CS2

Object Oriented

Programming

in

Java

Handout : Week3

prepared by N S kumar

Preface

Dear friends,

Another Week gone. Time again for another edition of the Lecture Notes.

There will be some overlap between these notes. Some aspects of programming are repeated to stress on the concepts.

I would like to add another feature called Frequently Asked Questions(FAQ). If you mail me or tell your teachers your doubts, we can add them to this handout.

I will also try to add a few assignments and quiz separately.

A couple of observations.

We are trying to enhance your learning experience. A Lot of effort goes behind the screen. There is total support by our CEO | Pro-Chancellor Prof. D. Jawahar. We have total commitment of our departments – thanks to our HODs Prof. Nitin Pujari and Dr. Shylaja. We have a team of teachers who are working towards this course. Even your seniors are giving inputs to make this course better. I think we should all be thankful to all these people involved as well as to our institution.

We do understand that you are all young and you are all brighter than any of us. You may have learnt all that is taught in the class. It does not mean that you can create ruckus in the class. It would demotivate the teachers. It will put those students who want to learn at disadvantage. We expect you to conduct the way a student should and contribute towards making the course better.

This document appears to be bigger than the earlier ones for two reasons. Quite a few programs with common code are directly put in this document. There are a few confusing concepts – these are discussed at length.

Through out the document, the masculine pronoun is used – but for one place. It does not mean anything. You may want to change that to the feminine pronoun while reading. Masculine pronoun required one letter less while typing!

Keep reading. But do not forget to contribute to this by sending your constructive feedback and lots of questions related to this course.
All the best.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com
Date : 25th Jan 2015

Treading the Object Path:

So far in these lectures, we have discussed the concepts of Object Oriented Programming. We have also discussed programs having constants, variables, simple input / output, assignment, expressions. We have dwelt in detail about the different aspects of operators like precedence, association and rank.

We have also made some simple classes with attributes(fields) and behaviour(methods or functions). We have also mentioned the special function to initialize an object - called its constructor.

We make a class to represent an entity or a thing or an idea. A class is like a user defined type. A class contains a number of attributes. We normally put a group of related classes into a file. Why should we segregate our classes into different files?

Let us consider a simple analogy. Would people at McDonalds make burgers and eat those themselves – (they may also eat a few – not sure though). Customers of McDonalds and people of McDonalds enjoy their own lives – but interact at the counter.

We would like to clearly distinguish between the creator of the class – he gives service and the user of the class(client) – he uses the service the creator provides. McDonald can cater to number of different types of customers. Customers may also choose a different joint if they do not like McDonald.

So, we use different files for the following reasons.

- producer and consumer are normally not the same person(s)
- these files can be shared with others – become a library
- these can be developed in parallel
- A bug can be traced easily. Modification (what is const in this world is change!) can be made easily without affecting others. So these are easy to maintain.

Let us look at some examples. In these discussions, we will use a simple example of a class - Rect – which abstracts a rectangle. My rectangle has two attributes length called l and breadth called b. Both are of type int. I am sure your rectangle could be different. Like a page in your book, it may have a dog ear or two!
Can you come out with a class to represent page with dog ear(s) ?

What is the difference between the following statements?
int a;
Rect r;
The first statement allocates enough memory to hold an integer value.
The second statement allocates memory just for a reference. It does not allocate memory for the fields of Rect.
r = new Rect();
The operator new causes allocation of memory for the fields of Rect and r would refer to it. This is the only way of allocating memory for an object. r is an object reference and on allocation, r refers to an object. By default, all attributes of a class are initialized to default value based on their types.

In this example, every attribute of the class Rect is an interface and therefore the client can access them directly and put values into the attributes and also find the area.

```
// L9/d1/Client.java
public class Client
{
	public static void main(String[] args)
	{
		Rect r;
		//System.out.println("r : " + r);
		 // Error. Why?
		r = new Rect();
		r.l = 20;
		r.b = 10;
		System.out.println("area : " + r.l * r.b);
	}
}
// L9/d1/Rect.java
// Class has only attributes
// everything is an interface
class Rect
{
	public int l;
	public int b;
}
```

What if the developer of the class decides to make the field names more meaningful? He changes l to length and b to breadth. What will happen to this client program? No wonder, program will not compile!

```
// L9/d2/Client.java
// Rect class has changed!
// this program does not work
```

```
public class Client
{
        public static void main(String[] args)
        {
                Rect r;
                r = new Rect(); // explain what happens here
                r.l = 20;
                r.b = 10;
                System.out.println("area : " + r.l * r.b);
        }
}

// naming convention is changed!
// clients are affected.
class Rect
{
        public int length;
        public int breadth;
}
```

Managing Change:

We know that everything changes in this world. How do we manage change? The developer of the class Rect should tell the client that he would not change those that the client can use and he may change those which the client cannot use.

