Java.lang.Package

**1. Introduction**

**2. Object Class**

**3. String class**

**4. String Buffer**

**5. String builder**

**6. Wrapper Class**

**7. AutoBoxing and AutoUnBoxing**

For writing any java program, the mostly commonly required classes and interfaces are grouped into this package.

Class Test {

public static void main(String[] args) {

sop(“Hello world”)

}

}

**Object:**

Root Java class

Why? The object class contains the most common methods which are required for every Java class.

**Class A extends B -> mean A is the child is B and B is the child of Object. This is called multi-level inheritance.**

**If our class does not extends any other class, our class is direct child class of object.**

**Either directly or indirectly java wont provide support multiple heritance.**

Objec class defines the following eleven methods.

Public String toString()

public native int hashCode()

public boolean equals(Object o)

protected native Object clone() throws CNSE

protected void finalize() throws Throwable

public final Class getClass()

public final void wait() throws IE

public final native void wait(long ms) throws IE

public final native void wait(long ms, int ns) throws IE

public native final void notify()

public native final void notifyAll()

Also there is a method class registerNative() method is also there.

Strictly speaking object class contains 12methods that extra method is private static native void **registerNatives();**

This method internally required for object class and not available to the child class. Hence we are not required to consider this method.

**toString():** We can use toString() method to get String representation of an object.

String s = obj.toString();

Whenever we are trying to print object reference, internally toString method will be called.

**hashCode()**

For every object a unique number generated by JVM is **hashCode.** HashCode wont represent address of object.

JVM will use hashCode, while saving object into hashing related DS like HashTable, HashMap, HashSet etc. The advantage of saving objects based on hashCode is search operation will become easy.

Linear search -> O(n)

Binary search -> O(log 2 n)

Hash search -> O(1)

**Overriding hashCode method**  is set to be proper. If and only if for every object we have to generate a unique number as hashCode.

**Equals Method:**

If our class doesn’t contain equals method, object class equals method will be executed.

We can override equals method for content comparision.

**In StringBuffer .equals method is not overriden for content comparison. Hence if object different .equal method return false, even though content is same.**

**GetClass method:**

To return **run type class definition** of an object.

Public final Class getClass();

By using this, we can access class level properties, like full qualified name of the class, methods information, constructor information etc.

GetDeclaredMethods()

**Finalize method:**

This is related to **Garbage Collection**

Justbefore destroying an object GC will call finalize method, to perform clean up activities, once finalize method complete automatically GC collector destroys that object.

**Wait(), notify(), notifyAll()**

These method can used for inter thread communication.

The thread which is expecting updation, it is responsible to call wait method. Then immediately the thread will entered into waiting state.

The thread which is responsible to perform updation, after performing updation, the thread can call notify method. The waiting thread will get the notification and continue its execution with those updates.

**String**

**Case 1**

**String s = new String(“suganthna”); | StringBuffer sb = new SB(“suganthan”)**

**s.concat(“madhavan”) | sb.append(“madhavan”)**

**sop(s) //suganthan | sop(sb) //suganthan madhavan**

Once we create a string object we can’t perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created. This non-changeable behavior is nothing but, immutability of String.

Above “s” pointing to suganthan, and we are trying to add something, then new object got created. But these new object doesn’t have any reference variable. So it is by default eligible for GC.

Once we create **StringBuffer** object, we can perform any change in the existing object, this changable behavior is nothing but mutability of StringBuffer object.

Above “sb” point to suganthan, after modification it is reflecting in the object.

**Case 2:**

**String** class is overriding .equal method is overriden, but not in **StringBuffer** is not.

**Case 3:**

String s = new String(“suganthan”) --> Two object will be created one at heap and another at String constanct pool and s is always pointing to heap object.

String s = “suganthan” --> One object will be created at String constant pool and s is pointing to SCP

Object creation at SCP is always optional. If the string is already there no object will be created. Existing object will be re-used.

GC is not allowed to access SCP area. Hence even though object does not contain reference variable hence it is not eligible for GC if it present in SCP.

