Files and I/O

By
Apparao G

Input/Output Concepts:

Data Storage :

Transient memory : RAM

- It is accessed directly by processor.

Persistent Memory: Disk and Tape

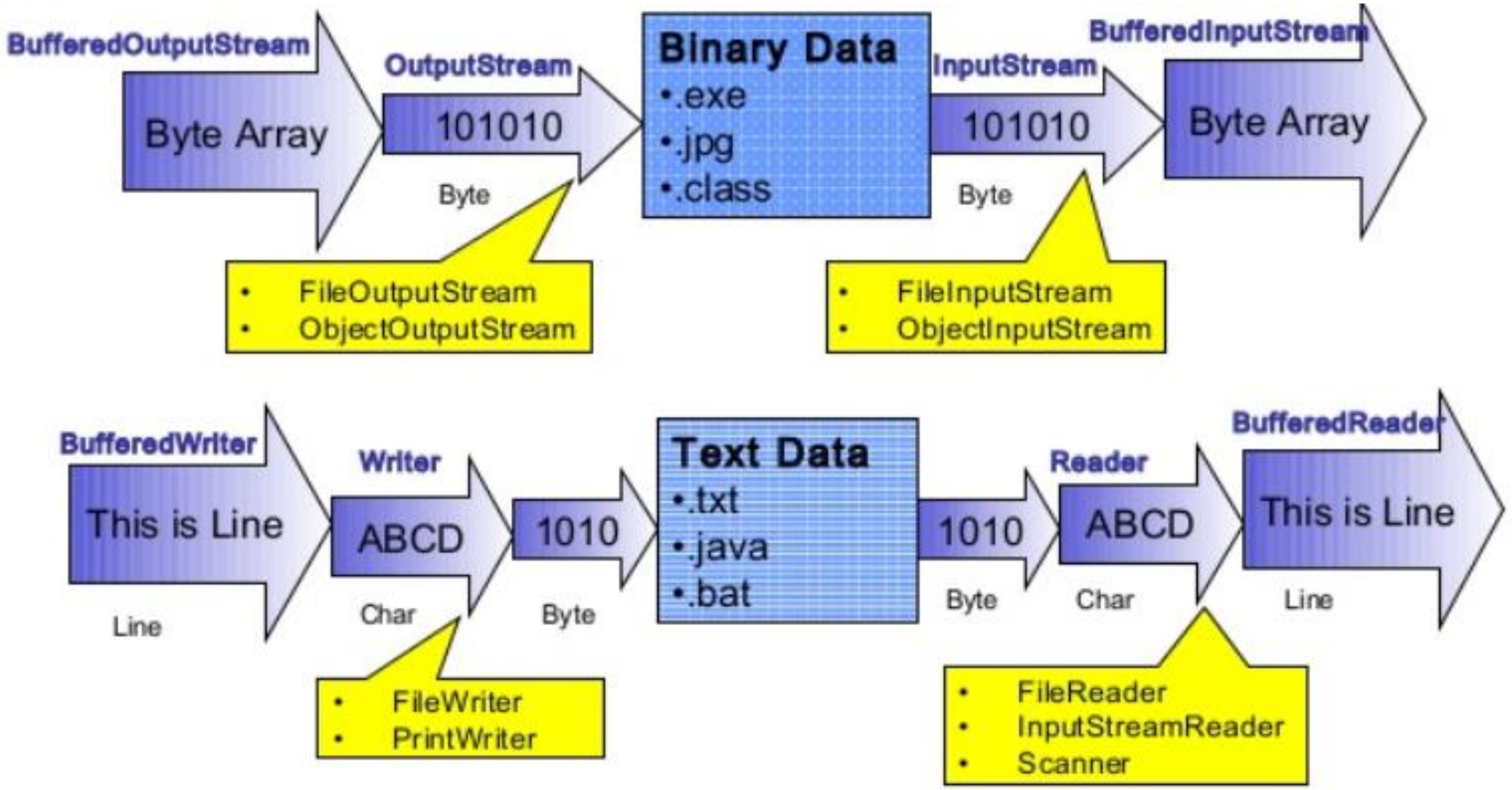
- It requires I/O operations.

I/O sources and destinations

- console, disk, tape, network, etc.

Streams

- It represents a sequential stream of bytes.
- It hides the details of I/O devices.


