CS2

Theme:

Object Oriented Programming in Java

Handout : Week5

prepared by N S kumar

Preface

Dear friends,

There is a delay in providing the Week5 Lecture Notes. I was under the wrong impression that I have to discuss only for one lecture. I realized later that I require to make notes for 5 lectures. Sorry for the delay. Trust this will not affect that much. I know that most of you will not touch these until the time for T2!

Trust you enjoyed the test and did well. I am getting disturbing intimation that many of you have managed to get a single digit score on fifty – it must have been extremely difficult to perform that way. It is a clear indication that you did not make a single attempt to read the lecture notes.

All the best.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com
Date : 1st February 2015

Recursion:

We have observed that any program in the world can be written by using three constructs – assignment, selection(if and switch) and loop(while, for, do..while).

This is a possible plan for study.

Keep on reading until no more topics are left for studying. This concept is called iteration.

```
void readPlan(TopicList topicList)
{
        while(topicList.hasNext())
        {
                read(topicList.next());
        }
}
```

We can express the same idea in a different way. We can say if there are topics, read the first topic and then read the remaining.

Observe we have removed the while and replaced it by if. We call the same method from within. This concept is called recursion.

```
void readPlan(TopicList topicList)
{
       if(topicList.hasNext())
       {
                read(topicList.next());
                readPlan(topicList); // Note: this will have one topic less!
       }
}
```

Recursion is an alternate for iteration. In iteration we use a conditional loop. In recursion, we express the solution for a problem, in terms of the problem itself,

but of a smaller size. We also have to have solution for some simple case or cases. So recursion has two steps

1. recursive call on a problem of smaller size
2. base cases or escape hatch for which the solution is directly available.

Please note that if and recursion are equivalent to a loop.

The concept of recursion is not new to you. You have come across the concept of recursion in successive differentiation. What happens if you differentiate the trigonometric function sin(x) twice with respect to x? One can define a train as a coach followed by a train or train is empty – the last clause is necessary without which the recursion will never come to an end.

Recursion is a very powerful tool. It is easy to express ideas in recursion if we are talking about those items which are inherently recursive. Look at the folders or directories in files. A directory can be empty or can have files and directories. Is this not recursive?

Let us look at some simple examples.
How to find the sum of the digits of a given number? Can we say that if the number is 0, then the sum is 0? Can we say that if the number has some digits, then the result is given by the unit digit + the sum of the digits of the number with that unit digit removed? If we keep removing the digits, ultimately the number has to become zero.

// L16/d1/Client.java

```
// recursion
import java.util.*;
public class Client
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		int n = scanner.nextInt();
		MyNum mynum1 = new MyNum(n);
		int sum = mynum1.findSum();
		System.out.println("sum : " + sum);
		scanner.close();
	}
}
```

// L16/d1/MyNum.java

```
public class MyNum
{
	private int val;
	public MyNum(int val)
	{
		this.val = val;
	}

	public int findSum() // wrapper for a recursive method
	{
		return findSumRecursive(val);
	}
```

```
    private int findSumRecursive(int val)
    {
        if(val == 0)
         // escape hatch or base case
        {
            return 0;
        }
        else
        {
            // recursive step
            return val % 10 + findSumRecursive(val / 10);
        }
    }
}
```

In recursion, something should keep changing to indicate that the problem size has decreased. In this example, each time we should remove the unit digit. If we do this directly on the attribute of the object, the object will change. **Object oriented programming does not support recursion concept directly**. So we write a method which in turn will make a copy of the attribute and send it to the recursive method. This sort of a function is called a **wrapper function**. A wrapper function gives the work to another function – **delegates** the work.

In case of recursive methods, each time the method is called, a new stack frame is created. This may make the program with recursion a bit slower on these computers based on Von Neumann Architecture. We are trying a project to convert recursive programs to iterative programs.

We can express of gcd of two numbers as follows:

gcd(m, n) : if two numbers are equal, then the number itself is the gcd

if m > n, gcd is gcd(m – n, n)

if m < n, gcd is gcd(m, n – m)

Check these programs.

// L16/d2/Pair.java

// L16/d2/Client.java

One of the classical puzzles goes with the name – Tower of Hanoi or Brahma's riddle. It states that when Brahma created the universe, he installed in Varanasi three golden pegs and 64 golden disks of decreasing diameter on peg 1. It seems he instructed the priests of Varanasi to move the disks from peg 1 to peg 3 using peg 2 if necessary. He also laid down the two rules.

1. move one disk at a time
2. Never place a bigger one over a smaller one.

The priests in Varanasi are doing this from the beginning of life on Earth. They take one second to move a disk. It seems Brahma has said that the world will perish when all the disks are moved. How come the world has not perished so far? Did Brahma go wrong somewhere in his calculations? How much time is required to move 64 disks? Can your computer which can do some 4×10^9 operations per second do this job? How much time would your computer require?

