Object class methods
++++++++++++++++++++

```
public class Object {

  //constructor
  public Object();

  //Commonly used in every class
  public final native Class<?> getClass();
  public native int hashCode();
  public java.lang.String toString();
  public boolean equals(Object obj);

  //Cloning
  protected native Object clone() throws CloneNotSupportedException;


  //MultiThreading
  public final native void notify();
  public final native void notifyAll();
  public final void wait() throws InterruptedException;
  public final native void wait(long) throws InterruptedException;
  public final void wait(long, int) throws InterruptedException;


  //Garbage Collector
  protected void finalize() throws java.lang.Throwable;
}
```

equals() :: check name and id,if both are equal then return true otherwise return false

passing different type of objects, it would result in "ClassCastException" so return false.

passing null type, it would result in "NullPointerException" so return false.

if 2 references are pointing to same object, then without comparison it should return true.

Case Studies
++++++++++++

```
case1:
Student s1 = new Student(10,"sachin");
Student s4 = s1;

s1==s4        :: true
s1.equals(s4) :: true

case2:
String s1 = new String("sachin");
String s2 = new String("sachin");
System.out.println(s1==s2);  //false(reference comparison)
System.out.println(s1.equals(s2));//true(content comparison)

StringBuffer s1 = new StringBuffer("sachin");
StringBuffer s2 = new StringBuffer("sachin");
System.out.println(s1==s2);//false(reference comparison)
System.out.println(s1.equals(s2));//false(object class equals() is meant for
reference comparison)
```

Explain the realtionsihp b/w "==" vs equals()?
1. if r1==r2 is true, then r1.equals(r2) will always return true.
2. if r1==r2 is false,then r1.equals(r2) may return true or false(depends on the reference type)
3. if r1.equals(r2) is true,then r1==r2 may return true or false(depends on the reference type)
4. if r1.equals(r2) is false,then r1==r2 is always false.
5. if r1==null is always false, then r1.equals(null) will always be false.

Note:
 In case of == operator,we can apply this operator on primtive types and reference type also.
 In case of == operator if we apply on primitive type, it is meant for content comparison
 where as on reference type is meant for reference comparison.
 In case of == operator, when we apply on reference type the rule is there should be a realtionship b/w
 both the argument, otherwise it would result in "Compiletime Error", where as in case of equals() if there
 is no relationship, then we wont get compiletime error rather it would return false.

Tricky Code
++++++++++
```
String s1 = new String("sachin");
StringBuffer sb = new StringBuffer("sachin");
System.out.println(s1==sb);//CE
System.out.println(s1.equals(sb));//false
```

Explain the relationship b/w equals() and hashCode() method?
```
public class Object
{
      public boolean equals(){return (this==obj);}

      public native int hashCode();
}
```

1. if 2 objects are equal by equals() method then their hashCode must be same, it means r1.hashCode() == r2.hashCode() is true.
2. if 2 objects are not equal by equals() method, then their hashCode may or maynot be same so
    r1.hashCode() == r2.hashCode() can give true or false.
3. if hashcode of 2 objects are equal, then equals() may return true or false.
4. if hashcode of 2 objects are different then equals() will always return false.

Note: As a good programming practise if we are overriding equals(),compulsorily the hashCode also should be overriden.
      Viloation of above rule would not lead to any compiletime or runtime error,but it is not a good programming practise.

eg::
```
public int hashCode()
{
      return 100;  //not a good practise
}
```

```
public int hashCode()
{
      return age+height;//good if we use age and height in equals()
}

public int hashCode()
{
      return name.hashCode()+age;//very good,because we are use internall hashCode
so hashcode will be different
}


eg#1.
class Person
{
      String name;
      int age;

      Person(String name,int age)
      {
            this.name = name;
            this.age  = age;
      }

      @Override
      public int hashCode()
        {
            //implementation of hashCode: using name and age
            return name.hashCode() + age;
      }

    @Override
      public boolean equals(Object obj)
      {
            if (this == obj)
                  return true;

            else if (obj instanceof Person)
            {
                  //content comparison
                  Person p  =(Person) obj;
                  if (name.equals(p.name) && age == p.age)
                        return true;
                  else
                        return false;
            }

            return false;
      }
}

//Client Code
public class Test
{
   public static void main(String[] args)
      {
         Person p1 = new Person("sachin",51);
         Person p2 = new Person("sachin",51);
```

```
         Integer i = new Integer(100);

         System.out.println(p1.equals(p2));//true
         System.out.println(p1.equals(i));//false
      }
}
```

hashCode() => unique address generated for every object by JVM.
           => since we are talking about address, we should not touch the logic of hashCode.
           => if we are overriding the hashCode, then we need to see what properties are used in equals() method.