He should hide the implementation and expose the interface.

In this example, the client accesses the interface of Rect. Changes to implementation of Rect will not affect the client.

Rule given by Gamma et al :

“Program to the interface and not the implementation”.

In this example, r.init(20, 10) will conceptually become Rect.init(r, 20, 10);
When we **invoke(another term for call!)** a method using an object of a class, the method of the class is called and the object is passed as the first argument.

Whenever a method is called, a block of memory for that method is created on the stack. This block of memory is called **stack frame.** The stack allows us to put something on top and remove that from the top. It is last in first out structure.

The call has number of expressions called **arguments**. These are copied within the stack frame and are accessed by the called method. These are called **parameters.**

The mechanism of associating parameters with arguments is called parameter passing technique. Java follows a technique called parameter passing by value. Arguments are copied to the corresponding parameters.

In java, arguments are copied to the corresponding parameters. Parameter Passing is by value.

So, a call to the method of a class will have an implicit argument which is the object through which the call is made. This is copied to a parameter whose name is always '**this**'. 'this' has special meaning in Java – it is called keyword.

Please note that 'this' is not present in any object. It manifests(comes into being) only when the method is called.

'this' refers to the object through which the call is made.

L9/d3/Client.java

```
// Client calls the interface methods of the Rect class
public class Client
{
	public static void main(String[] args)
	{
		Rect r;
		r = new Rect();
		r.init(20, 10);
		r.findArea();
	}
}
```

In this example, can the Client find the area of a rectangle within which a small rectangle is cut out? The answer is a NO as the client can only display the area and cannot get the area.
This implementation of findArea is flawed. It should return the area back to the client. The client can then decide what to do with it.

A method should normally return a value.
DO NOT print a value unless the method is called display!

L9/d3/Rect.java

```
// hide what could change; hide the implementation
// expose the interface; expose what should not change
class Rect
{
	private int length;
	private int breadth;
	public void init(int l, int b)
	{
		length = l;
		breadth = b;
	}

	// Rule: display in a method if the method is for displaying
	// always prefer to return a value

	public void findArea() // bad
	{
		System.out.println("area : " +  length * breadth);
	}
}
```

```
// L9/d4/Client.java
// Client calls the interface methods of the Rect class
public class Client
{
	public static void main(String[] args)
	{
		Rect r;
		r = new Rect();
		r.init(20, 10);
		System.out.println("area : " + r.findArea());
	}
}
```

This is better. The methid findArea returns the area back to the client.

```
//L9/d4/Rect.java
class Rect
{
	private int length;
	private int breadth;
	public void init(int l, int b)
	{
		length = l;
		breadth = b;
	}
	// method made better
	public int findArea()
	{
		return length * breadth;
	}
}
```

So far we have been telling you the truth, but not the whole truth. When we use the following statement

r = new Rect();

two things happen. The operator new allocates memory for the fields of Rect and then calls a special method called the constructor. This constructor is used to initialize an object. It does not create an object.

Constructor can have parameters. Compiler provides a constructor which has no parameters. This is called a default constructor. This will vanish when we make out own constructor. The client may want to use our class to initialize objects in different ways. You may order pizza with different toppings and with or without coke. We may have overloaded constructors to support what the client wants.

Constructor of a class has the name as the class itself. It has no return type.

```
// L9/d5/Client.java
// introduce parameterized constructor
public class Client
{
        public static void main(String[] args)
        {
                Rect r;
                r = new Rect(20, 10);
                System.out.println("area : " + r.findArea());
        }
}
// L9/d5/Rect.java
class Rect
{
        private int length;
        private int breadth;
        public Rect(int length, int breadth)
        {
                this.length = length;
                this.breadth = breadth;
        }
        // method made better
        public int findArea()
        {
                return length * breadth;
        }
}
```

Initialization of an object:

Giving a value or set of values to a variable or an object at the point of creation is called initialization.

