CS2

Theme:

Object Oriented Programming in Java

Handout : Week11

prepared by N S kumar

Preface

Dear friends,

In this notes, I will discuss the concept of packages and Inner classes.

Sorry for the delay. There is nothing much to state at this point.

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

class files:

We know that when we compile programs in Java using the program javac, we get one or more class files. On compiling a file called Test.java, we get a file called Test.class. To run this program, we would use the command java and specify this name Test as argument. This program is run by the jvm – Java Virtual Machine.

Do we require the Test.java to run the program? NO. In fact, these files can be ported to some other operating system running on a different hardware as long as that computer also has the java virtual machine of the same version.

Would the program work if the Test.class file is moved to a different directory? How does the JVM know where to find this class file? The operating system tells the JVM in which all directories the JVM to search for the class file. The path where the class files can be stored is stored in a variable of the operating system. This variable is called CLASSPATH. You can change the CLASSPATH as follows.

export CLASSPATH=<your dir path>:$CLASSPATH

The CLASSPATH has a sequence of paths separated by :. The order does matter. If we use a class called ABC and if ABC.class exists in more than one directory, then the one which appears leftmost in the CLASSPATH will be considered by the JVM.

Try the following experiments.

//L41/d1/Test.java

```
public class Test

{

	public void foo()

	{

		System.out.println("foo of Test");

	}

}

// Q1. do we require the Test.java file to run this program
//		is Test.class sufficient?
//		mv Test.java Test.bak and then try running the program
// Q2. can this class file be in a different directory?
//		make a directory MyLib; move Test.class to MyLib
//		try running the program
//		errors:
/*

// Exception if the class is not found in the CLASSPATH
Exception in thread "main" java.lang.NoClassDefFoundError: Test
	at Example1.main(Example1.java:8)
Caused by: java.lang.ClassNotFoundException: Test
	at java.net.URLClassLoader$1.run(URLClassLoader.java:372)
	at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
	at java.security.AccessController.doPrivileged(Native Method)
```

```
        at java.net.URLClassLoader.findClass(URLClassLoader.java:360)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        ... 1 more
*/

// Q3.
//$ export CLASSPATH=MyLib:$CLASSPATH
// try again
// it works
// This indicates how the java runtime looks for the classes

public class Example1
{
        public static void main(String[] args)
        {
                Test test = new Test();
                test.foo();
        }
}
```

What could happen if two different directories have file with the same class name? When we try to use this class, which class will we get? It depends on the value of the CLASSPATH variable. You can experiment by modifying the value of the CLASSPATH variable.

```
// directory MyLib has One.class
// directory MyLib1 has One.class
// Which One will be used here?
// depends on CLASSPATH
```

```
public class Example2
{
	public static void main(String[] args)
	{
		One test = new One();
		test.foo();
	}
}

// d1/MyLib/one.java
public class One
{
	public void foo()
	{
		System.out.println("foo of Test of MyLib");
	}

}

//d2/MyLib 1/one.java
public class One
{
	public void foo()
	{
		System.out.println("foo of Test of MyLib1");
	}
}
```

Let assume (please note I am **assuming**!) that you discuss about me with your friends and mention my name(?) - (This is just an assumption – I am sure something more interesting to discuss). What if your group knows more than one NSK? Then, you will have to qualify me (that old man who teaches and preaches!). This concept of qualifying is called namespace. Java implements using a concept called package. All names at the outermost level in a package should have unique names. The names can repeat across packages.
You may not believe – we had two S. Ramesh Babu and two K. Ravi in my class in high school. I cannot tell you how we used to distinguish them!

- **In java, the package name will translate to a directory name.**
- **So, this class One should be a file under MyLib directory.**
- **The CLASSPATH should contain the name of the directory which has MyLib.**
- **To use the features of this package, the client will import the package.**

import MyLib;
import MyLib1;

- **The client uses the fully qualified name to access the class in the package MyLib.**

MyLib.One x = new MyLib.One();

- **You can observe that there is no clash as fully qualified names are used.**
  MyLib.One : is a fully qualified name

//L41/d1/MyLib/One.java

```
package MyLib;
public class One
{
	public void foo()
	{
		System.out.println("foo of Test of MyLib");
	}
}
```

//L41/d1/MyLib1/One.java

```
package MyLib1;
public class One
{
	public void foo()
	{
		System.out.println("foo of Test of MyLib1");
	}
}
```

//L41/d1/Example2

```
import MyLib;
import MyLib1;
public class Example2
{
	public static void main(String[] args)
	{
		MyLib.One x = new MyLib.One();
```

```
            x.foo();
            MyLib1.One y = new MyLib1.One();
            y.foo();
        }
}
```

We do not require explicit import if we use the fully qualified name.

