—---

-Java.lang String

—---
-

```
String s = new String("Chandu");
s.append("Madhu");
System.out.println(s); //Chandu
```

****

```
StringBuffer s = new StringBuffer("Chandu");
s.append("Madhu");
System.out.println(s);//ChanduMadhu
```

—---

Once we create a string Object we cannot perform in the existing Object if we are trying to perform any changes then a new Object will be created , this non-changeable nature is called immutable.

—---

Once we create a stringBuffer Object we can perform in the existing Object this changable of nature is called mutable.

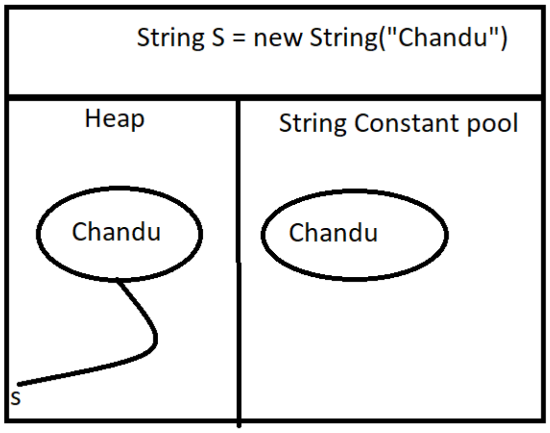
—---

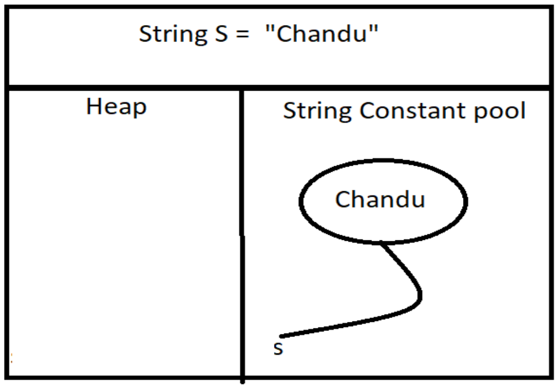
String VS StringBuffer

```
String s = new String("chandu")
String s1 = new String("chandu")
S == s1  //False
s.equals(s1) //True (content comparison) Overridden equals() to content.
```

—---

```
StringBuffer s = new StringBuffer("chandu")
StringBuffer s1 = new StringBuffer("chandu")
S == s1 //false
s.equals(s1)  //False (reference comparison) Object.equals() Executed.
```





Note:

Object creation in SCP is Optional First it will check if there is any Object already present in SCP with required Content if already present then existing Object will be reused, if Object is not only available then only new Object is created. But this rule is applicable only for SCP but not for Heap.

—--------------- Memory Areas -------------------

Heap area

Method area (SCP)

Stack area

Pc Register

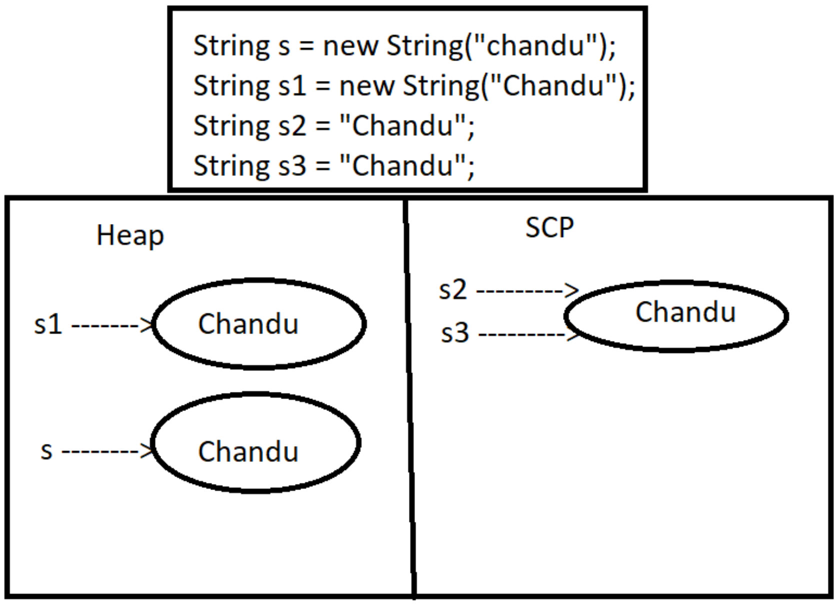
Native method Stacks

—---

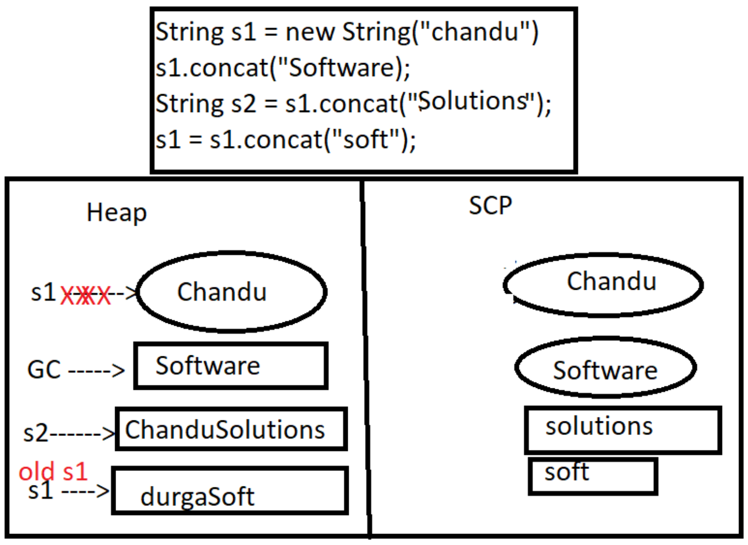
Note 2: garbage collectors not allowed to access SCP are even though Object does not contain reference variables not eligible for gc present in SCP area.

All SCP Objects will destroy automatically at the time of JVM ShutDown.

Example-2:



```
String s = new String("chandu");
String s1 = new String("Chandu");
String s2 = "Chandu";
String s3 = "Chandu";
```



```
String s1 = new String("chandu")
s1.concat("Software);
String s2 = s1.concat("Solutions");
s1 = s1.concat("soft");
```

STRING :

1. String Constructor:

- String s = new String(); //Empty String
- String s1 = "";   (Both are same)
- String s = new String("name"); //Create a String Object on Heap for given String Literal.
- String s = new String(StringBuffer sb);// Creates an Equal String Object for String Buffer .

- String s =new String(char[] ch);
    - Ex: Create a equivalent syring Object for Character Datatype.
    - Char[] ch = {'a',;b','c'};
    - String s = new String(ch);
- String s =new String(byte[] b);
    - EX:
    - byte[] b ={100,101,102,103};
    - String s = new String(b); //Output:- defg

String Methods :

1. Public char charAt(int index);

a. Returns char at Specified Index:
b. If Index is Out of Index then: StringIndexOutOfBoundsException.

2. Public String concat(String s);

a. "Chandu".concat("Ramireddy")
b. The Overloaded +,+= also for concatenation purpose only.
c. String s ="Chandu";
d. s.concat("Software");
e. s+="Software";

3. Public Boolean equals(Object o);

a. This is Case Sensitive.
b. Content Comparison.
c. This is the Overriding version of Object Class Equals(); For Content COmparison.

4. Public Boolean equalsIgnoreCase(Object o);

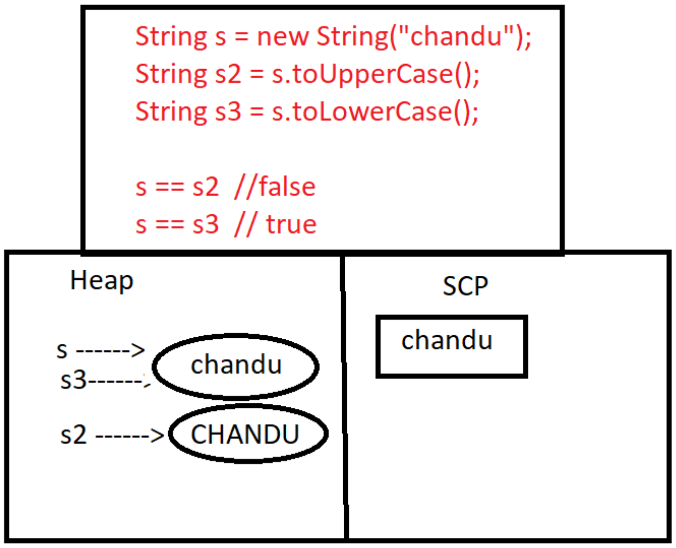
a. This is Not Case Sensitive.
b. This is the Overriding version of Object Class Equals(); For Content COmparison.
c. In general equalsIgnoreCase to validate to user mailID.