Local variables are not initialized by default.

Object is created by using the operator new. It allocates memory and calls the constructor.

There are a number of ways by which the fields of an object are initialized.

1. default initialization based on the type.

In this example you will find the int fields have become 0 and boolean is false.

```
// L10/d1/Client.java
public class Client
{
	public static void main(String[] args)
	{
		Rect r;
		r = new Rect();
		System.out.println("length : " + r.getLength());
		System.out.println("breadth : " + r.getBreadth());
		System.out.println("is square : " + r.getIsSquare());
	}
}
```

Our rectangle has changed with an additional field to indicate whether the rectangle is a square. It has methods to get the values of the attributes for the client. These are called **getters**. We may have methods to change the values of the attributes of an object. These are called **setters.**

```
// L10/d1/Rect.java
class Rect
{
	private int length;
	private int breadth;
	private boolean isSquare;
	public int getBreadth() { return breadth; }
	public int getLength() { return length; }
	public boolean getIsSquare() { return isSquare; }
}
```

Initialization (Continued) :

There are a number of ways by which the fields of an object are initialized.

1. default initialization based on the type.
2. Initialize in the constructors

Observe that isSquare depends on the other fields – we say it is a computed field. This is given a value in the constructors.

Constructors are overloaded to make the client life easier.

Default constructor initializes the fields to some default (not reasonable in this example – could not think of a default rectangle!).

We have a constructor which creates a square – takes only one argument.

We have also a constructor which takes two arguments.

```
// L10/d2/Client.java
public class Client
{
        public static void main(String[] args)
        {
                Rect r1;
                r1 = new Rect();
                System.out.println("length : " + r1.getLength());
                System.out.println("breadth : " + r1.getBreadth());
                System.out.println("is square : " + r1.getIsSquare());
                Rect r2 = new Rect(20);
                System.out.println("length : " + r2.getLength());
                System.out.println("breadth : " + r2.getBreadth());
                System.out.println("is square : " + r2.getIsSquare());
                Rect r3 = new Rect(20, 10);
                System.out.println("length : " + r3.getLength());
                System.out.println("breadth : " + r3.getBreadth());
                System.out.println("is square : " + r3.getIsSquare());
        }
}

// L10/d2/Rect.java
class Rect
{
        private int length;
        private int breadth;
        private boolean isSquare;
```

```
    public Rect() // default ctor
    {
        length = 100; // should be meaningful!
        breadth = 50;
        isSquare = false;
    }
    // Assume that this is a square
    public Rect(int length) // one arg ctor
    {
        this.length = this.breadth = length;
        isSquare = true;
    }
    public Rect(int length, int breadth) // two arg ctor
    {
        this.length = length;
        this.breadth = breadth;
        this.isSquare = this.length == this.breadth;
    }

    public int getBreadth() { return breadth; }
    public int getLength() { return length; }
    public boolean getIsSquare() { return isSquare; }
}
```

Initialization (Continued) :

There are a number of ways by which the fields of an object are initialized.

1. default initialization based on the type.
2. **Initialization within the class**

If members of the class need to be initialized in the same no matter which constructor is used and the initialization is some simple assignment, then we can do that in the class itself.

3. Initialize in the constructors

```
// L10/d3/Client.java
// not shown here; same as the last one

// L10/d3/Rect.java
// default initialization in the class
class Rect
{
        private int length = 100;
        private int breadth = 50;
        private boolean isSquare = false;
        public Rect() // is this still required?
        {
        }
        public Rect(int length) // one arg ctor
        {
                this.length = length;
        }
        public Rect(int length, int breadth) // two arg ctor
        {
                this.length = length;
                this.breadth = breadth;
                this.isSquare = this.length == this.breadth;
        }
        public int getBreadth() { return breadth; }
        public int getLength() { return length; }
        public boolean getIsSquare() { return isSquare; }
}
```

Initialization (Continued) :

There are a number of ways by which the fields of an object are initialized.

1. default initialization based on the type.
2. Initialization within the class
3. Initialize in the constructors
4. **constructor calling another constructor**

If a constructor does everything tht another constructor does and then something more, we can make one call the other.