BufferedOutputStream
Byte Array
OutputStream
101010
Byte
FileOutputStream
ObjectOutputStream
Binary Data
•.exe
•.jpg
•.class
InputStream
101010
Byte
FileInputStream
ObjectInputStream
BufferedInputStream
Byte Array
BufferedWriter
This is Line
Line
Writer
ABCD
Char
FileWriter
PrintWriter
1010
Byte
Text Data
•.txt
•.java
•.bat
1010
Byte
Reader
ABCD
Char
FileReader
InputStreamReader
Scanner
BufferedReader
This is Line
Line

Types Of Data:

character data:

- Represented as 1-byte ASCII .
- Represented as 2-bytes Unicode within Java programs. The first 128 characters of Unicode are the ASCII characters.
- Read by Reader and its subclasses.
- Written by Writer and its subclasses.
- You can use java.util.Scanner to read characters in JDK1.5 onward.

binary data:

- Read by InputStream and its subclasses.
- Written by OutputStream and its subclasses.

Example:

```
public class FileTest {

        public static void main(String[] args) {
                File f = new File("C:/shared_folder_centos7/text.txt");
                 //File f = new File("C:/shared_folder_centos7", "text.txt");
                if (f.exists()) {
                        System.out.println("Name:" + f.getName());
                        System.out.println("Absolute path: " + f.getAbsolutePath());
                        System.out.println(" Is writable:" + f.canWrite());
                        System.out.println(" Is readable:" + f.canRead());
                        System.out.println(" Is File:" + f.isFile());
                        System.out.println(" Is Directory:" + f.isDirectory());
                        System.out.println("Last Modified : " + new Date(f.lastModified()));
                        System.out.println("Length/size :" + f.length());
                }
        }

}
```

File:

Class java.io.File represents a file or folder (directory).

Constructors:

- public File (String path)
- public File (String path, String name)

Basic methods available in File class:

- boolean exists()
- boolean isDirectory()
- boolean isFile()
- boolean canRead()
- boolean canWrite()
- long length()
- long lastModified()
- boolean delete()
- boolean renameTo(File dest)
- boolean mkdir()
- String[] list()
- File[] listFiles()
- String getName()

Write program display all files and folders in given directory

```
import java.io.File;
public class DisplayFilesInDirectories {
	public static void main(String[] args) {
		File directory = new File("C:/shared_folder_centos7");
		String[] list = directory.list();
		for (int i = 0; i < list.length; i++) {
			System.out.println(list[i]);
		}
	}
}
```

Write a program display only files from a given folder:

```
public class DisplayOnlyFilesFromDirectory {
	public static void main(String[] args) {
		File directory = new File("C:/shared_folder_centos7");
		String[] list = directory.list();
		for (int i = 0; i < list.length; i++) {
			File f = new File("C:/shared_folder_centos7", list[i]);
			if (f.isFile()) {
				System.out.println(list[i]);
			}
		}
	}
}
```

Directories in Java:

A directory is a File which can contain a list of other files and directories. You use File object to create directories, to list down files available in a directory.

Creating Directories

There are two useful File utility methods, which can be used to create directories.

The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.

The mkdirs() method creates both a directory and all the parents of the directory.

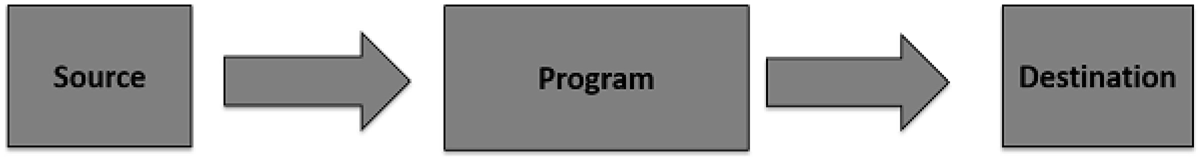
Example:

```
import java.io.File;
public class CreateDirectory {
        public static void main(String args[]) {
                String dirname = "C:/test/apparao";
                File d = new File(dirname);
                d.mkdirs();
        }
}
```

Stream :A stream can be defined as a sequence of data.

There are two kinds of Streams:

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.



InputStream:

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Frequently used methods:

public abstract int read()throws IOException

Reads the next byte of data from the input stream. It returns -1 at the end of the file.

public int available()throws IOException

Returns an estimate of the number of bytes that can be read from the current input stream.

public void close()throws IOException

It is used to close the current input stream.

OutputStream:

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes

Frequently used method:

public void write(int)throws IOException

It is used to write a byte to the current output stream.

public void write(byte[])throws IOException

It is used to write an array of byte to the current output stream.

public void flush()throws IOException

Flushes the current output stream.

public void close()throws IOException

It is used to close the current output stream.

