Object Oriented Programming

Java™

Unit 4

Useful JAVA API Classes

Working with strings: String Class, StringBuilder Class, StringBuffer, Implement StringBuffer

Working with Date and Time: Using Date class, Implement Date, SimpleDateFormat, Calendar Class

Objects of primitives: Wrapper Classes, Implement Autoboxing, Unboxing

Wrapper Classes

- **Recall:** primitive data types int, double, long are not objects

- **Wrapper classes:** convert primitive types into objects
  - **int:** Integer wrapper class
  - **double:** Double wrapper class
  - **char:** Character wrapper class

- 8 Wrapper classes:
  - Boolean, Byte, Character, Double, Float, Integer, Long, Short

Wrapper Classes

- Why are they nice to have?
  - Primitives have a limited set of in-built operations
  - Wrapper classes provide many common functions to work with primitives efficiently
- How to convert a **String “22”** ➔ **int**?
  - Cannot do this with primitive type int
  - Can use Integer wrapper class; Integer.parseInt(..)

```
String myStr = “22”;
int myInt= Integer.parseInt(myStr);
```

Wrapper Classes

- **Reverse:** How to convert **int 22 ➔ String**?
  - Cannot do this with primitive int
  - Can use Integer wrapper class; Integer.toString(..)

```
int myInt= 22;
String myStr= Integer.toString(myInt); // "22"
String myStr2 = String.valueOf(myInt); // "22"
```

equivalent

- Each Wrapper class has its own set of methods

Wrapper Classes With Constructor

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Autoboxing

- The automatic conversion of **primitive data types** into its **equivalent Wrapper type** is known as **boxing** and opposite operation is known as **unboxing.**
- This is the new feature of Java 5. So java programmer doesn't need to write the conversion code.

Autoboxing

```
class BoxingExample
{
  public static void main(String args[])
 {
         int z=50;
        Integer x=new Integer(z);   //Boxing
        int y=x;                    //Unboxing
        System.out.println(x+" "+y);
        Character a='Z';
        char b=a;
       System.out.println(a+" "+b);
  }
}
```

Using Date Class

- Java provides the **Date class** available in **java.util package**, this class encapsulates the current date and time.
- Time stored in long number. Holds milliseconds passed since January 1,1970(epoch time).
- Java assumes **1900** is the **start year** where the year is calculated ever since 1900.
- The Date class supports **two constructors:**

Constructor	Meaning
Date( )	This constructor initializes the object with the current date and time.
Date(long millisec)	This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Implement Date

```
import java.util.Date;
public class DateDemo
{
public static void main(String args[])
   {
             // Instantiate a Date object
             Date date = new Date();
            // display time and date using toString()
            System.out.println(date.toString());
    }
}
```

Implement Date

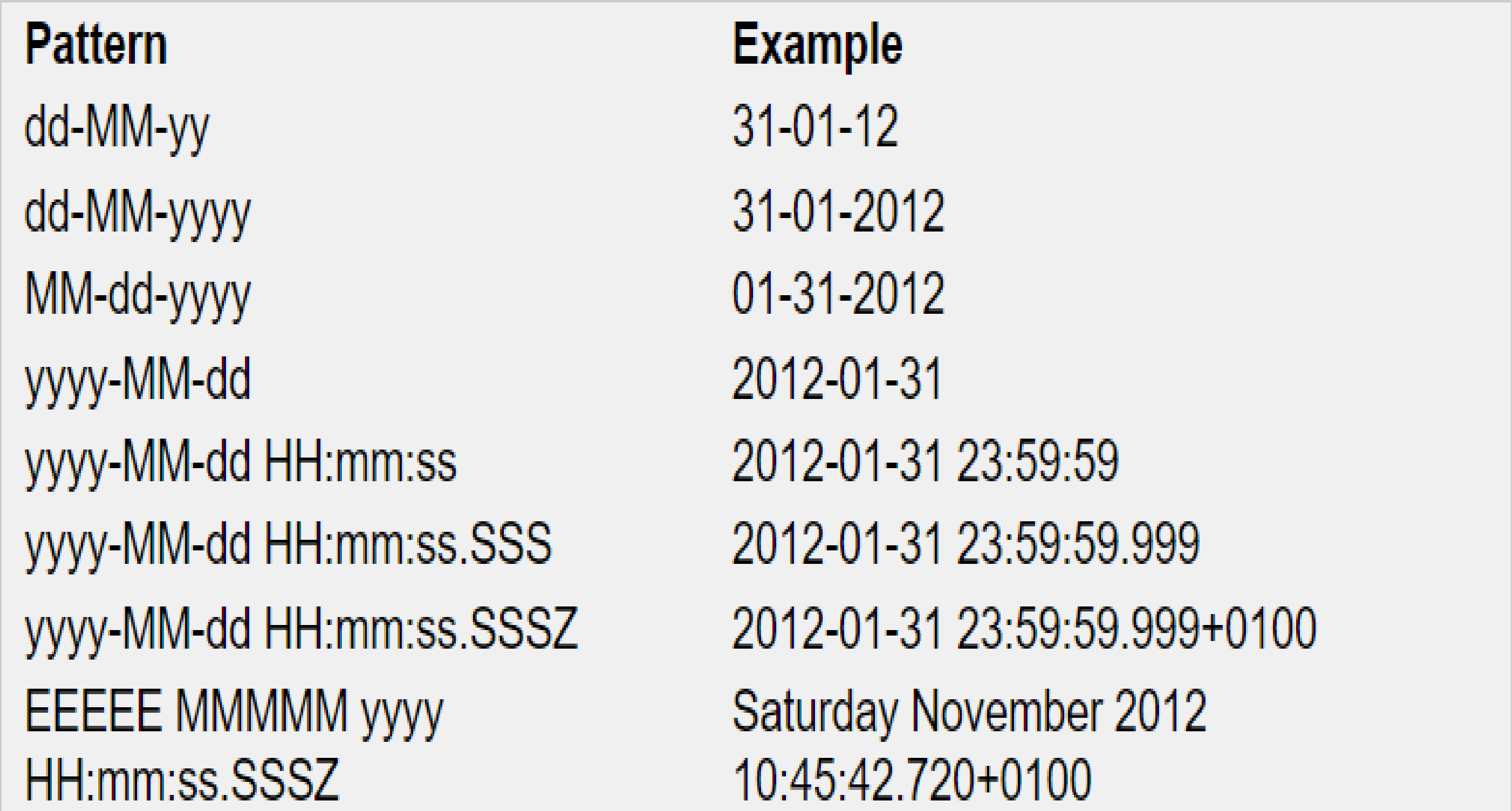
```
import java.util.Date;
public class DateDemo
{
public static void main(String args[])
    {
        System.out.println(new Date().toString());
     }
}
```