We want to make Dosa and Dosa-with-Ghee in our eatery. If the latter does everything of the former, then we can have one cook, make Dosa and then another cook(I can fit in here), add ghee to it.

```
// overloaded constructors
//	constructor calling another
//	we should not have common code across methods
//	reason: if the logic changes, we will have to change at number of places
//	moral: if one method can call the other, prefer that over copying code

// L10/d4/Client
// same as the earlier one; not shown here.

// constructor calling another in the class
class Rect
{
	private int length = 100;
	private int breadth = 50;
	private boolean isSquare = false;
	public Rect()
	{
	}
```

```
        public Rect(int length) // one arg ctor
        {
                // bad code; put only for demonstration purpose
                System.out.println("in one arg ctor");
                this.length = length;
        }

        public Rect(int length, int breadth) // two arg ctor
        {
                // this calls the ctor which takes one argument
                this(length); // calls another
                this.breadth = breadth;
                this.isSquare = this.length == this.breadth;
        }

        public int getBreadth() { return breadth; }
        public int getLength() { return length; }
        public boolean getIsSquare() { return isSquare; }
}
```

Initialization (Continued) :

There are a number of ways by which the fields of an object are initialized.

1. default initialization based on the type.
2. Initialization within the class
3. **Initialization block**

What if the batter for all types of dosas and idlis are same. Whatever is common between them can be done by a cook.

If the constructors have common code in the beginning, then that common code can be put in the initialization block.

4. Initialize in the constructors
5. constructor calling another constructor

```
// L10/d5/Client.java
// overloaded constructors
//      what if two (or many ctors) have some common code
//      capture that code in initialization block
// code removed removed; same as the one in the earlier example

// L10/d5/Rect.java
// initialization block
//      executed whenever an object is created
//      captures the commonality across ctors
class Rect
{
        private int length;
        private int breadth;
        private boolean isSquare;
         {
                // bad code; put only for demonstration purpose
                System.out.println("initialization block");
                isSquare = false;
        }
        public Rect()
        {
        }
        public Rect(int length) // one arg ctor
        {
                // bad code; put only for demonstration purpose
                System.out.println("in one arg ctor");
                this.length = length;
        }
```

```
	public Rect(int length, int breadth) // two arg ctor
	{
		this(length); // calls another
		this.breadth = breadth;
		this.isSquare = this.length == this.breadth;
	}
	public int getBreadth() { return breadth; }
	public int getLength() { return length; }
	public boolean getIsSquare() { return isSquare; }
}
-------------------------------------------------------------------------------------------------
```

Let us recollect the order of initialization in Java.

1. default initialization based on the type.
2. Initialization within the class
3. Initialization block
4. Initialize in the constructors
5. constructor calling another constructor

Something for you to experiment. In java, we use final to state that something is a constant – that it will not change once given a value. Your usn will not change during your stay in our college. Your pan numbers (assuming that you are paying income tax) will not change. Your passport number will not change even though it is not the same for your friend.

Experiment:

1. how to initialize a final member of a class?
2. Can we initialize more than once?
3. What is the default value if not initialized?
4. Can we assign to a final member after initialization?
5. Can reference be initialized? What becomes a constant? Reference or referent?

Life and Scope of variables:

A variable is given a location as the control enters the body of the method. When the control leaves the method, the memory given to the variable is taken back. So we say that the variable has life – has existence – in that block.

An object gets life when the new operator is called. At the end of the block the reference holding reference to the object is removed – dies. But the object remains but we have no way of accessing it. So, we have a location – in this case – an object – which has no access. Such a thing is called **garbage**. Creation of garbage decreases the memory available. When memory becomes low, Java clears all these locations and make them useful again. This concept is called **garbage collection**.

This concept is similar to what happens in our canteens. You all consume coffe and/or tea. After sometime there will be no cups available. Then the manager calls out for the boy to clean these cups and make them useful again. Sometime cups are not well cleaned – but not so in Java!

Overloading Methods:

We may want to change the sides of a rectangle – modify by a factor. We may want to change the length and the breadth by different factors. So, we require two methods. Should we call by different names or same name? Giving the same name makes the life of the client easier. He has to remember just one name. We may observe that the operator + stands for addition of all numeric types. The operations carried internally are different. This concept of giving the same name for a group of methods logically similar is called overloading.

We overload methods to make the client life easier.

