Object Oriented Programming (OOP)

Mohamed Ezz

Lecture 9

Lecture Objectives

- ✓ Define reusable classes based on inheritance
- ✓ Define abstract classes and abstract methods.
- ✓ Understand how to interface with class specification
- ✓ Differentiate the abstract classes and Java interface.
- ✓ Understand Multiple Inheritance
- ✓ Define Design pattern
- ✓ Explore alternative Design patterns

Review

- ✓ The following statement will give **Syntax** error or **Semantic** error :

  Shape someShape= new Shape ();
  Rectangle rec= (Circle) someShape;

- ✓ Write the signature of the following methods:
  - ➢equals
  - ➢toString
  - ➢clone

- ✓ Why using the above methods?

Review: Use IS or Has-A

- ✓ Which implementation is beter ?
  - ➢ Approach 1:

    ```
    class Faculty extends ArrayList {



    }
    ```

  - ➢ Approach 1:

    ```
    class Faculty {
     ArrayList  students;

    }
    ```

Review: Common mistakes

The Faculty class simply needs to reuse the service provided by the ArrayList class.

- ✓ **First approach called**: code reuse by inheritance
- ✓ **Second approach called**: reuse code by composition.

Suppose we need to modify the data structure class from ArrayList to HashMap for better performance as we did:

- ➢ With the **inheritance approach**, any client that uses the inherited methods of ArrayList needs to be rewritten.
- ➢ With the **composition approach**, the client that uses only the methods defined for the Faculty class will continue to work without change.

Review: Inheritance

```
class Shape{
    protected int color = 0;
    public void setColor(int color){
       this.color=color;
    }
    public int getColor(){
       return color;
    }
 public float computeArea (){
       return 0;
    }
}
```

```
class Circle extends Shape{
    private int radius = 0;

   public Circle (int r){
        radius =r;
    }
 public float computeArea (){
       return 22 /7* radius* radius;
    }
 public void doubleSize (){
       radius= 2 * radius;
    }
}
```

Are computeArea method implementation required?

ShapeCircle

What we should do?

- We need to improve the situation by preventing a **developer** from instantiating the **Super** class, because a developer has **marked it as having missing functionality**.
- It also provides **compile-time safety** so that you can ensure that any class that extend your **Super** class provide the bare minimum functionality to work
- Inheritors somehow have to magically know that they **have** to override a method in order to make it work.

Solution: Abstract Class

- This is a class with at least one method without implementation (abstract)
- You can not create instance from that class
- The inherited class from this abstract may implement the abstract methods

Shape Abstract Example

```
abstract class Shape{
    protected int color = 0;
    public void setColor(int color){
       this.color=color;
    }
    public int getColor(){
       return color;
    }
 abstract public float computeArea() ;
// need to be implemented by
//descendent class (child)
}
```

```
class Circle extends Shape{
    private int radius = 0;

  public Circle (int r){
       radius =r;
    }

 public float computeArea (){
       return 22 /7* radius* radius;
    }

 public void doubleSize (){
       radius= 2 * radius;
    }

}
```

Shape Abstract Example cont.

```
class Triangle extends Shape{
     private int base = 0;
     private int height = 0;

    public Triangle (int h, int b){
        base=b;
        height=h;
     }
 public float computeArea (){
        return 0.5 * base * height;
     }
}
```

```
class Rectangle extends Shape{
     private int width = 0;
     private int height = 0;

    public Rectangle(int h, int w){
        width=w;
        height=h;
     }

    public float computeArea (){
        return width* height;
     }

 public void swap(){
        int i= width;
         width = height;
         height = I;
     }
}
```

Abstract Classes in Java

- Abstract classes created using the **abstract** keyword:

  public abstract class Shape{ … }

- In an abstract class, several abstract methods are declared.

  - An abstract method is not implemented in the class, only declared. The body of the method is then implemented in subclass.
  - An abstract method is decorated with an extra "**abstract**" keyword.

- Abstract classes can not be instantiated! So the following is illegal:

  Shape s = new Shape();

Abstract methods Example

```
abstract class Shape{
     protected int color = 0;
     protected Point origin;
     public void setColor(int color){
         this.color=color;
     }
     public int getColor(){
         return color;
     }

    abstract public float computeArea() ;
    abstract public float computePerimeter () ;
   abstract void draw();
   abstract void resize();
  public void move(Point newPlace)  ? abstract

}
```

- Abstract methods are declared but do not contain an implementation.
- So Shape can NOT be instantiated
  (ie., can not be used to create an object.)

