CS2

Theme:

Object Oriented Programming in Java

Handout : Week12

prepared by N S kumar

Preface

Dear friends,

These notes are on time for a change!

We discuss the concept of callbacks while developing utility programs. We also discuss how to play with files. These topics are a bit deep. You may have to read some of these topics more than once to get a good feel of these topics.

All the best. Enjoy.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com

Date : 19th April 2015

Callbacks:

In the last lecture, you may recall that we discussed about making our methods flexible by using callbacks. We wrote a method to sort an array of int based on the predicate (method returning a boolean value). This method which is a predicate is part of an interface. Our sort method expected that we get an object of this interface on which we could invoke a method which would tell us whether the elements of the array should we swapped.

Effectively our method has become flexible – can work with different sorting criteria – without rewriting or even recompiling the method!

But there are a few limitations. The types of the parameters are fixed. So, component type of the array which is being sorted is also fixed. We will revisit this idea later with some more flexibility - you will have to wait for a concept called generics.

We shall now look at three special methods.

1. map.

It is extremely common to walk through an array of elements, do some operation on each element, collect the resultant into an array and then return an array. For example, my idea could be to square each element of an array and make a new array of squares of elements.

Or given an array of int, I may want to make a new array where each element is greater by 100 compared to the corresponding element of the array.

This concept is called mapping. Let us develop a method called map to do this. This method does an operation specified by the client through a callback. As to be expected, this callback takes one int argument and returns an int as the result.

Observe the following example.

```
//L45/Example1.java
import java.util.*;
public class Example1
{
        public static void main(String[] args)
        {
                int[] a = {1, 2, 3, 4, 5};
                // generates squares of each number in the array
                int[] b = Util.mymap(a, new UnaryOp()
                        {
                                public int compute(int x) { return x * x; }
                        });
                System.out.println(Arrays.toString(b));
                // adds 10 to each element in the array
                b = Util.mymap(a, new UnaryOp()
                        {
                                public int compute(int x) { return x + 10; }
                        });
                System.out.println(Arrays.toString(b));
                // cube each element in the array
                b = Util.mymap(a, new UnaryOp()
                        {
                                public int compute(int x) { return x * x * x; }
                        });
                System.out.println(Arrays.toString(b));
        }
}
```

```
class Util
{
	public static int[] mymap(int[] a, UnaryOp unaryop)
	{
		int[] res = new int[a.length];
		for(int i = 0; i < a.length; i++)
		{
			// calls back to compute to find mapping for each element
			// of the parameter array
			res[i] = unaryop.compute(a[i]);
		}
		return res;
	}
}

interface UnaryOp
{
	int compute(int x);
}
```

2. filter:

In this example, the callback method is unary as well as it is a predicate – returns a value true or false based on a single argument. We use this callback to filter out elements – select elements which satisfy the required criterion. We may want to find only odd numbers, numbers greater than a fixed number and so on.

Let us examine the code at length. Util.myfilter will faithfully return an array of elements which satisfy the given requirement from the array of elements passed as argument to it.

```
//L45/Examle2.java
// number of elements in the output less than or equal to the number of elements
// in the input
import java.util.*;
public class Example2
{
	public static void main(String[] args)
	{
		int[] a = {1, 2, 3, 4, 5};
		// find odd numbers
		int[] b = Util.myfilter(a, new UnaryOp()
			{
				public boolean compute(int x) { return x % 2 == 1; }
			});
		System.out.println(Arrays.toString(b));
		// find numbers with unit digit being 1.
		b = Util.myfilter(a, new UnaryOp()
			{
				public boolean compute(int x) { return x % 10 == 1; }
			});
		System.out.println(Arrays.toString(b));
	}
}
```

```
class Util
{
	public static int[] myfilter(int[] a, UnaryOp unaryop)
	{
		// make res the biggest possible array required.
		int[] res = new int[a.length];
		int j = 0;
		// walk through the given array and copy elements which satisfy the condition to the new array
		for(int i = 0; i < a.length; i++)
		{
			if(unaryop.compute(a[i]))
			{
				res[j++] = a[i];
			}
		}
		// return only selected elementss
		return Arrays.copyOf(res, j);
	}
}

interface UnaryOp
{
	boolean compute(int x);
}
```

You will observe that in map, the number of elements in the resultant array is same as the number of elements in the input array.
You will observe that in filter, the number of elements in the resultant array could vary from 0 to the number of elements in the input array.

3. reduce

This method will reduce a given array to a single element based on a callback which will repeatedly be applied to the elements of the array -each time reduce a pair of elements to a single element.