Compiler can make out which function to invoke(means call) based on matching of arguments(in method call) and of parameters(in method definition).

Overloading is resolved based on the number, type and/or order of arguments and parameters.

The next two examples illustrate this concept.

```
// L11/d1/Client.java
--------------------------
// methods
//	we would to scale the length and the breadth.
//	how many arguments are required?
//		should we also pass length and breadth?
//		should we pass an object explicitly?
//	what should the method return?
public class Client
{
	public static void main(String[] args)
	{
		Rect r1 = new Rect(20, 10);
		System.out.println("length : " + r1.getLength());
		System.out.println("breadth : " + r1.getBreadth());
		System.out.println("is square : " + r1.getIsSquare());
		int scaleFactor = 2;
		r1.scaleSides(scaleFactor);
		System.out.println("length : " + r1.getLength());
		System.out.println("breadth : " + r1.getBreadth());
		System.out.println("is square : " + r1.getIsSquare());
	}

}
```

```java
// L11/d1/Rect.java
class Rect
{
        private int length;
        private int breadth;
        private boolean isSquare;
        public Rect(int length, int breadth)
        {
                this.length = length;
                this.breadth = breadth;
                this.isSquare = this.length == this.breadth;
        }
        public int getBreadth() { return breadth; }
        public int getLength() { return length; }
        public boolean getIsSquare() { return isSquare; }

        public void scaleSides(int scaleFactor)
        {
                length *= scaleFactor;
                breadth *= scaleFactor;
        }
}
```

```
// L11/d2/Client.java
// methods
public class Client
{
	public static void main(String[] args)
	{
		Rect r1 = new Rect(20, 10);
		System.out.println("length : " + r1.getLength());
		System.out.println("breadth : " + r1.getBreadth());
		System.out.println("is square : " + r1.getIsSquare());
		int scaleFactorLength = 2;
		int scaleFactorBreadth = 3;
		r1.scaleSides(scaleFactorLength, scaleFactorBreadth);
		System.out.println("length : " + r1.getLength());
		System.out.println("breadth : " + r1.getBreadth());
		System.out.println("is square : " + r1.getIsSquare());
	}
}
// L11/d2/Rect.java
// methods
class Rect
{
	private int length;
	private int breadth;
	private boolean isSquare;
	public Rect(int length, int breadth)
	{
		this.length = length;
		this.breadth = breadth;
		this.isSquare = this.length == this.breadth;
	}

	public int getBreadth() { return breadth; }
	public int getLength() { return length; }
	public boolean getIsSquare() { return isSquare; }
	public void scaleSides(int scaleFactor)
	{
		length *= scaleFactor;
		breadth *= scaleFactor;
	}
	public void scaleSides(int scaleFactorLength,
		 int scaleFactorBreadth)
	{
		length *= scaleFactorLength;
		breadth *= scaleFactorBreadth;
		// Spot the bug!!
	}
}
```

Object as an argument for a method:

If we want to compare two rectangles, we can do that by calling the method using one object and passing a copy of the other. Parameter passing is by value(copy). In this example, 'this' will refer to r1 and the parameter called other will refer to r2.

```
//L11/d3/Client.java
// methods
//	can an object be an argument?
```

```
public class Client
{
	public static void main(String[] args)
	{
		Rect r1 = new Rect(20, 10);
		Rect r2 = new Rect(20, 10); // what if Rect r2 = null;  ?
		// can we compare whether two rectangles are same?
		// do we require accessors? can a method of the class do the job?
		// is so how to call?
		System.out.println("same : " + r1.isSame(r2));
	}
}

//L11/d3/Rect.java
// methods
class Rect
{
	private int length;
	private int breadth;
	private boolean isSquare;
	public Rect(int length, int breadth)
	{
		this.length = length;
		this.breadth = breadth;
		this.isSquare = this.length == this.breadth;
	}
	// what would happen if an object of some other class is passed as arg?
	//			if rect object passed as arg is not created?
```

```
        public boolean isSame(Rect other)
        {
                return length == other.length && breadth == other.breadth;
        }
}
```

Observe in the above example, the method isSame is accessing length and breadth of the object referred by other. Is this possible? We have made length and breadth implementation (that is, private). The answer is simple. We can.

Access control is with respect to the class and is not with respect to any particular object.

Within the method of a class, we can access any implementation of any object of the class which is visible in that method.

