***Agenda***

**1. Introduction**

**2. Java.lang.Object class**

 **toString( ) method**

 **hashCode() method**

 **toString() method vs hashCode() method**

 **equals() method**

 **Simplified version of .equals() method**

 **More simplified version of .equals() method**

 **Relationship between .equals() method and ==(double equal operator)**

 **Differences between == (double equal operator) and .equals() method?**

 **Contract between .equals() method and hashCode() method**

 **Clone () method**

 **Shallow cloning**

 **Deep Cloning**

 **getClass() method**

 **finalize( )**

 **wait( ) , notify( ) , notifyAll( )**

**3. java.lang.String class**

 **Importance of String constant pool (SCP)**

 **Interning of String objects**

 **String class constructors**

 **Important methods of String class**

 **Creation of our own immutable class**

 **Final vs immutability**

**4. StringBuffer**

 **Constructors**

 **Important methods of StringBuffer**

**5. StringBuilder (1.5v)**

 **StringBuffer Vs StringBuilder**

 **String vs StringBuffer vs StringBuilder**

 **Method chaining**

**6. Wrapper classes**

 **Constructors**

 **Wrapper class Constructor summery**

**7. Utility methods**

 **valueOf() method**

 **xxxValue() method**

 **parseXxx() method**

 **toString() method**

**8. Dancing between String, wrapper object and primitive**

**9. Partial Hierarchy of java.lang package**

 **Void**

**10. Autoboxing and Autounboxing**

 **Autoboxing**

 **Autounboxing**

 **Conclusions**

**11. Overloading with respect to widening, Autoboxing and var-arg methods**

 **Case 1: Widening vs Autoboxing**

 **Case 2: Widening vs var-arg method**

**==================================================================================================================================**

