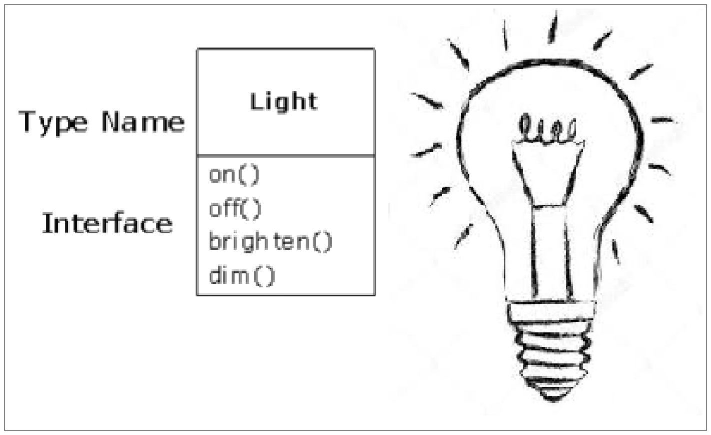
Chapter 1 . Introduction to objects

- Class is nothing but a type, i.e. class shows its represents some type. Whenever an object is been created from a class they all are nothing but image of the same class. Hence its clear that class which is a type gives an frame work or design upon which object is created. This is called **ABSTRACTION**.

- **A Object has an Interface :**

Putting member variables in a class and access to them is given only through by member method which are also called as interfaces in layman term (not java proper, java interface is something different). This way of keeping member variables private and allowing access only through interfaces is called **ENCAPSULATION**.

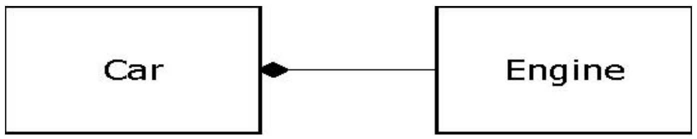


```
Light lt = new Light();
lt.on();
```

Above program we can see how encapsulation has been used.

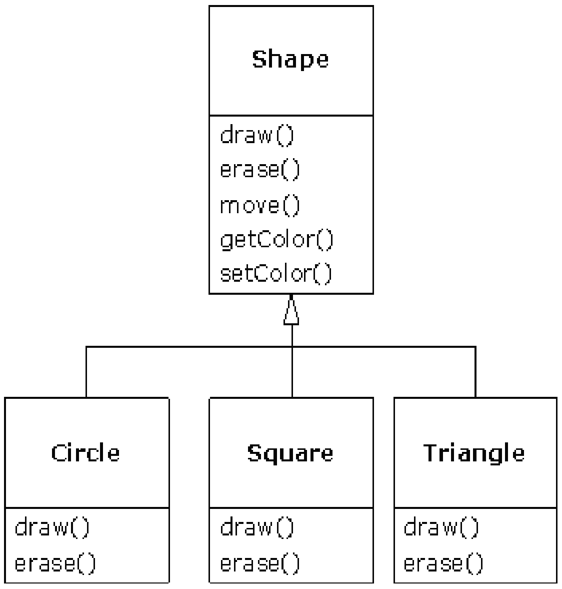
- Reusing the implementation :

Simplest way to reuse a class is by using a object of that class inside a new class. So a new class can be composed by creating many objects which is called **COMPOSITION**. If composition happens dynamically then its called **AGGREGATION**. Composition is often referred to as a “has-a” relationship, as in “A car has an engine”.



Diamond in UML diagram above shows composition.

➢ **<u>Inheritance and Interchangable objects with polymorphism</u> :**



We know that with inheritance we can have overridden methods too. Lets say Shape class have one more method doSomething() as shown below.

```
void doSomething(Shape shape) {
       shape.erase();
       // ...
       shape.draw();
    }
```

Above we can see method doSomething() speak to any shape. Whether its circle or square it can draw or erase. Now lets say some part of the program calls doSomething() as below

```
    Circle circle = new Circle();
    Triangle triangle = new Triangle();
    Line line= new Line();
    doSomething(circle);
    doSomething(triangle);
     doSomething(line);
```

Above we can see the in doSomething(circle) circle(a sub-class object) has been passed to the method who was expecting a shape. This treating of sub-class object as super-class is called **UPCASTING.** (This happens because of polymorphism where subclass ref can be assigned to superclass).