For example, we may want to find the sum of the elements of the array. We may start with an initial value say 0. To this we will add the 0th element of the array. To the result of this summation, we had the first element of the array. We repeat this until all the elements of the array are exhausted. By now, we would have reduced the array to a single element.

Here, the callback method will take two arguments and return a single result.

```
// L45/Example3.java
import java.util.*;
public class Example3
{
        public static void main(String[] args)
        {
                int[] a = {1, 2, 3, 4, 5};
                // Observe the callback is adding two numbers – so this will
                // return the sum of the elements of the array
                int b = Util.myreduce(a, 0, new BinaryOp()
                        {
                                public int compute(int x, int y) { return x + y; }
                        });
                System.out.println(b);
                // Observe the callback is multiplying  two numbers – so this
                // will return the product of the elements of the array
                // Also observe the initial value is 1
                b = Util.myreduce(a, 1, new BinaryOp()
                        {
                                public int compute(int x, int y) { return x * y; }
                        });
                System.out.println(b);
        // Add from an initial value of 100
        b = Util.myreduce(a, 100, new BinaryOp()
            {
                public int compute(int x, int y) { return x + y; }
            });
        System.out.println(b);
    }
}

class Util
{
    public static int myreduce(int[] a, int init, BinaryOp binaryop)
    {
        for(int i = 0; i < a.length; i++)
        {
            init = binaryop.compute(init, a[i]);
        }
        return init;
    }
}

interface BinaryOp
{
    int compute(int x, int y);
}
```

We have observed that all our methods so far were written to play with an array of int. With what we know so far, we are constrained to specify the type explicitly. Here are examples where the component type of the array is String.

In this first example, we map an array of strings to an array of int where each element in the resultant array is length of the corresponding string in the input array.
In the second example, every string is mapped to its corresponding uppercase form.

//L46/Example1.java

```
import java.util.*;
public class Example1
{
	public static void main(String[] args)
	{
		String[] a = { "tiger", "lion", "elephant", "cheeta"};
		int[] b = Util.mymapOne(a, new UnaryOpOne()
			{
				public int compute(String s) { return s.length(); }
			});

		System.out.println(Arrays.toString(b));
		String[] c = Util.mymapTwo(a, new UnaryOpTwo()
			{
				public String compute(String x) { return
					x.toUpperCase();
				}
			});
		System.out.println(Arrays.toString(c));
	}
}
```

```
class Util
{
        public static int[] mymapOne(String[] a, UnaryOpOne unaryop)
        {
                int[] res = new int[a.length];
                for(int i = 0; i < a.length; i++)
                {
                        res[i] = unaryop.compute(a[i]);
                }
                return res;
        }
        public static String[] mymapTwo(String[] a, UnaryOpTwo unaryop)
        {
                String[] res = new String[a.length];
                for(int i = 0; i < a.length; i++)
                {
                        res[i] = unaryop.compute(a[i]);
                }
                return res;
        }
}

interface UnaryOpOne
{
        int compute(String x);
}

interface UnaryOpTwo
{
        String compute(String x);
}
```

Observe the necessity of two different interfaces with two different implementations as the return types are different – one returns an int based on a String and the other a String itself.

Input-output operations:

In the good old days, sculptors would carve letters on stone – carving each letter would require umpteen number of strokes. Then came the palm leaves – made writing a bit easy. We can write on paper using pen. That is simpler to the earlier methods. Then came the letter press – where there could be mass production of printing. They are superceded by modern technologies like DTP, 3D printing etc.
It is equally true that there are number of ways of reading and writing from a file – a file provides mechanism to store the information even after the program terminates – and can be read by the same or some other program later. This concept is called serialization or persistence. I suggest that you may look at the Oracle (Sun) Tutorial.
https://docs.oracle.com/javase/tutorial/essential/io/index.html

We will go through a series of examples of input and output. We will mention the differences between them as we discuss them.

We may expect IO exceptions – file not found, no permission for read or writing …
There are a number of reasons why the input output operations could fail. We specify that our program may throw an exception and we also try to handle the exception. This feature shall be used in all the examples in this section.
We will close the opened file in finally block if the opening has succeeded. This feature is also used all the examples which follow.

1. byte streams:

we read the file byte by byte and write to another file.
FileInputStream supports read and returns -1 when the end of file is reached.