//L41/d2/Example3.java

```
public class Example3
{
        public static void main(String[] args)
        {
            MyLib.One x = new MyLib.One();
            x.foo();
            MyLib1.One y = new MyLib1.One();
            y.foo();
        }
}
```

We can also import a particular entity of the package. In this case, we are importing class One of MyLib. Now we can refer to this class without qualifying.

//L41/d2/Example4.java

```
import MyLib.One;
public class Example4
{
        public static void main(String[] args)
        {
            One x = new One(); // this is MyLib.One
            x.foo();
            // We can still qualify.
```

```
            MyLib1.One y = new MyLib1.One();
            y.foo();
        }
}
```

Can we import the class one of both the packages MyLib and MyLib1?
No. This will introduce the clash we were trying to avoid. If we refer to One without qualifying the name, we will not know which One we are talking about.

//L41/d2/Example5.java

```
import MyLib.One;
import MyLib1.One; // Error
public class Example5
{
        public static void main(String[] args)
        {
            One x = new One();
            x.foo();
            One y = new One();
            y.foo();
        }
}
```

---

nested classes or types :

Long long back, we wrote a program using a constant or a string literal. We then said that it is better to use a variable so that we can give different values each time we run the program. We also associated a variable with a type capturing the essence of the variable itself. We then said that the types we have in a language may not be sufficient. We made our own type – called it a class. We then

developed relationship between classes. We called that relationship of “IS A” as inheritance. We then developed a class with no commitment – no implementation – we called this an interface which is a pure type. We also developed classes having objects within them. We called this composition. Trust all these are comfortable so far.

Let us consider an example. We may want to furnish some chairs in our rooms. We may find a type called chair. Make a few instances of these chairs in our room. We may pick up comfortable big chairs for our room. So far it is fine. Can we take these chairs to a flight? Can these chairs we picked up for rooms have room in an aircraft? Those who do not have had the privilege to travel business class in an aircraft would know the comfortable chairs that the aircrafts provide!
So the chairs of the aircraft are different from the chairs we normally use in our rooms – unless you have fallen for the aircraft chairs!
So, as these chairs are special for aircrafts, we can have a type called chairs of aircraft and pack any number of these chairs as possible in an aircraft. So, here we require a type in a type. This concept is called nested type or inner class.

We use inner classes for the following reasons.

- We can increase the encapsulation. We can hide the class. This shall be used only by the outer class and no other class.
- As only this class requires the inner class, we can avoid polluting the namespace.
- Implementation of this inner class may require or may use the outer class.

Four types of inner classes in java:
a) non-static inner class
b) static inner class
c) local inner class
d) anonymous inner class

a) Inner class or non-static inner class:

Here, the object of the inner class is created using an object of outer class. The object of inner class remembers the object of outer class using which the inner class object is created – is called outer this.

In this example, there is a class called Domain. Each Domain has a name. The Domain contains a class called Member. Every member has to belong one or the other Domain. So the member object is created using the Domain object.

```
// non static inner class
// every inner class object refers to an outer class object
//       has an outer this
public class Example1
{
        public static void main(String[] args)
        {
                Domain d1 = new Domain("RainBow");
                // John and peter belong to the domain RainBow
                Domain.member m1 = d1.new member("john");
                Domain.member m2 = d1.new member("peter");
                Domain d2 = new Domain("Sea");
                // mary belongs to the domain Sea.
                Domain.member m3 = d2.new member("mary");
                m1.disp();
                m2.disp();
                m3.disp();
        }
}
```

```
class Domain
{
	private String name;
	public Domain(String name) { this.name = name; }
	public class member
	{
		private String memberName;

		public member(String memberName)
		{
			this.memberName = memberName;
		}
		// name here refers to the field of the outer class
		public void disp()
		{
			System.out.println("name : " + name + " member : " +
memberName);
		}
	}
}
```

2) static inner class:

The inner class does not remember anything of the outer class. We do not require an object of outer class to create an object of inner class. This is the way many utility classes are developed,

