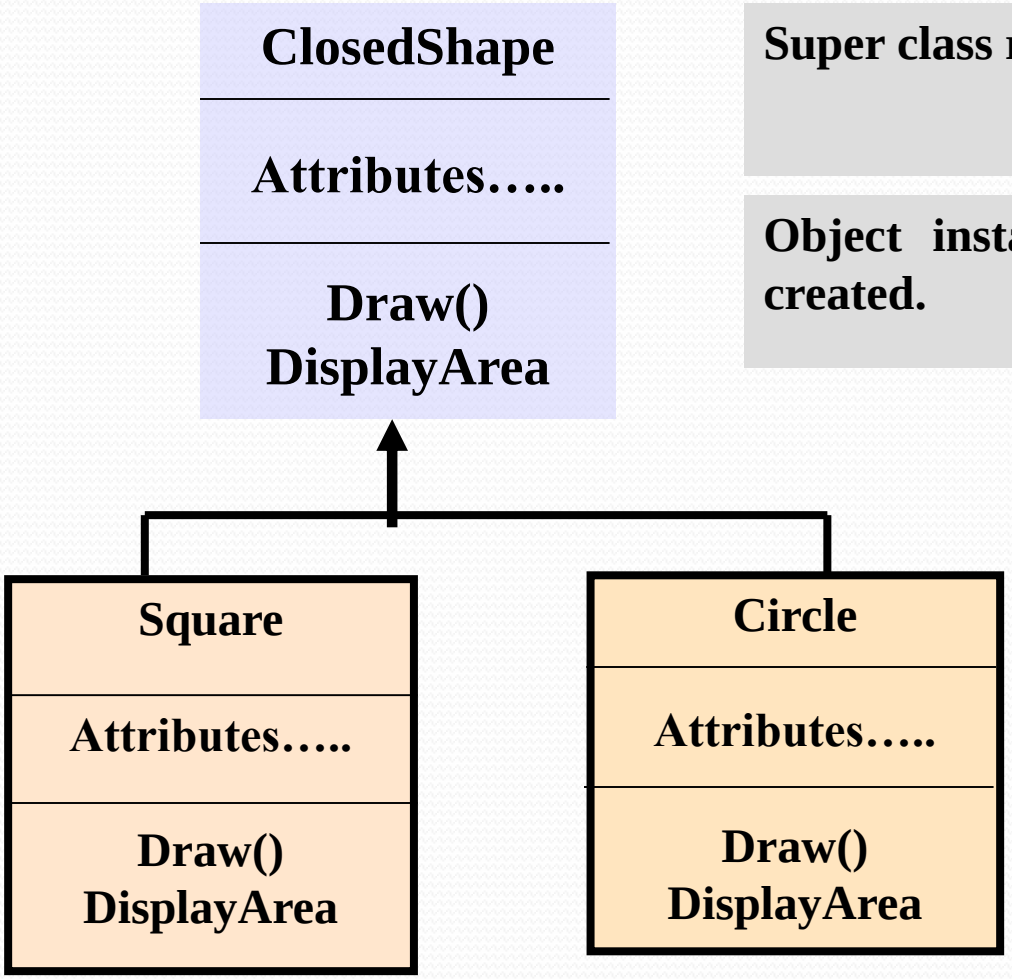
Session - 9

Abstract Classes and Interfaces

Abstract Classes

- If a class contains one or more abstract methods then the class has to be declared as an abstract class
- Abstract classes provide the basic structure to its sub-classes when it is inherited
- The abstract class cannot be instantiated using new operator
- The constructor of an abstract class cannot be declared abstract

Abstract Class



Super class renders the class layout for sub class

Object instance of the Super class can not be created.

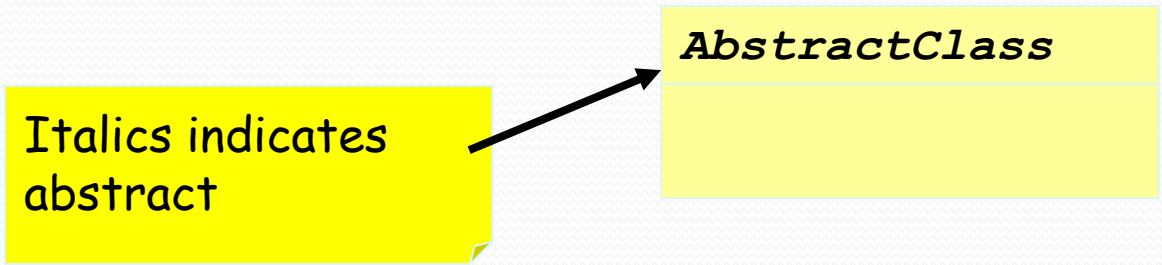
Sub classes implements super class methods , thus inherits the super class properties and behaviour.