5. Public String substring(int Begin);

6. Public String substring(int Begin,int end);

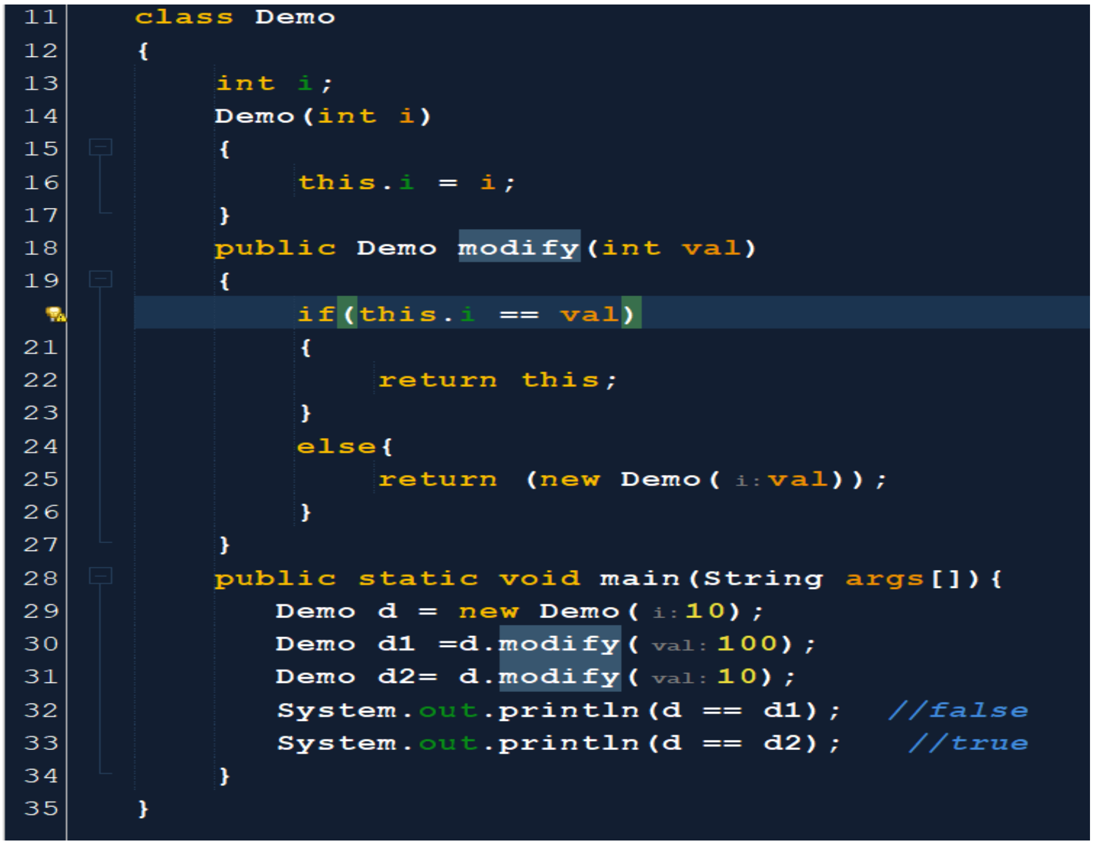
    a. Return Substring from begin to end-1 index;

7. **Public int length();**
    a. Returns the length of String.
    b. “Chandu”.length();   // Number of characters present in string.
    c. Length   —> Applicable for Arrays not for Strings.
8. **Public String replace(char oldchr, char newch);**
    a. s.replace(‘a’,’b’);
9. **Public String toLowerCase()**
10. **Public String toUpperCase()**
11. **Public String trim();**
12. **Public int indexOf(char c);**
13. **Public int lastIndexOf(char ch);**



```
String s = new String("chandu");
String s2 = s.toUpperCase();
String s3 = s.toLowerCase();

s == s2  //false
s == s3  // true
```

    a.
    b. The String Object, If we change the content of string then a new Object is created in heap,else existing Object will be reused.
14. **How we can create Our Own immutable Classes**

a. 

```java
class Demo
{
        int i;
        Demo(int i)
        {
                this.i = i;
        }
        public Demo modify(int val)
        {
                if(this.i == val)
                {
                        return this;
                }
                else{
                        return (new Demo( i: val));
                }
        }
        public static void main(String args[]){
            Demo d = new Demo( i: 10);
            Demo d1 =d.modify( val: 100);
            Demo d2= d.modify( val: 10);
            System.out.println(d == d1);   //false
            System.out.println(d == d2);    //true
        }
}
```

Ex: final StringBuffer sb = new StringBuffer("Chandu");

b. sb.append(" Software");
c. // we can append any methods that are possible, Only reassignment is not Possible.

StingBuffer:

1. If our content is fixed then Go for String Concept.
2. If Our Content is changing always then StringBuffer is Recommended to use.
3. All Changes in StringBuffer, Won't create a new Object, Changes in Existing Objects.

StringBuffer Constructors:

1. **StringBuffer SB =new StringBuffer();**
    a. Default initial Capacity - 16;
    b. If Default capacity is full then a new Object is created and is referred to as a new Object. [(CurrentCapacity+1)*2]

c. SB.capacity(); //16
d. Sb.append("abcdefghijklmnop");
e. Added 16 characters to StringBuffer now the capacity is full;
f. sb.append('q'); Now (16+1)*2; ===>34
g. Performance is down because when sb is full then it will create a new Object and copy all data to the new StringBuffer. So Performance down.

2. StringBuffer sf = new StringBuffer(int initialCapacity);
3. StringBuffer s = new StringBuffer(String s);
   a. Once we create a stringBuffer with String.
   b. Capacity = s.length()+16; ⇒21

Methods For StringBuffer();

1. Public int length();
2. Public int capacity();
3. Public char charAt(int index);
   a. If out of index: StringIndexOutOfBoundsException.
4. Public void setCharAt(int index,char ch);
   a. To replace the character located at specified index with provided Character.
5. Public StringBuffer append(String s);
   a. String,int,long,char,boolean, … all are overloaded. Means we can pass these all as arguments.
6. Public StringBuffer insert(int index, String s);
   a. String,int,long,char,boolean, … all are overloaded. Means we can pass these all as arguments.
   b. We can insert a string at the specified index.
7. Public StringBuffer delete(int begin,int end);
   a. To delete character located from begin index to end-1;
8. Public StringBuffer deleteCharAt(int index);
   a. To delete characters located at Specified Index.

9. **Public StringBuffer reverse();**
10. **Public void setLength(int length);**
    a. Sb = "ChandraSekharReddy"
    b. sb.setLength(7);
    c. Output – "Chandra"
11. **Public void ensureCapacity(int capacity);**
    a. To Increase Capacity on fly based on our required;
    b. StringBuffer sb = new StringBuffer();
    c. System.out.println(sb.capacity()); //16
    d. sb.ensureCapacity(1000);
    e. System.out.println(sb.capacity()) //1000
12. **Public void trimToSize();**
    a. To deallocate the extra allocated free memory
    b. StringBuffer sb = new StringBuffer(1000);
    c. System.out.println(sb.capacity()); //1000
    d. sb.append("abc");
    e. sb.trimToSize();
    f. System.out.println(sb.capacity()); //3

StringBuilder:

Every Method in StringBuffer is Synchronized; And Hence only one thread is allowed to operate on StringBuffer Object At a time Which may Create performance program; To avoid this the sun people introduced StringBuilder in 1.5 version

StringBuffer vs StringBuilder :

- Where ever Buffer is there replaced with Builder;
- Where ever Synchronized is there in StringBuffer it is Removed in StringBuilder.
- StringBuilder(1.5v) and StringBuffer(1.0v) Everything is Same, only Above 2 points are different.

WrapperClasses :

The Main Objectives of wrapper class are:

1. To wrap Primitives into Object form, so that we can handle primitives also just like Objects.
2. To define several utility methods which are required for primitive.

Constructors :

Almost All Wrapper Classes contain 2 Constructors one can take Corresponding primitive as argument and Other can take String as Arguments.