Note: In all String class, Collection class, Wrapper class equals() method is overriden for content comparison.

Cloning
+++++++
=> The process of creating a exactly duplicate object is called "Cloning".
=> The main objective of cloning is to mantain backup.
=> To perform clonning we use "clone()" from Object class.

```
protected native Object clone() throws CloneNotSupportedException;


//Client Code
public class Test  implements Cloneable
{
      int i =10;
      int j =20;

   Test()
   {
            System.out.println("Constructor got called");
   }
   public static void main(String[] args) throws CloneNotSupportedException
      {
         Test t1 =  new Test();
         Test t2 = (Test)t1.clone();

         t2.i = 1000;
         t2.j = 2000;

         System.out.println(t1.hashCode() == t2.hashCode());
         System.out.println("T1 Object i and j value ===>"+t1.i + " " +t1.j);//10
20
         System.out.println("T2 Object i and j value ===>"+t2.i + " " +t2.j);//1000
2000

      }
}
```

=> we can perform cloning only on "Cloneable" objects.
=> An object is said to Cloneale,if and only if the corresponding class has implemented "Cloneable" interface.
=> Cloneable interface is present in java.lang package and it doesn't contain any

```
abstract method so we say that
   interface as "marker interface".
=> if we try to perform cloning on Non-Cloneable objects, then at runtime jvm will
throw an exception called as
   "CloneNotSupportedExeption".

Output
Constructor got called
false
T1 Object i and j value ===>10 20
T2 Object i and j value ===>1000 2000

In java we have 2 types of Cloning
 a. Shallow Cloning
      => If the main object contains reference variable, then corresponding object
won't be created rather
           reference will be shared to clone object, this type of clonning is
called "Shallow Cloning".
      => Using the main object reference, if we perform any changes to the
contianed object then those changes would
         be automatically reflected to main object also.
      => main object :: Dog, contained object : Cat
      => Shallow cloning is done by Object class clone() as default.
      => Shallow Cloning best suited only when the object have primitive variable
types.

eg#1.
class Cat
{
      int j;

      Cat(int j){
            this.j = j;
      }
}
class Dog implements Cloneable
{
      int i;

      //HAS-A relationship
      Cat c;

      Dog(int i,Cat c){
            this.i = i;
            this.c = c;
      }

      @Override
      public  Object clone() throws CloneNotSupportedException
      {
            //Object class clone() is getting called :: Shallow Cloning
            return super.clone();
      }
}

//Client Code
public class Test
{
   public static void main(String[] args) throws CloneNotSupportedException
```

```
      {
         Cat c = new Cat(10);
         Dog d = new Dog(20,c);

         Dog d1 = (Dog) d.clone();

         //Changes are made using Cloned Copy
         d1.i = 9999;
         d1.c.j = 8888;

         System.out.println("D Object data is :: "+d.i+" "+d.c.j);
         System.out.println("D1 Object data is :: "+d1.i+" "+d1.c.j);

      }
}

Output
D Object data is :: 20 8888
D1 Object data is :: 9999 8888
```

b. Deep Cloning

If the main object contains reference variable, then corresponding object copy also will be created during clonning, such type of cloning is refered as "Deep Cloning".

Object class clone() is meant for "ShallowCloning",if we want deep cloning then we need to go for "DeepCloning".

eg#1.

```
class Cat
{
      int j;

      Cat(int j)
      {
            this.j = j;
      }
}
class Dog implements Cloneable
{
      int i;

      //HAS-A relationship
      Cat c;

      Dog(int i,Cat c){
            this.i = i;
            this.c = c;
      }

      @Override
      public  Object clone() throws CloneNotSupportedException
      {
            //Perform Deep Clonning
            Cat c1 = new Cat(c.j);
            Dog d1 = new Dog(i,c1);
            return d1;
      }
}
```

```
//Client Code
public class Test
{
   public static void main(String[] args) throws CloneNotSupportedException
      {
         Cat c = new Cat(10);
         Dog d = new Dog(20,c);

         Dog d1 = (Dog) d.clone();

         //Changes are made using Cloned Copy
         d1.i = 9999;
         d1.c.j = 8888;

         System.out.println("D Object data is :: "+d.i+" "+d.c.j);
         System.out.println("D1 Object data is :: "+d1.i+" "+d1.c.j);

      }
}
Output
D Object data is :: 20 10
D1 Object data is :: 9999 8888

Note:
 Object class clone()
 a. primitive variable in object :: deep cloning
 b. reference varaible in object :: Shallow cloning
 c. If our object contains reference varaible and if we want deep cloning to happen
then we need to override clone().
```