Example of Abstract Classes

```
abstract class Stack {
   abstract void push(Object o);
   abstract Object pop();
}

public class ArrayStack extends Stack {
   .... // declare elems[] and top;
   void push(Object o) {   elems[top++] = o; }
   Object pop() {    return elems[--top]; }
}

class LinkedStack extends Stack {
   .... // declare ...
   void push(Object o) {  .... }
   Object pop() {  ....  }
}
```

All methods abstract

What we should do?

- Describe what operations can be performed on an object.
- Provides **compile-time safety** so that you can ensure that any class that extend your **Super** class provide the bare all functionality to work
- Inheritors somehow have to magically know that they **have** to override a method in order to make it work.

Interfaces

Defines a set of methods that a class must implement

- An *interface* is like a class with nothing but abstract methods and final and static fields (constants).
- An *interface* can also have fields, but the fields must be declared as final and static.
- All methods are abstract implicitly.
- *Interface* can be added to a class that is already a subclass of another class. (implements)
- To declare an interface:

```
public interface ImportTax {
        public double calculateTax( );
}
```

- Use **implement** instead of **extend** when inherit from **interface**

Example of *Interfaces* (1/2)

```
interface Stack {
  void push(Object o);
  Object pop();
}

public class ArrayStack implements Stack {
  .... // declare elems[] and top;
  void push(Object o) {   elems[top++] = o; }
  Object pop() {    return elems[--top]; }
}

class LinkedStack implements Stack {
  .... // declare ...
  void push(Object o) {  .... }
  Object pop() {  ....  }
}
```

Example of *Interfaces* (2/2)

```
public class TestStack{
    public static void main (String arg[]){
        ArrayStack as= new ArrayStack ();
        LinkedStack ls= new LinkedStack ();
       testStack(as);
       testStack(ls);

   }
   static public int testStack(Stack s) { ....
       s.push(5);     s.push(4);
       System.out.println(s.pop());
       System.out.println(s.pop());
   }
}
```

Interfaces definition Oracle

- There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts.
- Each group should be able to write their code without any knowledge of how the other group's code is written.
- Generally speaking, *interfaces* are such contracts.
- Try to anticipate all uses for your interface and specify it completely from the beginning?

Difference between Abstract Class and *Interface*

- An abstract class is a class containing several abstract methods.