Above when doSomething(circle) is called it gets into the body and found that shape.draw() or shape.erase() needs to be called but at compile time compiler will be not sure that which piece of code will be executed. In traditional programming execution flow was known during compile time which is called **EARLY BINDING** but this will produce falls results with java . So java uses **LATE BINDING** i.e. execution flow will be determined during Run Time.

- Example for the Late binding scenario is given below

```java
class BaseClass
{
	int a=5,b=5;
	int sumInt()
	{
		int sum = 0;
		sum = a+b;
		return sum;
	}

	void printSum(BaseClass obj)
	{
		System.out.println("this is base class method "+obj.sumInt());
	}
}

class DerivedClass extends BaseClass
{
	int i=10,j=10;
	int sumInt()
	{
		int sum = 0;
		sum = i+j;
		return sum;
	}

}

public class PolymrphClass {

	public static void main(String[] args)
	 {
		// TODO Auto-generated method stub
		BaseClass bc = new BaseClass();
		DerivedClass dc = new DerivedClass();
		System.out.println(bc.sumInt());
		System.out.println(dc.sumInt());

		bc.printSum(dc);

		bc = dc;
		System.out.println(bc.sumInt());
	}

}
```

```
Output :
10
20
this is base class method 20
20
```

```java
package Chap1TIJ;

class BaseClass1
{
	String str1;
	BaseClass1(String str)
	{
		str1 = str;
	}
	void PrintBaseMet1(BaseClass1 obj)
	{
		System.out.println(str1);
	}
	void PrintBaseMet2(BaseClass1 obj)
	{
		System.out.println("overloaded base");
	}
	void PrintMet3()
	{
		System.out.println("overridden base");
	}
}

class DeriveClass1 extends BaseClass1
{
	String str2;
	DeriveClass1(String str)
	{
		super("derive base");
		str2 = str;
	}
	void PrintDeriveMet1(DeriveClass1 obj)
	{
		System.out.println(str2);
	}
	void PrintDeriveMet2(DeriveClass1 obj)
	{
		System.out.println("overloaded derive");
	}
	void PrintMet3()
	{
		System.out.println("overridden derive");
	}
}

public class CheckUpcastingThoroughly
{
	public static void main(String[] args)
	{
		BaseClass1 bc1 = new BaseClass1("Base");
		DeriveClass1 bc2 = new DeriveClass1("Derive");

		bc1.PrintBaseMet1(bc1);
		bc1.PrintBaseMet2(bc1);
		bc1.PrintMet3();

		bc2.PrintDeriveMet1(bc2);
```

```
            bc2.PrintDeriveMet2(bc2);
            bc2.PrintMet3();

            bc1.PrintBaseMet1(bc2);
            bc1.PrintBaseMet2(bc2);
            bc2.PrintMet3();
      }
}
Output:
Base
overloaded base
overridden base
Derive
overloaded derive
overridden derive
Base
overloaded base
overridden derive
```

➢ In above example we can see that when method has been overloaded or overridden then it will work accordingly and upcasting don't take place. Only in case of method which is unique to superclass and doesn't present in subclass like method PrintBaseMet1() upcasting take place.

➢ **Containers and Parameterized type (Generics)** :

- In general, one wont know how many objects he needs to be created and how to store them until during Run Time. Java always provides solution by creating more objects. The new type of objects that solves this particular problem holds references to other objects. This new objects are generally called as Container(or collection also).

- Collection stores only objects. So when references are stored in containers they upcast them to objects hence they lose their character and while fetching back we get objects. To get the particular object references we need to downcast this objects which is called **DOWNCASTING.**

- As during upcasting we go specific to the more general(derived to base) type so in downcasting its opposite, we go from general to specific. Upcasting is easy as we need circle is shape but downcasting may arise problem because a object can be shape or circle.

- However downcasting is not that dangerous because if you got to the wrong object then Error will occur called exception. But still downcasting and runtime checks required extra time for running program and extra effort from the programmer. Hence we use **PARAMETERIZED TYPE** mechanism.

- Parameterized type is a class which compiler customized directy to use with particular type. Example for parameterized container compiler customize the container so that it could accept only shapes and fetch only shapes.

- One big change in Java SE5 is addition of parameterized types called **GENERICS.** Generics are recognized with the angle brackets with types inside. Example a ArrayList that holds shapes can be created like this.

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```