- Float class contains 3 constructors with float double and String Arguments.
- Character Class has only 1 Constructor which can take char arguments.
- Boolean (Boolean, String)
    - Boolean b = new Boolean(true);   // true
    - Boolean b = new Boolean(false); //false
    - Boolean b = new Boolean(True);   Error
    - Boolean b = new Boolean("Yes"); //false
    - Boolean b = new Boolean("no");   //false
    - Boolean b = new Boolean("true"); // True
    - Boolean b = new Boolean("TRUE"); // True
    - If Anything in the form of String it is false except if the content is exactly "true"or "TRUE" or in any case Insensitive form.
- In all Wrapper Class "toString()" is Overridden to written content directly. In all Wrapper Classes .equal() is Overridden for content Comparison.

Example 1:

```
Integer i = new Integer(10);
Integer i1 = new Integer("10");
Integer i2 = new Integer("ten")       XXNumberFormatException.
```

- **Utility Methods:**

1. valueOf();

2. XXXValue();
3. parseXXX();
4. toString();

valueOf : [public static wrapper valueOf(String s)]

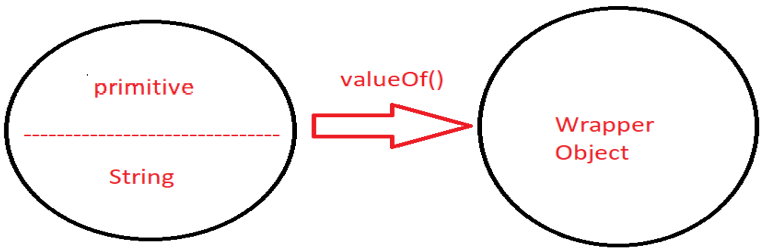
Integer i = new Integer(100);
Integer i = Integer.valueOf('10'); //Recommended
Double d = Double.valueOf("10.2");

We can use valueOf() to create wrapper Object for Given Primitives or String.
Every wrapper class except character class contains a static valueOf() to create wrapper Object for the given String.(Char is not accepted).

- ☐ Integer i = Integer.valueOf('1111'); // Output - 1111
- ☐ Radix: like
- ☐ Integer i = Integer.valueOf('1111',2); // Output - 15
- ☐ (radix option is Only For Integral type) byte, short, int, long.
- ☐ Allowed radix is 2-36; else NumberFormatException
- ☐ Reason for only 2-36
- ☐ Binary -0,1 | Octal = 0- 8 | Decimal = 0-10
- ☐ hexa= 0-9,a-f
- ☐ (2-36) 0-9,a-z (After z what should we take so we have radix upto 36).

Every wrapper class including character class contain a static valueOf() to create wrapper Object for the given primitives.

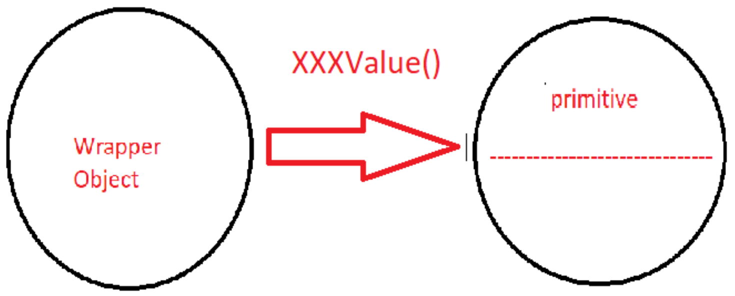
- Integer i = Integer.valueOf(10);
- Character v = Character.valueOf('c');



-

XXXValue() 👍

- We can use XXXValue() to get primitives for the given Wrapper Object.
- To get primitive for the given Wrapper Objects.



Every NumberType Wrapper Class

1) byteValue()
2) shortValue()
3) intValue()
4) longValue()
5) floatValue()
6) doubleValue()

-
- Above Image applies for all 6 datatypes (byte,short,int,long,double,float);
- Ex: Integer i = Integer.valueOf(10);
- System.out.println(i.int(Value());
- i.byteValue(); i.shortValue(); i.intValue(); i.longValue(); i.floatValue(); i.doubleValue(); //-126, 130, 130, 130, 130.0, 130.0 ;

parseXXX():

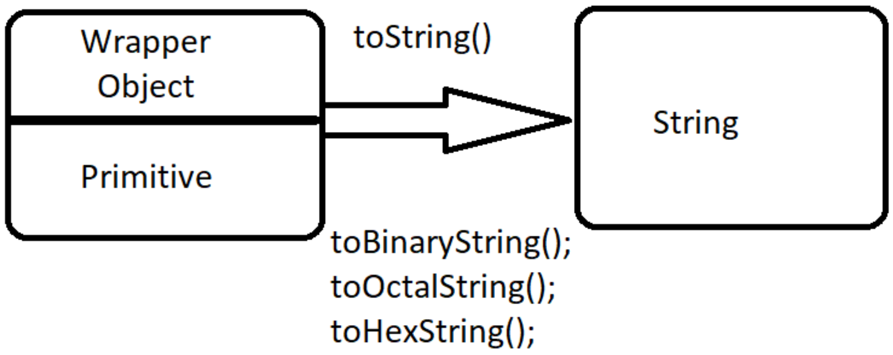
- We can use parseXXX() to convert String to primitive.

1. Form-1:
   a. Public static primitive parseXXX(String s);
   b. Every Wrapper CLass except character Class contains the following parseXXX() to find primitives for the given String Objects.
   c. Int i =Integer.parseInt("10");
   d. Boolean b = Boolean.parseBoolean("true");
2. Form-2
   a. Public static primitive parseXXX(String s , int radix)
   b. Int i =Integer.parseInt("10",2); —> Binary //15(Integrals)

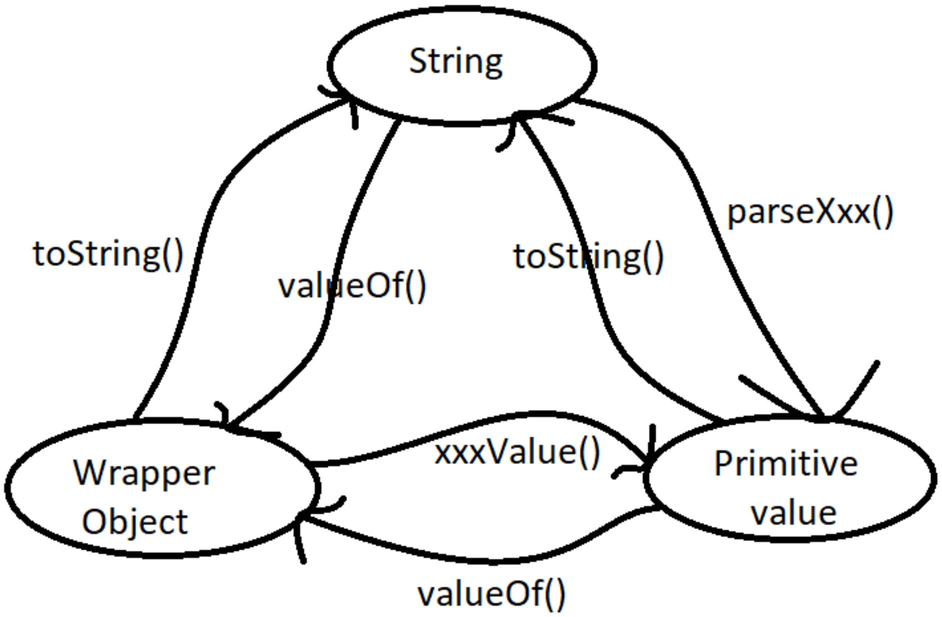
toString() :

- We can use toString() to convert Wrapper Object or primitive to String.
- Form-1
    - Public String toString();
    - Every Wrapper class contains the following toString() to convert Wrapper Object to String type.
    - It is the Overriding version of Object Class toString();
    - Whenever we are trying to print Wrapper Object reference internally this toSting() will be called .
    - Integer i = new Integer(10);
    - String s = i.toString();
    - System.out.println(i); —-> i.toString() is called..
- Form-2;
    - Public static String toString(Primitives p)
    - Every Wrapper class including Character class Contains the following the static toString() to convert Primitives to String.
    - String s = Integer.toString(10);
    - String s = Character.toString('c');
- Form -3
    - Public static String toString(Primitives p,int radix)
    - String s = Integer.toString(10,2);   //radix is for int , Long only.
    -         System.out.println(s); //OutPut: -   (1010)
    - Allowed range of radix (2-36).
    - Integer and Long CLasses the following toString() to convert primitive to specified radix to String type;
- Form-4 👍
    - toXxxString():
    - Integer and Long classes contains the following toXxxString()
    - Public static String toBinaryString(primitive p)
    - Public static String toOctalString(primitive p)
    - Public static String toHexString(primitive p)
    - Ex:

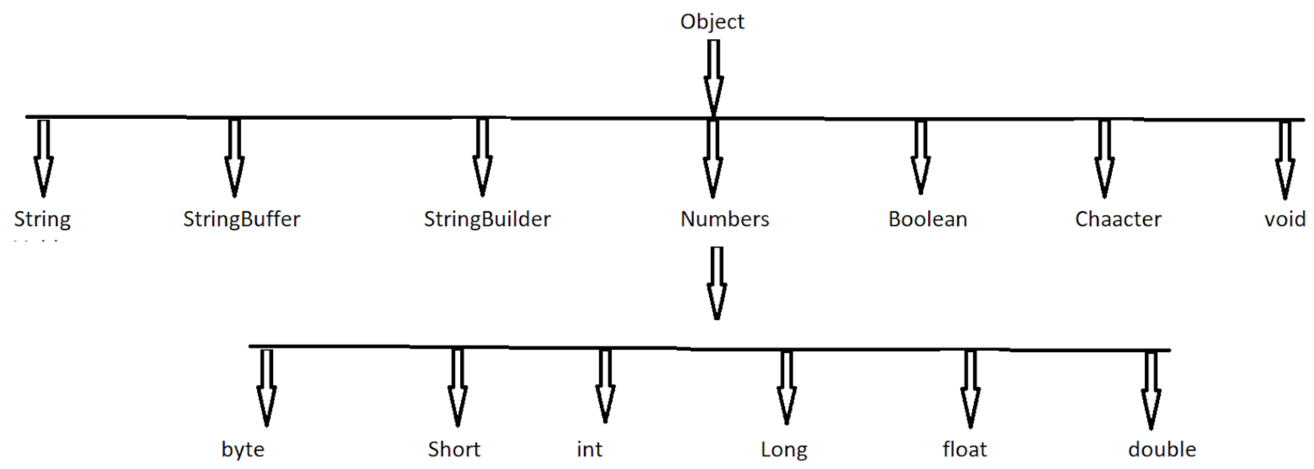
- String s = Integer.toBinaryString(10); //1010
- String s = Integer.toOctalString(10);//12
- String s = Integer.toHexString(10); //a
- //***



-



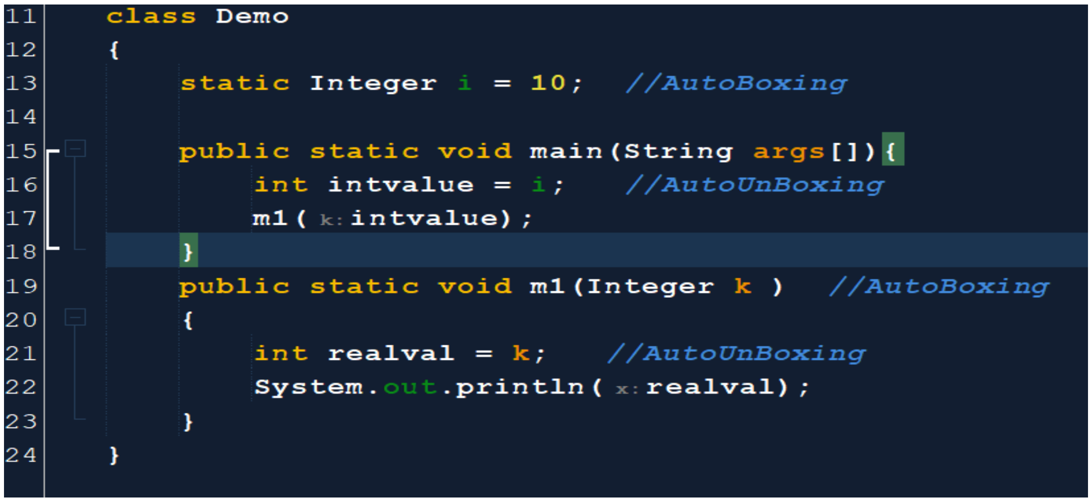
-



1. The Wrapper classes which are not child classes of Number are Boolean Characters.
2. The wrapper classes which are not direct child class of Objects are Byte short Integer long Float,Double
3. String,StringBuffer,StringBuilder are final Classes.
4. In Addition to String Object all Wrapper class Objects are immutable.
5. SomeTimes Void class is also considered as a Wrapper class.
    a. Void Class: It is a Final class and it is the direct child class of Object. It doesn't contain any Methods and it contains one Variable called void Type;
    b. In General we can use Void class in Reflections to check the return is Void or not.

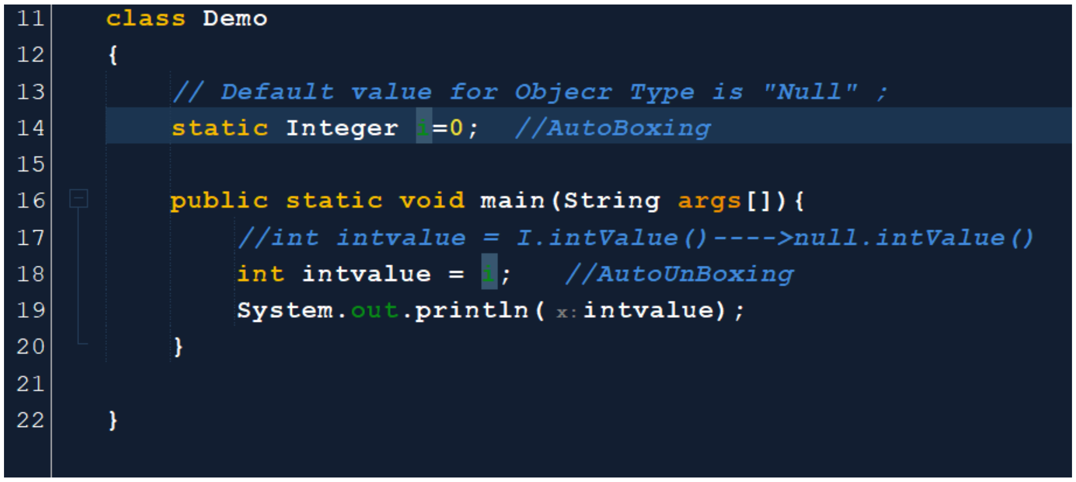
AutoBoxing and AutoUnboxing:

1. Automatic Conversion of Primitive to Wrapper Object by compiler is called AutoBoxing.
2. Integer i = 10; (After Compiling internally Integer i = new Integer(10);)
3. The AutoBoxing concept is implemented by using valueOf().
4. AutoUnboxing:
    a. Integer i = new Integer(10);
    b. Int i1 = I ===> Internally the compiler will convert WrapperObject to Primitives.

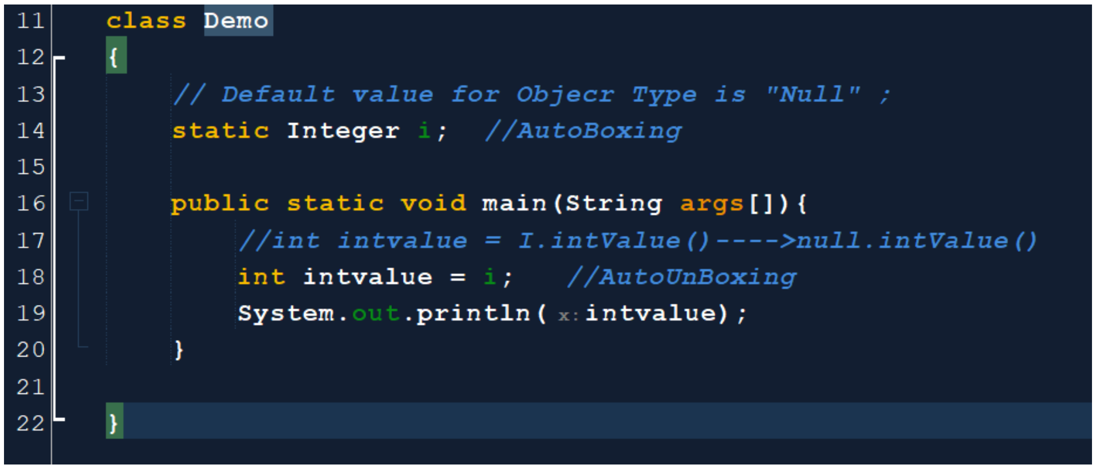
c. 

```java
class Demo
{
    static Integer i = 10;   //AutoBoxing

    public static void main(String args[]){
        int intvalue = i;    //AutoUnBoxing
        m1( k: intvalue);
    }
    public static void m1(Integer k )   //AutoBoxing
    {
        int realval = k;    //AutoUnBoxing
        System.out.println( x: realval);
    }
}
```

d. Above is the Example for AutoBoxing and Auto Unboxing.

