CS2

Theme:

Object Oriented

Programming

in

Java

Handout : Week13

prepared by N S kumar

Preface

Dear friends,

For a change, this document is being prepared before its time!

we introduce to a new paradigm called generic programming and a library called Java Collections. You may have to read these documents more than once to get a good feel of these concepts.

Please read the ending note – epilogue and provide us with a constructive feedback. Let us also know whether any of you want to volunteer to improve these documents during the summer time.

All the best.

N S Kumar

Visiting Professor

Dept. Of Computer Science and Engineering

PES Institutions.

Email : ask.kumaradhara@gmail.com

Date : 19th April 2015

Boxing and Unboxing:

There are 8 primitive types in Java. Java supports single rooted hierarchy where it is possible to trivially convert an object of any class to a reference to the celestial class Object. Most of the methods in Java are overloaded for 9 different types – there are 8 overloads to take care of all the primitive and one to take care of all reference types. The one which takes care of reference types can invoke a method polymorphically.

There are cases where we may have to convert a value of a primitive type to an object reference. To support this feature, java provides classes corresponding to each of the eight primitive types. Conversion of a value of a primitive to its corresponding reference type is called **Boxing**. Conversion of a value of a reference type back its corresponding primitive type is called **Unboxing**.

When we assign a value of primitive to a reference of its corresponding reference type, the boxing happens automatically – is called **autoboxing**. Conversion of a value in a reference type to its corresponding primitive type happens when a reference is assigned to a variable of primitive type. This is called **autounboxing.**

//L49/Ex.java

```
// Boxing and unboxing:
// autoboxing and autounboxing
public class Ex
{
	public static void main(String[] args)
	{
		int a = 10; // primitive type
		Integer b = new Integer(20); // reference type
	// Integer : wrapper class for primitive int type
		// boxing : primitive -> corresponding reference type
		Integer c = 30; // c = new Integer(30)
		// unboxing : reference type -> corresponding primitive
		int d = c;
	}
```

}

Some box this:

Let us make our own box to put something in and take it out later.

We may want to put any thing in the world. We know anything - including

values of primitive types by boxing – can become Object. So we create a box which can hold an Object. We have a put method in our box for the client to put something into it. How can the client retrieve this object which he has put in? Our Box can only return an Object – the client should remember what he put in and therefore downcast to the right type this Object on calling the get method.

Let us state these couple of points.

1. Client should downcast to the right type
2. If the client downcasts to the wrong type, this will cause an exception at runtime – ClassCastException

//L49/Ex1.java

```
// box of object
//        requires casting
//        can result in runtime exception

public class Ex1
{
        public static void main(String[] args)
        {
                Box box = new Box();
                String s = "pes";
                box.put(s);
                // require downcasting
                String s1 = (String)box.get();
                System.out.println(s1);

                box.put(20); // box.put(new Integer(20))
```

```
                int x = (int)box.get();
                System.out.println(x);
                // wrong type; compiler cannot make out
                // runtime exception : classcast
                s1 = (String)box.get();
                System.out.println(s1);
        }
}

class Box
{
        private Object o;
        public void put(Object o)
        {
                this.o = o;
        }
        public Object get()
        {
                return this.o;
        }
}
```

Generic Box:

Can we create a Box which is specially made for the type of object we would like to put? Can we specify what type of object we would like to put at the point of instantiating the class? We can. This opens a new paradigm of programming called Generic Programming. (In fact, a full fledged 4 credit elective course has been offered in our department). We will just have glimpses of this paradigm.

class Box<T> indicates that this generic type requires a type parameter while creating an object of the class. Conceptually at compile time, for every type we specify, a new Box is created. At the end of compilation in Java, all these T are replaced by Object type. Effectively at runtime we will have only

class called Box. This feature is called type erasure. This unusual concept in Java is a serious limitation in generic programming.

As every type will be reduced to Object type, we can instantiate these classes with reference types only and we cannot instantiate with primitive types.

At compile time, java knows the type of Object being put in. Therefore, downcasting is not required. If the client casts the Object wrongly while getting from the box, he will get a compile time error and not a runtime exception.

We use identifiers like T, U for these placeholders for type. This is only a convention.

Observe the changes in the code.

