Encapsulation means putting together variables and methods as a single unit called a Class.

An external method can access the public methods and properties i.e value of an attribute of an encapsulated class.

A fully encapsulated class is one in which all properties are private and can only be accessed by methods in the same class. This provides security against unauthorized attempts to modify data. Access to these private properties is provided via public getter() and setter() methods.

Consider the following example.

```
public class Computer
{
	//Private Data Members/attributes
	private String model;
	private int memory_gb;
	private String operating_system;

	//Public Constructor
	public Computer()
	{
		this.model="X1Y2";
		this.memory=32;
		this.operating_system="Windows";
	}
	//Public Getter() methods (Naming convention : getXx() where Xx is the name of the property
	public String getModel()
	{
		return this.model;
	}
	//Public setter() methods
	public boolean setMemory_gb(int m)
	{
		//validation for property
		if(m<0)
		{
			//memory cannot be negative
			return false;
		}
```

```
                else
                {
                        this.memory_gb=m;
                        return true;
                }
        }
}
```

If a programmer needs to access the "model" of a computer, he can call the getModel method of the class as shown below

```
        Computer c = new Computer(); //Call to constructor
        String model=c.getModel();
        System.out.println(model);
```

OUTPUT :

```
        X1Y1
```

If a programmer needs to change the value for "memory" of a computer, he can call the setMemory_gb method of the class as shown below

```
        Computer c = new Computer(); //Call to constructor
        boolean r=c.setMemory_gb(8);
        boolean r1=c.setMemory_gb(-2);
        //true indicates success and false indicates failure
        System.out.print(r+"  "+r2+"  ");
        System.out.print(c.getMemory_gb());
```

OUTPUT :

```
        true  false  8
```

Inheritance is a concept that allows us to create classes that share certain features and also have their own unique features. In simple terms, it eliminates redundant code for shared features.

Deriving a new class from an existing class definition is called Inheritance.

In the following example, the class Computer is a more general definition. This forms the PARENT/BASE class. The classes Laptop and Desktop are specific forms of Computer. Hence, these are called CHILD/SUB/DERIVED classes. The sub classes have access to non-private members of the base class and can define their own unique attributes or behavior.

```
public class Computer
{
	public String model;    //Public member can be accessed by other classes
	//Private Data Members/attributes
	private int memory_gb;
	private String operating_system;

	//Public Constructor
	public Computer()
	{
		this.model="X1Y2";
		this.memory=32;
		this.operating_system="Windows";
	}
	public String getModel()
	{
		return "Desktop : Model - "+model;
	}
}
class Laptop extends Computer
{
	public Laptop()
	{
		super();   //call to constructor in parent
	}
	public String getModel()
```

```
        {
                return "Laptop : Model - "+model;          //model is the derived attribute
        }
}
class Desktop extends Computer
{
        private String monitor_version; //unique feature of desktop
        public Desktop()
        {
                super();   //call to constructor in parent
        }
        public String getModel()
        {
                return "Desktop : Model - "+model;
        }
}
```

Implementation:

```
        Computer c = new Computer();
        System.out.println(c.getModel());
        Laptop l = new Laptop();
        System.out.println(l.getModel());
        Desktop d = new Desktop();
        System.out.println(d.getModel());
```

OUTPUT :

```
        X1Y1
        Laptop : Model -  X1Y1
        Desktop : Model - X1Y1
```

Polymorphism means the ability to take multiple ("poly") forms("morphos"). It is a feature that allows a word to be interpreted differently in different contexts. In java there are two categories of polymorphism: Static/Compile-time and Dynamic/Run-time.

Compile-time polymorphism can be implemented using method overloading. Method overloading is a feature that allows a method to execute differently by altering its argument list. In simple terms, it allows us to have multiple functions with the same name but having different arguments, within the same class.

For example:

```
class MyClass
{
	// method 1
	public void display(String s)
	{
		System.out.println("Argument is a String- "+ s);
	}
	// method 2
	public void display (int i)
	{
		System.out.println("Argument is an int- "+ i);
	}
	//method 3
	public void display (String s, int i)
	{
		System.out.println("Mixed arguments- "+ s + "--" + i);
	}
}
```

Implementation

```
	MyClass m=new MyClass();
	m.display(20);   //calls method 2
	m.display("Twenty");     //calls method 1
	m.display("twenty",20); //calls method 3
```

Run-time polymorphism or method overriding plays a key role in inheritance. A base class derives the methods of its parent, but in some cases it needs to override this default behavior. This creates two method definitions having the exact signature, one in the base class and one in the derived class. Since the signature is identical, the method call is resolved at run-time instead of compile-time based on the object that is used in the call.

Consider the example below:

```
class ParentClass
{
	// parent method
	public void display(int s)
	{
		System.out.println("This is the Parent class");
	}
}
class ChildClass extends ParentClass
{
	//child method having same signature as parent
	//NOTE: If this method does'nt exist, the parent's method will be called by default
	public void display(int s)
	{
		System.out.println("This is the Child class");
	}
}
```

Implementation:

```
	ParentClass p=new ParentClass();
	p.display();   //call to parent class method
	ChildClass c=new ChildClass();
	c.display();  //call to child class method
```

OUTPUT :

```
	This is the Parent class
	This is the Child class
```

Abstraction deals with hiding complex implementation details and providing a basic interface with which you can interact. There are two ways of implementing abstraction in java. One is by creating abstract class - a class that cannot be instantiated but can be inherited by other concrete classes. Second is via an interface that is implemented by a concrete class.

In the following example, I have an abstract class called bake consisting of an abstract method setTime that varies for different objects, and a common method bake(). The classes Cake and Cookie inherit from the abstract class Bake and provide their implementation for the abstract method setTime(). Hence the abstract class hides the implementation of the setTime method.

If a new class is created extending Bake, it will have to provide an implementation for the setTime method.

```
public abstract class Bake
{
	public abstract void setTime();
	public void bake()
	{
		System.out.println("Baking...");
	}
}
class Cake extends Bake
{
	@Override
	public void setTime()
	{
		System.out.println("Baking time for cake : 1hr");
	}
}
class Cookie extends Bake
{
	@Override
	public void setTime()
	{
		System.out.println("Baking time for cookies : 20min");
	}
}
```

Implementation :

```
Cake cake=new Cake()
cake.setTime();
cake.bake();
Cookie cookie=new Cookie();
cookie.bake();
```

OUTPUT :

```
Baking time for cake : 1hr
Baking...
Baking time for cookies : 20min
Baking...
```

The above example can also be implemented using an interface instead of abstract class. The output would be the same as shown above. The key difference is that an interface can contain only abstract methods. Hence each concrete class that implements an interface needs to provide implementation for every method declared in the interface.

```
//interfaces are abstract by default
public interface class Bake
{
	//interface methods are abstract by default
	public void setTime();
	public void bake();
}
class Cake implements Bake
{
	@Override
	public void setTime()
	{
		System.out.println("Baking time for cake : 1hr");
	}
	@Override
	public void bake()
	{
		System.out.println("Baking...");
```

```
	}
}
class Cookie implements Bake
{
	@Override
	public void setTime()
	{
		System.out.println("Baking time for cookies : 20min");
	}
	@Override
	public void bake()
	{
		System.out.println("Baking...");
	}
}
```