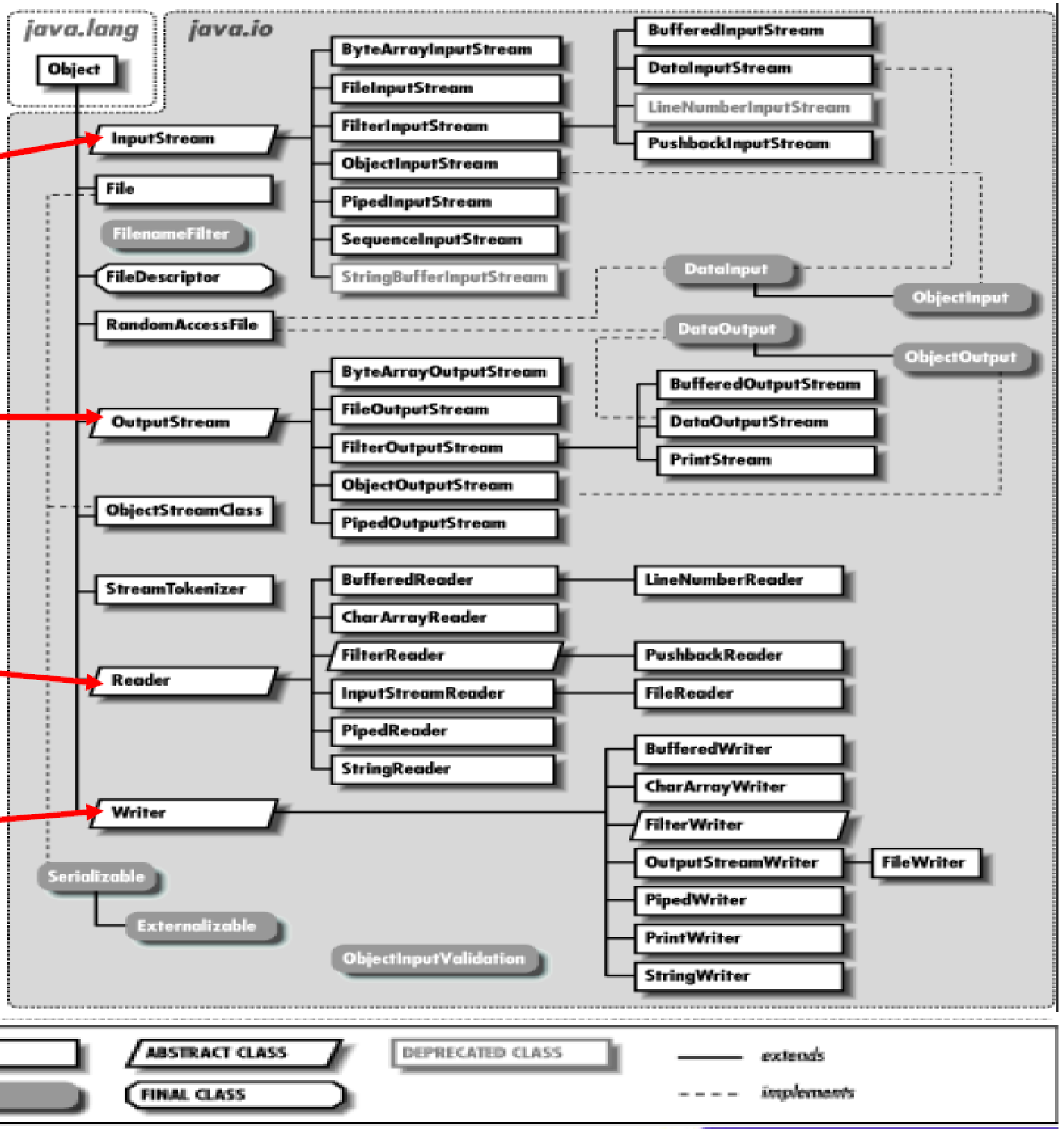
Streams

InputStream

OutputStream

Reader

Writer



Byte Stream:

Java byte streams are used to perform input and output of (8-bit) bytes.

Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream.

```
import java.io.*;
public class FileCopy {
        public static void main(String args[]) throws IOException {
                FileInputStream in = null;
                FileOutputStream out = null;
                try {
                        in = new FileInputStream("C:/test/input.txt");
                        out = new FileOutputStream("C:/test/output.txt");
                        int c;
                        while ((c = in.read()) != -1) {
                                out.write(c);
                        }
                } finally {
                        if (in != null) {
                                in.close();
                        }
                        if (out != null) {
                                out.close();
                        }
                }
        }
}
```

Character Streams: Java Character streams are used to perform input and output for 16-bit unicode.