SimpleDateFormat

The **java.text.SimpleDateFormat** class provides methods **to format and parse date and time in java.**

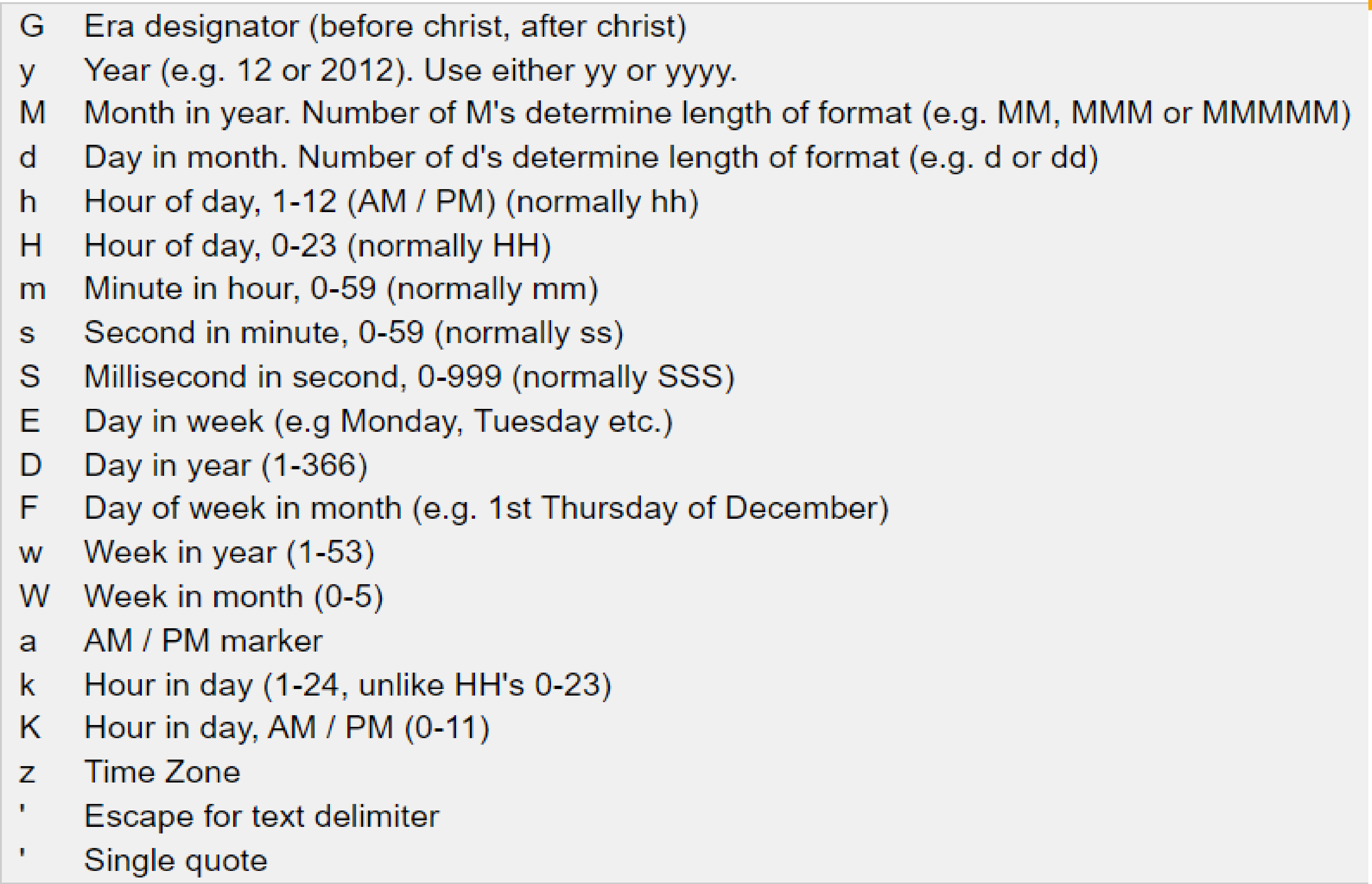
The SimpleDateFormat is a concrete class for formatting and parsing date which inherits java.text.DateFormat class.