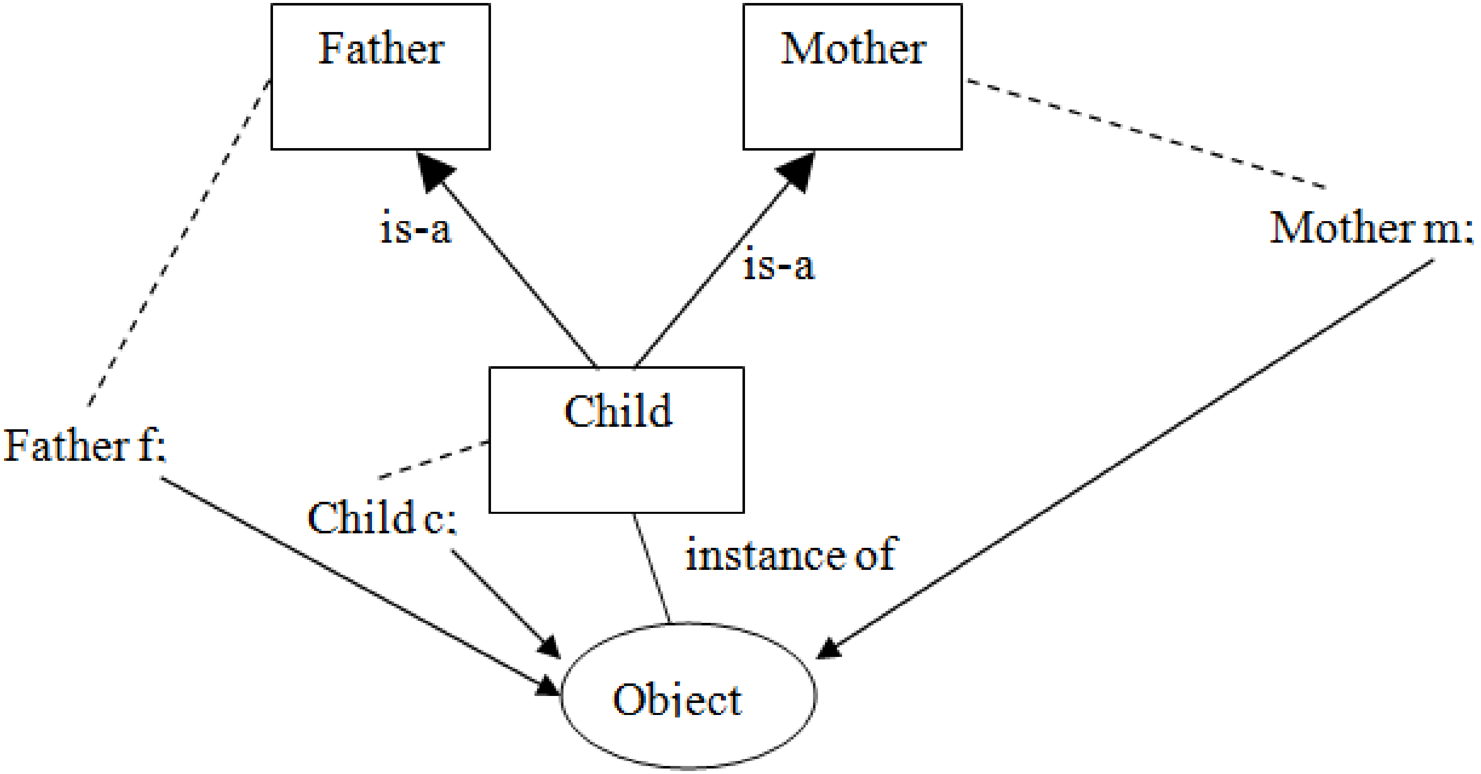
  Each abstract method is prefixed with the keyword “abstract”.
- An abstract class can not be instantiated but can be extended (subclassed).
- An *interface* contains only abstract methods and constants.

  Each abstract method is not prefixed with the keyword “abstract”.
- An *interface* can only be implemented.

Multiple inheritance

- *Class can inherit characteristics /features from more than one parent class.*

Multiple inheritance and using interface



The child class can extends only **one class**, and implements **any no. of interfaces**
Where f, m ,c are references

Multiple inheritance

```
public Class Mother {
 public void a()
   {
     System.out.println("Mother:a");
   }
    public void b()
   {
     System.out.println("Mother:b");
   }
    public void y()
   {
     System.out.println("Mother:y");
    }
}
```

```
public interface Father{
        public void c();
        public void d();
        public void y();
}
```

