CS2

Theme:

Object Oriented Programming in Java

Handout : Week4

prepared by N S kumar

Preface

Dear friends,

Week 4 edition for you. In this we discuss control structures. I am not including recursion in this week edition even though it is part of our schedule as it is not part of your test. Next edition will include recursion.

In this edition, you will find lots of examples. You will find good ones as well as bad ones. I am trying to teach you how to program well. Walk through the lecture notes and the examples with a big convex lens!

The section of FAQ is delayed as I have received a very few questions. I will make this section by the middle of this week based on your inputs. So, ask lots of questions.

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

'this' again!

I remember my trip to Egypt in 2005. We had an Egyptian guide whose mastery over the Queen's language was not exemplary. He would keep using the phrases 'this one' and 'that one' – of course we would not know which one he was referring to as we would normally have more than one thing in the context.

Assume that you take the dog to a doctor or a beautician. I am sure you will have a very good name for your dog – like Alexander or Caesar. It is very likely that the doctor or the beautician may not have very good memory for names(like me!). He may refer to your dog as 'this' dog. Thats what Java does when we invoke a method using an object.

What if we take a pair of dogs to the beautician (forget the doctor – your idea could be to arrange a grand wedding!), then one of them will be 'this' and the other has to be given a suitable name.

Trust this above article puts to rest any doubt you may have about this!

'this' is a parameter – comes into being whenever a method is called using an object (this is half lie! Do not worry at this point). It is not part of the object.

In the last article, we learn how to initialize objects using constructors. We also learn how to write selection statements – both if and switch.

Looping Structures:

When I was a child, I used to pester my mother for stories. Some time she would start telling me a story like this. A sparrow brought a grain of rice. Another sparrow brought a grain of rice. One more sparrow brought a grain of rice. My goodness. The story would never end. It would keep repeating with sparrows after sparrows. Can we do something similar in programming?

We can. We call it a loop. There are four ways of looping in Java.

1. **while loop**
2. **for loop**
3. **do .. while loop**
4. **enhanced for loop**.

This week, we will discuss the first three and let the last one remain dormant until the next week.

Let us start with a simple example. My requirement is to find the sum of the digits of a given number. If I am given a number like 7 – a single digit, it is so nice. That itself is the number. What if I a given 47? I can get the unit digit by modulo 10 arithmetic. Then I can divide by 10 to get 4 and add the two. What if the number is having number of digits? I can keep getting the unit digit by modulo 10 arithmetic and remove the unit digit by dividing by 10. As the unit digits become available, I can add them to a variable initialized to 0 in the beginning – we call such a variable an accumulator. When the number becomes 0, we can stop and whatever the accumulator contains will be the result.

There are two ways of solving this problem. The client can pass the number explicitly or make an object which contains a number and ask that object to find the sum of digits.

By convention, we should use the former method. Recall Math.sqrt(25.0). We do not make an object of Math type and call sqrt on it!

We shall break the convention. we all like breaking – break the law, break the queue! So, we will program using truly object oriented programming paradigm. We will make a MyNum object which represents an integer and we will ask it to find the sum.

Files: L13/d1/Client.java L13/d1/MyNum.java

//L13/d1/Client.java

```
public class Client
{
	public static void main(String[] args)
	{
		MyNum mynum1 = new MyNum(1729);
		int sum = mynum1.findSum();
		System.out.println("sum : " + sum);
	}
}
```

//L13/d1/MyNum.java

```
public class MyNum
{
	private int val;
	public MyNum()
	{
		this.val = 0;
	}
	public MyNum(int val)
	{
		this.val = val;
	}

	public void disp()
	{
		System.out.println("val : " + val);
	}
```

```
	public int findSum()
	{
		int s = 0;
		while(val != 0)
		{
			s += val % 10;
			val /= 10;
		}
		return s;
	}
}
```

Let us concentrate on the method findSum. The variable s acts like an accumulator. We initialize that to 0. We execute the block (open and close braces – and what all goes within that pair) repeatedly until val becomes 0. As we keep dividing val by 10, there is a guarantee that val will ultimately become 0 and therefore we will definitely come out of the loop.

While loop : We repeat statements based on the condition. We check the condition in the beginning. So, we execute the loop zero or more times.

While loop is top tested, conditional, executed 0 or more times.

Something is wrong in our program. Murphy always makes its appearance. Hint: Display the MyNum object before and after summation! Our object has changed! To avoid that, we may have to make a copy of what the object contains before modifying in the method.