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

String s1 = new String(“Suganthan”)

String s2 = new String(“Madhavan”)

String s3 = “Suganthan”

String s4 = “Suganthan”

In total 3 object will be created, 2 in heap and one in SCP. Duplicate object is possible in Heap but not in SCP

String s1 = new String(“suganthan”)

s1.concat(“software”)

String s2 = s1.concat(“engineer”)

s1 = s1.concat(“blore”)

|  |  |
| --- | --- |
| Heap | SCP |
| Suganthan | suganthan |
| suganthansoftware | Software |
| suganthanengineer | Engineer |
| suganthanblore | Blore |

Total 8 objects will be created. For concat(**runtime**) operation object will be created only in heap. **For every string literal, one object will be created at SCP.**

**Constructors of String class**

String s = new String()

String s = new String(String literal)

String s = new String(StringBuffer sb)

String s = new String(char[] ch)

char[] ch = {‘a’,’b’,’c’,’d’}

String s = new String(ch)//abcd

String s = new String(byte[] byte);

byte[] byte = {101,102,103,104}

String s = new String(byte)//defg

**Important methods of String class**

charAt(int index)

concat(String s)

replace

substring

trim

length

indexOf

lastIndexOf

**Note**

String s1 = “suganthan”

String s2 = s1.toUpperCase()

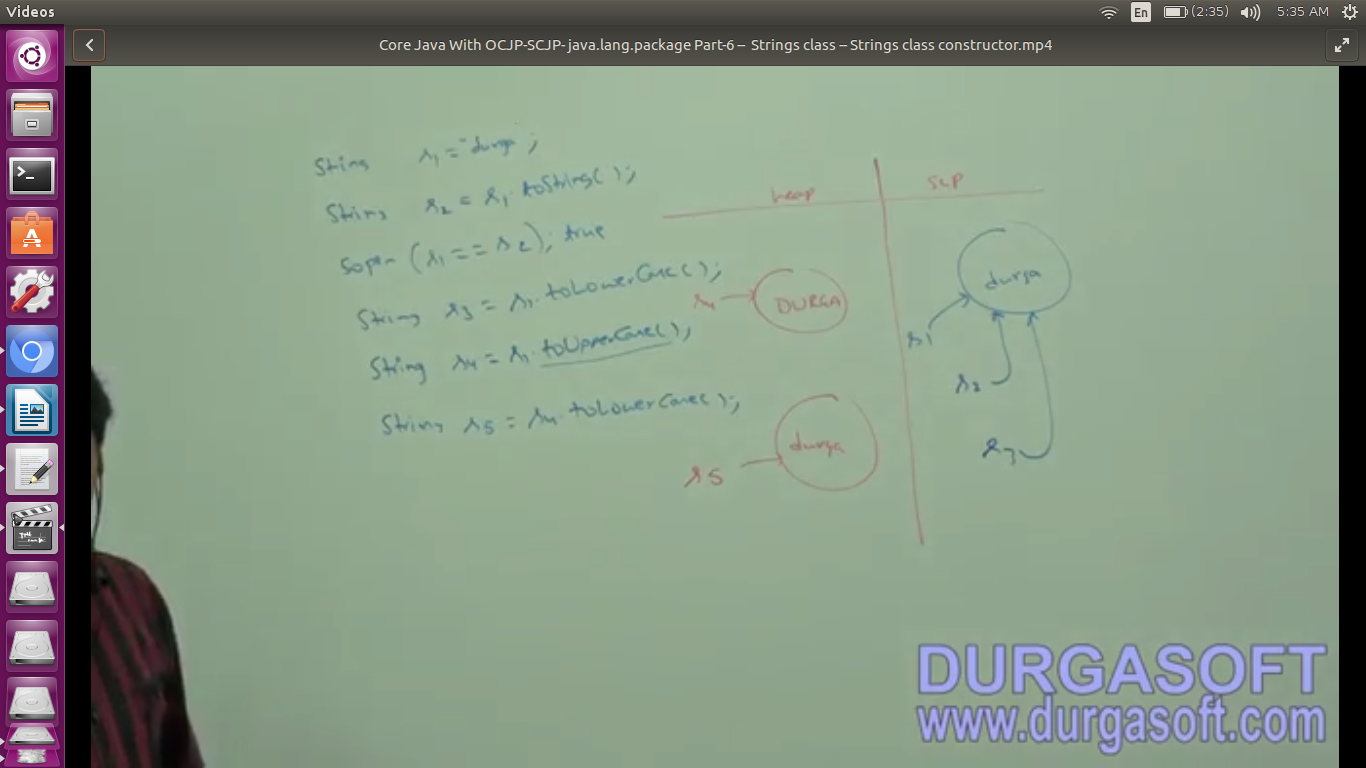
String s3 = s1.toLowerCase()

s1 == s2 // false

s1 == s3 //true

**Because of runtime operation if there is a change in the content, then with those changes a new object will be create at the heap. If their is no change in the content, then existing object will be reused and new object wont be created.**

Whether the object present in heap or SCP, only if there change in content new object will be created or existing object will be reused



**How to create our own immutable class**

Easy, test the content change in the method implementation. If there is a change create new object else return the same.

