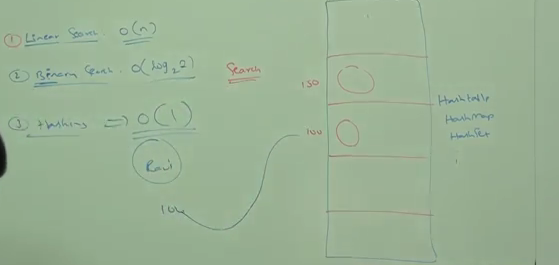
**Hashcode:**

Hashcode and address of object are not same. Both are different. It is not possible to find the address of an object. They are no way related.

**Definition**:

1. For every object, an unique number is created by JVM which is nothing but hashcode.
2. Hashcode wont represent address of object
3. JVM will use hashcode while saving the objects into hashing related data structures like Hashtable, Hashmap, Hashset, etc.,
4. The main advantage of saving objects based on hashcode is search operation will become easy (The most powerful search algorithm upto today is hashing).

**Time Complexity:**



**Method Declaration:**

* Public native int hashcode() -> contains in Object class

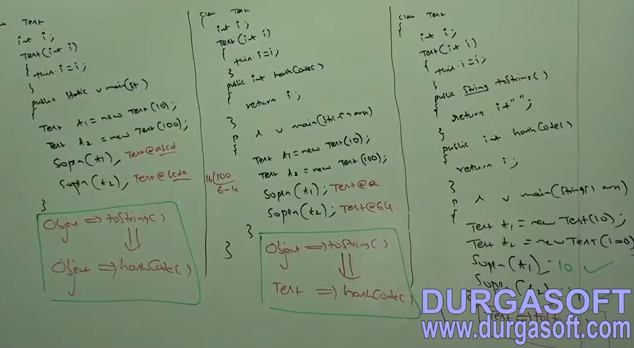
If we are giving the chance to Object class hashcode(), It will generate hashcode based on address of the object. It doesn’t mean hashcode represent address of the object.

Based on our requirement, we can override hashcode() in our class to generate our own hashcode which should be unique for every object.

**toString() VS hashCode():**

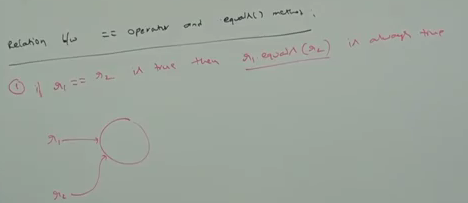
If we are giving the chance to Object class toString(), it will internally calls hashCode().

If we are overriding toString(), then our toString() may not call hashCode().

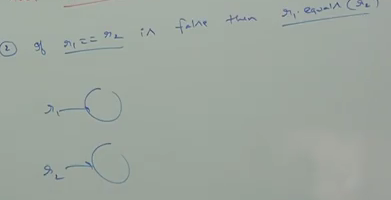


**Relation b/w == operator and equals():**

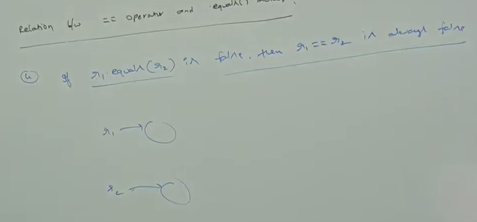
1. *If two objects are equal by == operator then this objects are always equal by equals()*
   1. If r1==r2 is true, then r1.equals(r2) is always true



1. If two objects are not equals by ==operator then we cannot conclude anything about equals(). It may returns true or false
   1. If r1==r2 is false, then r1.equals(r2) may returns true or false. We cannot expect exactly.



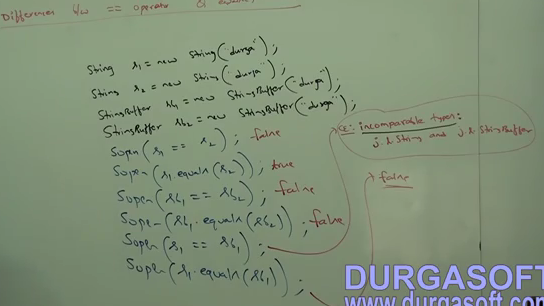
1. If two objects are equal by equals(), then we cannot conclude anything about ==operator. It may returns true or false.
   1. If r1.equals(r2) is true, then we cannot conclude anything about r1==r2. It may returns true or false
2. *If two objects are not equal by equals(), then this objects are always not equal ==operator.*
   1. If r1.equals(r2) is false, then r1==r2 is always false.



**Differences b/w == operator and equals():**

1. To use ==operator, compulsory there should be some relation b/w argument types (Either child to parent or Parent to child or same type). Otherwise we will CE: Incomparable types.
2. If there is no relation b/w argument types, then equals() will not rise any exception. Simple it return false.