Check:

// L16/d3/MyNum.java

// L16/d3/Client.java

Observe the way the solution is formed to move n disks from peg 1 to peg 3 using peg 2. If the number of disks is 0, nothing to do -that is the escape hatch. If there are n disks, somehow move n – 1 disks from 1 to 2 using 3 is necessary. Now, there is only one disk in peg 1, move that to peg 3. Now there are n – 1 disks in peg 2. Move them to peg 3 using peg 1 if necessary. If we keep repeating this, at sometime n becomes 0, there is nothing to move and the problem is solved.

It is interesting to check how a recursive function works. You may want to follow this for val being 25 and find the output.

```
// L17/d1/Client.java
// L17/d1/MyNum.java
public int whatRecursive(int val)
	{
		if(val == 0)
		{
			return 0;
		}
		else if(val % 2 == 1)
		{
			return 1 + whatRecursive(val / 2);
		}
		else
		{
			return whatRecursive(val / 2);
		}
	}
```

Exercise:

1. Write programs using MyNum class to support the following recursive methods.

a) factorial

b) find number of 1 in the binary representation of the number

c) reverse a number (this is a bit tricky)

d) nth fibonacci number

2. Write programs using Pair class to support the following recursive methods.

a) a to the power m

b) Russian Peasant method

c) Generate the nth term of an arithmetic progression

d) Generate the nth term of a geometric progression

Final look at Final concept with respect to fields:

The concept of Final with respect to fields says that the field cannot be changed once initialized. There are two types of fields to observe.

a) Primitive type fields:

There is no default initialization based on the type. These can be initialized within the class or in the initialization block or in the constructor. Once initialized, it cannot be assigned.

A local variable does not have the concept of default initialization. So, it might be initialized at the point of declaration or can be assigned later, but only once.

Go through the example given below carefully and observe the comments provided.

// L17/d2/Client.java

```
public class Client
{
	public static void main(String[] args)
	{
		// test 1:
		final int a; // ok why?
		a = 10;	// ok
		// a = 20; // NO
		final int b = 10;
		//b = 20; // Error
	}
}
```

```
class Test
{
	// compare with a of main of Client
	//private final int t1; // error
	// what about this?
	private final int t2;
	{
		t2 = 2;
	}

	// what about this?
	private final int t3;
	{
	}
	Test() { t3 = 3; }
	private final int t4 = 4;
	{
		// Error
		//t4 = 44; // what if t4 = 4 ?
	}
}
```

b) Primitive reference fields:

Reference becomes a constant and not its reference. You may observe the example given below.

```
// L17/d3/Client.java
// reference becomes constant and not its reference
public class Client
{
	public static void main(String[] args)
	{
		MyNum n1 = new MyNum(10);
		MyNum n2 = n1;
		// we say that n1 and n2 are aliases
		n2.set(20);
		System.out.println("n1 : " + n1.get()); // 20 why?

		// what happens what n2 referred to earlier?
		n2 = new MyNum(200);
		 // No relation between n1 and n2
		// They refer to two different objects
		System.out.println("n1 : " + n1.get()); // 20 why?

		// n3 is final; cannot be changed
		final MyNum n3 = new MyNum(30);
		System.out.println("n3 : " + n3.get());
		//n3 = n1; // Error
		//n3 = new MyNum(300); // Error

		// whatever n3 refers to, can be changed
		n3.set(300);
		System.out.println("n3 : " + n3.get());
	}
}
```

This is definitely not the final call on final – as it happens in airports – last and final calls keep repeating even in Java!

Exercises:

a) Can every field of a class be final?

b) Why should we make the fields of a class final?

c) If a field is final, does it mean that its value will be same for all the objects of the class?

d) If a constructor calls another constructor, can the final field be initialized in any one of them? Can we initialize in both?

Static members in a class:

We have said earlier that no language can support all possible types we would like to have in our worlds – real or hypothetical. We said that a class would have attributes and behaviour. Every object of the class would get all those attributes and exhibit those behaviour.

This is not always true. There are certain attributes which are shared with all the objects of the class. The number of students in a class is not a property of every student. It is the property of the class to which the students belong. The name of our college is shared with all our students. If we make the name of the college part of every student, then if the name of the college is changed, every student will have to change the name of his/her college explicitly!

Let us try to understand this concept with a few more examples.

A road may have a number of houses. The road is not owned by any house. It is shared. We sometimes have fights about this – 'is it your father's property?'.

In good old days, we used to have a single TV at home – a shared resource. My son and My grand mother would have a fight – to watch football or ramayana!