In this example, the inner class Pair provides a mechanism to hold a pair of integers and access them using getters for the first and the second elements.
The outer class provides a method to receive an array and return a pair indicating the first and the last elements in the array. This could also have been return the max and the min elements or the max and the second max and so on.

```
// static inner class
// only relation between types;
public class Example2
{
	public static void main(String[] args)
	{
		int[] x = {1, 1, 2, 3, 5, 8, 13, 21, 34 };
		MyNumbers n = new MyNumbers(x);
		System.out.println(n.firstTwo().getFirst());
		MyNumbers.Pair p = n.lastTwo();
		System.out.println(p.getFirst() + " : " + p.getSecond());
	}
}

class MyNumbers
{
	private int[] a;
	public MyNumbers(int[] x)
	{
		a = new int[x.length];
		for(int i = 0; i < x.length; i++)
		{
			a[i] = x[i];
		}
	}
	public Pair firstTwo()
```

```
	{
		return new Pair(a[0], a[1]);
	}
	public Pair lastTwo()
	{
		return new Pair(a[a.length - 2], a[a.length - 1]);
	}

	static public class Pair
	{
		private int first;
		private int second;
		public Pair(int first, int second)
		{
			this.first = first;
			this.second = second;
		}
		public int getFirst() { return first; }
		public int getSecond() { return second; }
	}
}
```

c) local inner class

In this case, we create a class within a method. So, this can be used only in that method and nowhere else.

What I have for my breakfast is something that is totally mine. Why should I tell you?

```
// local inner class
//	class within a method; no body can use it outside of this method

public class Example3
{
```

```
	public static void main(String[] args)
	{
		foo();
	}



	// no body outside of foo can know MyBreakfast!
	public static void foo()
	{
		class MyBreakfast
		{
			private String name;
			public MyBreakfast(String name) { this.name = name; }
			public void disp() { System.out.println("breakfast : " +
name);}
		}
		MyBreakfast mbf = new MyBreakfast("dosa");
		mbf.disp();
	}
}
```

d) Anonymous inner class:

In the good old days, the husbands of the house would not call their wife by name and the wives forbidden to call their husbands by name. The lady would refer to the husband as “that person – avaru in kannada” or as “father of chinnu or munnu”. So clearly name is not a necessity to refer to a thing or a person. I am sure your parents would have struggled to give a name for you. Why should everything have a name? If I want to use something only once, why give a name at all? This is the concept behind anonymous entities in programming. There are other benefits which we can not discuss at this point.

We can make a new class from an existing class and make an object immediately

– all in one go. For this, in Java, we use Anonymous inner class.
This is used very heavily in Java.

```
// anonymous inner class
public class Example4
{
	public static void main(String[] args)
	{
		Tester t1 = new Tester(); t1.test();
		// makes a new class with no name; overrides whichever methods
		// it wants to.
		// avoids unnecessary name
		Tester t2 = new Tester() {
			public void test() { System.out.println("this is a new test"); }
		};
		t2.test();
	}
}

class Tester
{
	public void test() { System.out.println("this is a test"); }
}
```

Let us consider the following code.
Tester is a class. The code new Tester() would create an object of the class Tester. But this code is followed by a block of code. This block of code is extending the class Tester – thereby creating a new class with no name – anonymous class. This new class overrides the method test. Therefore t2 instantiates this new anonymous class which extends Tester. Call to the method test using tester would call this overridden method.

```
Tester t2 = new Tester() {
        public void test() { System.out.println("this is a new test"); }
};
```

The concept of Callback :

The methods developed for a library should be flexible. It should be able to serve different varieties of requests. To provide this flexibility we use the concept of callbacks. We shall explain the concept through the following examples.

Let us look at the method test1.

```
public static void test1(MyInterface mi)
{
    mi.foo(); //  we do not know which method will be called
    // MyInterface should support foo
}
```

This method expects an object which implements the interface MyInterface and therefore override the method called foo. By looking at this method test1, we cannot deduce which foo will be called. It depends on the object implementing the interface MyInterface which will be passed as argument to this method.

Let us also look at another interface :

```
public static int test2(Computable comp, int x, int y)
{
    return comp.compute(x, y);
}
```

We can infer that this method calls the method compute on an object of a class implementing the Computable interface – takes two arguments – returns the result. We cannot make out what compute does by looking at this code.