The second looping structure is similar to while loop. This is the for loop.

for(<initial step>;<condition>;<modification step>)

{

<stmt>

}

First the initial step is executed and is executed only once. Then the condition is checked. If it is true, the body of the loop is executed. Then the modification step. Back to condition. This goes on and on and on until the condition becomes false.

Here is our first attempt to replace the while loop by the for loop. Would this change affect the Client program?

```
//L13/d2/MyNum.java
public int findSum()
    {
        int s = 0;
        for(int tempVal = val; tempVal != 0; s += tempVal % 10)
        {
            tempVal /= 10;
        }
        return s;
    }
```

It does not seem to work. Does it?

Have we understood how exactly for loop works?

I leave it to you to figure out what is wrong with this program.

Technically, there is no difference between the while and the for loop. We say that they are isomorphic. So,

for loop is top tested, conditional and executed 0 or more times.

As the initialization, the modification and the condition are together at one place, we prefer this while generating progressions (arithmetic or geometric). In such cases, the number of iterations (repetition) can be determined. Such a loop is called a counting loop. We mimic the counting loop using the for loop.

Let us consider another variation of the same problem. Can we keep on repeating the operation by copying the sum to the number until the sum becomes a single digit? This concept is called the digital root. We can associate a decimal number(could be cheque amount) with its digital root. If a crook modifies a single digit, the digital root will change and the modification can be caught!

```
//L13/d3/MyNum.java
	public int findDigitalRoot()
	{
		int s = 0;
		int tempVal = val;
		while(s > 9)
		{
			for(s = 0; tempVal > 0; tempVal /= 10)
			{
				s += tempVal % 10;
			}
			tempVal = s;
		}
		return s; // always returns a nice value!
	}
```

Can we realize that we have to repeat finding the sum and therefore we require a loop inside a loop? This concept is called nested loop. But something is wrong with this method. We will always get 0 as the loop is not even entered as s > 9 is false in the beginning. We cannot compare the sum before the sum is computed! So, we require a mechanism to at least do once and then check whether the sum is a

single digit or not. We require a bottom tested looping. So, we have do .. while loop.

do .. while loop : conditional, bottom tested, executed one or more times

Here is the right solution.

```
public int findDigitalRoot()
{
        int s = 0;
        int tempVal = val;
        do
        {
                for(s = 0; tempVal > 0; tempVal /= 10)
                {
                        s += tempVal % 10;
                }
                tempVal = s;
        }
        while(s > 9);
        return s;
}
```

Prime of Programming:

Prime numbers fascinate not only the Mathematicians, but also the Computer Scientists. Srinivasan Ramanujan has produced some weird formula to find the number of prime numbers between a pair of really long numbers. You may want to check http://en.wikipedia.org/wiki/Ramanujan_prime . It seems that the Goddess of Namakkal would come in his dreams and tell him the problem and the solution and he would just note that down! You may want to find your own Goddess for this course!

Let us try whether we can check whether a given number is prime. A prime number in case you have forgotten your Elementary Mathematics, is a number divisible by 1 and itself and no other number between 1 and itself. We shall use Object Oriented Programming style to solve this problem. We shall make an object which represents an integer and then ask the object “Are you Prime …?”.

```
//L14/d1/Client.java
import java.util.*;
public class Client
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		int val = scanner.nextInt();
		MyNum mynum1 = new MyNum(val);
		boolean isPrime = mynum1.isPrime();
		System.out.println("prime ? " + isPrime);

		scanner.close();
	}
}
```

```
//L14/d1/MyNum.java
public class MyNum
{
// some portion removed
// Wrong method 1
        public boolean isPrime()
        {
                for(int i = 2; i < val; i++)
                {
                        if(val % i == 0)
                        {
                                return false;
                        }
                        else
                        {
                                return true;
                        }
                }
        }
}
```

The method isPrime returns a boolean – makes sense. It checks whether the given number is divisible by numbers between 2 and itself – is a countng loop. So we prefer for loop over while. If the loop variable i divides val, clearly val is not prime, so we return false. So far very good. Otherwise we return true. So this appears fine.

Oh NO, just because a number between 2 and val does not divide the number, we cannot conclude that the number is prime. Some other number might divide the given number. This is a terrible program. Our first attempt fails. We will try again.

```
// 2. incomplete program; also uses  break
    public boolean isPrime()
    {
        for(int i = 2; i < val; i++)
        {
            if(val % i == 0)
            {
                break;
            }
        }
        // TODO; add some code here
    }
```

So, we can not decide whether the number is a prime until we test with all possible numbers. Can we come out the loop when i divides val? We can by using break. The statement break takes the control out of the loop.

break: exits the loop or switch statement.