Example:



|  |  |
| --- | --- |
| **== operator** | **Equals()** |
| It is a operator in java applicable for both primitive and object types | It is a method applicable only for object types but not for primitive types. |
| In the case of object references, == operator is meant for reference(address) comparison | By default equals() present in object class also meant for reference comparision. |
| We cannot override == operator for content comparision. | We can override equals() for content comparision |
| Compulsory there should be some relation b/w argument types (Either child to parent or parent to child or same type). Otherwise we will get compile time error saying incomparable types. | If there is no relation b/w argument types, then equals() wont rise any exception. It simply return false. |

In simple word, == operator is meant for reference comparision and equals() is meant for content comparision.

Note: if r is any object references, then r==null and r.equals(null) always return false.

Example:

Thread r=new Thread();

Sop(r==null) -> false

Sop(r.equals(null)) - >false

**Hashing related data structure follow the following fundamental rule.**

1. Two equivalent should be placed in same bucket but all the objects present in same bucket need not be equal.
2. Two equivalent should not be placed in different bucket otherwise search operation fails.
3. If Two equivalent should be placed in same bucket, then compulsory hashCode() of two objects should be equal.

**Contract b/w equals and hashCode:**

1. *If two objects are equal by equals(), then their hashcode must be equal.* 
   1. If r1.equals(r2) is true, then r.hashCode()==r2.hashCode()

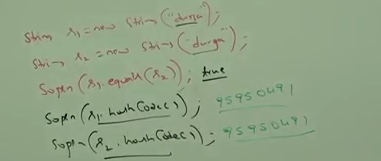
Object class equals() and hashCode() follows above contract. Hence whenever overriding equals(), compulsory we should override hashCode() to satisfy above contract.

1. If two objects are not equal by equals(), then there is no restriction on hashcodes, may be equal or may not be equals
2. If hashcodes of two objects are equal, then we cannot conclude anything about equals(). It may return true or false.
3. *If hashcodes of two objects are not equal, then this objects are always not equal by equals().*

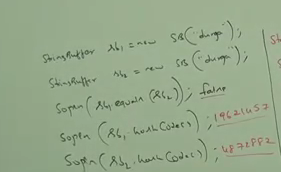
**Note: To satisfy contract b/w equals() and hashCode(), whenever we are overriding equals() compulsory we have to override hashCode(). Otherwise we wont get any exception, but it is not good programming practice.**

**Example:**

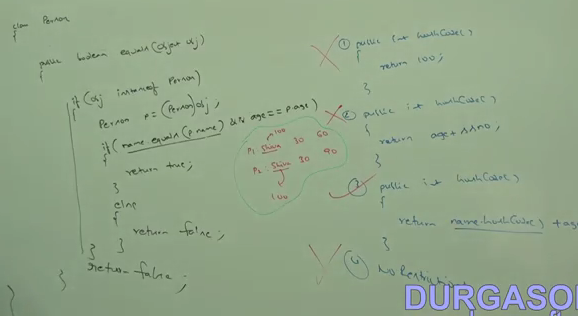
In String class equals() is overridden for content comparision and hence hashCode() is also overridden to generate hashCode() based on content.



In StringBuffer class equals() is not overridden for content comparision and hence hashCode() is not overridden.



**Ex: Consider the following below person class:**

****

**In all Collection classes, In all Wrapper classes and in String class equals() is overridden for content comparision. Hence it is highly recommended to override equals() in our class for content comparision.**

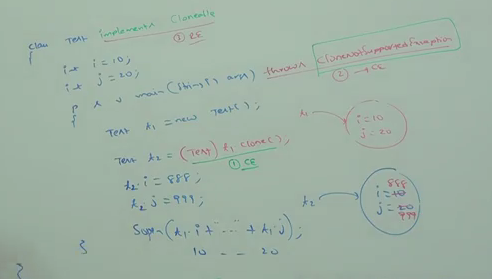
**Clone:**

1. It is present in Object class
2. The process of creating exactly duplicate object is called cloning
3. Purpose of cloning is to maintain backup copy and to preserve the state of an object.
4. We can perform cloning by clone() of Object class.
5. We can perform cloning on cloneable objects. If an object is said to be cloneable, if and only the corresponding class should implements cloneable interface which is present in java.lang. It is a marker interface.
6. If we are trying to perform cloning on non cloneable objects then we will get CloneNotSupportedException