**The following are some of important classes present in java.lang package.**

**1. Object class**

**2. String class**

**3. StringBuffer class**

**4. StringBuilder class (1.5 v)**

**5. Wrapper Classes**

**6. Autoboxing and Autounboxing(1.5 v)**

**class:**

**1. For any java object whether it is predefine or customized the most commonly**

**required methods are encapsulated into a separate class which is nothing but**

**object class.**

**2. As object class acts as a root (or) parent (or) super for all java classes, by default**

**its methods are available to every java class.**

**3. Note : If our class doesn't extends any other class then it is the direct child class**

**of object**

**If our class extends any other class then it is the indirect child class of Object.**

**The following is the list of all methods present in java.lang Object class :**

**1. public String toString();**

**2. public native int hashCode();**

**3. public boolean equals(Object o);**

**4. protected native Object clone()throws CloneNotSupportedException;**

**5. public final Class getClass();**

**6. protected void finalize()throws Throwable;**

**7. public final void wait() throws InterruptedException;**

**8. public final native void wait()throws InterruptedException;**

**9. public final void wait(long ms,int ns)throws InterruptedException;**

**10. public final native void notify();**

**11. public final native void notifyAll();**

**toString( ) method :**

**1. We can use this method to get string representation of an object.**

**2. Whenever we are try to print any object reference internally toString() method**

**will be executed.**

**3. If our class doesn't contain toString() method then Object class toString()**

**method will be executed.**

**To provide our own String representation we have to override toString() method**

**in our class.**

**Ex : For example whenever we are try to print student reference to print his a**

**name and roll no we have to override toString() method as follows.**

**In String class, StringBuffer, StringBuilder, wrapper classes and in all collection**

**classes toString() method is overridden for meaningful string representation.**

**Hence in our classes also highly recommended to override toString() method.**

**hashCode() method :**

**1. For every object jvm will generate a unique number which is nothing but**

**hashCode.**

**2. Jvm will using hashCode while saving objects into hashing related data**

**structures like HashSet, HashMap, and Hashtable etc.**

**3. If the objects are stored according to hashCode searching will become very**

**efficient (The most powerful search algorithm is hashing which will work based**

**on hashCode).**

**4. If we didn't override hashCode() method then Object class hashCode() method**

**will be executed which generates hashCode based on address of the object but it**

**doesn't mean hashCode represents address of the object.**

**5. Based on our programming requirement we can override hashCode() method to**

**generate our own hashcode.**

**6. Overriding hashCode() method is said to be proper if and only if for every object**

**we have to generate a unique number as hashcode for every object.**

**Note :**

**1. if we are giving opportunity to Object class toString() method it internally calls**

**hashCode() method. But if we are overriding toString() method it may not call**

**hashCode() method.**

**2. We can use toString() method while printing object references and we can use**

**hashCode() method while saving objects into HashSet or Hashtable or HashMap.**

**equals() method:**

**1. We can use this method to check equivalence of two objects.**

**2. If our class doesn't contain .equals() method then object class .equals() method**

**will be executed which is always meant for reference comparison[address**

**comparison]. i.e., if two references pointing to the same object then only .equals(**

**) method returns true .**

 **In the above program Object class .equals() method got executed which is always**

**meant for reference comparison that is if two references pointing to the same**

**object then only .equals(() method returns true.**

**In object class .equals() method is implemented as follows which is meant for**

**reference comparison.**



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

 **return (this == obj);**

 **}**

 **Based on our programming requirement we can override .equals() method for**

**content comparison purpose.**

**When ever we are overriding .equals() method we have to consider the following things :**

**1. Meaning of content comparison i.e., whether we have to check the names**

**are equal (or) roll numbers (or) both are equal.**

**2. If we are passing different type of objects (heterogeneous object) our**

**.equals() method should return false but not ClassCastException i.e., we**

**have to handle ClassCastException to return false.**

**3. If we are passing null argument our .equals() method should return false**

**but not NullPointerException i.e., we have to handle NullPointerException to return false.**

**4. The following is the proper way of overriding .equals() method for**

**content comparison in Student class.**

**In StringBuffer class .equals( ) is not overriden for content comparision hence**

**Object class .equals( ) will be executed which is meant for reference comparision , hence if objects are different .equals( ) method returns false , even though content is same.**

**Relationship between .equals() method and ==(double equal operator) :**

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

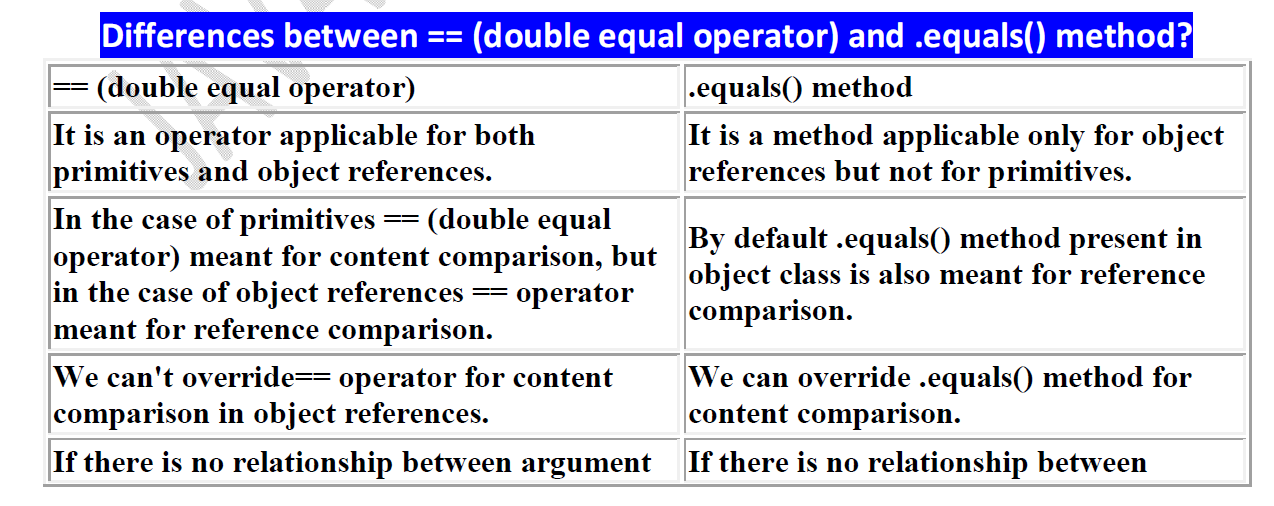
**== operator then these objects are always equal by .equals( ) method also.**

**2. If r1==r2 is false then we can't conclude anything about r1.equals(r2) it may return true (or) false.**

**3. If r1.equals(r2) is true then we can't conclude anything about r1==r2 it may**

**returns true (or) false.**

**4. If r1.equals(r2) is false then r1==r2 is always false.**



**Clone () method:**

1. **The process of creating exactly duplicate object is called cloning.**

**2. The main objective of cloning is to maintain backup purposes.(i.e., if something**

**goes wrong we can recover the situation by using backup copy.)**

1. **We can perform cloning by using clone() method of Object class.**

**protected native object clone() throws CloneNotSupportedException;**

**We can perform cloning only for Cloneable objects.**

 **An object is said to be Cloneable if and only if the corresponding class**

**implements Cloneable interface.**

 **Cloneable interface present in java.lang package and does not contain any**

**methods. It is a marker interface where the required ability will be provided**

**automatically by the JVM.**

 **If we are trying to perform cloning or non-clonable objects then we will get**

**RuntimeException saying CloneNotSupportedException.**

**Shallow cloning vs Deep cloning :**

**Shallow cloning:**

1. **The process of creating bitwise copy of an object is called Shallow Cloning .**