//L49/Ex2.java

```
// generic:
//        no casting while accessing
//  compile time errors instead of runtime errors
//                  if the types do not match
// - make generics of only reference types
// - all types are reduced to Object type at the end
//        of compilation
//        - type erasure




public class Ex2
{
        public static void main(String[] args)
```

```
	{
		Box<String> box = new Box<String>();
		String s = "pes";
		box.put(s);
		String s1 = box.get(); // no casting
		System.out.println(s1);
//		Box<int> box1 = new Box<int>(); // NO primitive types
		Box<Integer> box1 = new Box<Integer>();
		box1.put(20);
		int x = box1.get(); // OK; no casting required
		System.out.println(x);
//		if the types do not match, we get a compile time error
//		String s2 = box1.get(); // Compile time error
	}
}
// T is a place holder for some reference type
class Box<T>
{
	private T o;
	public void put(T o)

	{
		this.o = o;
	}
	public T get()
	{
		return this.o;
	}
}
```

Generic Collection Library:

We have seen earlier that there are many algorithms which are required to be used day in and day

out. These are made available in the libraries. You are not required to code them at all. You can just use them. We have seen Arrays.sort utility method to sort an array of objects.
Array provides one way of storing objects. The size is fixed at the point of creation. The access to an element takes the same constant time and it does not depend on the position of the element in the array.

There are a number of various ways the objects can be stored. Java provides libraries where we can store and manipulate a number of objects. These are called collections in java. These also support the client to specify the component type at the point of creation. These collections are also generic.

The arrangement of these components could vary. We may arrange the books on our book shelves in different ways. We may keep vertically one by the side of another. We may keep them one above the other. Each way of arrangement will have its own advantages and disadvantages. There is no unique way of arrangement at all.

To support the required operations, these collections provide a well defined interface – having different ways of implementation. An interface may have multiple implementations.

You may want to check the Java Collection Tutorial.
https://docs.oracle.com/javase/tutorial/collections/TOC.html

Let us briefly examine two such collections.

a) List :

The list data structure supports dynamic size – we can add and remove elements. It also provides methods to check whether the element exists. There are a number of other interfaces available. It is sufficient at this point to note that the List interface is implemented as an ArrayList or a LinkedList. Let us chose one of them and not worry how they are internally implemented in this course.

b) Set

The Set data structure also supports dynamic size – we can add and remove elements. The set stores unique elements. It does not store the duplicates. The set interface has three different

implementations. Let us use one of them and forget about the rest as far as this course is concerned.

This unit is just an introduction. Treat this chapter in that sense. It is just the tip of the ice berg.

An example of usage of List Collection:

Given a file containing a number of lines – each line containing name of a city and a place of interest in the city, find the unique names of cities. The city names do repeat. A city may have more than one place of attraction.

We read a file using Scanner object, If the file has some more input lines, we read two words each time in the loop. The first is the city and the second in the place of interest in the city.

We create a list of String to hold the names of the city. Observe that on the left of assignment we use an interface and on the right of assignment, we create an object of the class implementing that interface.

```
List<String> citylist = new ArrayList<String>();
```

The List interface provides a method called add. We can add the city which is a String to the citylist each time we encounter a city. But this will end up with duplicates. How to avoid this?

The List interface provides a search method called indexOf – similar to what Strings provide. If the element is found, this method returns the position, else -1. So we add the city only if the city is not already present in the list.

```
if(citylist.indexOf(city) == -1)
        citylist.add(city);
```

We used the enhanced for loop to walk through the collection.

```
for(String s : citylist)
```

```
        {
            System.out.println(s);
        }

import java.util.*;
import java.io.*;
public class Ex1
{
    public static void main(String[] args)
    {
        Scanner in = null;
        List<String> citylist = new ArrayList<String>();
        try
        {
            String city;
            String place;
            in = new Scanner(new File("places.txt"));
            while(in.hasNext())
            {
                city = in.next();
                place = in.next();
                //System.out.println(city);
                //System.out.println(place);

                if(citylist.indexOf(city) == -1)
                    citylist.add(city);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
```

```
                }
                finally
                {
                        if(in != null)
                                in.close();
                }

                for(String s : citylist)
                {
                        System.out.println(s);
                }
                //System.out.println(citylist.indexOf("mysuru"));
                //System.out.println(citylist.indexOf("mysore"));
        }
}
```

An example of usage of Set Collection:

The Set is an ideal data structure to remove the duplicates and store the elements uniquely – we call this deduplication. The reading of file is similar to the one in the earlier example.