Abstract Class – An example

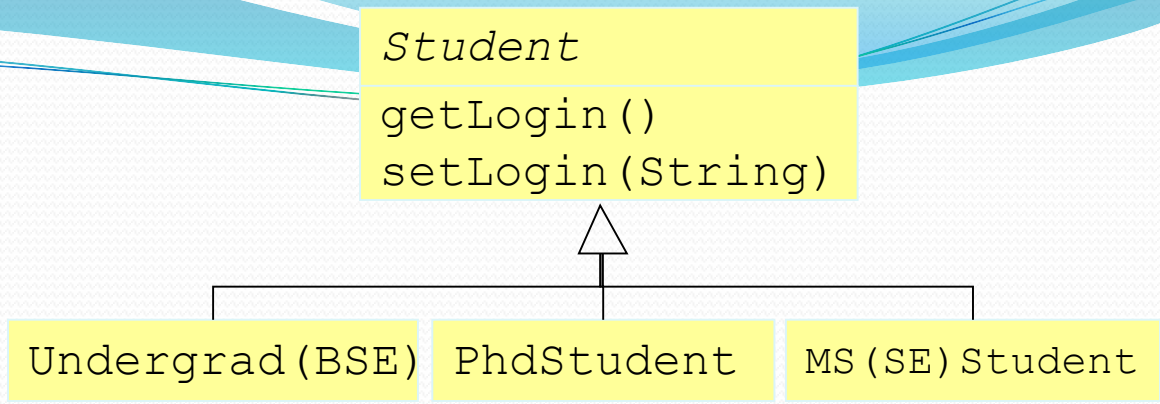
```
abstract class ClosedShapes
{
……..// data members
……. //Constructors  for circle, rectangle …

abstract void draw();                    //function in the base class
abstract void displayarea();
}
class Circle extends ClosedShapes{
  Circle(int r)   {
   super(r);    // calling super class constructor
  }
  void draw() {
   System.out.println("Draw Circle with radius " +radius);
  }
  void displayarea()   {
  System.out.println("Area of Circle = " + 3.14*radius*radius);
  }
}
```

Abstract Classes

- Like classes, they introduce types.
  - but no objects can have as actual type the type of an abstract class.
- Why use them?
  - Because there is a set of common features and implementation for all derived classes but…
    - We want to prevent users from handling objects that are too generic
    - We cannot give a full implementation for the class



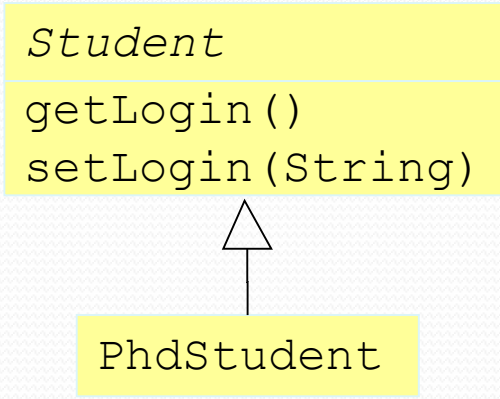
Example 1



- The problem:
  - Students are either undergraduate, PhD or `MS(SE)`.
  - We want to guarantee that nobody creates a Student object. The application always creates a specific kind of Student.
- The solution:
  - Declare Student as abstract.
- Why have the Student class in the first place?
  - A common implementation of common aspects of all students. (e.g. setLogin() and getLogin())
  - To handle all students independently of their subclass using type Student and polymorphism.