**Method:**

Protected native Object clone() throws CloneNotSupportedException

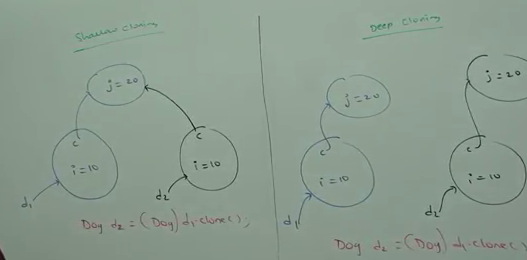
**Example:**

****

**Two types of cloning:**

1. Shallow cloning
2. Deep cloning

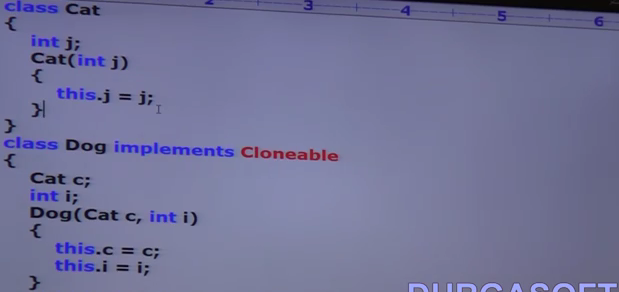
**Shallow Vs Deep Cloning**

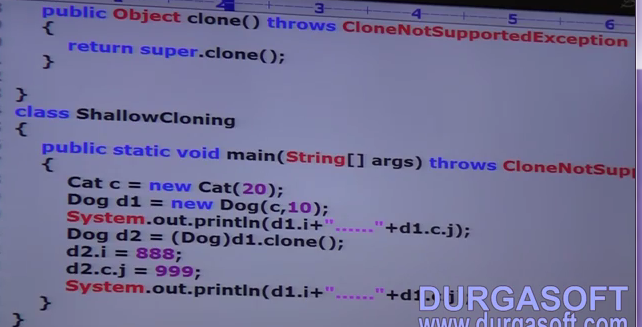
****

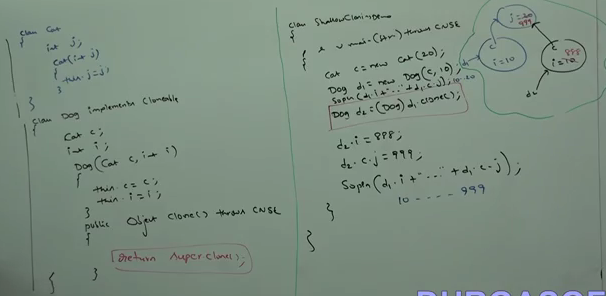
**Shallow:**

1. The process of creating bitwise copy of an object is called Shallow Cloning. Bit wise copy means copying primitive variables.
2. If the main object contains primitive variables, then exactly duplicate copies will be created in the cloned object.
3. If the main object contains any reference variable, then corresponding object wont be created, just duplicate reference variable will be created pointing to old contained object.
4. Clone() of Object class is meant for shallow cloning.

**Example:**

****

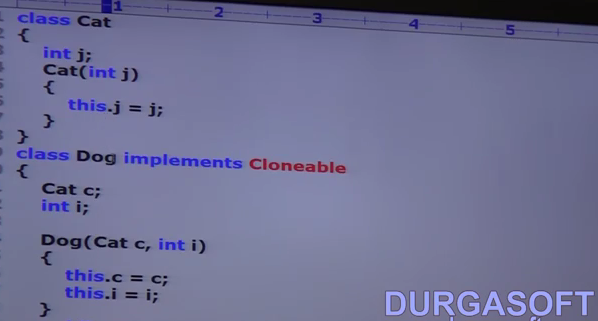
****

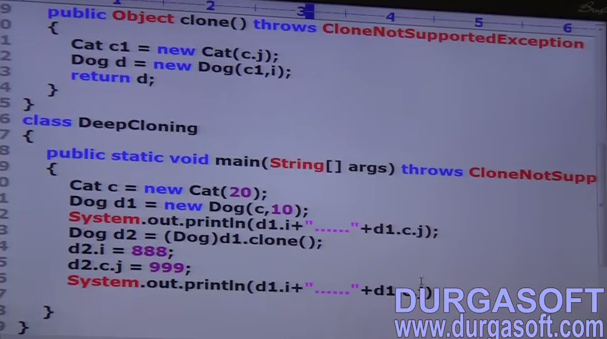
****

**Deep:**

1. The process of creating exactly independent duplicate object including contained object is called deep cloning
2. If the main object contain any primitive variables, then in the cloned object duplicate copies will be created.
3. If the main object contain any reference variables, then the corresponding contained object will also be created in the cloned copy.
4. By default Object class clone() meant for shallow cloning. But we can implement deep cloning explicitly by overriding clone() of Object class in our class.

**Example:**

****

****