**2. If the main object contain any primitive variables then exactly duplicate copies**

**will be created in cloned object.**

**3. If the main object contain any reference variable then the corresponding object**

**won't be created just reference variable will be created by pointing to old**

**contained object.**

**4. By using main object reference if we perform any change to the contained object**

**then those changes will be reflected automatically to the cloned object , by**

**default Object class clone( ) meant for Shallow Cloning**

**Which cloning is best ?**

**If the Object contain only primitive variable then Shallow Cloning is the best choice ,**

**If the Object contain reference variables then Deep cloning is the best choice.**

**Deep Cloning :**

**1. The process of creating exactly independent duplicate object(including contained**

**objects also) is called deep cloning.**

**2. In Deep cloning , if main object contain any reference variable then the**

**corresponding Object copy will also be created in cloned object.**

**3. Object class clone( ) method meant for Shallow Cloning , if we want Deep**

**cloning then the programmer is responsible to implement by overriding clone( )**

**method.**

**4. In Deep cloning by using main Object reference if we perform any change to the**

**contained Object those changes won't be reflected to the cloned object.**

**Shallow cloning:**

1. **The process of creating bitwise copy of an object is called Shallow Cloning .**

**2. If the main object contain any primitive variables then exactly duplicate copies**

**will be created in cloned object.**

**3. If the main object contain any reference variable then the corresponding object**

**won't be created just reference variable will be created by pointing to old**

**contained object.**

**4. By using main object reference if we perform any change to the contained object**

**then those changes will be reflected automatically to the cloned object , by**

**default Object class clone( ) meant for Shallow Cloning**

**37. Shallow cloning is the best choice , if the Object contains only primitive values.**

**38. In Shallow cloning by using main object reference , if we perform any change to**

**the contained object then those changes will be reflected automatically in cloned**

**copy.**

**39. To overcome this problem we should go for Deep cloning.**

**finalize( ) :**

**Just before destroying an object GC calls finalize( ) method to perform CleanUp**

**activities .**

**wait( ) , notify( ) , notifyAll( )::**

**We can use these methods for inter thread communication**

**Note :**

**1. Object creation in SCP is always optional 1st JVM will check is any object**

**already created with required content or not. If it is already available then it will**

**reuse existing object instead of creating new object. If it is not already there then**

**only a new object will be created. Hence there is no chance of existing 2 objects**

**with same content on SCP that is duplicate objects are not allowed in SCP.**

**2. Garbage collector can't access SCP area hence even though object doesn't have**

**any reference still that object is not eligible for GC if it is present in SCP.**

1. **All SCP objects will be destroyed at the time of JVM shutdown automatically.**

**Note :**

**When ever we are using new operator compulsory a new object will be created on the**

**Heap . There may be a chance of existing two objects with same content on the heap but**

**there is no chance of existing two objects with same content on SCP . i.e., duplicate**

**objects possible in the heap but not in SCP .**

**Creation of our own immutable class:**

**Once we created an object we can't perform any changes in the existing object.**

**If we are trying to perform any changes with those changes a new object will be created.**

**If there is no change in the content then existing object will be reused. This behavior is**

**called immutability.**

**final class CreateImmutable {**

**private int i;**

**CreateImmutable(int i) {**

**this.i=i;**

**}**

**public CreateImmutable modify(int i) {**

**if(this.i==i)**

**return this;**

**else**

**return (new CreateImmutable(i));**

**}**

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

**CreateImmutable c1=new CreateImmutable(10);**

**CreateImmutable c2=c1.modify(100);**

**CreateImmutable c3=c1.modify(10);**

**System.out.println(c1==c2);//false**

**System.out.println(c1==c3);//true**

**CreateImmutable c4=c1.modify(100);**

**System.out.println(c2==c4);//false**

**}**

**}**

**Final vs immutability :**

**1. final modifier applicable for variables where as immutability concept applicable**

**for objects**

**2. If reference variable declared as final then we can't perform reassignment for**

**the reference variable it doesn't mean we can't perform any change in that**

**object.**

**3. That is by declaring a reference variable as final we won't get any immutability**

**nature .**

**4. final and immutability both are different concepts .**

**Every method present in StringBuffer is syncronized hence at a time only one thread is**

**allowed to operate on StringBuffer object , it increases waiting time of the threads and**

**creates performence problems , to overcome this problem we should go for**

**StringBuilder.**

**StringBuilder (1.5v)**

**1. Every method present in StringBuffer is declared as synchronized hence at a**

**time only one thread is allowed to operate on the StringBuffer object due to this,**

**waiting time of the threads will be increased and effects performance of the**

**system.**

**2. To overcome this problem sun people introduced StringBuilder concept in 1.5v.**