Abstract Classes in Java

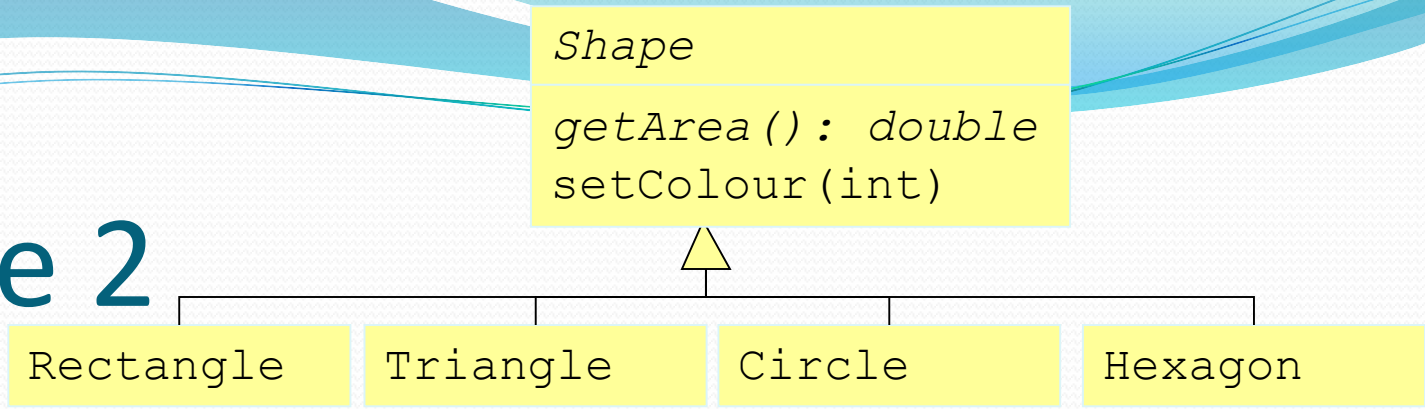
```
public abstract class Student {
   protected String login, department, name;

   public Student() {
         login = “”; department = “”; name = “”;
   }

   public void setLogin(String login) {
         this.login = login;
   }

   public String getLogin() {
         return login;
   }
}
```



PhdStudent is said to be a **concrete class**

```
public class PhdStudent extends Student{
   private String supervisor;

   public void setSupervisor(String login) {
   ...
```

Example 2

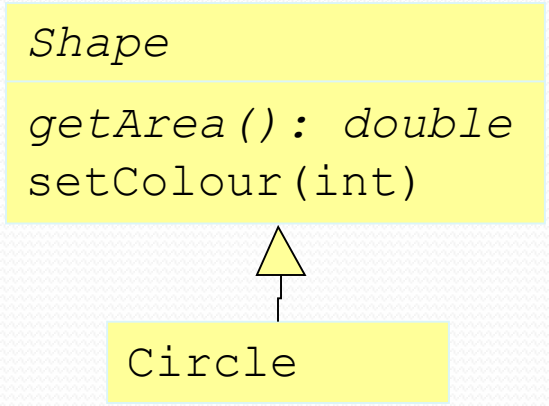


- The Problem
  - How do we calculate the area of an arbitrary shape?
  - We cannot allow Shape objects, because we cannot provide a reasonable implementation of getArea();
- The Solution
  - So we declare the Shape to be an abstract class.
  - Furthermore, we declare getArea() as an abstract method because it has no implementation
- Why have the Shape class in the first place?
  - Same reasons as for Student: a common implementation, a placeholder in the hierarchy and polymorphism.
  - Plus that we want to force all shapes to provide an implementation for getArea();

Abstract Methods in Java

```
public abstract class Shape {
   final static int BLACK = 0;
   private int colour;

   public Shape() {
         colour = BLACK;
   }

   public void setColour(int c) {
         this.colour = c;
   }

   public abstract double getArea();
}
```

Abstract methods have no body




```
public class Circle extends Shape {
   final static double PI = 3.1419;
   private int radius;

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

   public double getArea() {
         return (radius^2)*PI;
}
}
```

If Circle did not implement getArea() then it would have to be declared abstract too!

Abstract Classes

- What are the differences between both examples?
- In Example 1
  - I **choose** to declare Student abstract because I think it is convenient to prevent the existence of plain Students
- In Example 2
  - I **must** declare Shape abstract because it lacks an implementation for getArea();

Using abstract classes

```
// Shape s = new Shape(); // ERROR
Shape s = new Circle(4); // Ok
double area = s.getArea(); // Ok - Remember polymorphism?
Circle c = new Circle(3); // Ok
c.setColour(GREEN); // Ok
area = c.getArea(); // Ok
```

- Class *Shape* cannot be instantiated (it provides a partial implementation)
- Abstract methods can be called on an object of apparent type *Shape* (they are provided by *Circle*) (Polymorphism)

Abstract Base Classes – C#

- Abstract classes are classes that can be inherited from, but objects of that class cannot be created.