SimpleDateFormat



Pattern	Example
dd-MM-yy	31-01-12
dd-MM-yyyy	31-01-2012
MM-dd-yyyy	01-31-2012
yyyy-MM-dd	2012-01-31
yyyy-MM-dd HH:mm:ss	2012-01-31 23:59:59
yyyy-MM-dd HH:mm:ss.SSS	2012-01-31 23:59:59.999
yyyy-MM-dd HH:mm:ss.SSSZ	2012-01-31 23:59:59.999+0100
EEEEE MMMMM yyyy HH:mm:ss.SSSZ	Saturday November 2012 10:45:42.720+0100

SimpleDateFormat



G	Era designator (before christ, after christ)
y	Year (e.g. 12 or 2012). Use either yy or yyyy.
M	Month in year. Number of M's determine length of format (e.g. MM, MMM or MMMMM)
d	Day in month. Number of d's determine length of format (e.g. d or dd)
h	Hour of day, 1-12 (AM / PM) (normally hh)
H	Hour of day, 0-23 (normally HH)
m	Minute in hour, 0-59 (normally mm)
s	Second in minute, 0-59 (normally ss)
S	Millisecond in second, 0-999 (normally SSS)
E	Day in week (e.g Monday, Tuesday etc.)
D	Day in year (1-366)
F	Day of week in month (e.g. 1st Thursday of December)
w	Week in year (1-53)
W	Week in month (0-5)
a	AM / PM marker
k	Hour in day (1-24, unlike HH's 0-23)
K	Hour in day, AM / PM (0-11)
z	Time Zone
'	Escape for text delimiter
'	Single quote

Implement SimpleDateFormat

```
package datedemo;
import java.util.*;
import java.text.*;
public class SimpleDateDemo
{
  public static void main(String args[])
{
    Date dNow = new Date();
    SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");
    System.out.println("Current Date: " + ft.format(dNow));
  }
}
```

Note: M (capital M) represents month and m (small m) represents minute in java.

Implement SimpleDateFormat

```
import java.util.*;
import java.text.*;
public class SimpleDateDemo {
   public static void main(String args[]) {
      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

Question Time

Is it safe to store the number of milliseconds in a variable of type long?

Overview of java.time package

- ***java.time*** - the base package containing the value objects
- ***java.time.chrono*** -provides access to different calendar systems [like Thai Buddhist]
- ***java.time.format*** - allows date and time to be formatted and parsed
- ***java.time.temporal*** - the low level framework and extended features
- ***java.time.zone*** - support classes for time-zones

LocalDate

This object only contain date component.

LocalDate

It is consist of Day, Month, Year.

Date using LocalDate

Today Date:

```
import java.time.*;

LocalDate today = LocalDate.now();

System.out.println("Today's Date : "+ today);
```

Output :- **Today's Date : 2021-09-15**

Previous Date:

```
LocalDate today = LocalDate.now();
System.out.println("Previous Date : "+  today.minusDays(1));
```

Output :- **Previous Date : 2021-09-14**

Next Date:

```
LocalDate today = LocalDate.now();
System.out.println(“Next Date : "+  today.plusDays(1));
```

Output :- **Next Date : 2021-09-16**

LocalDate using printf

Days in a Year:

```
LocalDate today = LocalDate.now();

System.out.printf("%d days in  %d\n",today.lengthOfYear(),today.getYear());
```

Output :- **365 days in 2021**

Is a Leap Year?

```
import java.time.temporal.*;

LocalDate today = LocalDate.now();

System.out.printf("%d is leap year?  %s\n",today.get(ChronoField.YEAR),
today.isLeapYear());
```

Output: **2021 is leap year ? false**

LocalTime

- LocalTime object only contains **time component.**
- It is consist of **hour, minute and second**.
- Example: What is the time now ? “01:40 PM”

LocalTime

Current Hour

```
LocalTime timeNow=LocalTime.now();
 System.out.println("Today's Time : "+ timeNow);
```

Output: **Today’s Time: 08:44:58.900935900**

Adding an hour to current hours