5. AutoUnBoxing and AutoBoxing is valid from 1.5Version.
6. Just because of AutoUnBoxing and AutoBoxing, We can use primitives and Wrapper Object Interchangeable from the 1.5 version.
7. 

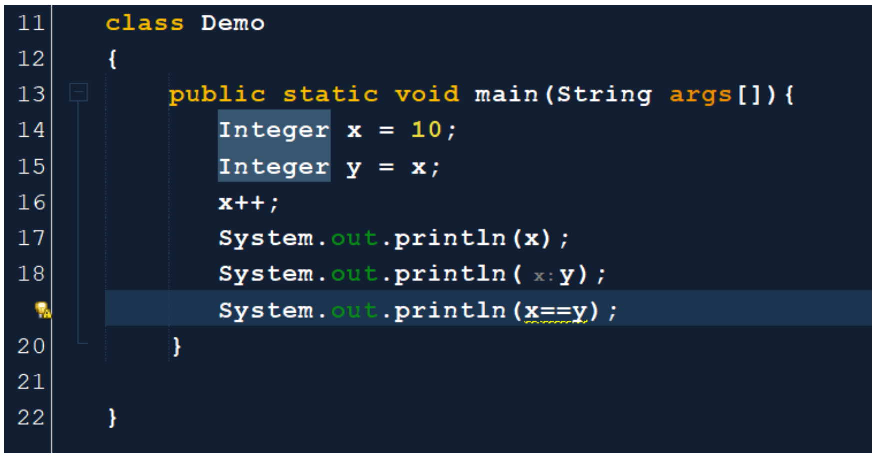
```java
class Demo
{
    // Default value for Objecr Type is "Null" ;
    static Integer i=0;  //AutoBoxing

    public static void main(String args[]){
        //int intvalue = I.intValue()---->null.intValue()
        int intvalue = i;   //AutoUnBoxing
        System.out.println( x: intvalue);
    }

}
```

8. 

```
11  class Demo
12  {
13      // Default value for Objecr Type is "Null" ;
14      static Integer i;   //AutoBoxing
15
16      public static void main(String args[]){
17          //int intvalue = I.intValue()---->null.intValue()
18          int intvalue = i;    //AutoUnBoxing
19          System.out.println( x: intvalue);
20      }
21
22  }
```

9. Above image throws NullPointerException, Explanation in line 17-code.(static Integer i; means Object's Default value is Null).

10. On Null Reference if we are trying to perform autoboxing then we will get a runtimeException saying NullPointerException.

11. 

```
class Demo
{
    public static void main(String args[]){
        Integer x = 10;
        Integer y = x;
        x++;
        System.out.println(x);
        System.out.println( x: y);
        System.out.println(x==y);
    }

}
```

a. Note: All Wrapper Class are immutable,i.e once we create Wrapper class Object we cant perform any changes in that Object if we trying to perform any changes with those changes a new Object is created.

```
//CASE -I
    //Integer x = new Integer(10);   //10
    //Integer y =  new Integer(10);  //10
    //System.out.println(x==y);     //false

 //CASE-II
    //Integer x = new Integer(10);
```

```
        //Integer y = 10;
        //System.out.println(x==y);  //false

 //CASE-III
        //Integer x = 10;
        //Integer y =  10;
        //System.out.println(x==y);   //True

  ///CASE-IV
        //Integer x =100;
        //Integer y = 100;
        //System.out.println(x==y);   //True

//CASE - V
       //Integer x = 1000;
        //Integer y = 1000;
        //System.out.println(x==y);   //false
```

Conclusion: Internally to support for AutoBoxing a buffer of Wrapper Objects will be created at the time Wrapper Class Loading, by Autoboxing if an Object is required is create First JVM will check whether this Object already present in buffer or not If it already peasant in Buffer, Existing Buffer Object is used, if already not in Buffer then JVM will create a new Object.

-128	-127	----	-----	-----	-----	-----------------------------------	0	----10	------	100	---------	127

10 and 100 Is Present in Buffer itself so no new Object is Created, esle like 1000 is not in Buffer So new Object is created.

Buffer concept is available only in the following ranges.

Byte	Always
Short	-128 to 127
Int	-128 to 127
Long	-128 to 127
character	-128 to 127

Boolean Always

Except this range in all remaining cases a new Object will be Created.
For float and double Buffer concept is not there, because 0-1 we have infinite buffer values like 0.1,0.00001,0.00002,
EX:

```
Double d = 10.0;
Double d1 = 10.0;
d.equals(d1); //false
```

Internally AutoBoxing COncept is implemented by using ValueOf() Hence Buffering Concept Applicable for valueOf() also;
Example:-
CASE-I

```
//Integer x = new Integer(10);   //10
Integer y =  new Integer(10);  //10
System.out.println(x==y);     //false
```

CASE-II

```
Integer x = Integer.valueOf(10);
Integer y = Integer.valueOf(10);
System.out.println(x==y);  //TRue
```

CASE-III

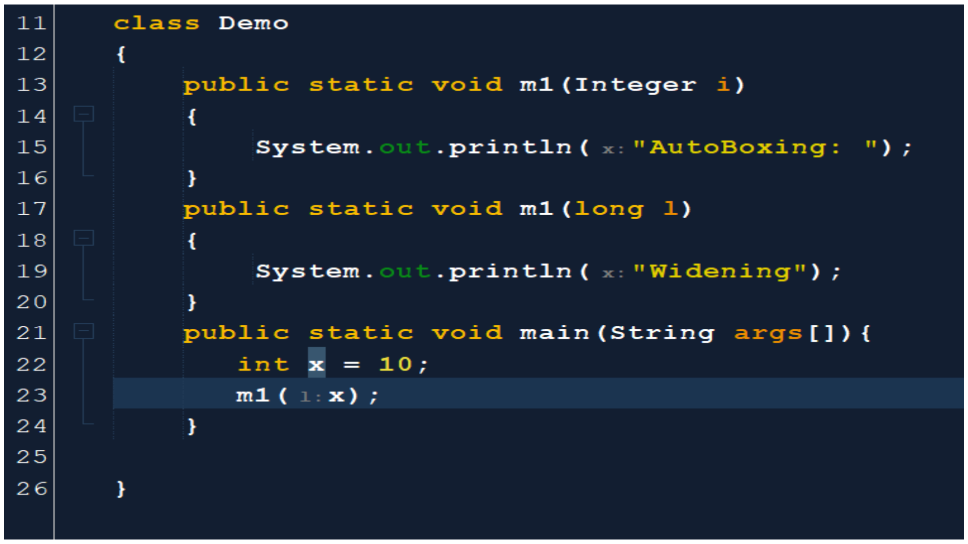
```
//Integer x = 10;
//Integer y =  10;
System.out.println(x==y);  //TRue
```

CASE-IV

```
Integer y = Integer.valueOf(10);
//Integer x = 10;
System.out.println(x==y);  //TRue
```

Overloading w.r.t AutoBoxing widening and varArgs():

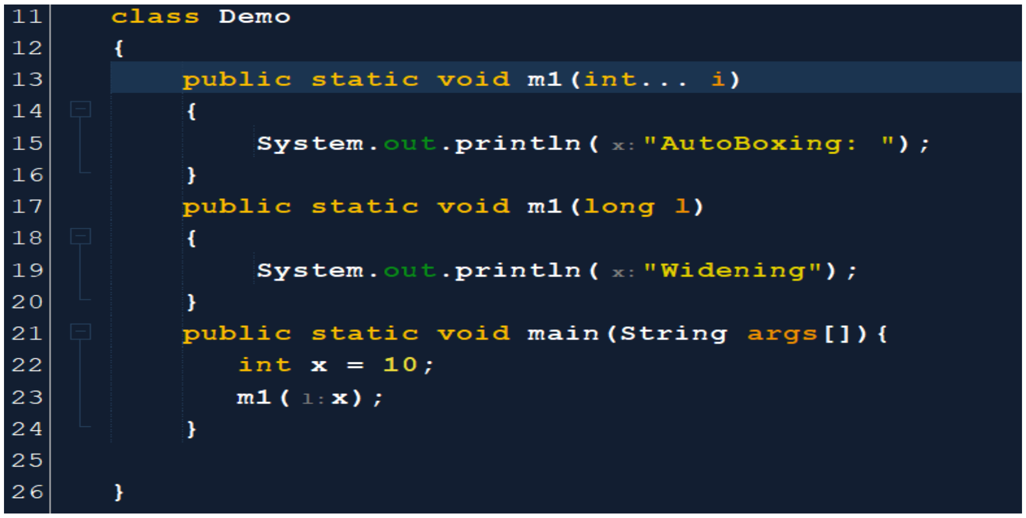
1. CASE -I (AutoBoxing VS Widening)