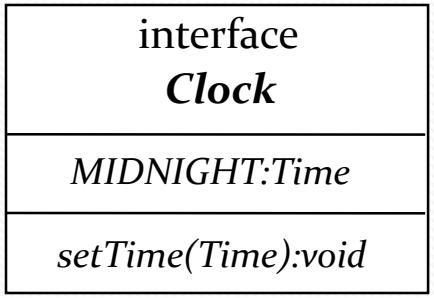
- C# allows creation of Abstract Base classes by an addition of the abstract modifier to the class definition.

Abstract Base Classes (2)

```
using System;

abstract class ABC
{
    public abstract void AFunc();

    public void BFunc()
       {
       Console.WriteLine("This is the BFunc() method!");
       }
}

class Derv : ABC
{
    public override void AFunc()
       {
       Console.WriteLine("This is the AFunc() method!");
       }
}

class Test
{
    static void Main() {
        Derv b = new Derv();
        ABC a = b;
        a.AFunc();
        b.BFunc();
    }
}
```

Interfaces

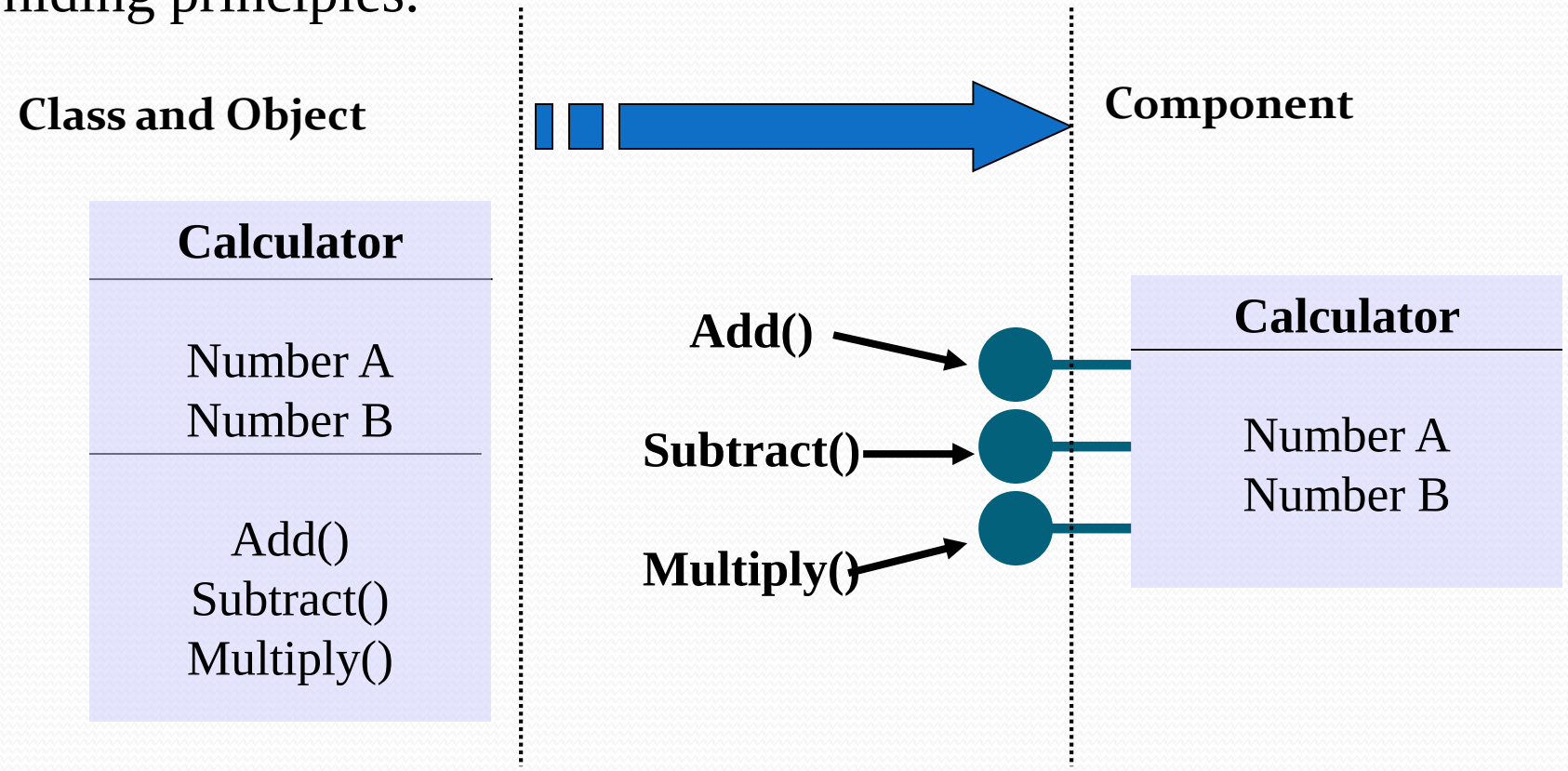
- An interface is a set of methods and constants that is identified with a name.