We choose one of the implementations of the Set interface and create a Set object.

Set<String> cityset = new TreeSet<String>();

We add elements uniquely by using the add interface of the Set Collection.

cityset.add(city);

We use the enhanced for loop to walk through the collection.

```
for(String s : cityset)
                {
                        System.out.println(s);
```

```
                }
import java.util.*;
import java.io.*;
public class Ex2
{
        public static void main(String[] args)
        {
                Scanner in = null;
                Set<String> cityset = new TreeSet<String>();
                try
                {
                        String city;
                        String place;
                        in = new Scanner(new File("places.txt"));
                        while(in.hasNext())
                        {
                                city = in.next();
                                place = in.next();
                                //System.out.println(city);
                                //System.out.println(place);
                                cityset.add(city);
                        }
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
                finally
                {
                        if(in != null)
                                in.close();
```

```
                }
                for(String s : cityset)
                {
                        System.out.println(s);
                }

                //System.out.println(citylist.indexOf("mysuru"));

                //System.out.println(citylist.indexOf("mysore"));
        }
}
```

Generic Collection Algorithms:

sort:

In the last section, we discussed a couple of data structures: the List and the Set.
The libraries also provide utility methods which can process a number of these collections. We shall examine a couple of them in this course.

In this example, we create a List of Strings. We sort them by using the generic method of the Collections. This will arrange the strings in the lexicographic order.

Collections.sort(mylist);

Can this sorting be flexible? Can we sort based on the length of the Strings? Can we sort ignoring the case? I am sure we can come out with various ways of sorting.

We have seen earlier that Arrays.sort requires that we implement an interface called Comparable – override the method compareTo in the component class of the Array.

In case of Generic Collections, there is a Generic Interface called Comparator.

```
interface Comparor<T>
{
        int compare(T x, T y);
}
```

Collections.sort is overridden to take an object implementing this interface. This method compare will decide the order of elements in sorting.

```
import java.util.*;
public class Example1
{
        public static void main(String[] args)
        {
                List<String> mylist = new ArrayList<String>();
                String[] a = { "dhoni", "raina", "kohli", "yuvaraj", "rahane" };
                for(String e : a)
                {
                        mylist.add(e);
                }
                System.out.println(mylist);
                Collections.sort(mylist);

                System.out.println(mylist);
                Collections.sort(mylist, new Comparator<String>()
                {
                        public int compare(String lhs, String rhs)
                        {
                                return lhs.length() - rhs.length();
                        }
```

```
        });

        System.out.println(mylist);
        Collections.sort(mylist, new Comparator<String>()
        {
            public int compare(String lhs, String rhs)
            {
                return lhs.charAt(lhs.length() -1)
                - rhs.charAt(rhs.length() - 1);
            }
        });
        System.out.println(mylist);
    }
}
```

binarySearch:

Collections also provide an interface to do an efficient search on a sorted collection. Think of the way we search in a dictionary for a word like zebra. Definitely we would not start from the first page. We start somewhere in the middle and based on the words in that page, we discard the left or the right portion for further searching. The Collections.binarySearch does the same thing. Do remember that this works only on sorted collections.

In this example, we sort a List. Then we search for an element which exists. We get the position as an index – positive integer.

```
int res = Collections.binarySearch(mylist, "raina");
```

Then we search for an element which does not exist. We get a negative integer value indicating the element is not found. But the negation of the value indicates where it could have been if found. Then we can insert that element if we want to.

res = Collections.binarySearch(mylist, "sehwag");

System.out.println("res : " + res);

mylist.add(-res-1, "sehwag");