**Final** class Test {

int i;

Test(int i) {

}

public void modify(int i) {

if(this.i == i) {return this}

else {new Test(i)}

}

}

**Final vs Immutable**

Final applicable for variables but not for objects. Whereas immutablity applicable for objects but not for variables.

**By declaring a reference variable as final we wont get any immutablity nature even though reference variable is the final, we can perform any type of change in the corresponding object but we can perform reassignment for that variable.**

Hence final and immutable both are different concepts

final StringBuffer sb = new StringBuffer(“Suganthan”)

sb.append(“engg”) // no issues

sb = new SB(“something”) //this is not possible

**final variable ==> valid**

**immutable variable ==> invalid**

**final object ==> invalid**

**immutable object ==> valid**

**StringBuffer**

If the content is fixed and wont change frequently then it is recommended to go for String.

If the content is not fixed and keep on changing, then it is not recommeded to use String. Because for every change a new object will be created which effects performance of the system. To handle this requirement, we should go for **StringBuffer**

**Adv: All required changes will be performed in the existing object only.**

**StringBuffer sb = new StringBuffer()** -> creates an empty StringBuffer object with default intial capacity **16.**

**Autogrow = (currentCapacity + 1) \* 2**

**StringBuffer sb = new StringBuffer(int capacity) -> performance will be improved**

**StringBuffer sb = new StringBuffer(String s); ->** For this capacity is s.length+16

**Important methods:**

**append -> appkicable for int, long, boolean etc**

**length**

**capacity**

**charAt**

**setCharAt**

**insert(index,str) -->applicable for int, long, boolean etc**

**delete**

**deleteCharAt**

**reverse**

**setLength**

**ensureCapacity**

**trimToSize**

But every method is **synchronized -->** here comes **StringBuilder**

**Chaining is possible here**

**Wrapper class**

The main objectives of wrapper classes are

1. Two wrap primitive into object form, so that we can handle primitives also, just like object.

2. We can define several utility methods which are required for primitives

**Constructors**

Almost all wrapper class contain two contructor, one can take corresponding **primitive** as argument and the other can take **string** as argument

**Integer int = new Integer(10)**

**Integer int = new Integer(“10”)**

**Double d** = new **Double**(10.5)

**Double** d = new **Double**(“10.5”) //RE if the string argument not representing a number.

\***Float class** contains with **float, double, and String** argument

Float f = new Float(10.5f)

Float f = new Float(“10.5f”)

**Float f = new Float(10.5)//double**

**Float f = new Float(“10.5”)//double stirng**

**\*Character class** contains only one constructor which can take char argument

**Character ch = new Character(‘a’)**

**Character ch = new Character(“a”) //invalid**

**\*Boolean**

Boolean bool = **new** Boolean(**true**);

Boolean bool1 = **new** Boolean(**false**); //false

Boolean bool2 = **new** Boolean(**"true"**);

Boolean bool3 = **new** Boolean(**"TRUE"**);

Boolean bool4 = **new** Boolean(**"True"**);

Boolean bool5 = **new** Boolean(**"suganthan"**); //false

System.***out***.println(bool+**", "**+bool1+**", "**+bool2+**", "**+bool3+**", "**+bool4+**", "**+bool5);

If we are passing String type as argument, then case and content both are not important. If the content is case insensitive string of “true”, then it is treated as true. Otherwise it is treated as false.

Example: Boolean b = new Boolean(“true”)//true

Boolean b = new Boolean(“True”)//true

Boolean b = new Boolean(“TRUE”)//true

Boolean b = new Boolean(“Malika”)//false

Boolean b = new Boolean(“Suganthan”)//false

All wrapper class has overriden **toString** method

**Few methods**

**1. valueOf(String s) or valueOf(primitive) or valueOf(String s, int radix)**--> we can use this method to create wrapper object for the given primitive or string. Which takes string parameter as well.