```
LocalTime timeNow=LocalTime.now();
System.out.println("Meeting in next hour : "+  timeNow.plus(1, ChronoUnit.HOURS));
```

Output: **Meeting in next hour : 09:56:10.968536100**

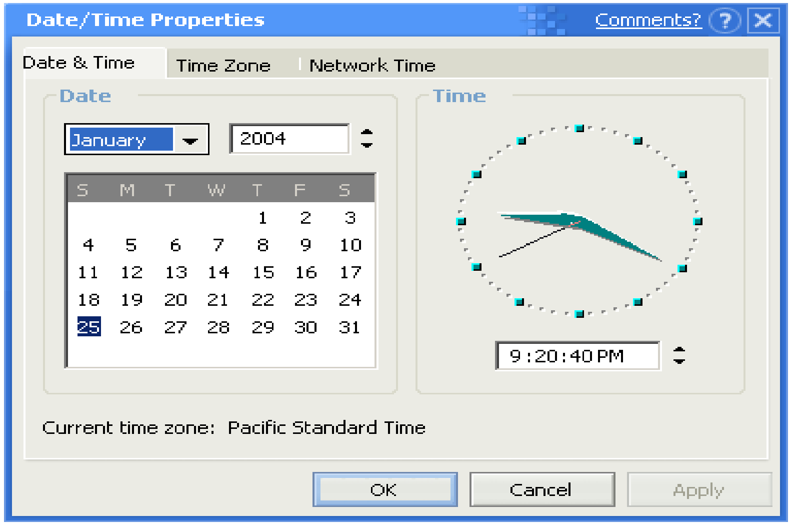
Only take out minutes

- LocalTime timeNow = LocalTime.*now();*
- System.*out.println("Only upto minutes : "+ timeNow.truncatedTo(ChronoUnit.MINUTES));*

- *Output:* ***Only upto minutes : 13:21***

LocalDateTime

- This object contain date and time components.
- *It is consist of year,month,day,hour, minute and second.*



January 2004

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

9:20:40 PM

Current time zone: Pacific Standard Time

ZoneOffset

- This object represent the time zone difference from Greenwich/UTC.
- This is usually a fixed number of hours and minutes.

ZoneDateTime

- It contain date and time along with time zone details.

Commonly Used Methods

Prefix	Use
of	Creates an instance where the factory is primarily validating the input parameters, not converting them.
from	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
parse	Parses the input string to produce an instance of the target class.
format	Uses the specified formatter to format the values in the temporal object to produce a string.
get	Returns a part of the state of the target object.
plus	Returns a copy of the target object with an amount of time added.
minus	Returns a copy of the target object with an amount of time subtracted.

DateTimeFormatter

- This class is primarily responsible for formatting.

- *DateTimeFormatterBuilder* class allow a *DateTimeFormatter* to be created.

- Formatter can be created with desired Locale, Chronology, ZoneId and DecimalStyle.

```
LocalDate date = LocalDate.now();
String text = date.format(formatter);
```

STRING

What is a String?

String ➔ a sequence of characters Example of **strings:**

“The cow jumps over the moon”

“This is a lecture on 5 Oct2020”

“PRESIDENT OBAMA”

“12345”

What is a String class in Java?

- String class in Java: holds a “sequence of characters”

String **greeting** = new String("Hello world!“);

- Creating new String object
- Contains sequence of chars

String Class

- Why use String class vs. char array?
- **Advantage:** provides many useful methods for string manipulation
  - – Print length of string – **str.length()**
  - – Convert to lowercase – **str.toLowerCase()**
  - – Convert to uppercase – **str.toUpperCase()**

* Many others

Creating String Objects

There are 2 ways to create String objects

Method 1: String **greeting1** = **new** String(“Hello World!”) ;

Method 2: String **greeting1** = “Hello World!” ;

Numbers as strings

String **s3** =“12345”;

String **s4** = **new** String(s3); //s4 will hold same value as s3

Char array as strings

char[] **helloArray** = { 'h', 'e', 'l', 'l', 'o', '.' };

String **s5**= **new** String(helloArray);

String Constructor

- **Recall:** when new object created → the constructor method is always called first

- Pass initial arguments or empty object

- String class has multiple constructors

String Constructor

```
String str1= new String() ; //empty object

String str2= new String(“string”) ; //string input

String str3= new String(char[]) ; //char array input

String str4= new String(byte[]) ; //byte array input
```

* few others

Understanding String Creation

String **greeting1** = “Good day!” String **greeting2** = “Good day!”

Does Java create 2 String objects internally?

Without **“new”** operator for String creation:

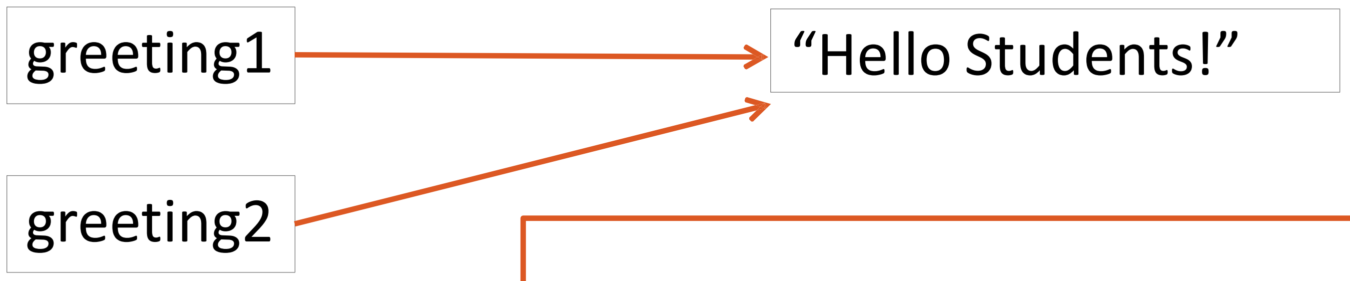
- Java looks into a String pool (collection of String objects)
  - Try to find objects with same **string** value
- If **object exists** ➔ new variable points to existing object
- If **object does not exist** ➔ new object is created
- Efficiency reasons – to limit object creation

Understanding String Creation