Most frequently used classes are, FileReader and FileWriter. Internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

```
public class FileCopyUsingCharacterStream {
        public static void main(String args[]) throws IOException {
                FileReader in = null;
                FileWriter out = null;
                try {
                        in = new FileReader("C:/test/input.txt");
                        out = new FileWriter("C:/test/output.txt");
                        int c;
                        while ((c = in.read()) != -1) {
                                out.write(c);
                        }
                } finally {
                        if (in != null) {
                                in.close();
                        }
                        if (out != null) {
                                out.close();
                        }
                }
        }
}
```

Write a program read file line by line:

```
package com.test.io;
import java.io.*;
public class ReadFileLineByLine {
		public static void main(String args[]) throws IOException {
				FileReader in = null;
				BufferedReader br = null;
				try {
						in = new FileReader("C:/test/input.txt");
						br = new BufferedReader(in);
						while (true) {
								String line = br.readLine();
								if (line == null) {
										break;
								}
								System.out.println(line);
						}
				} finally {
						if (in != null) {
								in.close();
						}
						if (br != null) {
								br.close();
						}
				}
		}
}
```

Standard Streams:

Java provides the following three standard streams.

Standard Input: This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output: This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

Standard Error: This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Reads data from keyboard and writes into a file:

```
import java.io.*;
public class ReadInputFromKeyBoardWriteToFile {
        public static void main(String[] args) {
                FileWriter writer = null;  PrintWriter printWriter = null; InputStreamReader isReader = null;
                String target = "C:/test/output.txt";
                try {
                        writer = new FileWriter(target);
                        printWriter = new PrintWriter(writer);
                        // Reading input from keyboard.
                        System.out.println("Enter text here:");
                        isReader = new InputStreamReader(System.in);
                        BufferedReader in = new BufferedReader(isReader);
                        String line = in.readLine();
                        while (!line.equals("exit")) {
                                printWriter.print(line);
                                line = in.readLine();
                        }
                } catch (IOException e) {
                        System.err.print(e.getMessage());
                } finally {
                        printWriter.close();
                        try {
                            writer.close(); isReader.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

Serialization:
Serialization is a process of writing the state of an object into a byte-stream.

The reverse operation of serialization is called deserialization where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.
We need to implement the Serializable interface for serializing the object.

Advantages of Serialization:

It is mainly used to travel object's state on the network (which is known as marshaling).

Serializable interface:

Serializable is a marker interface. It has no data member and method.

Examples of marker interfaces are: Cloneable interface and Remote interface.

If you want to serialize the object, class needs to implement serializable interface.

The String class and all the wrapper classes implement the java.io.Serializable interface by default.

ObjectOutputStream:

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor:

public ObjectOutputStream(OutputStream out) throws IOException {}

Creates an ObjectOutputStream that writes to the specified OutputStream.

Methods:

public final void writeObject(Object obj) throws IOException

Writes the specified object to the ObjectOutputStream.

public void flush() throws IOException

Flushes the current output stream.

public void close() throws IOException

Closes the current output stream.

ObjectInputStream:

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor:

public ObjectInputStream(InputStream in) throws IOException {}

Creates an ObjectInputStream that reads from the specified InputStream.

Methods:

public final Object readObject() throws IOException, ClassNotFoundException

Reads an object from the input stream.

public void close() throws IOException

Closes ObjectInputStream.

Example:

```
import java.io.Serializable;
public class Employee implements Serializable {
        public int id;
        public String name;
        public String address;
        public double salary;

        public Employee(int id, String name, String address, double salary) {
                super();
                this.id = id;
                this.name = name;
                this.address = address;
                this.salary = salary;
        }
        public void displayEmployeeDetails() {
                System.out.println("Employee details  id:" + id + " name::" + name +" salary:"+salary);
        }
}
```

```
import java.io.*;
public class SerilizationTest {
        public static void main(String[] args) throws IOException {
                FileOutputStream fos = null;
                ObjectOutputStream oos = null;
                Employee e = new Employee(123, "Bob", "Miami FL", 3000);
                try {
                        fos = new FileOutputStream("C:/test/employee.ser");
                        oos = new ObjectOutputStream(fos);
                        oos.writeObject(e);
                        System.out.printf("Employee object is Serialized and data is saved in C:/test/employee.ser");
                } catch (IOException ioe) {
                        ioe.printStackTrace();
                } finally {
                        if (null != oos)
                                oos.close();
                        if (null != fos)
                                fos.close();
                }
        }
}
```

