Core Java 8

Lesson 08 : String, I/O Formatting and Parsing

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to:

- Work with String Handling
- Understand new Date and Time API
- Best Practices

Lesson Objectives:
This lesson introduces to the fundamental Java API that is used in almost every type of Java applications.

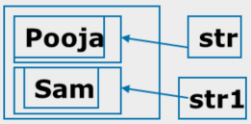
Lesson 5: Exploring Java Basics
- 5.1: The Object Class
- 5.2: Wrapper Classes
- 5.3: Type casting
- 5.4: Using Scanner Class
- 5.5: The System Class
- 5.6: String Handling
- 5.7: Date and Time API
- 5.7: Best Practices

String Handling

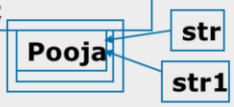
String is handled as an object of class String and not as an array of characters

- String class is a better & convenient way to handle any operation
- String objects are immutable

```
String str = new String("Pooja");
String str1 = new String("Sam");
```




```
String str = new String("Pooja");
String str1 = str;
```



String is not an array of characters but it is actually a class and is part of core API. We can use the class String as a usual data-type. It can store up to 2 billion characters.
Note: A String in java is not equivalent to character array.
Strings are built-in objects & thus have a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a sub string, concatenate two strings etc. In addition, String objects can be constructed in number of ways.

String objects are immutable objects. That is, once you create a String object you cannot change the characters, which are part of String. This seems to be a major restriction, but that is not the case. Every time you perform some modification operation on the object, a new String object is created that contains the modifications. The original string is left unchanged. Hence, the number of operations performed on one particular string creates those many string objects.

For those cases in which modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.

String - Important Methods

length(): length of string

indexOf(): searches an occurrence of a char, or string within other string

substring(): Retrieves substring from the object

trim(): Removes spaces

valueOf(): Converts data to string

isEmpty(): Added in Java 6 to check whether string is empty or not

concat(String s) : Used to concatenate a string to an existing string. Eg

```
String string = "Core ";
System.out.println( string=string.concat(" Java") );
Output -> "Core Java"
```

String.isEmpty() method added in Java 6 to check whether the given string is empty or not. Code prior to JDK 6 is as shown below:

```
//code prior to JDK 6

public boolean checkStringForEmpty(String str) {
    If(str.equals("")) {            //str.length==0
        return true;
    else
        return false;
}


//now with JDK 6 enhancement

public boolean checkStringForEmpty(String str) {
    If(str.isEmpty()) {             //much faster than the previous code
        return true;
    else
        return false;
}
```

String Concatenation

Use a "+" sign to concatenate two strings Examples:

- Example: String string = "Core " + "Java"; -> Core Java
- String a = "String"; int b = 3; int c=7
  System.out.println(a + b + c); -> String37

System.out.println(a + (b + c)); -> String10

The concat() method seen in previous page allows one string to be concatenated to another. But Java also supports string concatenation with the "+" operator.
In general, Java does not support operator overloading. The exception to this rule is the + operator, which concatenates two strings, and produces a new string object as a result.

```
class SimpleString  {
   public static void main(String args[])  {
      // Simple String Operations
      char c[] = {'J', 'a', 'v', 'a'};
      String s1 = new String(c); // String constructor using
      String s2 = new String(s1);
      // String constructor using string as arg.
      System.out.println(s1);
      System.out.println(s2);
      System.out.println("Length of String s2 : " + s2.length());
      System.out.println("Index of v : " + s2.indexOf('v'));
      System.out.println("s2 in uppercase : " + s2.toUpperCase());
      System.out.println("Character at position 2 is  : " +  s2.charAt(1));
      // Using concatenation to prevent long lines.
       String longStr = "This could have been " +
                        "a very long line that would have " +
                        "wrapped around.  But string concatenation " +
                        "prevents this.";
        System.out.println(longStr);
   } }
```

String Comparison

```
Output :  Hello equals Hello -> true
     Hello == Hello -> false
```

```
class EqualsNotEqualTo {
   public static void main(String args[]) {
        String str1 = "Hello";
        String str2 = new String(str1);
        System.out.println(str1 + " equals " + str2 + " -> " +
                   str1.equals(str2));
        System.out.println(str1 + " == " + str2 + " -> " + (str1 ==str2));
   }
}
```

The String class includes various methods that compare strings or substrings within each string. The most popularly used two ways to compare the strings is either using = = operator or by using the equals method.
The equals() method compares the characters inside a String object. The = = operator compare two object references to see whether they refer to the same instance. The program above shows the difference between the two.