Walk through the program carefully and understand how the calls take place.

```
import java.util.*;
public class CallBack1
{
	public static void main(String[] s)
	{
		MyInterface x1 = new MyClass();
		test1(x1);
		MyInterface x2 = new MyInterface() {
			@Override
			public void foo() { System.out.println("new foo"); }
		};
		test1(x2);
		System.out.println("res : " +  test2(
			new Computable() { @Override public int compute(int x, int y)
				{ return x + y; }}
		, 10, 20));
		System.out.println("res : " +  test2(
			new Computable() { @Override public int compute(int x, int y)
				{ return x * y; }}
		, 10, 20));
	}

	public static void test1(MyInterface mi)
	{
		mi.foo(); //  we do not know which method will be called
		// MyInterface should support foo
	}
```

```
	public static int test2(Computable comp, int x, int y)
	{
		return comp.compute(x, y);
	}
}

interface MyInterface
{
	void foo();
}

class MyClass implements MyInterface
{
	@Override
	public void foo()
	{
		System.out.println("foo of MyClass");
	}
}
interface Computable
{
	int compute(int x, int y);
}
```

We use anonymous inner cass to implement a callback. These are the properties of anonymous inner class.

- cannot have a constructor
- extends the class whose object is being created
- can downcast in the hierarchy
- can override methods of the super class
- can have attributes

- should not introduce new methods

Let us understand how we use the concept of callback. Here we implement a class called Sorter with a method called mysort. This method sorts an array of integer using some exchange sort.
We also have an overload of the method mysort which takes an object of implementation of MyInterface interface. This class supports a method which compares the two elements – called mycompare. The mysort arranges the elements in order based on this result of call to mycompare method.
Now, the client can decide how to arrange the elements of an array by changing the implementation of mycompare method.

Observe the following.

- The class SortReverse implements MyInterface with mycompare comparing the numbers for descending order. The mysort call with the object of SortReverse would cause the arrange elements in descending order.
- In the second case, we pass an object of an anonymous inner class implementing MyInterface – overriding mycompare to compare the unit digits. So the second call to mysort will arrange the elements of the array in the increasing order of unit digit.

The program is a bit long. So go through slowly. Slow and steady wins the race.

```
import java.util.*;
public class CallBack2
{
	public static void main(String[] s)
	{
```

```
		int[] a = { 15, 32, 123, 67, 26, 24 };
		System.out.println(Arrays.toString(a));
		MyInterface my1 = new SortReverse();
		Sorter.mysort(a, my1);
		System.out.println("Sort Reverse " + Arrays.toString(a));
		// anonymous inner class
		MyInterface my2 = new MyInterface()
		{
			public boolean mycompare(int x, int y)
			{
				return x % 10 < y % 10;
			}
		};

		Sorter.mysort(a, my2);
		System.out.println("what " + Arrays.toString(a));
	}
}
class Sorter
{
	// Observe : this method is not flexible; always sort in ascending
	// order
	public static void mysort(int[] a)
	{
		int i, j;
		int len = a.length;
		for(i = 0; i < len - 1; ++i)
		{
			for(j = i + 1; j < len; ++j)
			{
				if(a[j] < a[i])
				{
```

```
                        // swap(a[i], a[j]); // NO
                        myswap(a, i, j);
                }
        }
}

// mycompare is not a static fn
// can also recieve objects which are of subclass of MyPred
// these subclasses can override mycompare
// not possible if the method is static
// if we use inheritance, class can not also have
//      another base class
// coupling is high
// this is flexible; comparison is based on a method call
// method mycompare can be of any class which extends MyPred
public static void mysort(int[] a, MyPred mypred)
{
        int i, j;
        int len = a.length;
        for(i = 0; i < len - 1; ++i)
        {
                for(j = i + 1; j < len; ++j)
                {
                        if(mypred.mycompare(a[j], a[i]))
                        {
                                // swap(a[i], a[j]); // NO
                                myswap(a, i, j);
                        }
                }
        }
}
```

```
    // more flexible; no commitment for any concrete class
    public static void mysort(int[] a, MyInterface mypred)
    {
        int i, j;
        int len = a.length;
        for(i = 0; i < len - 1; ++i)
        {
            for(j = i + 1; j < len; ++j)
            {
                if(mypred.mycompare(a[j], a[i]))
                {
                    // swap(a[i], a[j]); // NO
                    myswap(a, i, j);
                }
            }
        }
    }

    public static void myswap(int[] a, int i, int j)
    {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }


}
```

```
class MyPred
{
	public boolean mycompare(int x, int y)
	{
		return x % 10 < y % 10;
	}
}

interface MyInterface
{
	boolean mycompare(int x, int y);
}

class SortReverse implements MyInterface
{
	public boolean mycompare(int x, int y)
	{
		return x   > y  ;
	}
}
class SortTenthDigit implements MyInterface
{
	public boolean mycompare(int x, int y)
	{
		return x / 10 % 10 < y / 10 % 10;
	}
}
```

Now, your turn to add a few more implementations of the interface to make mysort dance to your tune.