The college bus is shared by all the students traveling by it. You do not have right to try your artwork on the back of the seat in front of you or try to satisfy your curiosity by finding what is within the seat by using a knife.

The word static has umpteen number of meanings in Computer Science. Whatever is done during the compilation process(I am sure you know by now what is compiling a program!) is said to be static. Whatever happens when the program is run as dynamic.

In 'C', a variable can be declared static outside a block – called external static or inside a block – called internal static. In java, we do not have this concept at all. We can not declare variables outside a class. We can not declare a variable as static within a method. Henceforth we shall not discuss the 'C' static concept in these notes.

Let us examine a few programs. We shall start with a wrong program.

```
//L18/d1/Client.java
public class Client
{
	public static void main(String[] args)
	{
		// So nice, we have great students!
		Student s1 = new Student("ashoka", 9.8, 1);
		Student s2 = new Student("akbar", 9.9, 2);
		s1.disp();
		s2.disp();
	}
}
```

```
// L18/d1/Student.java
public class Student
{
	private String name;
	private double gpa;
	// Wrong design :
	//	num of students cannot be member of every object
	//		we create. It is not an instance member.
	//	occupies memory in each object
	//	if we query two objects, we get two diff results
	//	if we want to update, we should change all the objects
	private int numOfStud;
	public Student(String name, double gpa, int numOfStud)
	{
		this.name = name;
		this.gpa = gpa;
		this.numOfStud = numOfStud;
	}
	public void disp()
	{
		System.out.println(
			name + ":" + gpa + ":" + numOfStud
		);
	}
}
```

If we run this program, we will find that the number of students to be different for each student. Observe that the field numOfStud exists for each student – therefore occupies extra memory proportional to the number of students. Changing one will break the integrity. Modifying all objects is a nightmare. So, we should have only one instance no matter how many objects we have.

Let us now have a look at an example with a shared variable – shared with all objects of the class – is also called class variable – but syntactically, it is declared as static.

```
// L18/d2/student.java
public class Student
{
	private String name;
	private double gpa;
	// right design:
	// This is shared with all objects of the class
	// It is a class attribute and an instance attribute
	// can be accessed using the class name or the instance name
	// experiment:
	//	created first time the class is used
	//	initialization:
	//		default based on type
	//		in the class
	//		static initialization block
	// we can provide static methods to access these fields.
	private static int numOfStud = 0;
	public Student(String name, double gpa)
	{
		this.name = name;
		this.gpa = gpa;
		numOfStud++;
		//this.numOfStud++; // how about this?
	}
```

```
        // can we display a static member here?
        public void disp()
        {
                System.out.println(
                        name + ":" + gpa + ":" + numOfStud
                );
        }
}
```

Please note that a static field of a class is created once and only once when the class is loaded – this happens when the class is used for the first time – when an object is created or a method of the class is called.

A static field of a class can be initialized in the following ways.

1. initialize based on its type – default initialization
2. initialize within the class
3. initialize within a static initialization block

How do we access a static field of a class? Can we use an instance method (method we call with an object or instance of the class)? What if there are no objects at all? Can we still access the static field?

For this, a class can have static method. A method qualified as static can be invoked by using either an instance name(object reference) or the class name.

We have said earlier that when a method is invoked using an object reference, the object reference is passed as the first argument. It does not happen in static methods. Even when we call a static method using an object reference. the compiler will change the code so that the call happens using the class name alone. So, a static method will not have 'this' reference. So, in a static method, we cannot have unqualified access to a field. There is no 'this' - There is no

implied object – therefore no implied field.

We use static methods for number of reasons.

1. When we start executing a program, we have no objects. We cannot create objects before we enter into a java class. Therefore the entry point (the main method) has to be static.
2. We also make methods static so that we can deal with static fields of a class.

We will see other special uses later in the course.

Please read the following programs.

// L18/d3/Client.java

// L18/d3/Student.java

Exercises:

1. Design classes for the following. Decide the attributes and check whether any of them should be static.

a) Four teachers use a car pool – One particular teacher gets the car.

b) Four teachers use a car pool – every day one of the teachers will bring his car

c) Four teachers use a car pool – One particular teacher gets one of his cars.

2. can a field of a class be static and final? Can that be of primitive type? Can that be of reference type? What is the difference?

Let us start the next topic with a series of questions. We will answer them as we go along. You may want to pause a while, ponder over these questions before turning over to the next pages.

1. Can the instance field of a class be a reference to the same class by type?
2. Can the static field of a class be a reference to the same class by type?
3. Can we make a constructor of the class private?
4. Can we make all constructors of the class private?
5. Can a method of the class return a reference to an object back?
6. Can a method of the class return the reference to a static field of the class?
7. If a field of the class is of reference type (may be to the same class type), is it initialized to null by default?