StringBuffer Class

Following classes allow modifications to strings:

- java.lang.StringBuffer
- java.lang.StringBuilder

Many string object manipulations end up with a many abandoned string objects in the String pool, since String objects are immutable

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // output is "sb = abcdef"
```

Let us understand StringBuilder with an example.

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x); // output is "x = abc"
```

Because no new assignment was made, the new String object created with the concat() method was abandoned instantly. We also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x); // output is "x = abcdef"
```

We got a nice new String out of the deal, but the downside is that the old String "abc" has been lost in the String pool, thus wasting memory. If we were using a StringBuffer instead of a String, the code would look like this:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // output is "sb = abcdef“
```

Note: Refer Javadocs to know more about other methods of StringBuilder.

StringBuilder Class

Added in Java 5

Exactly the same API as the *StringBuffer* class, except:

- It is not thread safe
- It runs faster than StringBuffer

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb ); // output is "fed---cba"
```

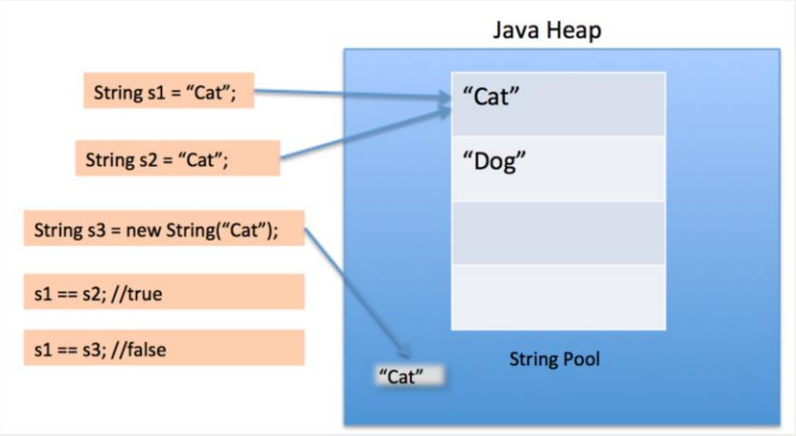
The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So, apart from
Synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa.

Note: Refer Javadocs to know more about methods of StringBuilder.

Important Facts about Strings and Memory

String is immutable :

String Pool in java is a pool of Strings stored in Heap Memory .This is possible only because String is immutable in java.



Demo

Execute the following programs:

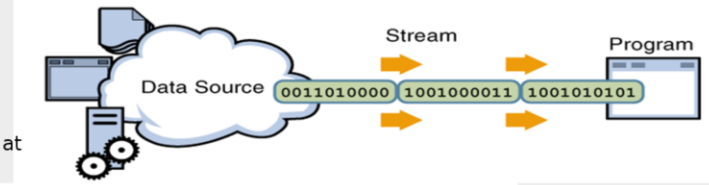
- SimpleString.java
- ToStringDemo.java
- StringBufferDemo.java
- CharDemo.java

File Navigation And IO

Most programs need to access external data.

Data is retrieved from an input source. Program results are sent to output destination.

Figure 7-1: A program uses an input stream to read data from a source, one item at a time



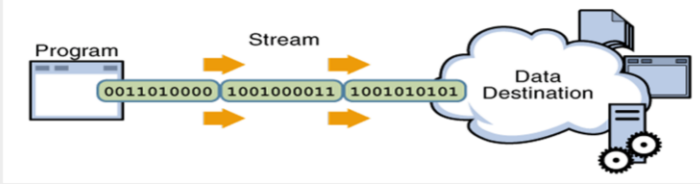


Figure 7-2:A program uses an output stream to write data to a destination, one item at time

Most programs need to use data. To read some data, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. To write some data, a program opens a stream to a data source and writes to it in a serial fashion.

Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same.

The java.io package provides an extensive library of classes dealing with input and output. Each class has a variety of member variables & methods. java.io is layered. i.e. it does not attempt to put too much capability into one class. Instead, you can get the features you want, by layering (chaining streams) one class over another.

Types of I/O Streams

Byte Streams: Handle I/O of raw binary data.

Character Streams: Handle I/O of character data. Automatic translation handling to and from a local character.

Buffered Streams: Optimize input and output with reduced number of calls to the native API.

Data Streams: Handle binary I/O of primitive data type and String values.

Object Streams: Handle binary I/O of objects.

Scanning and Formatting: Allows a program to read and write formatted text.

There are different types of I/O (Input/Output) Streams:

Byte Streams: They provide a convenient means for handling input and output of bytes. Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes descend from InputStream and OutputStream class.

Character streams: They provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

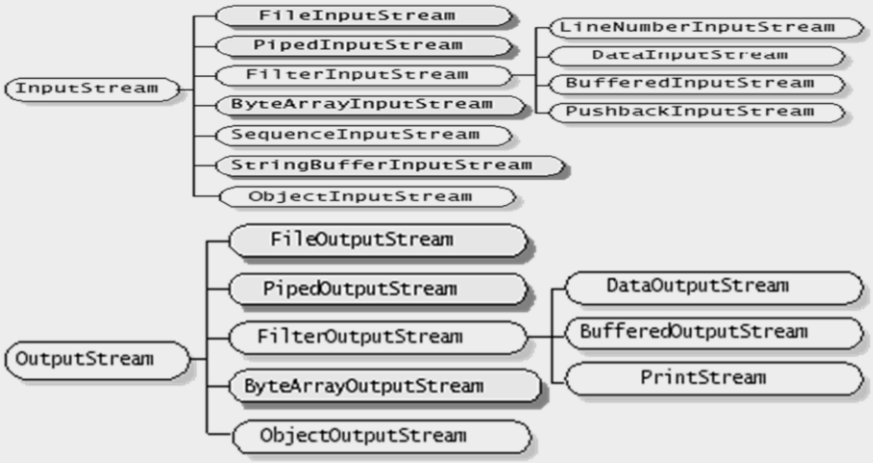
Buffered Streams: Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Data Streams: Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.

Object Streams: Just as data streams support I/O of primitive data types, object streams support I/O of objects.

Scanning and Formatting: It allows a program to read and write formatted text.

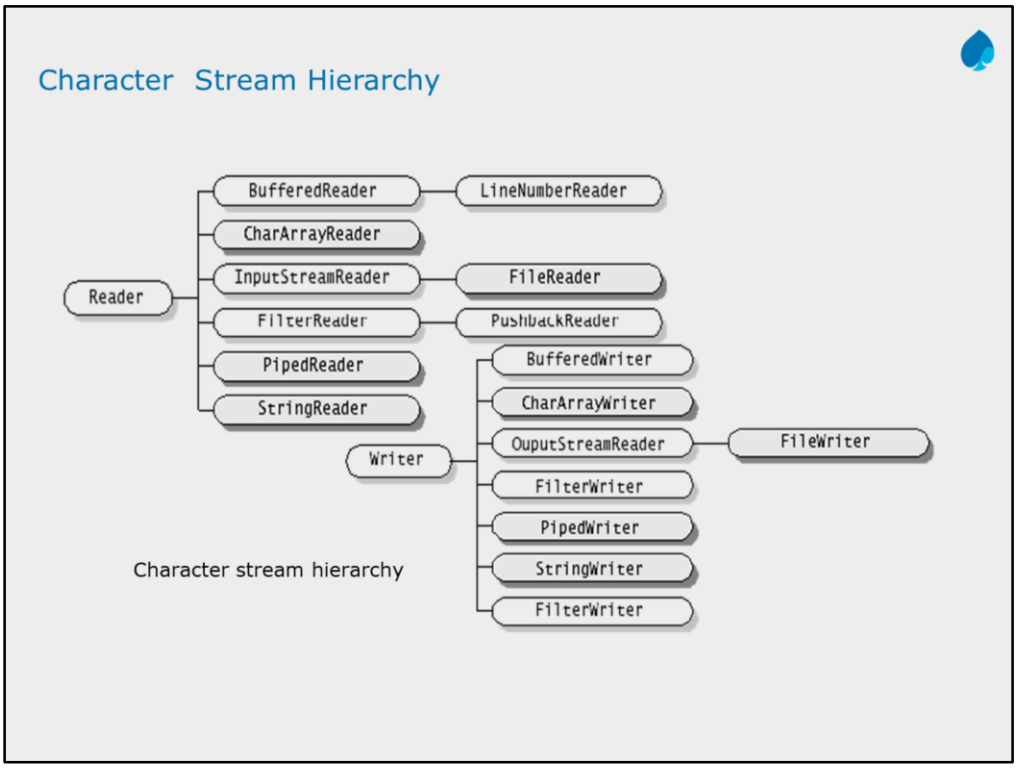
Byte Stream I/O Hierarchy



Byte-stream I/O hierarchy

At the top of the hierarchy are two abstract classes: InputStream and OutputStream.

Each of these abstract classes serves as base class for all other concretely implemented I/O classes. Each of the abstract classes defines several key methods that the other stream classes implement.



The byte stream classes support only 8-bit byte streams and doesn't handle 16-bit Unicode characters well. A character encoding is a scheme for representing characters. Java represents characters internally in the 16-bit Unicode character encoding, but the host platform might use different character encoding.
The abstract classes Reader and Writer are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding.
A reader is an input character stream that reads a sequence of Unicode characters, and a writer is an output character stream that writes a sequence of Unicode characters.

RandomAccessFile Class

This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown

Constructor	Description
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

The java.io.Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.

Serializing Objects

Object Serialization:

- Process to read and write objects.
- Provides ability to read or write a whole object to and from a raw byte stream.
- Use object serialization in the following ways:
  - Remote Method Invocation (RMI): Communication between objects via sockets.
  - Lightweight persistence: Archival of an object for use in a later invocation of the same program.

Object Serialization allows an object to be transformed into a sequence of bytes that can be later re-created (deserialized) into an original object.
Java provides this facility through ObjectInput and ObjectOutput interfaces, which allow the reading and writing of objects from and to streams. These interfaces extend DataInput and DataOutput respectively
The concrete implementation of ObjectOutput and ObjectInput interfaces is provided in ObjectOutputStream and ObjectInputStream classes respectively. These two interfaces have the following methods:
final void writeObject(Object obj) throws IOException.
final Object readObject() throws IOException, ClassNotFoundException
The writeObject() method can be used to write any object to a stream, including strings and arrays, as long as an object supports java.io.Serializable interface, which is a marker interface with no methods.
Serializing an object requires only that it meets one of two criteria. The class must either implement the Serializable interface (java.io.Serializable) which has no methods that you need to write or the class must implement the Externalizable interface which defines two methods. As long as you do not have any special requirements, making a serializable is as simple as adding the ``implements Serializable'' clause.

Example : Object Serialization