a. 

```
class Demo
{
    public static void m1(Integer i)
    {
        System.out.println( x: "AutoBoxing: ");
    }
    public static void m1(long l)
    {
        System.out.println( x: "Widening");
    }
    public static void main(String args[]){
        int x = 10;
        m1( l: x);
    }

}
```

b. Output:Widening

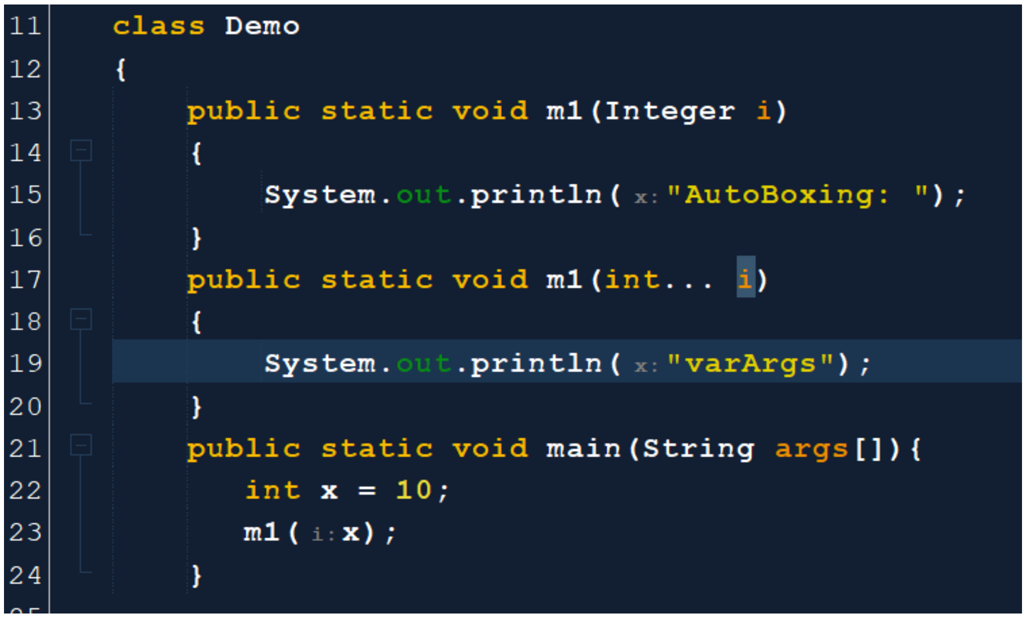
2. CASE -II (Varargs vs Widening) overloadding

a. 

```
class Demo
{
    public static void m1(int... i)
    {
        System.out.println( x: "AutoBoxing: ");
    }
    public static void m1(long l)
    {
        System.out.println( x: "Widening");
    }
    public static void main(String args[]){
        int x = 10;
        m1( i: x);
    }

}
```

b. Output: Widening

3. CASE -III (AutoBoxing vs VarArgs)

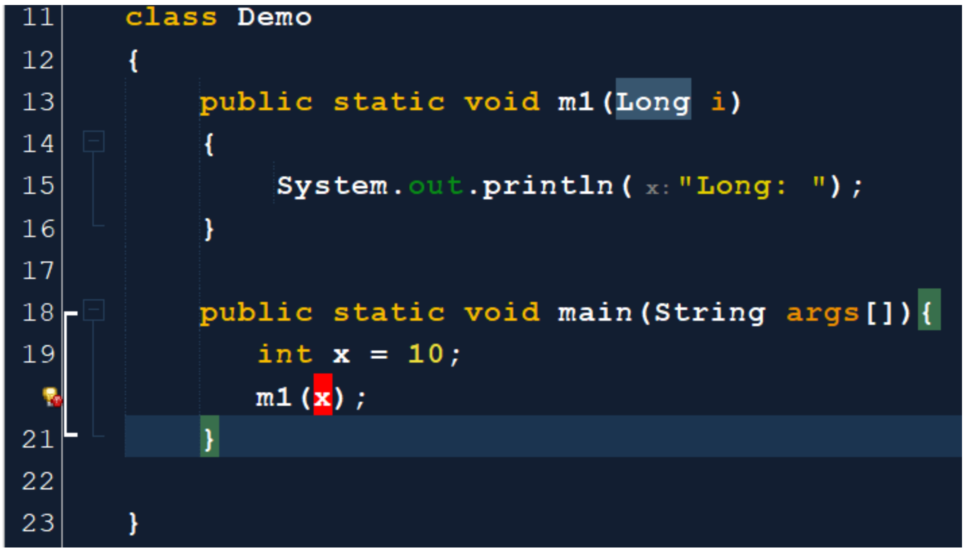
a. 

```
class Demo
{
    public static void m1(Integer i)
    {
        System.out.println( x: "AutoBoxing: ");
    }
    public static void m1(int... i)
    {
        System.out.println( x: "varArgs");
    }
    public static void main(String args[]){
        int x = 10;
        m1( i: x);
    }
```

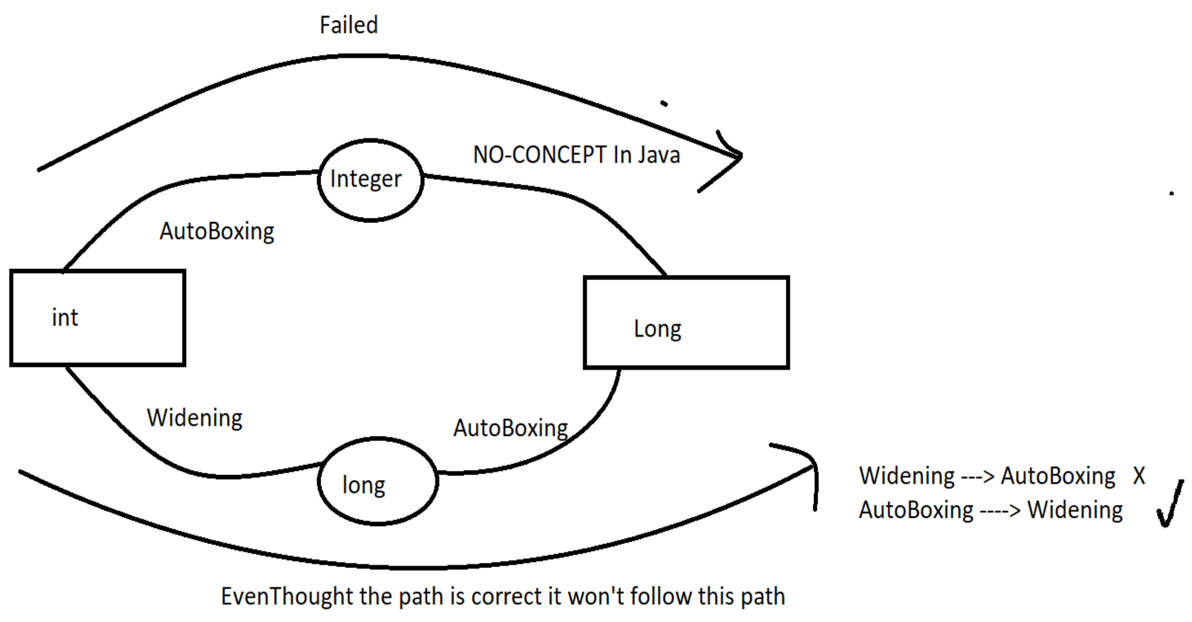
b. AutoBoxing

c. Priority : Widening , AutoBoxing, VarArgs.