So we can reach the point after the loop in two ways – when the condition of the loop becomes false or when the condition preceding break becomes true. It is for you to complete the method. You can expect such questions in the test T1.

But if a block of code can be exited in number of ways, it becomes very difficult to understand. If a thief enters a building which has a number of exits, it is so difficult for the police to catch the thief. It is difficult remove the bug for the programmers if the loop has multiple exits.

Good rule of programming :
The statement break only if switch.
We prefer single entry – single exit constructs.

Let us consider another way of writing the code. We have a cousin of break called continue.

The statement continue causes the next iteration.

Control is transferred to the condition of the loop in while and do .. while and the modification step in for.

```
// 3. Terrible
public boolean isPrime()
{
        int i = 2;
        while(i < val)
        {
                if(val % i != 0)
                        continue;
                i++;
        }
        // fill up the blank ? another if or just a single expression?
}
```

This code is terrible. The loop may never exit. Such a loop is called an infinite loop. Can you figure out what is wrong?

We shall try again. Never give up.

http://en.wikipedia.org/wiki/William_Edward_Hickson.

```
// 4. another attempt
        public boolean isPrime()
        {
                int i = 2;
                while(i * i < val)
                {
                        if(val % i == 0)
                                break;
                        i++;
                }
                return val % i != 0;
        }
```

To find whether a number is prime, should we check for a factor for the given number between 2 and val – 1. Or can we stop early? Can we make our programs to do less and therefore become faster?

We observe that if 4 divides 24, so also 6 which is the quotient of the first division. Factors occur in pairs. An exception to the rule is that for a perfect square, the two factors coincide. So, we can stop our exploration of finding a factor at its square root. But we do not want to get into floating point values as they are approximately stored. So we prefer to play with integers. So we keep checking for the factors as long as the loop variable squared is less than the given number. (Or should we check for less than or equal to the number?). So, this method is more efficient, but still evil – it applies 'break'!

```
// 5. one more attempt
        public boolean isPrime()
        {
                int i = 2; boolean tempIsPrime = true;
                while(tempIsPrime && (i * i <= val) )
                {
                        if(val % i == 0)
                                tempIsPrime = false;
                        i++;
                }
                return tempIsPrime;
        }
```

In this example, we start with the assumption that the number is a prime. That is fine. Thats what we do in proving by induction, Make a hypothesis. Then check whether it holds good. If the given number val is likely to be a prime and we are yet to exhaust all possible factors, we keep repeating. Once we find a factor, we toggle the variable tempIsPrime to false. If we fail to find a factor, our assumption is fine. We return the value of tempIsPrime.

This method is a better method compared to the earlier ones. Many programmers have a tendency to call a variable like tempIsPrime as flag. Sometimes flag takes more than two values! The name flag does not tell what it stands for. This name is not recommended.

We have discussed the way of developing looping structures at length. I do not want bore you. I will leave the rest of the programs for you to examine – but for a line or two of comments. Being a teacher, how can I avoid not saying anything? Give a microphone to a politician and you know what follows!

This method finds the greatest common divisor by repeated subtraction. Verify the claim. This is called Euclid's algorithm – is supposed to be the oldest algorithm ever known. A few questions for you.

- Can you replace subtraction by modulo arithmetic?
- Are there any conditions on the values of m and n?
- Can they be negative? Can they be zero? Can one of them be zero?

```
//L14/d1/Pair.java
	public int gcd()
	{
		int tempM = m; int tempN = n;
		while(tempM != tempN)
		{
			if(tempM > tempN)
			{
				tempM = tempM - tempN;
			}
			else
			{
				tempN = tempN - tempM;
			}
		}
		return tempM; // or should that be tempN?
	}
```

This method finds m to the power n – does exponentiation by repeated multiplication. A few questions for you.

- What shall be the conditions on the initial value of m and n?
- Can we reduce the number of multiplications?
- Can we replace the condition of while by n-- > 0 ?

```
public int power()
{
        int m = this.m; int n = this.n;
        int res = 1;
        while(n > 0)
        {
                res = res * m;
                n--;
        }
        return res;
}
```

Most of us forgot the multiplication tables long back. We can compute, but am sure we can not recite. So nice to hear children recite those in chorus!

