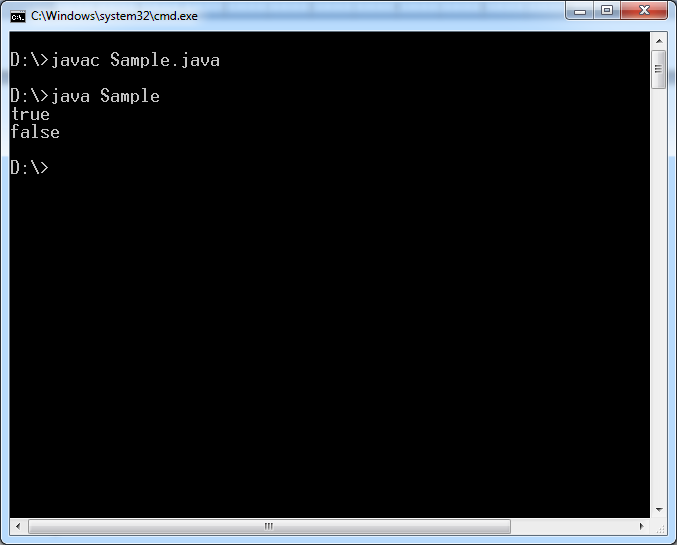
**Employee e2=new Employee(10,20);**

**Employee e3=new Employee(40,50);**

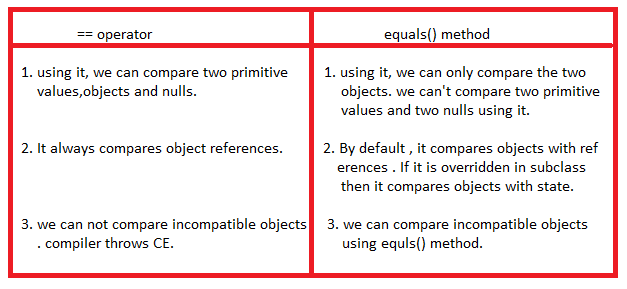
**System.out.println(e1.equals(e2));**

**System.out.println(e2.equals(e3));**

**}}**



**Diff between == operator and equals() method.**

****

**4.toString:-**

Syntax:

Public String toString();

This method returns class name and hashcode of current object in the format:

**classname@hashcode** in hex string format.

Example:-

import java.util.\*;

import java.lang.Object;

class Employee{

int a;

int b;

Employee(int a,int b){

this.a=a;this.b=b;

}}

public class Sample{

public static void main(String args[]) {

Employee e1=new Employee(10,20);

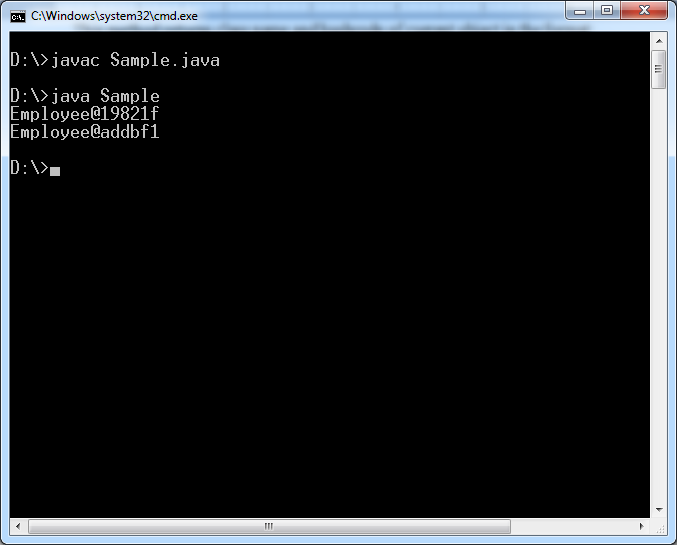
Employee e2=new Employee(10,20);

System.out.println(e1.toString());

System.out.println(e2.toString());

}

}

****

**We**  can override it in subclass. The method logic can also be written as per our requirements.

Example:-

import java.util.\*;

import java.lang.Object;

class Employee{

int a;

int b;

Employee(int a,int b)

{

this.a=a;this.b=b;

}

public String toString()

{

return "FirstNo:"+a+"\n"+"second No:"+b;

}

}

public class Sample

{

public static void main(String args[])

{

Employee e1=new Employee(10,20);