Multiple inheritance

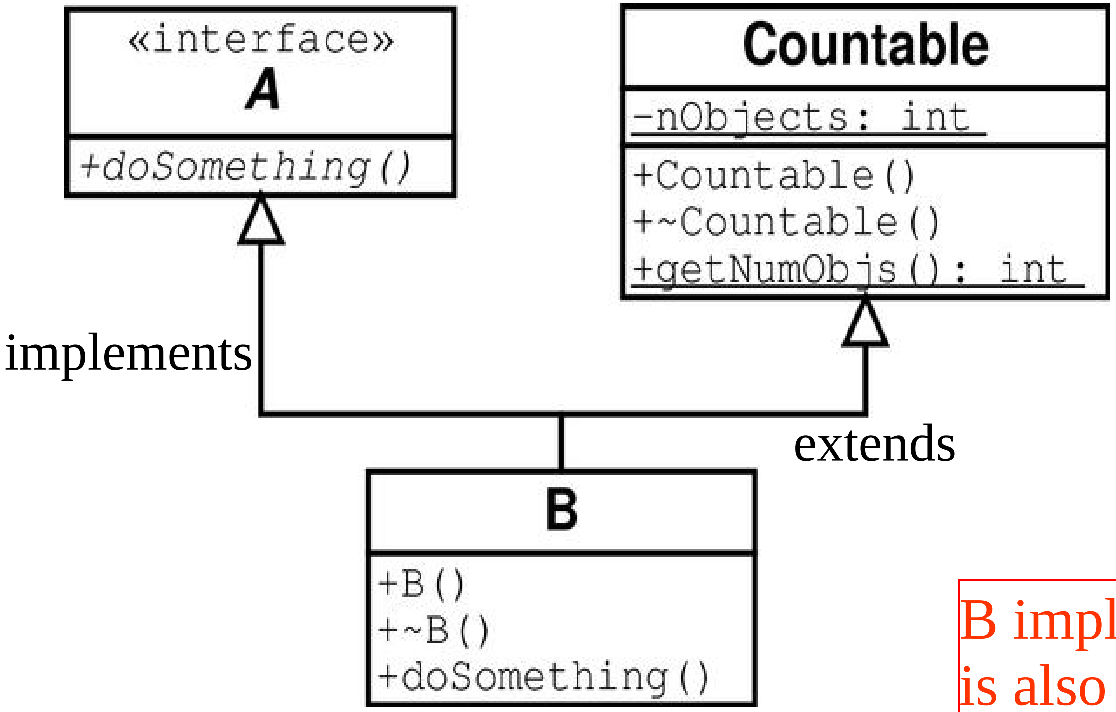
```
public class Child extend Mother implements Father{
    public void c(){
    System.out.println("Child:c");
    }
    public void d(){
    System.out.println("Child:d");
    }
}
```

Multiple inheritance

```
public static void main(String arg[])
{
  Child c= new Child();
  Father f=(Father)c;
  Mother m=(Mother)c;
  c.a();
  c.b();
  c.c();
  c.y();
   m.a();
  m.b();
  m.y();
   f.d();
  f.y();
  f.a(); // correct? no
 }
```

- Multiple inheritance can cause the diamond problem (Ambiguity Problem)
- Using Java Multiple inheritance : NO Ambiguity Problem

Multiple Inheritance (Java)



B implements the interface A and is also a "countable" class since it inherits class Countable

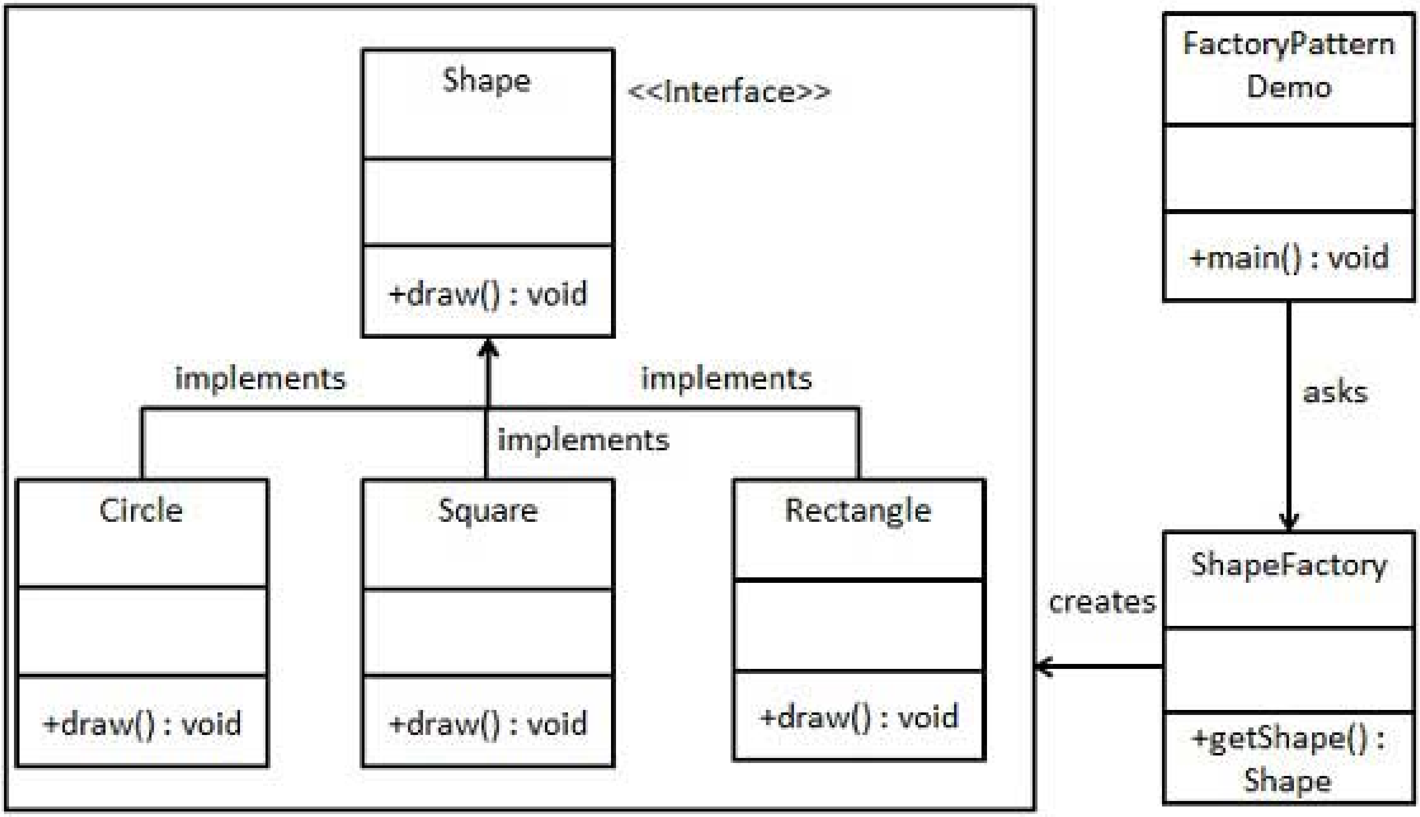
class B extends Countable implements A { /*…*/ }