**Byte, Short, Interger, Long are integral data type**

Every integral type wrapper class Byte, Short, Integer, Long contains the following **valueOf** method. To create wrapper object for the given specified radix string.

**public static wrapper valueOf(String s, int radix)**

The allowed range of **radix is** 2 to 36

**Integer.valueOf(“111”, 2)**

**2.** We can use xxxValue() method to get primitive for the given wrapper object.

Every number type wrapper class contains, the following six methods to get primitive for the given wrapper object

**public byte byteValue()**

**public short shortValue()**

**public int intValue()**

**public long longValue()**

**public float floatValue()**

**public double doubleValue()**

**3. parseXXX -> to convert String to primitive**

**4. toString -> take a look**

**Void – class:**

It is final class and direct child of object, it doesnot contain any method and it contain only one variable Void.TYPE

In general, we can use void class in reflection, to check whether the method return type is void or not.

if(getMethod(m1).getReturnType() == Void.TYPE) //

**Void is a wrapper class representation is primitive void in java**

**AutoBoxing:**

**Automatic** conversion of primitive to wrapper object by compiler.

Integer i = 10;

using valueOf()

**AutoUnBoxing:**

Integer I = new Integer(10);

int i = I

**Automatic** conversion of wrapper object to primitive by compiler.

Using intValue()

class Test {

static Integer I;

p s v m(S[] args) {

int i = I; //On null reference, if we are trying perfrom auto unboxing then we will get RE saying NPE.

}

}

**All Wrapper class by default final**

Integer X = 10;

Integer Y = X;

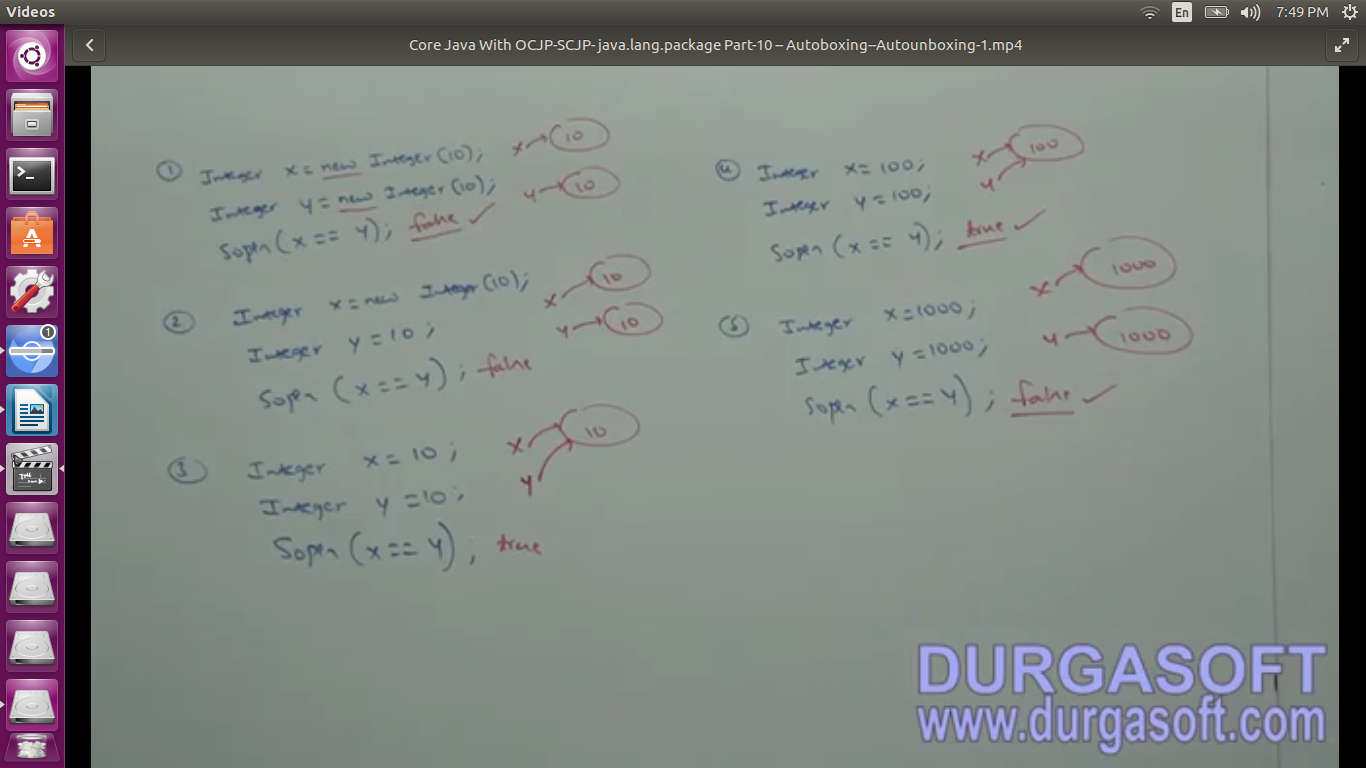
X++;