```
class Student implements Serializable{
    int roll;
    String sname;
    public Student(int r, String s){
        roll = r;
        sname = s;     }
     public String toString(){
                    return "Roll no is : "+roll+"   Name is : "+sname;
    } }
```

```
public class demo{
    public static void main(String args[]){
    try{ Student s1 = new Student (100,"Varsha");
        System.out.println("s1 object : "+s1);
```

Example: Object Serialization (contd..)

```
FileOutputStream fos = new FileOutputStream("student");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(s1);
oos.flush();
oos.close();
  }  catch(Exception e){  }
  try{
Student s2;
FileInputStream fis = new FileInputStream("student");
ObjectInputStream ois = new ObjectInputStream(fis);
s2 = (Student)ois.readObject();
ois.close();
System.out.println("s2 object : "+s2);   }
catch(Exception e){  }  }
```

Output :
s1 object : Roll no is : 100 Name is : Varsha
s2 object : Roll no is : 100 Name is : Varsha

Date ,Number And Currency

we look at the Date, Calendar, Locale, DateFormat and NumberFormat classes that allow us to create and manipulate dates, times, numbers and currencies for different regions of the world.

Class	Description
`java.util.Date`	The `Date` class allows us to create an object that represents a specific instant in time.
`java.util.Calendar`	The `Calendar` class allows us get an instance of a `Calendar` object which we can use to convert and manipulate dates and times.
`java.util.Locale`	The `Locale` class allows us to create an object that represents a specific geographical, political, or cultural region of the world. We can then use the `Locale` object in conjunction with the `DateFormat` or `NumberFormat` classes to get *locale* specific dates, times, numbers and currencies for that *locale*.
`java.text.DateFormat`	The `DateFormat` class provides us with methods to format dates in various styles and for different *locales*.
`java.text.NumberFormat`	The `NumberFormat` class provides us with methods to format numbers and currencies for different *locales*

Working with Date