A method returning an object:

Any method can return an object which it knows. In this example a new object is created by calling new operator – fields are filled – reference is returned.

You may want to think: Can we use a statement like the one below?

return this;

```
// methods
//      can a method return an object?
//      want to create a new Rect with length and breadth doubled?
//      Think: can the method modify the given rect itself and return it?
```

```
//L11/d4/Client.java
public class Client
{
        public static void main(String[] args)
        {
                Rect r1 = new Rect(20, 10);
                Rect r2 = r1.DoubleRect();
                r1.disp();
                r2.disp();
        }
}

//L11/d4/Rect.java
// methods:
//      returning an object
class Rect
{
        private int length;
        private int breadth;
        private boolean isSquare;
        public Rect() { }
        public Rect(int length, int breadth)
        {
                this.length = length;
                this.breadth = breadth;
                this.isSquare = this.length == this.breadth;
        }
        public void disp()
        {
                System.out.println("length : " + length + " breadth : " + breadth);
        }
        public Rect DoubleRect()
        {
//              Error: why?
//              Rect temp; temp.length = length * 2; temp.breadth = breadth
* 2;

                Rect temp = new Rect(); // would fail if there is no default ctor
                temp.length = length * 2; temp.breadth = breadth * 2;
                return temp;

//      how about this?
//              return new Rect(length * 2, breadth * 2);
        }

}
```

Parameter passing once again:

Let us discuss this topic one more time – we will put this to rest – RIP.

Parameter passing is by value.

- If we change the parameter of a simple type in the called method, the variables of the caller will not change. You would have changed the copy!
- If the argument is an object reference, then the parameter is a copy of the reference. If the parameter is changed (which is a copy of the argument), argument will not change. The Object remains unaltered.

Your friend has a key to her locker in the bank – has something very valuable for her (look at this. Suddenly the keyboard changed the gender!). Let us say you have made a copy of your friend's key. If you file(mechanical filing – not file on your computer), your friend's key will not change.

- If the argument is an object reference, then the parameter is a copy of the reference. If you change the object using the parameter, whatever the argument refers to would be changed.

Your friend has a key to her locker in the bank – has something very valuable for her. Let us say you have made a copy of your friend's key. If you access the locker through your key(copy of your friend's key) – you are a nice person – so you may put something in the locker – not a snake of course), then your friend's locker has changed. The next time your friend opens the locker, she will get the shock of her life!

Now you may walk through these examples.

```
//L11/d5/Client.java
// methods
//	Some experiments in parameter passing
public class Client
{
	public static void main(String[] args)
	{
		Rect r1 = new Rect(20, 10);
		Rect r2 = r1;
		r1.disp();
		r2.disp();
		System.out.println();
		r1.changeLength(30);
		r1.disp();
		r2.disp(); // would r2 change? why?
		System.out.println();
		r2.disp();
		r1.mystery1(r2);
		r2.disp();
		System.out.println();
		r2.disp();
		r1.mystery2(r2);
		r2.disp();
		System.out.println();
	}
}

//L11/d5/Rect.java
// methods:
class Rect
{
	private int length;
	private int breadth;
	private boolean isSquare;
	public Rect() { }
	public Rect(int length, int breadth)
	{
		this.length = length;
		this.breadth = breadth;
		this.isSquare = this.length == this.breadth;
	}
	public void disp()
	{
		System.out.println("length : " + length + " breadth : " + breadth);
	}

	public void changeLength(int length)
	{
		this.length = length;
	}
	public void mystery1(Rect that)
	{
		that = new Rect(111, 222);
	}
	public void mystery2(Rect that)
	{
		that.length = 333;
		that.breadth = 444;
	}
}
```

Let us make another class – a triangle – to represent a Triangle of geometry. We will specify the three sides and we will assume that the triangle is formed. Can we classify the triangle as equilateral, isosceles or scalene? I am a bit sloppy and providing some computed members as interfaces – equi, iso and scalene. Please do not change them in the client code. If you do, then you may end up with a triangle which is all of them simultaneously. You may change this class and make it better by providing methods to check whether the triangle is equilateral, isosceles or scalene. Some homework for you!

The files are in the directory L12. The client program remains same. The method to find the triangle type keeps changing. We will walk through these changes.