```
String greeting1 = “Hello Students!”
String greeting2 = “Hello Students!”
```

Local Variable Table

Pool of String Objects



greeting1

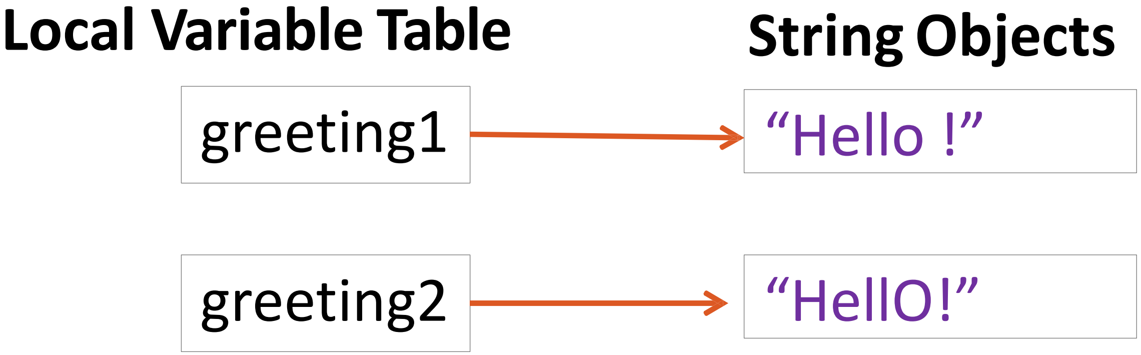
greeting2

“Hello Students!”

Concept of String pooling

Understanding String Creation

```
String greeting1 = new String (“Hello!”);
String greeting2 = new String (“Hello!”);
```



Local Variable Table

String Objects

greeting1

“Hello !”

greeting2

“HellO!”

STRINGS - METHODS, CONCATENATION

String Methods

Advantage of String class: many built-in methods for String manipulation

```
str.length();           // get length of string
str.toLowerCase()       // convert to lower case
str.toUpperCase()       // convert to upper case
str.charAt(i)           // what is at character i?
str.contains(..)        // String contains another string?
str.startsWith(..)      // String starts with some prefix?
str.indexOf(..)         // what is the position of a character?
....many more
```

String Methods - length, charAt

str.length() ➔ Returns the **number of chars** in String

str.charAt(i) ➔ Returns the **character at position** i

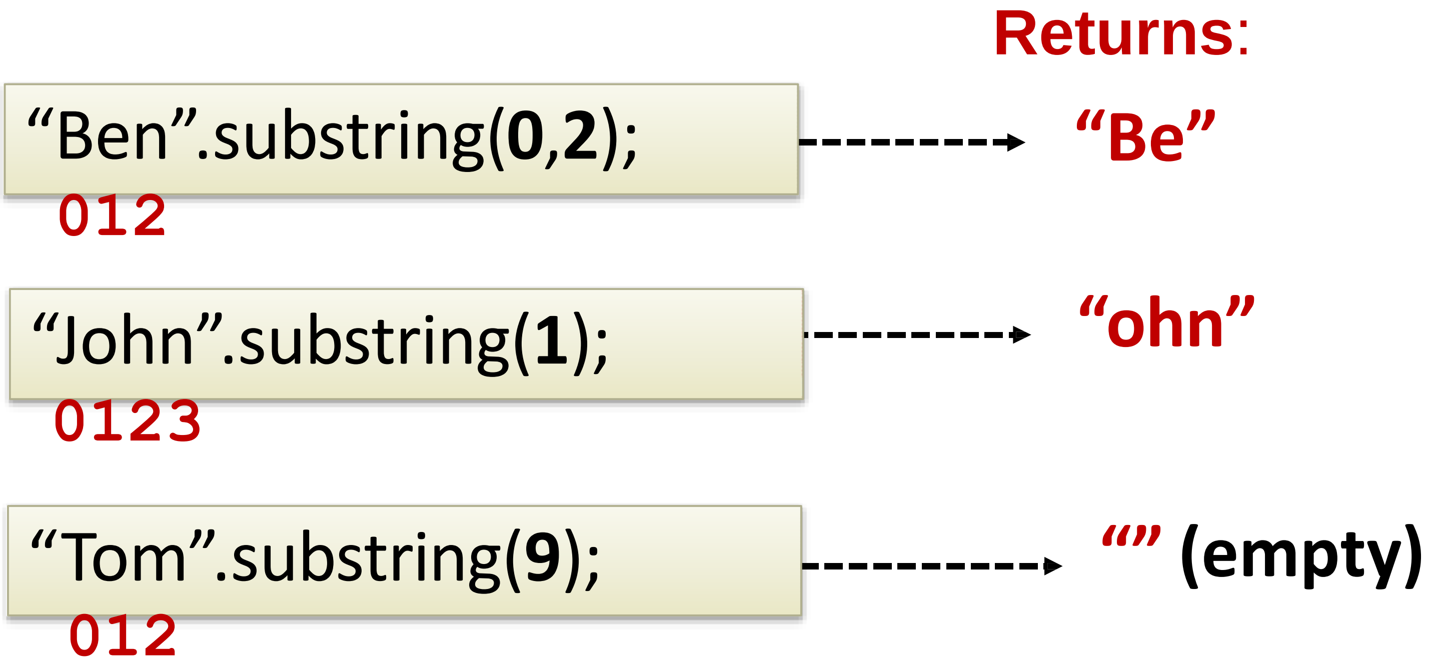
Character positions in strings are numbered starting from 0 – just like arrays

String Methods – substring(..)

str.substring(..) ➔ returns a new String by copying characters from an existing String.

- str.**substring (i, k)**
  - – returns substring of chars from pos i to k-1
- str.**substring (i)**;
  - – returns substring from the i-th char to the end

String Methods – substring(..)



Returns:

“Ben”.substring(**0,2**);
012
“Be”

“John”.substring(**1**);
0123
“ohn”

“Tom”.substring(**9**);
012
“” (empty)

String Concatenation – Combine Strings

- What if we wanted to combine String values?

String **word1** = “re”;

String **word2** = “think”;

String **word3** = “ing”;

How to combine and make ➔ **“rethinking” ?**

- Different ways to concatenate Strings in Java

String Concatenation – Combine Strings

String **word1** = “re”;
String **word2** = think”;
String **word3** = “ing”;

Method 1: Plus “+” operator String **str** = word1 **+** word2;

–*concatenates word1 and word2* ➔ ***“rethink”***

Method 2: Use String’s “concat” method

String **str** = word1.**concat** (word2);

–*the same as word1 + word2* ➔ ***“rethink”***

String Concatenation – Combine Strings

Now **str** has value **“rethink”,** how to make **“rethinking”**?

String **word3** = “ing”;

Method 1: Plus “+” operator

str = str + word3;   //results in ***“rethinking”***

Method 2: Use String’s “concat” method

str = str.**concat**(word3);   //results in ***“rethinking”***

Method 3: Shorthand

str += word3;   //results in ***“rethinking”*** (same as method 1)

STRING EQUALITY/INEQUALITY

Testing String Equality

- In general, both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

1. The main difference between the .equals() method and == operator is that one is a method and the other is the operator.
2. We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

Testing String Equality

How to check if two Strings **contain same value**?