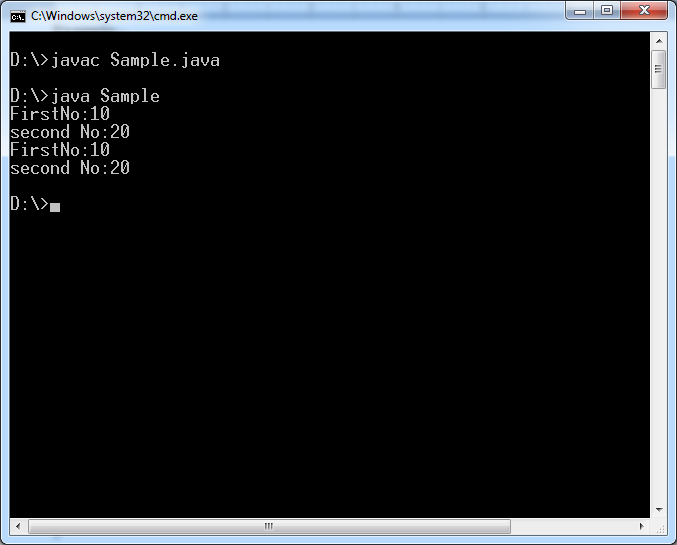
Employee e2=new Employee(10,20);

System.out.println(e1.toString());

System.out.println(e2.toString());

}

}

****

**5.cloning:-**

**Def:** cloning object means creating duplicate copy with current object state is called object cloning.

To perform cloning, we should call clone() method.

Syntax:

Protected native Object clone() throws CloneNotSupportedException.

Rule1:- The object which to be cloned. That object class must be subclass of “Clonable “ Interface. Otherwise “java.lang.CloneNotSupoortedException” is thrown.

The Clonable is a marker interface which is an empty interface. It provides permission to execute clone() method.

Rule2:- The method which invokes the clone() should handle the “java.lang.CloneNotSupportedException”.

Note:- the Original object and Cloned objects have different references and different hashcodes.

**5.1. Types Of Cloning:- java**  supports two types of cloning.

a. shallow cloning.

b. Deep Cloning.

**5.1.a. Shallow cloning:-** A shallow copy of object is a new object whose instance variables are identical to old object.

Shallow clone only copies the top level structure of object not the lower levels. So cloned objects referenced variables are still pointing to original objects.

Example:

import java.util.\*;

import java.lang.Object;

class Address{

String dno="13-8";

String city="nlr";

}

public class Sample implements Cloneable

{

byte eno=1;

String ename="suku";

Address adr=new Address();

public static void main(String args[])throws CloneNotSupportedException

{

Sample e1=new Sample();

Sample e2=(Sample)e1.clone();

System.out.println(e1.eno==e2.eno);

System.out.println(e1.adr==e2.adr);

}

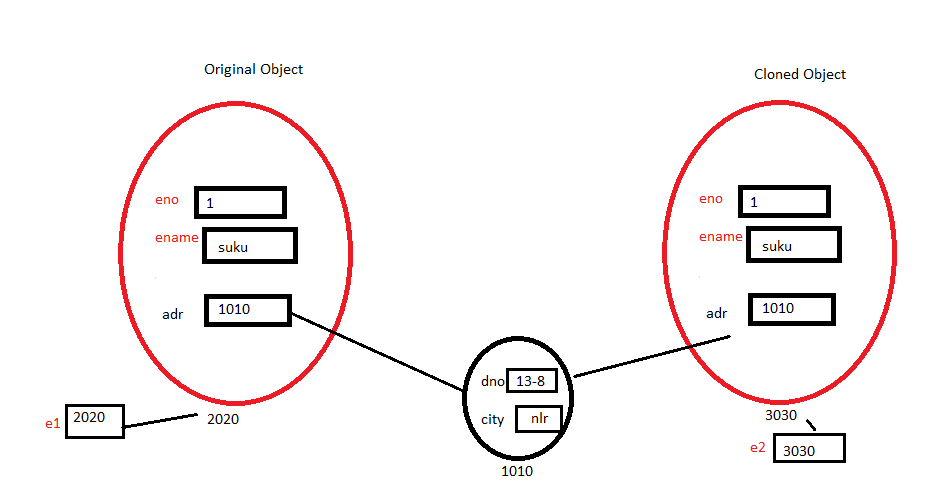
}

Output:

True

True





**Note:- By** **default clone () method creates shallow copy of object.**

5.1.b. **Deep cloning:-** In deep cloning , the object is copied along with its internal objects. Developer must over ride the clone() method in sub class for implementing the deep cloning.

Example:-

import java.util.\*;

import java.lang.Object;

class Address{

String dno="13-8";

String city="nlr";

}

public class Sample implements Cloneable

{

byte eno=1;

String ename="suku";

Address adr=new Address();

public Object clone()

{

Sample s1=new Sample();

s1.eno=this.eno;

s1.ename=this.ename;

s1.adr=new Address();

s1.adr.dno=this.adr.dno;

s1.adr.city=this.adr.city;

return (Object)s1;

}

public static void main(String args[])throws CloneNotSupportedException

{

Sample e1=new Sample();

Sample e2=(Sample)e1.clone();

System.out.println(e1.eno==e2.eno);

System.out.println(e1.adr==e2.adr);