Observe that we have tried a few possible variations to check whether the given Triangle class is correct. This is called testing. You may wonder why we have tried 3 different cases for isosceles triangle!

```
//L12/d1/Client.java
// methods
//	selection
public class Client
{
	public static void main(String[] args)
	{
		Triangle t1 = new Triangle(5, 5, 5);
		t1.disp();
		Triangle t2 = new Triangle(5, 5, 4);
		t2.disp();
		Triangle t3 = new Triangle(5, 4, 4);
		t3.disp();
		Triangle t4 = new Triangle(5, 4, 5);
		t4.disp();
		Triangle t5 = new Triangle(5, 4, 3);
		t5.disp();
	}
}
```

```
//L12/d1/Triangle.java
// classify triangle
class Triangle
{
	private int a;
	private int b;
	private int c;
	public boolean equi; // there are interfaces!
	public boolean iso;
	public boolean scalene;
	}



	public Triangle(int a, int b, int c)
	{
		this.a = a;
		this.b = b;
		this.c = c;
		classifyTriangle();
	}
	public void disp()
	{
		System.out.println("sides : " + a + " : " + b + " : " + c);
		System.out.println("equi : " +  equi + ": iso : " + iso +
			" : scalene : " + scalene);
		System.out.println();
	}

	private void classifyTriangle()
	{
		if(a == b)
			if(b == c) // should we compare c and a?
				equi = true;
			else
				iso = true;
		else if(b == c)
			iso = true;
		else if(c == a) // do we require this?
			iso = true;
		else
			scalene = true;
	}
}
```

We are using selection (if statement) to choose between the alternatives.
We are using boolean fields to hold the result of these comparisons. Observe by default, all these boolean fields become false because of default initialization based on type.
In this example, we have if within if, Such a construct is called nested if statement.

We may modify the above method by using logical and, or and not operators.

```
// L12/d2/Triangle.java
// is the client affected because of the change?
	private void classifyTriangle() // logical operator
	{
		if(a == b && b == c)
			equi = true;
		else if(a == b || b == c || c == a)
			iso = true;
		else
			scalene = true;
	}
```

&& : is a binary operator. The result is true if both of its operands are true.
|| : is a binary operator. The result is true if any one of them is true.
! : is a unary operator. It toggles the operand. True becomes false and false becomes true.

Short Circuit Evaluation:
If the first operand of && operator is false, then the whole expression is false. There is no necessity to evaluate the second operand.
If the first operand of || operator is true, then the whole expression is true. There is no necessity to evaluate the second operand.
We can stop as soon as the truth or falsehood is found out. That what happens in Java. This is called short circuit evaluation.

Let us look one more variation of the code.

```
// L12/d3/Triangle.java
// use boolean variables
        private void classifyTriangle()
        {
                equi = a == b && b == c;
                iso = !equi && (a == b || b == c || c == a); // short ckt;
                // don't care in K map
                scalene = ! equi && ! iso; // can we use DeMorgan's theorem?
        }
```

We use the results of earlier computation. Scalene is neither isosceles not scalene.

Dijkstra said in his article “The Humble Programmer” that programming is a human activity. There is no unique way of doing it. So you cannot easily copy! (I know you would not!). So another way of solving this problem.

```
// L12/d4/Triangle.java
        // no shortage of number of ways of writing !
        private void classifyTriangle()
        {
                int count = 0;
                if(a == b) ++count;
                if(b == c) ++count;
                if(c == a) ++count; // can the count be 2 ?
                switch(count)
                {
                        case 3 : equi = true; break;
                        case 1 : iso = true; break;
                        // case 0 :
                        default : scalene = true; break;
                }
        }
```

We count the number of times a pair of sides match. It can be 3 or 1 or 0. Think: why it is never 2? Based on that we can classify a triangle. Here, we have to compare a single integer variable with number of constants for equality. In such cases, we prefer the switch statement. The value of the expression of switch decides the point of entry into the switch clauses (and not exit. Thats why we use break to take the control out of the block!). We can have a 'catch all' clause – that is called default.

Observe that the switch statement differs from if statement in the following points.

- same variable compared
- compared with constants
- compared for equality
- In Java, expression of switch should be any of the integer types or string.

Thats all for now. We will continue next week with looping structures – we all like Merry-Go-Round. Don't we? Or should I say “London Eye”?