```
String str1=new String(“Hello World!”);

String str2=new String(“Hello World!”);

if(str1==str2)                                    //false

{

   System.out.println(“same”);

}
if(str1.equals(str2))                          //true
{
 System.out.println(“same with equals”);
}
```

Testing String Equality

- **Point to note:** String variables are references to String objects (i.e. memory addresses)

- **“str1==str2”** on String objects compares memory addresses, not the contents

- Always use **“str1.equals(str2)”** to compare contents

STRING IMMUTABILITY

String Immutability

- Strings in Java **are immutable**
- **Meaning:** cannot change its value, once created

```
String str;

str = “Java is Fun!”;
str = “I hate Java!”;
```

Did we change the value of **“Java is Fun!”** to **“I hate Java!”**?

String Immutability

- **Problem:** With frequent modifications of Strings
  - Create many new objects – uses up memory
  - Destroy many unused ones – increase JVM workload
  - **Overall:** can slow down performance

- **Solution** for frequently changing Strings:
  - StringBuilder Class instead of String

StringBuffer

- **A thread-safe, mutable sequence of characters**. A string buffer is like a String, but **can be modified(length and content).**

- String buffers are safe for **use by multiple threads**. The methods **are synchronized** where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

- **The principal operations on a StringBuffer are the append and insert methods,** which are overloaded so as to accept data of any type.

- **Every string buffer has a capacity.** As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

- The **append** method always adds these characters at the end of the buffer; the **insert** method adds the characters at a specified point.

StringBuffer

StringBuffer()

- Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer(int capacity)

- Constructs a string buffer with no characters in it and the specified initial capacity.

StringBuffer(String str)

- Constructs a string buffer initialized to the contents of the specified string.

Example:

```
class StringBufferExample2
{
 public static void main(String args[])
 {
   StringBuffer sb=new StringBuffer("Hello ");
   sb.insert(1,"Java");//now original string is changed
   System.out.println(sb);//prints HJavaello
 }
}
```

StringBuilder Class

- **StringBuilders** used for String concatenation

- StringBuilder class is “**mutable**”

Meaning: values within StringBuilder can be changed or modified as needed

- In contrast to Strings, where Strings are immutable

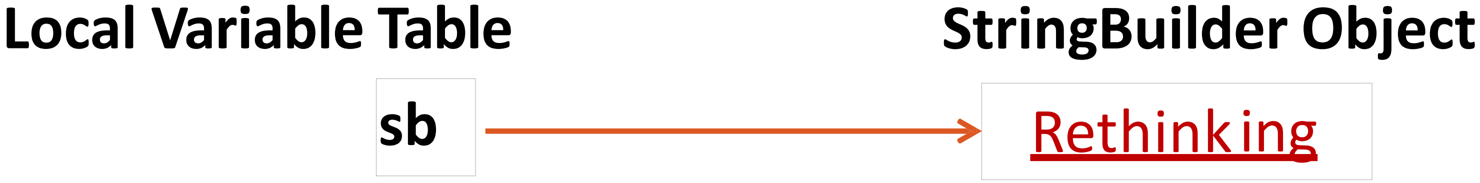
StringBuilder – “append” method

- append(X) ➔ most used method
  - Appends a value **X** to end of StringBuilder
  - ***X: int, char, String, double, object (almost anything)***

StringBuilder for String Construction