4. CASE -IV:

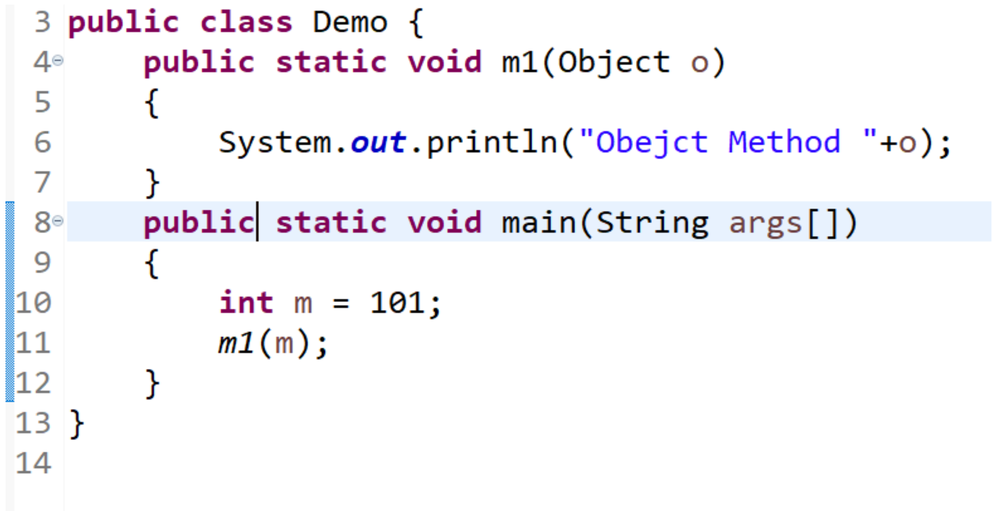
a. 

```
class Demo
{
    public static void m1(Long i)
    {
        System.out.println( x: "Long: ");
    }

    public static void main(String args[]){
        int x = 10;
        m1(x);
    }

}
```


Failed
NO-CONCEPT In Java
Integer
AutoBoxing
int
Long
Widening
AutoBoxing
long
Widening ---> AutoBoxing X
AutoBoxing ----> Widening √
EvenThought the path is correct it won't follow this path

b.

5. Long l = 10; Invalid,
6. long l = 10 valid



```java
public class Demo {
    public static void main(Object o)
    {
        System.out.println("Obejct Method "+o);
    }
    public static void main(String args[])
    {
        int m = 101;
        m1(m);
    }
}
```

7.
8. int > Integer >Object
9. Object o = 10; valid
10. Number n = 10; valid

Valid:

Int i = 10; valid

Integer I = 10; valid AutoBoxing

Int i = 10L; Invalid Compiler Error

Long l = 10L;	valid AutoBoxing
Long l = 10;	Invalid
long l =23;	Valid Widening
Object o = 10;	Valid Autoboxing-widening
double d = 10;	valid widening
Double D = 10;	invalid
Number n = 10;	valid AutoBoxing Widening

.Equals() :

Relation Between "==" and .equals():

- If r1 == r2 is true then r1.equals(r2) is always true.
- If 2 Objects are not equal by "==", then we can't conclude anything by .equals(), It may return true or false; if r1 == r2 is false r1.equals(r2) may be true or false.
- If r1.equals(r2) is true , then r1==r2 returns true or false based on reference.
- If r1.equals(r2) is false then r1 == r2 return always true.

DIfference::

String s = new String("chandu");
String bf = new stringBuffer("chandu");

String.equals() is overridden for content comparison.
StringBuffer.equals() is not Overridden so Object Class .equals() will execute.

System.out.println(s == sb) //invalid Incompatible type java.lang.string, StringBuffer.

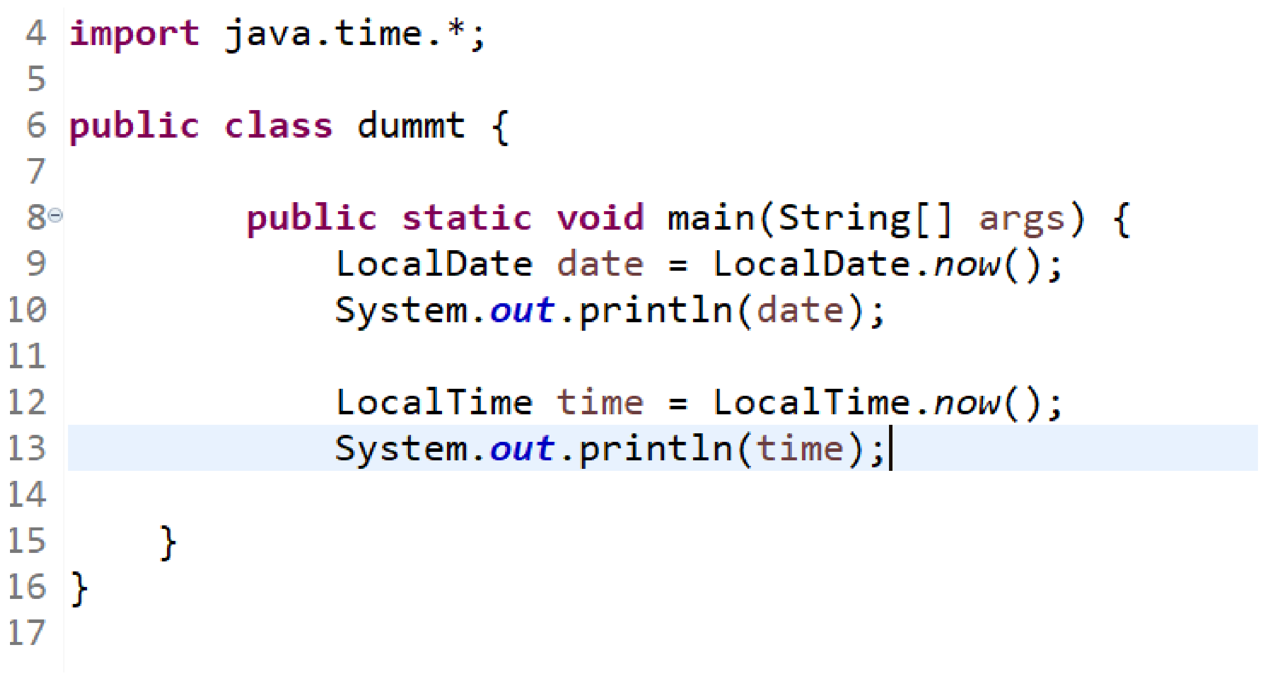
****

Date & Time API

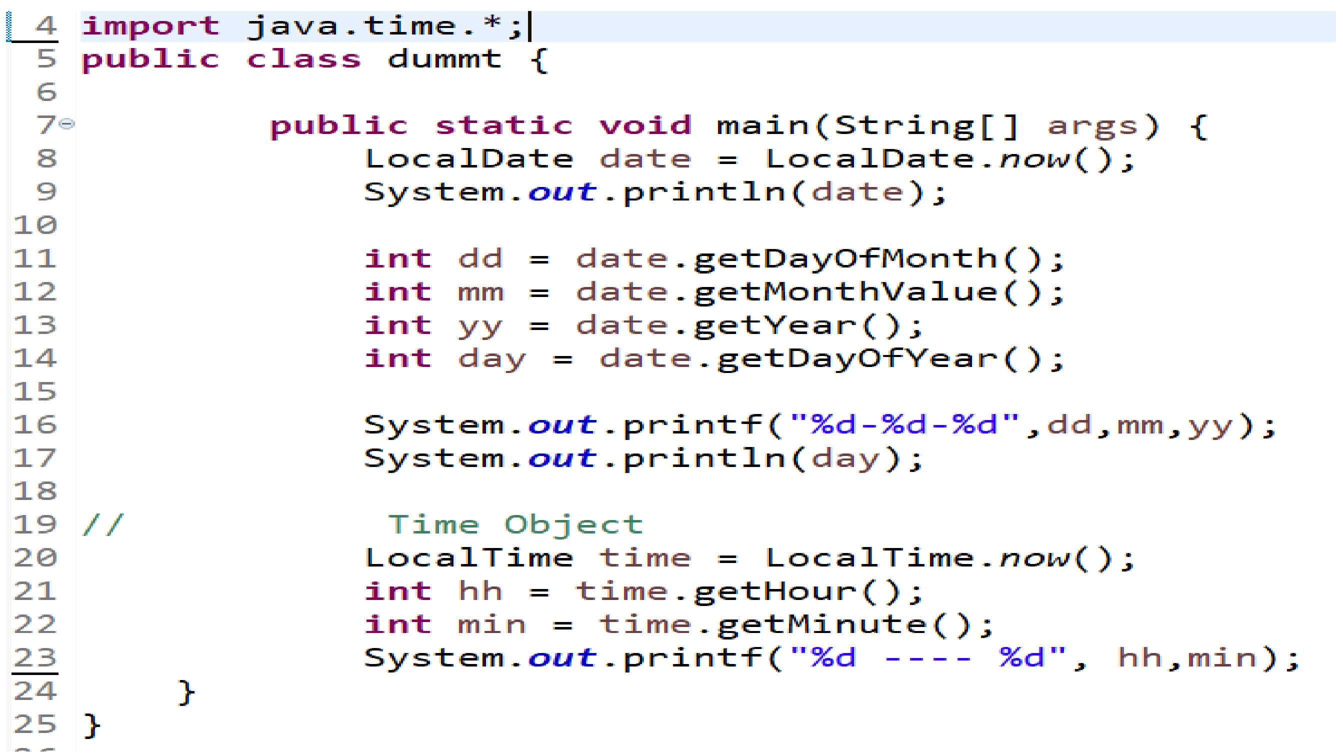
****

Date , Calender , TimeStamp —-> 1.5V (Classes)
Performance , recommendation is low.