Design Patterns

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Design Pattern - Factory Pattern

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.
- will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs



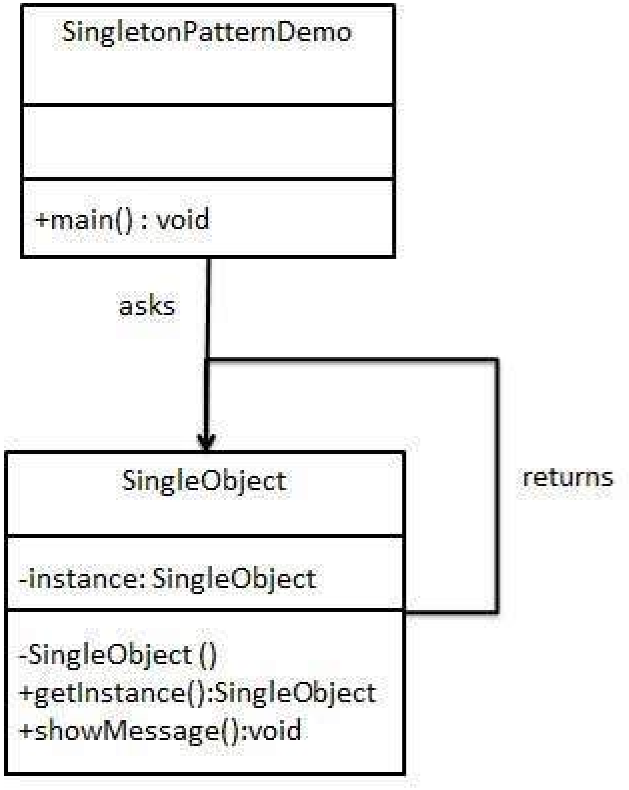
Factory Pattern example

```
public class ShapeFactory {
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else
       if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE"))
       {
         return new Square();
      }
      return null;
   }
}
```

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      shape1.computeArea ();

        Shape shape2 =
        shapeFactory.getShape("RECTANGLE");
        shape2.computeArea ();