```
StringBuilder sb = new StringBuilder(); //obj creation
sb.append("Re");        //add "Re" to sb
sb.append("think"); //add "think" to sb
sb.append("ing");       //add "ing" to sb
String str= sb.toString();
```

to same object



Local Variable Table

StringBuilder Object

sb

Rethinking

Bottom-line: *Use StringBuilder when you have frequent string modification*

String, StringBuilder and StringBuffer

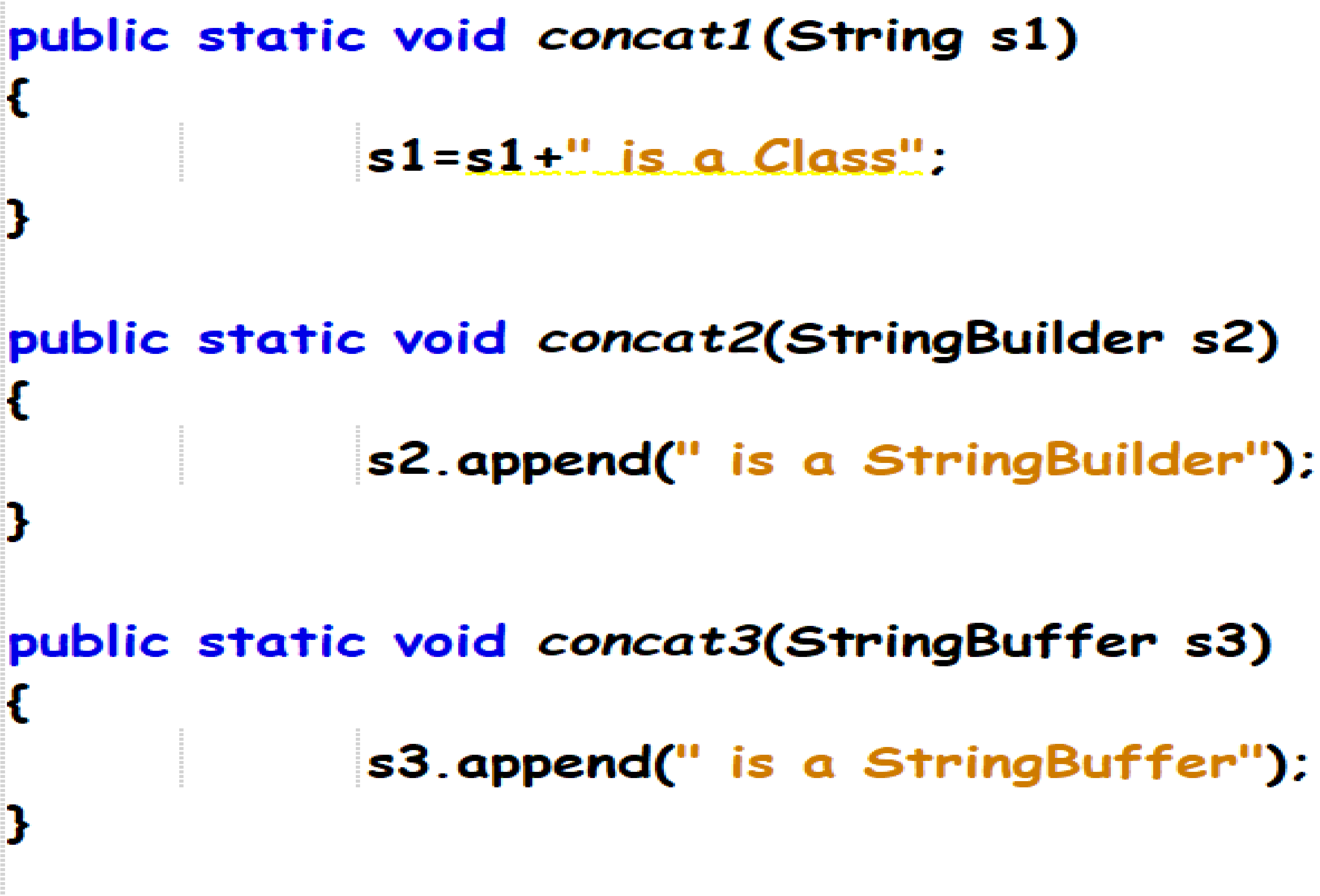
- If a string is going to **remain constant throughout the program**, then use the String class object because a String object is immutable.

- If a string can change (for example: lots of logic and operations in the construction of the string) and **will only be accessed from a single thread, using a StringBuilder** is good enough.

- If a string can **change and will be accessed from multiple threads**, use a StringBuffer because **StringBuffer is synchronous**, so you have thread-safety.

- If you don’t want thread-safety than you can also go with StringBuilder class as it is not synchronized.

Example

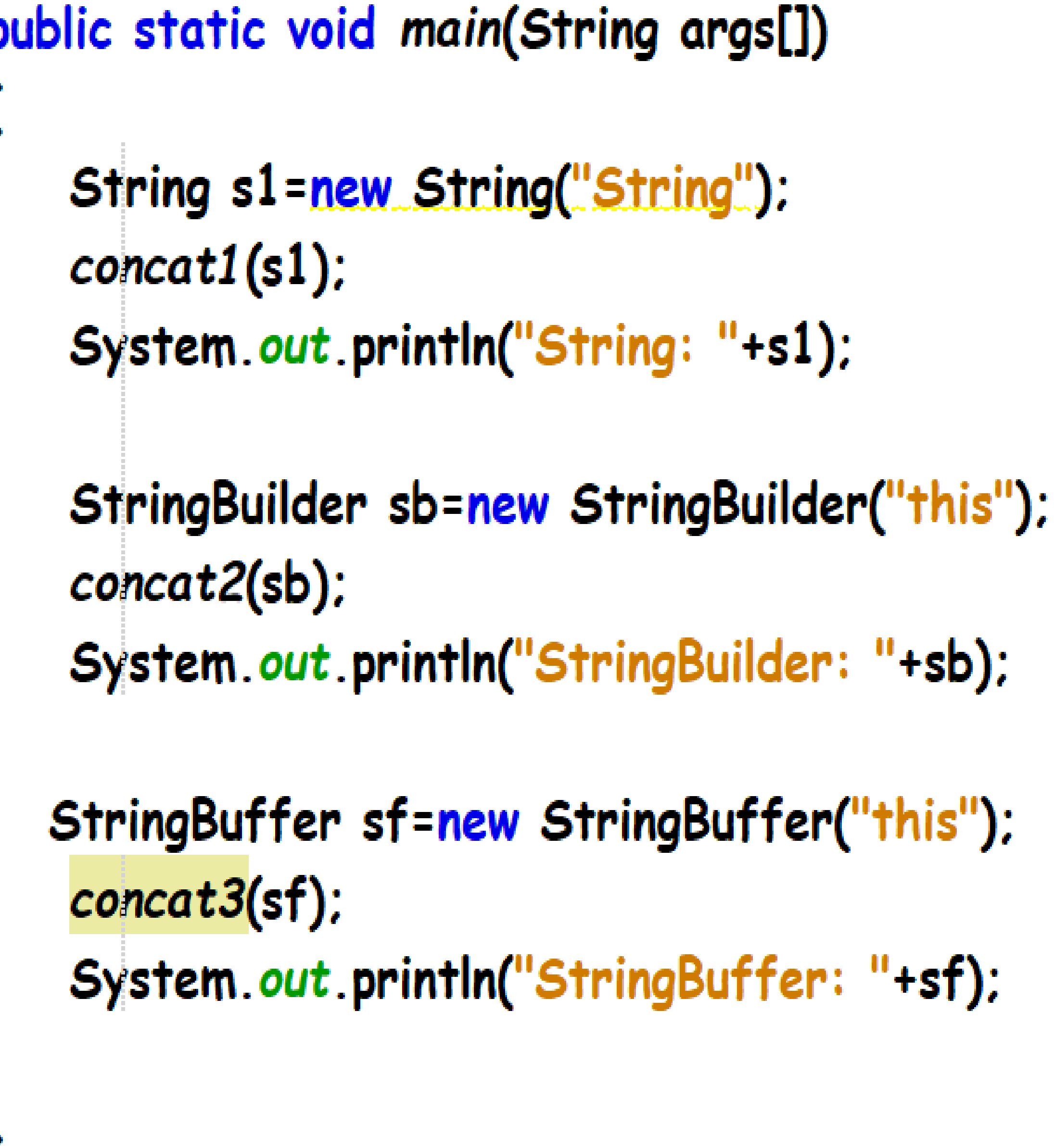