The Generic method binarySearch can also take a Comparator as the second argument. In that case, the searching is based on that predicate. The requirement is that the Generic Collection is sorted using the same predicate.

```
import java.util.*;
public class Example2
{
	public static void main(String[] args)
	{
		List<String> mylist = new ArrayList<String>();
		String[] a = { "dhoni", "raina", "kohli", "yuvaraj", "rahane" };
		for(String e : a)
		{
			mylist.add(e);
		}
		Collections.sort(mylist);
		System.out.println(mylist);
		int res = Collections.binarySearch(mylist, "raina");
		System.out.println("res : " + res);
		res = Collections.binarySearch(mylist, "sehwag");
		System.out.println("res : " + res);
		mylist.add(-res-1, "sehwag");
		System.out.println(mylist);
		System.out.println("----------------------------");

		// Sort based on the last character
```

```
        Collections.sort(mylist, new Comparator<String>()
        {
            public int compare(String lhs, String rhs)
            {
                return lhs.charAt(lhs.length() -1)
                - rhs.charAt(rhs.length() - 1);
            }
        });
        System.out.println(mylist);
        // do a binarySearch based on the last character.
        // In this case,  search  for a name ending with i
        // So, we may not find dhoni !
        res = Collections.binarySearch(mylist, "dhoni",
            new Comparator<String>() {
                public int compare(String lhs, String rhs)
                {
                    return lhs.charAt(lhs.length() -1)
                    - rhs.charAt(rhs.length() - 1);
                }
            });
        System.out.println("res : " + res + ":" + mylist.get(res));
    }
}
```

We shall end the lecture notes at this point. Trust you find the information discussed in these notes helpful.

All the best.

Epilogue

The Lecture Notes for this semester have become ready. Hopefully these lecture notes have made your journey in learning Object Oriented programming through Java easier and enjoyable. Trust this has kindled the light of thinking in you. I can imagine seeing all of you as bright starts in the horizon of future.

Let me acknowledge a few who have made this happen.
It was the continued support and the encouragement of our CEO and Pro-Chancellor Prof. D. Jawahar. Each time I meet him, I get inspired to try some new experiment.

I am grateful to our HODs Prof. Nitin Pujari and Dr. Shylaja, who not only supported this activity, but also read through these notes -should have been so boring for them - and always had a few words of encouragement for me.

I also would like to thank Dr. Ram Rustogi, Dr. Srinath and many others who gave me very valuable inputs without which the notes would not have been in this shape.

I want to thank my team of teachers – who have tried their best to deliver the lectures and conduct the lab sessions – as per my whims and fancies. I am sorry if I have been a bit harsh at times with any of them.

Special mention has to be made of two teachers without whom the course would have been a terrible failure. Mrs. Ranjani and Mrs. Vidhu – in spite of their family commitments – gave all their time to me. I do not know how many times I have troubled them through out the course. I run short of words to express my gratitude to them. Thanks to you and all the best.

Three of my young friends went through the notes every week – spotted all the mistakes – helped me correct them. I really appreciate their help. These young students have sacrificed their valuable time to help me. I just want to tell them – 'keep the spirit of helping others'. Thanks to you – Ullas, Sneha and Sharon.

We come to the end of this journey. It is just a comma and not a period. Your journey goes on beyond this course. I have placed this notes in your hands. It just happened. It is not mine. "Na mama". It is for you all to improve these notes. I can try to help you. Let me know whether any of you want to volunteer to make this document – which is yours – better.

Now the ball is in your court. You will have to work hard. There is no single simple way of learning other than hard work. Learning is one percent inspiration and 99 percent perspiration.

A deer will not enter into the mouth of a sleeping lion.
udyamena hi sidhyanti kAryANi na manorathaiH |
na hi suptasya siMhasya pravishanti mukhe mRigAH ||
उद्यमेनकार्याणि न मनोरथैः ।
न हि सुप्तस्य सिंहस्य प्रविशन्ति मुखे मृगाः ।।This Subhashita captures the essence of knowledge – a wealth which will increase as you spend it. Please help others learn programming well.

Na chora haryam, Na cha raja haryam
Na bhratu bhajyam Na cha bhajakari
Vyaye krute vardharta eva nityaam
Vidya dhanam sarva dhane pradhanam

No one can steal it, not authority can snatch,
Not divided in brothers, not heavy to carry,
As you consume or spend, it increases; as you share,
it expands,
Education (Vidhya) is the best wealth among all the wealth anyone can have.

I will appreciate if you can give me a feedback as how this or any other similar notes be made better.
Please send your feedback to my email : ask.kumaradhara@gmail.com
State your name(optional), section, branch of Engineering.
State some three points you liked about this couse.

State at least three points you did not like about this course.

State some three points you liked about this notes.

State at least three points you did not like about this notes.

State how the course and the lectures can be improved.

Let me also know whether you are interested to contribute towards these sort of activities.

I am expecting that my mail box will be full in the next couple of days.

All the best.

May all of you learn programming well.

May all of you do well in your examinations.

May all of you have blissful joy.