sop(X) // 11

sop(Y) // 10

sop(X==Y) //false.

While we try to modify X, a new object will be created.



**Overloading with Autoboxing, Widening & var-args methods**

**Widening**

byte -> short --V

|--int --> long -->float --> double

char ---^

**var-args method**

**m1(int... x){**

**}**

**m1(10);**

**m1(10, 20);**

**m1(10, 20, 30);**

**Autoboxing vs Widening**

**Case 1: Widening wins AutoBoxing because of experience**

class Test {

p s v m1(Integer i){sop(“autoboxing”)}

p s v m1(long l){sop(“widening”)}

p s v main(String[] args) {

int i = 10 ;

m1(i); //widening

}

}

**case 2:**

**Windening vs Var-args**

class Test {

p s v m1(int... i){sop(“var-args”)}

p s v m1(long l){sop(“widening”)}

p s v main(String[] args) {

int i = 10;

m1(i); //widening

}

}

Case 3:

**Autoboxing vs var-args**

class Test {

p s v m1(int... i){sop(“var-args”)}

p s v m1(Integer i){sop(“autoboxing”)}

p s v main(String[] args) {

int i = 10;

m1(i); // autoboxing

}

}

Var-args always get least priority. It is same default as switch.

**\*Note:** While resolving overloaded methods, compiler will always gives the precendence in the following order.

1. Widening

2. Autoboxing

3. Var-args method

**Case 4:**

class Test {

p s v m1(Long l){}

p s v main(String[] args) {

int i = 10;

m1(i); // CE

}

}

This is not possible, because widening(int to long) to autoboxing(long to Long) is not implemented.

Long l = 10; //not possible

long l = 10; //widening is possible

W -> A is not possible

**Case 5**:

class Test {

p s v m1(Object o){}

p s v main(String[] args) {

int i = 10;

m1(i); // **Perfectly valid this is Autoboxing(Integer) to Widening(Object)**

}

}

Object o = 10;

Number n = 10;

**Summary:**

**int i=10; //valid**

**Integer I = 10; //valid**

**int i = 10L;** //invalid: possible loss of precision found long required int

**Long L = 10L; //valid: auto boxing**

**Long L = 10; //**invalid: incompatible types found int required Long

**long l = 10; //valid: widening**

**Object o = 10;** //valid: (autoboxing -> widening) int -> Integer -> Object

**double d = 10;** //**valid**: widening

**Double D = 10;** //invalid int -> Long, int -> Double incompatible types

**Number n = 10; //valid autoboxing**

**Equals methods**

**Relation b/w == operator and .equals method**

1. If two objects are equal by == operator then these objects are always equal by .equals method

if r1 == r2 is true, then r1.equals(r2) is always true

2. If two objects are not equal by == operator then we can’t conclude about .equals method. It may returns true or false

if r1 == r2 is false, then r1.equals(r2) is may true or false. And we can’t expect exactly

**Two 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 get compile time error saying incomparable types.

If there is no relation between argument type, then **.equals method wont rise any compile time or run time errors**. Simply it returns false.

String s = new String(“sugu”)

StringBuffer sb = new SB(“sugu”)

s==sb; //CE: incomparable types

s.equals(sb); //false

For any object reference r

r==null

r.equals(null)

always returns false.

**Contract b/w equals and hashcode**

Two objects are equal by .equals method, then there hashcode must be equal. Two equivalent objects should have same hashCode

Two objects are not equal by .equals method, then there no restriction on hashcode. May be equal or may not be equal.

**Clone: Fine**