We can multiply two numbers if we know the tables of two. Here is the algorithm called Russian Peasant Method.

```
Algorithm multiply(m, n)
s ← 0
while m is not zero
      if m is odd, add n to s
      halve m
      double n
output s
```

But something has gone wrong with our program. It seems to work for m = 15 and n = 40 and not for m = 25 and n = 40. Can you fix the bug?

```
public int russianPeasant() // Buggy....
{
        int m = this.m; int n = this.n;
        int s = 0; // should this be 1?
        while(m != 0)
        {
                if(m % 2 == 1)
                {
                        s += n;
                        m /= 2;
                        n *= 2;
                }
        }
        return s;
}
```

We will also observe (once you correct the program) that the following expression always remains the same as we enter the loop, in the loop and as we exit.

s + m * n

Such an expression is said to be a loop invariant.

We shall now attempt a few programs of generating some outputs.

```
//L15/d1/MyNum.java
        public void dispIdentityMatrix()
        {
                int n = val;
                for(int i = 1; i <= n; i++)
                {
                        for(int j = 1; j <= n; j++)
                        {
                                System.out.printf("%4d ", (i/j) * (j/i));
                        }
                        System.out.println();
                }
        }
```

As the name indicates, this method is supposed to generate identity matrix of a given size. Observe a few aspects. We require a loop in a loop as we have to generate a number of columns for each row and we have to generate a number of rows. The method printf gives us control over displaying as in 'C'. Observe the println in the outer loop is used to make the cursor jump to the next line after each row.

But this is not considered a good program. The programmer is playing tricks. Why is the expression (i/j) * (j/i) not giving 1 all the time? That is what we were taught in school. I am sure you will observe that int division of Java breaks all the good rules of Mathematics!

How about this expression? This is very clean. We understand well.

```
System.out.printf("%4d ", (i == j) ? 1 : 0); // preferred
```

Good rule of programming :

Make the code simple. Never use tricks.

What if we have to generate the following pattern for a given value (say 4).

```
          1  = 1
      1 + 2  = 3
   1+ 2 + 3 = 6
1 + 2 + 3 + 4 = 10
```

// check: //L15/d2/MyNum.java

Whenever we are faced with a complex task, we should break that into smaller manageable parts. We can first write a method to generate one line of the output.

```
// step 1; get one row
	private void dispOneRow(int n)
	{
		int s = 0;
		for(int i = 1; i < n; ++i)
		{
			System.out.printf("%d + ", i);
			s += i;
		}
		s += n; // I missed this first time I wrote - shows how bad I am as a programmer!
		System.out.printf("%d = %d\n", n, s);
	}
```

Then We can write a method to display all the lines by calling this method repeatedly.

```
public void dispSummationPattern()
{
	for(int i = 1; i <= val; i++)
	{
		dispOneRow(i);
	}
}
```

We can write a method to take care of white spaces in the beginning of each line. We can call that method in the loop before calling the method to display one row.

```
public void dispSpaces(int n)
{
	// can we avoid the loop?
	for(int i = 0; i < n; ++i)
		System.out.print("       "); // 7 spaces?
}

public void dispSummationPattern()
{
	for(int i = 1; i <= val; i++)
	{
		dispSpaces(val - i);
		dispOneRow(i);
	}
}
```

You will observe that the summation for subsequent rows starts from 1 each time. I will leave that to you to make the method better.

I am using 'leave' a number of times in this document. Every body likes 'leave', Students tend to get sick around the test week – want medical leave! A few months from now spring will dawn and we will see green leaves around us. You may remember the ironical story 'The Last Leaf'. English is an unusual language.

The next example shows how we can build programs. The requirement is to find number of factors for a range of numbers and report the highest number of factors.

```
//L15/d3/MyNum.java
	int findNumberOfFactors()
	{
		int s = 0;
		for(int i = 1; i <= val; i++)
		{
			if(val % i == 0)
			{
				s++;
			}
		}
		return s;
	}
```

The above method counts the number of factors for a given value. Can you make this better by stopping at square root of val?

The method below makes an object on the fly for numbers from 1 to n and calls the method to the find number of factors on those numbers. It tracks the highest number of factors so far. It assumes that the highest number of factors to be 0 to start with –this is a reasonable assumption. At the end returns the highest number of factors.

Go through the program and understand the logic. You may walk through the program for a small value like 10.

Good programming practice.
We learn a lot by reading and understanding programs.

```
int findHighestNumberOfFactors()
{
        int highest = 0;
        int n = this.val;
        for(int i = 1; i <= n; i++)
        {
                int count = new MyNum(i).findNumberOfFactors();
                if(count > highest)
                {
                        highest = count;
                }
        }
        return highest;
}
```

It has been proved that any program in the world can be written if we know the three constructs.

1. Assignment.
2. Selection (if and switch)
3. Looping (while, for, do .. while)

You know all of them. So you can solve all the problems in the world.

Keep doing that until I trouble you again next week.
“Do not trouble trouble till trouble troubles you!”.