Lots of questions? Do not worry. I am here to answer them all. Simple answer. All of them are possible.

Let us start with the first example. An instance field of the class is a reference to the same class. This is definitely possible. A coach of a train leads to the next coach. Such a class is called a self referential class. If you study a course called data structures, you will be using such classes a lot. Here is an example of such a class.

A field of a class could be a reference to the same class type.

```
// L19/d1/MyNum.java
public class MyNum
{
	private int val;
	private MyNum ref; // some name!
	public MyNum()
	{
		this.val = 111;
		this.ref = null;
	}
	public MyNum(int val)
	{
		this.val = val;
		this.ref = new MyNum();
	}
	public void disp()
	{
		System.out.println(this.val + ":" +
			this.ref.val);
	}
}
```

```
// L19/d1/Client.java
public class Client
{
	public static void main(String[] args)
	{
		MyNum x = new MyNum(222);
		x.disp();
		MyNum y = new MyNum(333);
		y.disp();
	}
}
```

x will refer to an object whose val field becomes 222 and ref field will now refer to one more object whose value field becomes 111 and ref field is null – say grounded.

Observe that the one argument constructor calls in turn the default constructor.

A static field of a class could be a reference to the same class type.
In that case, it is shared with all of the objects of that class.

Observe the following examples.

```
//L19/d2/Client.java
public class Client
{
	public static void main(String[] args)
	{
		MyNum x = new MyNum(222);
		x.disp();
		MyNum y = new MyNum(333);
		y.disp();
	}
}
```

```
//L19/d2/MyNum.java
public class MyNum
{
	private int val;
	private static MyNum ref = new MyNum(1000);
	public MyNum(int val)
	{
		this.val = val;
	}
	// Observe the second part of the display is same for
	// all objects. ref is shared with all objects.
	public void disp()
	{
		System.out.println(this.val + ":" +
			this.ref.val);
	}
}
```

We can make all constructors of a class private. In that case, we may have to provide some other mechanism to make objects. We can have a static method return an object. The static method in turn would call the private constructor. The client can call the static method using the class name to create an object.

```
// L19/d3/MyNum.java
public class MyNum
{
	private int val;
	// private constructor; cannot be used by the client.
	private MyNum(int val)
	{
		this.val = val;
	}
```

```
	public void disp()
	{
		System.out.println(this.val);
	}

	// This allows the user to make objects of the class
	// is an object creator
	public static MyNum makeMyNum(int val)
	{
		return new MyNum(val); // calls the private constructor
	}
}
// L19/d3/Client.java
public class Client
{
	public static void main(String[] args)
	{
		// cannot call ctor
		//MyNum x = new MyNum(222);
		//x.disp();
		//MyNum y = new MyNum(333);
		//y.disp();

		// can we call any instance method of the class?
		//	NO
		// can we call a class method(static method)
		// YES
		// can that method make an object? YES
		MyNum x = MyNum.makeMyNum(11);
		x.disp();
```

```
        MyNum y = MyNum.makeMyNum(22);
        y.disp();
    }
}
```

It is a very natural question to ask as to why we should go around to make objects in this way. This gives better control over object creation. We may want to make only some number of objects – this reflects some reality. It could stand for the number of students in a class or the number of class rooms in a particular building (already completed). There are some extreme requirements – only one object. PESU university should have one and only one VC, the state of Karnataka should have one and only chief minister and so on. Such a class is called a singleton class. There are number of ways of implementing this concept. Here is a possible implementation of the Singleton class.

Idea: have a static field of the class type – initialize it to null (that is the default any way) – When a static method is called for the first time, if that member is null, create an object and assign to the static member – return it. When the client calls next time, if statement is skipped and the same object is returned.

```
private static MyNum ref = null;
public static MyNum makeMyNum(int val)
    {
        if(ref == null)
        {
            ref = new MyNum(10000);
        }
        return ref;
    }
```

Exercise:

Modify the singleton class. Use a counter. Initialize it to 0. If the counter is 0, make it 1 and create an object. Next time the method is called, counter is not 0, therefore return the object created.

Modify the above solution to have n (some small integer) objects in stead of 1.

It is my experience that the students find the concept of static members (fields and methods) a bit uncomfortable.

We use the static field to indicate that it is an attribute of the whole class and not any particular object. It is shared with all the objects.

We use a static method for number of reasons.

1. **play with static fields**
2. **provide an entry into the program**
3. **simulate the way we call mathematical functions: ex: Math.sqrt(25.0)**
4. **control the way of creation of objects.**

At this point, I will sign off. We have discussed the lectures L16 -L19 so far.

I will try to share the notes for the topics L20 onwards and an additional FAQ section at the earliest.

Until that time, Sayonara!