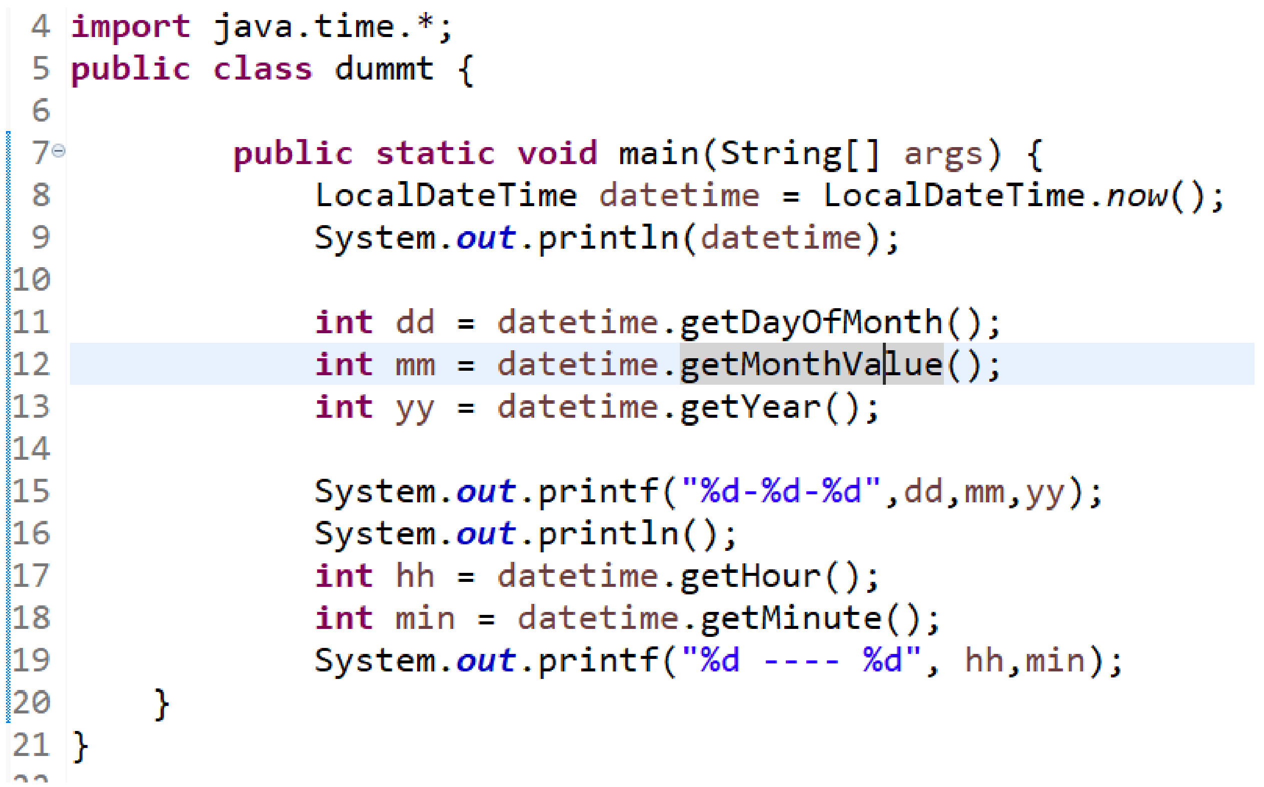
In 1.8 v a new API is introduced
Date and time API —> called as joda Time API(Name)---> by Joda.ORG



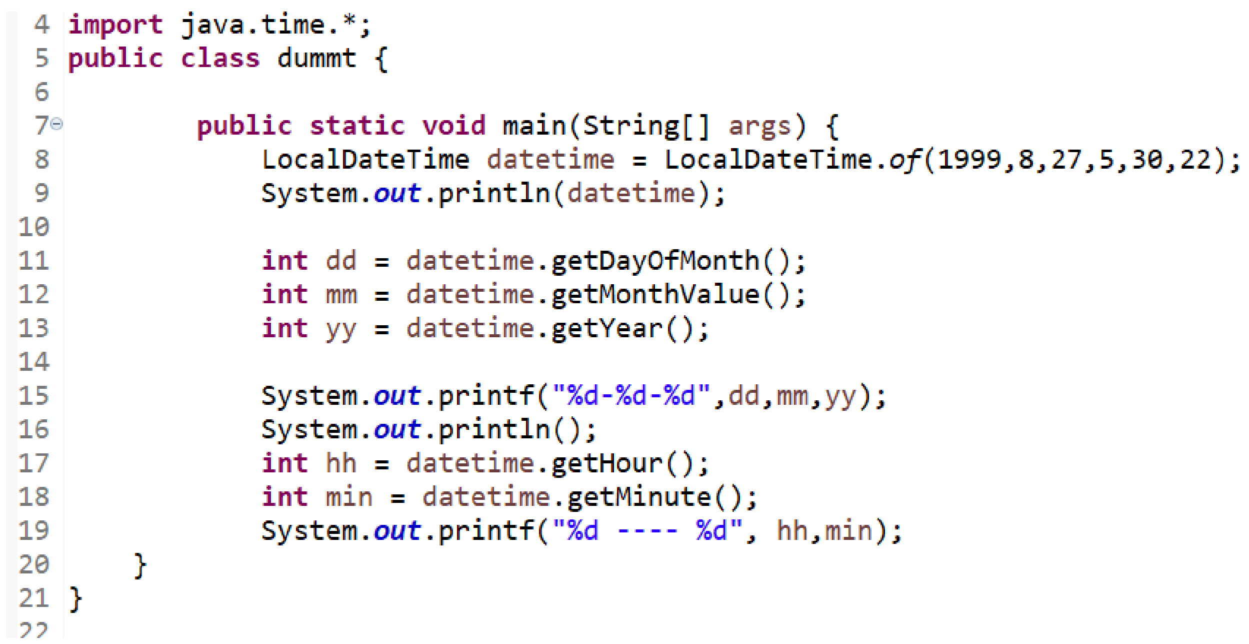
```java
import java.time.*;

public class dummt {

	public static void main(String[] args) {
		LocalDate date = LocalDate.now();
		System.out.println(date);

		LocalTime time = LocalTime.now();
		System.out.println(time);

	}
}
```



```java
import java.time.*;
public class dummt {

        public static void main(String[] args) {
            LocalDate date = LocalDate.now();
            System.out.println(date);

            int dd = date.getDayOfMonth();
            int mm = date.getMonthValue();
            int yy = date.getYear();
            int day = date.getDayOfYear();

            System.out.printf("%d-%d-%d",dd,mm,yy);
            System.out.println(day);

//           Time Object
            LocalTime time = LocalTime.now();
            int hh = time.getHour();
            int min = time.getMinute();
            System.out.printf("%d ---- %d", hh,min);
    }
}
```



```java
import java.time.*;
public class dummt {

        public static void main(String[] args) {
            LocalDateTime datetime = LocalDateTime.now();
            System.out.println(datetime);

            int dd = datetime.getDayOfMonth();
            int mm = datetime.getMonthValue();
            int yy = datetime.getYear();

            System.out.printf("%d-%d-%d",dd,mm,yy);
            System.out.println();
            int hh = datetime.getHour();
            int min = datetime.getMinute();
            System.out.printf("%d ---- %d", hh,min);
    }
}
```



```java
import java.time.*;
public class dummt {

        public static void main(String[] args) {
            LocalDateTime datetime = LocalDateTime.of(1999,8,27,5,30,22);
            System.out.println(datetime);

            int dd = datetime.getDayOfMonth();
            int mm = datetime.getMonthValue();
            int yy = datetime.getYear();

            System.out.printf("%d-%d-%d",dd,mm,yy);
            System.out.println();
            int hh = datetime.getHour();
            int min = datetime.getMinute();
            System.out.printf("%d ---- %d", hh,min);
    }
}
```

****

REGULAR EXPRESSIONS

****

1. Regular Expressions
2. Pattern
3. Matcher
4. Character Classes
5. Predefined Character Classes
6. Quantifiers
7. Pattern Class Split()
8. String Class split()
9. String Tokenizer

If we want to represent a Group of strings according to a Particular Pattern, then we should go for Regular Expression.