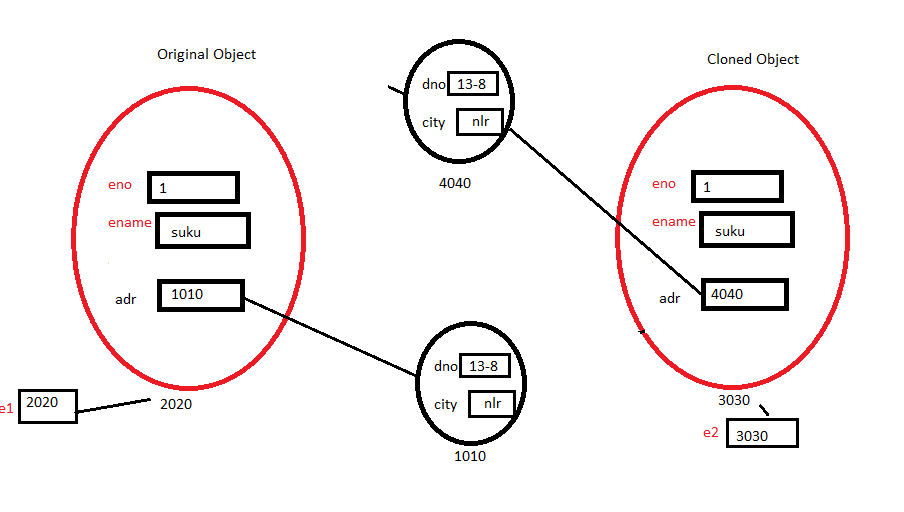
}

}

Output:

True

False



**5.2. Cloning object with IS-A Relation:** clone() method clone the object inheritance graph starts from root super class to current cloning object.

Example:-

import java.util.\*;

import java.lang.Object;

class Address{

String dno="13-8";

String city="nlr";

}

class Employee extends Address{

byte eno=1;

String ename="suku";

}

public class Sample extends Employee implements Cloneable

{

public static void main(String args[])throws CloneNotSupportedException

{

Sample e1=new Sample();

Sample e2=(Sample)e1.clone();

System.out.println(e2.eno);

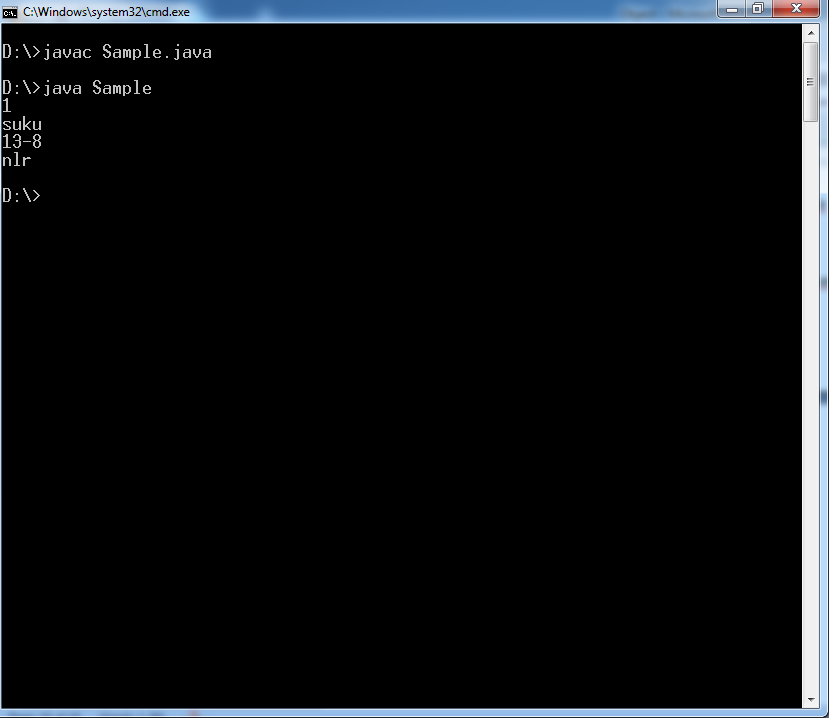
System.out.println(e2.ename);

System.out.println(e2.dno);

System.out.println(e2.city);

}

}



**6.Object Finalization:-** It is empty method.

Syntax:

Protected void finalize() throws Exception

The resource releasing logic will be changed from one resource to another resourse. That’why finalize() method was defined as empty method. So sub class developer should override finalize method in subclass with that class object’s resource releasing logic.

The finalized method is automatically called by garbage collector just before object is destroying. After finalize method execution is completed if still current object is unreferenced object, then garbage collector will destroy current object.

**Example:**

import java.util.\*;

import java.lang.Object;

class Address{

String dno="13-8";

String city="nlr";

}

class Employee extends Address{

byte eno=1;

String ename="suku";

protected void finalize()throws Exception{

System.out.println("finalized executed");

}}

public class Sample extends Employee implements Cloneable{

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

Sample e1=new Sample();

e1=null;

System.gc();

}}