        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape2.computeArea ();
   }
}
```

Design Pattern - Singleton Pattern

- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.



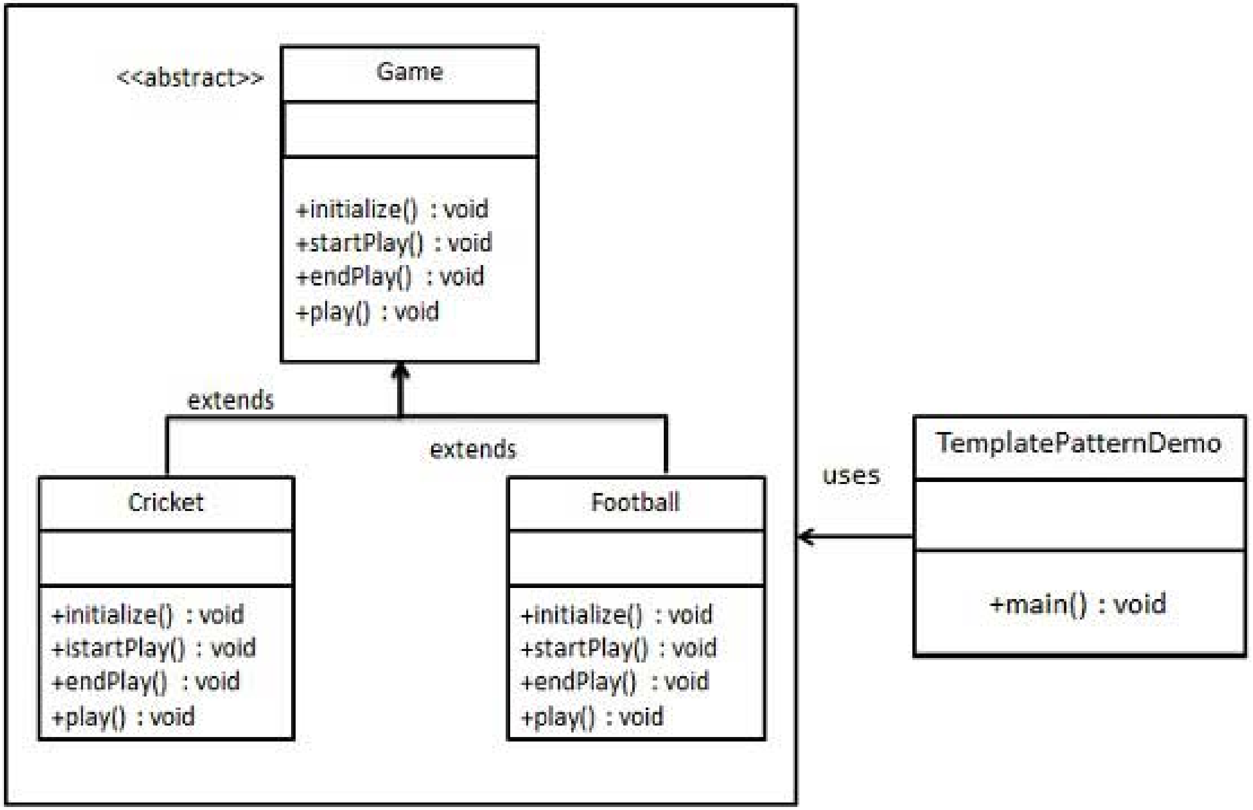
Singleton Pattern example

```
public class SingleObject {
   //create an object of SingleObject
   private static SingleObject instance;
   //make the constructor private so that this class
  //cannot be   instantiated
   private SingleObject(){       }

   //Get the only object available
   public static SingleObject getInstance(){
       if (instance == null)
                instance = new SingleObject();
      return instance;
   }
   public void showMessage(){
      System.out.println("Hello World!“ + this);
   }
}
```

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject()
     //is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();

        //this will get same object
        SingleObject object2= SingleObject.getInstance();
        object2.showMessage();
   }
}
```

Design Patterns - Template Pattern

- An abstract class exposes defined way(s)/template(s) to execute its methods.
- Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.



Template Pattern example

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){
      initialize();    //initialize the game
      startPlay();    //start game
      endPlay();   //end game
   }
}
public class Cricket extends Game {
void endPlay() {
      System.out.println("Cricket Game Finished");
   }
```

```
void initialize() {
      System.out.println("Cricket Game Initialized! Start.");
   }
void startPlay() {
      System.out.println("Cricket Game Started");
   }
}

public class Football extends Game {
void endPlay() {
      System.out.println("Football Game Finished!");
   }
void initialize() {
      System.out.println("Football Game Initialized.");
   }
void startPlay() {
      System.out.println("Football Game Started.!");
   }
}
```

Questions