```java
public class TestDate {
    public static void main(String[] args) {
        Date date1 = new Date();
        Date date2 = new Date(99999999999L);
        Date date3 = new Date(999999999999L);
        System.out.println("Our first date is: " + date1);
        System.out.println("Our second date is: " + date2);
        System.out.println("Our third date is: " + date3);
        if (date1.after(date2)) {
           System.out.println("Our first date is after our second date.")
        }
        if (date2.before(date3)) {
           System.out.println("Our second date is before our third date."
        }
    }
}
```

```
Our first date is: Tue Jan 29 11:13:03 IST 2019
Our second date is: Sat Mar 03 15:16:39 IST 1973
Our third date is: Sun Sep 09 07:16:39 IST 2001
Our first date is after our second date.
Our second date is before our third date.
```

Output

Working With Currency

```
public class CurrencyDemo {
    public static void main(String[] args) {

        Locale locale= Locale.getDefault();
        int amount=10000;
        Currency c=Currency.getInstance(locale);
        System.out.println("using currency code : "+ amount+" "+c.getCurrencyCode());
        System.out.println("using currenycy symbol : "+amount+c.getSymbol());
        System.out.println("using currency name : "+amount+" " +c.getDisplayName());

    }
}
```

```
using currency code : 10000 USD
using currenycy symbol : 10000$
using currency name : 10000 US Dollar
```

Output

Parsing ,Tokenizing And Formatting

Parsing, Tokenizing and Formatting are concepts you would use invariably in any application that is handling string data and displaying stuff on screen. The purposes of these are to help the programmer read/understand and represent data in a format that is understandable to all concerned parties.

Parsing :

```
Int a= Integer.parseInt("34");
Double d= Double.parseDouble("34.11");
```

Parsing ,Tokenizing And Formatting Continue…

Tokenizing Tokens are the actual piece of data.
2 classes for Tokenizing.

1. String class with String [] split (String regex) method.
2. Scanner class

```
public class TokenizingDemo {
    public static void main(String[] args) {

            String str="Hello i want to create string tokens...";
            String arr[]= str.split(" "); //parameter can be any regex
            for(String token: arr){
                System.out.println(token);
            }

    }
}
```

```
Hello
i
want
to
create
string
tokens...
```

Output

Locating Data with Pattern Matching

The java.util.regex package primarily consists of the following three classes:

- Pattern
- Matcher
- PatternSyntaxException

The java.util.regex package primarily consists of the following three classes:
Pattern Class: An instance of this class is a compiled representation of a regular expression. This class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
Matcher Class: An instance of this class is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. A Matcher instance is created by invoking the matcher() method on a Pattern object.
PatternSyntaxException: A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern. It has various methods like getDescription(), getIndex(), getMessage() and getPattern() that provide details of the error.

Locating Data with Pattern Matching : Example

```
public class RegExpTest {
    public static void main(String[] args) {
        String inputStr = "Test String";
        String pattern = "Test String";
        boolean patternMatched =
                                Pattern.matches(pattern,
inputStr);
        System.out.println(patternMatched);
    }
}
```

Output: true

Locating Data with Pattern Matching : Example

java.util.regex.Matcher interprets the pattern and performs match operations against an input string.
It provides a full set of methods to do the scanning.

```
String input = "Shop,Mop,Hopping,Chopping";
Pattern pattern = Pattern.compile("hop");
Matcher matcher = pattern.matcher(input);
System.out.println(matcher.matches());
while (matcher.find()){
   System.out.println(matcher.group() + ": " +matcher.start()
+ ": " + matcher.end());
}
```

Displays : false

Displays:
hop: 1: 4
hop: 18: 21

Please refer to Javadocs for details about the methods of this class.
In the code snippet above, the first output returns false since the entire input string " Shop,Mop,Hopping,Chopping " does not match the regular expression pattern “hop" and hence matches() method returns false.

Lab

Summary

In this lesson you have learnt:

- String Handling
- Best Practices

Add the notes here.

Review Questions

Question 1: String objects are mutable and thus suitable to use if you need to append or insert characters into them.

- True/False

Question 2: Which of the following static fields on wrapper class indicates range of values for its class:

- Option 1:MIN_VALUE
- Option 2: MAX_VALUE
- Option 3: SMALL_VALUE
- Option 4: LARGE_VALUE