```
public static void concat1(String s1)
{
                s1=s1+" is a Class";
}

public static void concat2(StringBuilder s2)
{
                s2.append(" is a StringBuilder");
}

public static void concat3(StringBuffer s3)
{
                s3.append(" is a StringBuffer");
}
```

Example (Contd.)

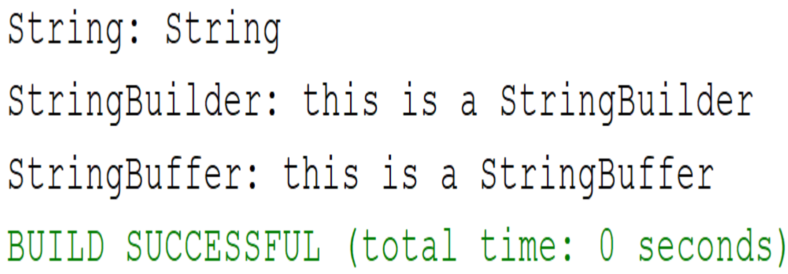


```
public static void main(String args[])
{
   String s1=new String("String");
   concat1(s1);
   System.out.println("String: "+s1);

   StringBuilder sb=new StringBuilder("this");
   concat2(sb);
   System.out.println("StringBuilder: "+sb);

  StringBuffer sf=new StringBuffer("this");
   concat3(sf);
   System.out.println("StringBuffer: "+sf);

}
```



```
String: String
StringBuilder: this is a StringBuilder
StringBuffer: this is a StringBuffer
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example (Contd.)

	String	StringBuffer	StringBuilder
Storage	Heap area, SCP	Heap area	Heap area
Objects	Immutable	Mutable	Mutable
Memory	More memory if data changes frequently	Less memory	Less memory
Thread-safe	Not thread safe (non-synchronized)	All methods are synchronized, so thread safe (multiple threads)	All methods are non synchronized, so not thread safe
Performance	Slow	Fast as compared to String	Fast as compared to StringBuffer
Uses	When data does not change frequently	Frequent data Changes	Frequent data changes