FileOutputStream supports write which writes a single byte to the output file.
These are very low level routines. This stream is a very lower stream.
This program looks a lot similar to a 'C' program reading and writing a file character by character.

```
//L47/Example1.java
// input-output:
//	please check Oracle tutorial
//	streams:
//		bytestreams
//			FileInputStream extends InputStream
//			FileOutputSteam extends OutputStream
//	create the input file
//	run the pgm
//	compare the input and the output files



import java.io.*;
public class Example1
{
	public static void main(String[] args) throws IOException
	{
		FileInputStream fin = null;
		FileOutputStream fout = null;
		try
		{
			fin = new FileInputStream("test.txt");
			fout = new FileOutputStream("newtest.txt");
			int ch;
			while((ch = fin.read()) != -1)
			{
				fout.write(ch);
```

```
                **}**
            }
            catch(IOException e)
            {
                System.out.println(e);
            }
            finally
            {
                if(fin != null)
                    fin.close();
                if(fout != null)
                    fout.close();
            }
        }
}
```

2. **Character Streams**

This treats the character as the smallest unit of reading and writing.

FileReader allows reading a character – could be unicode.

FileWriter allows writing a unicode character into the file.

Otherwise this is similar to bytestreams.

```
//L47/Example2.java
// input-output:
//	please check Oracle tutorial
//	streams:
//		bytestreams
//			FileInputStream extends InputStream  : read()
//			FileOutputSteam extends OutputStream : write(byte)
//		characterStreams
//			FileReader extends Reader
//			FileWriter extends Writer
//				can support unicode
//

import java.io.*;
public class Example2
{
	public static void main(String[] args) throws IOException
	{
		FileReader fin = null;
		FileWriter fout = null;
		try
		{
			fin = new FileReader("test.txt");
			fout = new FileWriter("newtest.txt");
			int ch;
			while((ch = fin.read()) != -1)
			{
				fout.write(ch);
			}
		}
```

```
        catch(IOException e)
        {

            System.out.println(e);
        }
        finally
        {
            if(fin != null)
                fin.close();
            if(fout != null)
                fout.close();
        }
    }

}
```

3. Line oriented streams

These streams provide mechanism to read line and write line. We can also make a string and write it in one stroke. This concept of buffering helps in matching memory speeds with I/O speeds.

//L47/Example3.java

```
// input-output:
//      please check Oracle tutorial
//      streams:
//          bytestreams
//              FileInputStream extends InputStream  : read()
//              FileOutputSteam extends OutputStream : write(byte)
//          characterStreams
//              FileReader extends Reader
//              FileWriter extends Writer
//                  can support unicode
//          Line Oriented IO:
```

```
//                      provides line functionality over CharacterStreams
//                      BufferedReader
//                      PrintWriter
import java.io.*;
public class Example3
{
        public static void main(String[] args) throws IOException
        {
                BufferedReader fin = null;
                PrintWriter fout = null;
                try
                {
                        fin = new BufferedReader(new FileReader("test.txt"));
                        fout = new PrintWriter(new FileWriter("newtest.txt"));
                        String ch;
                        int i = 0;
                        while((ch = fin.readLine()) != null)
                        {
                                fout.println(++i + " : " + ch);
                        }
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        if(fin != null) fin.close();
                        if(fout != null) fout.close();
                }
        }
}
```

4. **Scanner object**

In the earlier cases, we used read the input as characters. This scanner class allows to specify whether we expect an int or a double or a string as the next input. The input is broken into parts – called tokenizing.

We ask whether there are more elements to read. Check the condition of the while loop. If there are more elements, we specify the type of input we expect next.

```
        String s;
        while(fin.hasNext()) // is not end of file
        {
                s = fin.next(); // get the next String
                fout.println(s);
        }
```

```
//L47/Example4.java
// input-output:
//      please check Oracle tutorial

//      streams:
//              bytestreams
//                      FileInputStream extends InputStream  : read()
//                      FileOutputSteam extends OutputStream : write(byte)
//              characterStreams
//                      FileReader extends Reader
//                      FileWriter extends Writer
//                              can support unicode
//              Line Oriented IO:
//                      provides line functionality over CharacterStreams
//                      BufferedReader
//                      PrintWriter
//              Scanner:
```

```
//                        break input into tokens

import java.io.*;
import java.util.*;
public class Example4
{
        public static void main(String[] args) throws IOException
        {
                Scanner fin = null;
                PrintWriter fout = null;
                try
                {
                        fin = new Scanner(new BufferedReader(
                                        new FileReader("test.txt")));
                        fout = new PrintWriter(new FileWriter("newtest.txt"));
                        String s;
                        while(fin.hasNext())
                        {
                                s = fin.next();
                                fout.println(s);
                        }
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        if(fin != null)
                                fin.close();
                        if(fout != null)
                                fout.close();
```

```
		}
	}
}
```

5. Data Streams

These streams allow values of different types to be written into the files.