```
public class DeserilizationTest {
        public static void main(String[] args) throws IOException {
                Employee e = null;  FileInputStream fis = null;  ObjectInputStream ois = null;
                try {
                        fis = new FileInputStream("C:/test/employee.ser");
                        ois = new ObjectInputStream(fis);
                        e = (Employee) ois.readObject();
                } catch (IOException i) {
                        i.printStackTrace();
                } catch (ClassNotFoundException c) {
                        c.printStackTrace();
                } finally {
                        if (null != ois)   {ois.close(); }
                        if (null != fis) { fis.close();  }
                }
                System.out.println("Deserialized Employee object");
                System.out.println("Name: " + e.name);
                System.out.println("Address: " + e.address);
                System.out.println("id: " + e.id);
                System.out.println("salary: " + e.salary);
                e.displayEmployeeDetails();
        }
}
```

Output:

```
Deserialized Employee object
Name: Bob
Address: Miami FL
id: 123
salary: 3000.0
Employee details  id:123 name::Bob salary:3000.0
```

Points to remember with respect to serialization :

1. Serialization with respect to Inheritance:

If a class implements serializable then all its sub classes will also be serializable, otherwise serialization process will not be performed.

2. Serialization with respect to Aggregation:

If a class has a reference to another class, all the references must be Serializable otherwise serialization process will not be performed.

3. If any static data member in a class, it will not be serialized because static is the part of class not object.

4. In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization process will be failed.

5. Transient Keyword:

If you don't want to serialize any data member of a class, you can mark data member as transient.

6. SerialVersionUID:

The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, SerialVersionUID is used. The sender and receiver must have the same SerialVersionUID, otherwise, InvalidClassException will be thrown when you deserialize the object. We can also declare our own SerialVersionUID in the Serializable class.

Externalizable interface:

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format.

It is not a marker interface.

The Externalizable interface provides two methods:

public void writeExternal(ObjectOutput out) throws IOException

public void readExternal(ObjectInput in) throws IOException

java.util.Scanner:

A simple text scanner which can parse primitive data types and strings using regular expressions.

Reads character data as Strings,or converts to primitive values.

Does not throw checked exceptions.

Frequently used Constructors:

Scanner (File source) // reads from a file

Scanner (InputStream source) // reads from a stream

Scanner (String source) // scans a String

Frequently used methods:

boolean hasNext()

boolean hasNextInt()

boolean hasNextDouble()

String next()

String nextLine()

int nextInt()

double nextDouble()

```
import java.io.*;
import java.util.Scanner;
public class ScannerExample {
        public static void main(String[] args) throws Exception {
                readFileUsingScanner();
                // readInputFromKeyBoard();
        }
        private static void readFileUsingScanner() throws FileNotFoundException, IOException {
                FileReader reader = new FileReader("C:/test/input.txt");
                Scanner sc = new Scanner(reader);
                while (sc.hasNext()) {
                        System.out.println(sc.nextLine());
                }
                reader.close();
                sc.close();
        }
        private static void readInputFromKeyBoard() {
                Scanner in = new Scanner(System.in);
                System.out.print("Enter your address: ");
                String name = in.nextLine();
                System.out.println("Address is: " + name);
                in.close();
        }
}
```

StringTokenizer:

Breaks string into tokens based on delimeter string provided.

Constructor:

StringTokenizer(String str)

creates StringTokenizer with specified string.

StringTokenizer(String str, String delim)

creates StringTokenizer with specified string and delimeter.

StringTokenizer(String str, String delim, boolean returnValue)

creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods:

boolean hasMoreTokens() : checks if there is more tokens available.

String nextToken() : returns the next token from the StringTokenizer object.

String nextToken(String delim) : returns the next token based on the delimeter.

boolean hasMoreElements() : same as hasMoreTokens() method.

Object nextElement() : same as nextToken() but its return type is Object.

int countTokens() : returns the total number of tokens.

Example:

```
import java.util.StringTokenizer;
public class StringTokenizerExample {
        public static void main(String[] args) {
                String str = "Java is Object Oriented Language";
                StringTokenizer stn = new StringTokenizer(str, " ");
                String token = null;
                System.out.println("Token are : ");
                while (stn.hasMoreElements()) {
                        token = stn.nextToken();
                        System.out.println(token);
                }
        }
}
```

Output:

```
Token are :
Java
is
Object
Oriented
Language
```