There are different methods to write data of different types.

You may use cat or od command to check what the created new file contains!

//L48/Example1.java

```
// input-output:
//	please check Oracle tutorial
//	data streams:
//		can have combination of different types
//		all simple types and strings
// create a data stream
import java.io.*;
import java.util.*;
public class Example1
{
	public static void main(String[] args) throws IOException
	{
		DataOutputStream fout = null;
		String[] s = { "gavaskar", "srikanth", "amarnath", "kapildev" };
		int[] runs = {  36, 66, 85, 175 };
		try
		{
			fout = new DataOutputStream(new BufferedOutputStream(
					new FileOutputStream("newtest.txt")));
```

```
                        for(int i = 0; i < s.length; i++)
                        {
                                fout.writeUTF(s[i]);
                                fout.writeInt(runs[i]);
                        }
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        if(fout != null)
                                fout.close();
                }
        }
}
```

Even though the file appears to contain junk when we cat the file, we can read the content of the file using data streams. The way we check for the end of file here is unusual. We keep reading until an exception is thrown.

```
//L48/Example2.java
// input-output:
//      please check Oracle tutorial
//      data streams:
//              can have combination of different types
//              all simple types and strings

// read a data stream
```

```
import java.io.*;
import java.util.*;
public class Example2
{
        public static void main(String[] args) throws IOException
        {
                DataInputStream fin = null;
                String s;
                int r;
                try
                {
                        fin = new DataInputStream(new BufferedInputStream(
                                        new FileInputStream("newtest.txt")));
                        while(true)
                        {
                                s = fin.readUTF();
                                r = fin.readInt();
                                System.out.println(s +  " : " + r);
                        }
                }
                catch(IOException e)
                {
                        //System.out.println(e);
                }
                finally
                {
                        if(fin != null) fin.close();
                }
        }
}
```

6. Object Streams

In this example, we will store the objects into a file and bring them back later.

This serialization of objects requires that we implement the marker interface called Serializable. AS in the case of Cloneable interface, JVM will take care of all the operations. We only have to indicate that we want this service.

In this example, we create an array of 4 objects. We write the object into a file.

//L48/Example3.java

```
// create a serialization of object(s)
import java.io.*;
import java.util.*;
public class Example3
{
	public static void main(String[] args) throws IOException
	{
		ObjectOutputStream fout = null;
		String[] s = { "gavaskar", "srikanth", "amarnath", "kapildev" };
		int[] runs = {  36, 66, 85, 175 };
		Score[] score = new Score[4];
		for(int i = 0; i < score.length; i++)
		{
			score[i] = new Score(s[i], runs[i]);
		}
		try
		{
			fout = new ObjectOutputStream(new BufferedOutputStream(
					new FileOutputStream("out.txt")));
			for(int i = 0; i < score.length; i++)
			{
				fout.writeObject(score[i]);
			}
		}
		catch(IOException e)
```

```
        {
            System.out.println(e);
        }
        finally
        {
            if(fout != null)
                fout.close();
        }
    }
}

// Serializable is a marker interface ; no need to implement any method in the
class
class Score implements Serializable // marker interface
{
    private String name;
    private int runs;
    public Score(String name, int runs)
    {
        this.name = name;
        this.runs = runs;
    }
    public String toString()
    {
        return name + ":" + runs;
    }
}
```

In this example, we read from the object created in the last example as an Object and downcast to the right class. Now the object is ready for consumption.

```
//L48/Example4.java
// input-output:
import java.io.*;
import java.util.*;
public class Example4
{
	public static void main(String[] args)
		throws IOException, ClassNotFoundException
	{
		ObjectInputStream fin = null;
		try
		{
			fin = new ObjectInputStream(new BufferedInputStream(
					new FileInputStream("out.txt")));
			while(true)
			{
				Score s = (Score)fin.readObject();
				System.out.println(s);
			}
		}
		catch(IOException e)
		{
			System.out.println(e);
		}
		finally
		{
			if(fin != null) fin.close();
		}
	}
}
```

```
class Score implements Serializable
{
        private String name;
        private int runs;
        public Score(String name, int runs)
        {
                this.name = name;
                this.runs = runs;
        }

        public String toString()
        {
                return name + ":" + runs;
        }
}
```

The input-output requires the knowledge of the class hierarchy. Do not try to memorize. Whenever you are required to use them, I am sure you will have the Java documentation and the tutorial with you.
So, do not memorize the class hierarchy. Do not waste your grey matter to remember all this terrible stuff – use that to remember some nice incidents which you may want to recall sometime later in your life. Make those incidents persist in your brains.