- They are similar to abstract classes
  - You cannot instantiate interfaces
  - An interface introduces types
  - But, they are completely abstract (no implementation)
- Classes and abstract classes realize or implement interfaces.
  - They must have (at least) all the methods and constants of the interface with public visibility

Interfaces

Abstracting the implementation of a class enriches the data hiding principles.



Features of Interface

- Are similar to a class but they do not contain instance variables and the methods contained in them
- Are similar to abstract classes with all methods declared as abstract
- Are support dynamic method resolution at run time
- Disconnect the definition of a method from the inheritance hierarchy
- Possible for classes

Defining an Interface

<access Specifier> interface <name>
{

final <data type> variable name = value;
<access specifier> <return type> method name(parameter list)

}

Example:

```
interface Student
{
void Learn(String sub);
}
```

Implementing an Interface

```
interface Area
{
final double  pi=3.14;
 void displayarea();
}
class Circle implements Area      {
  private int radius;
  Circle(int r)
  {
   radius=r;
   }
  public void displayarea()
 {
  System.out.println("Area of Circle = " + pi*radius*radius);
 }
}
```

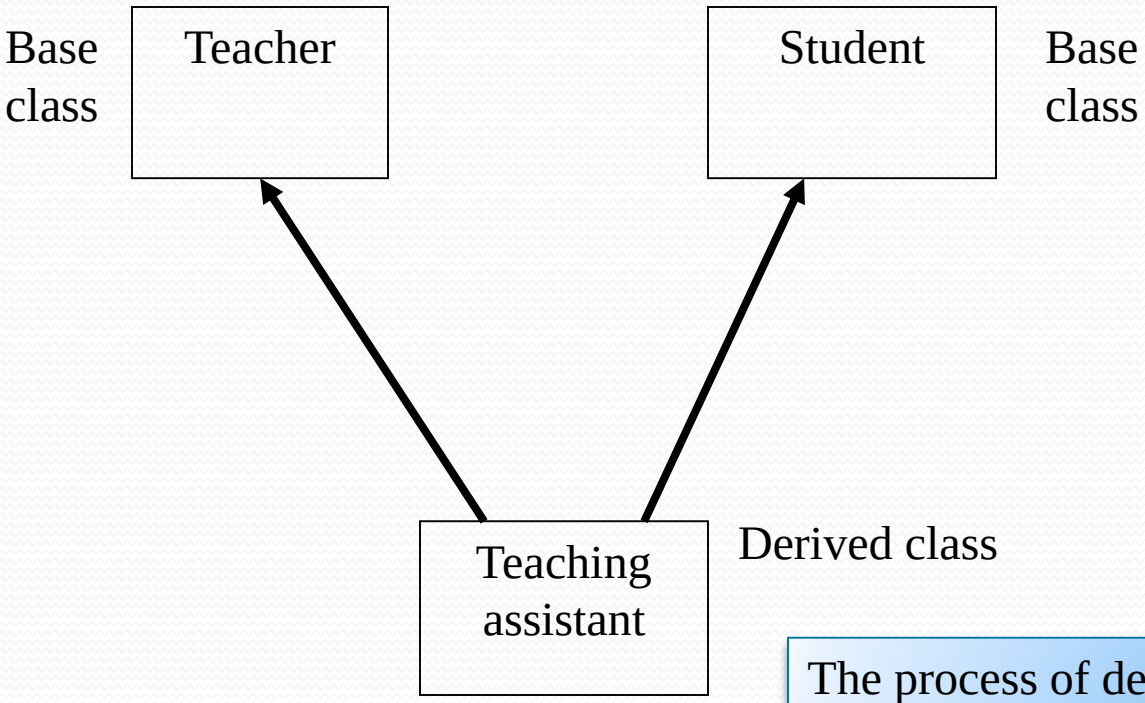
Implementing an Interface

```
class Rectangle implements Area
{
  private int length;
  private int width;
  Rectangle(int l, int w)
  {
   length=l;
   width=w;
   }


  public  void displayarea()
  {
  System.out.println("Area of  Rectangle=" + length*width);
  }

}
```

Implementing an Interface

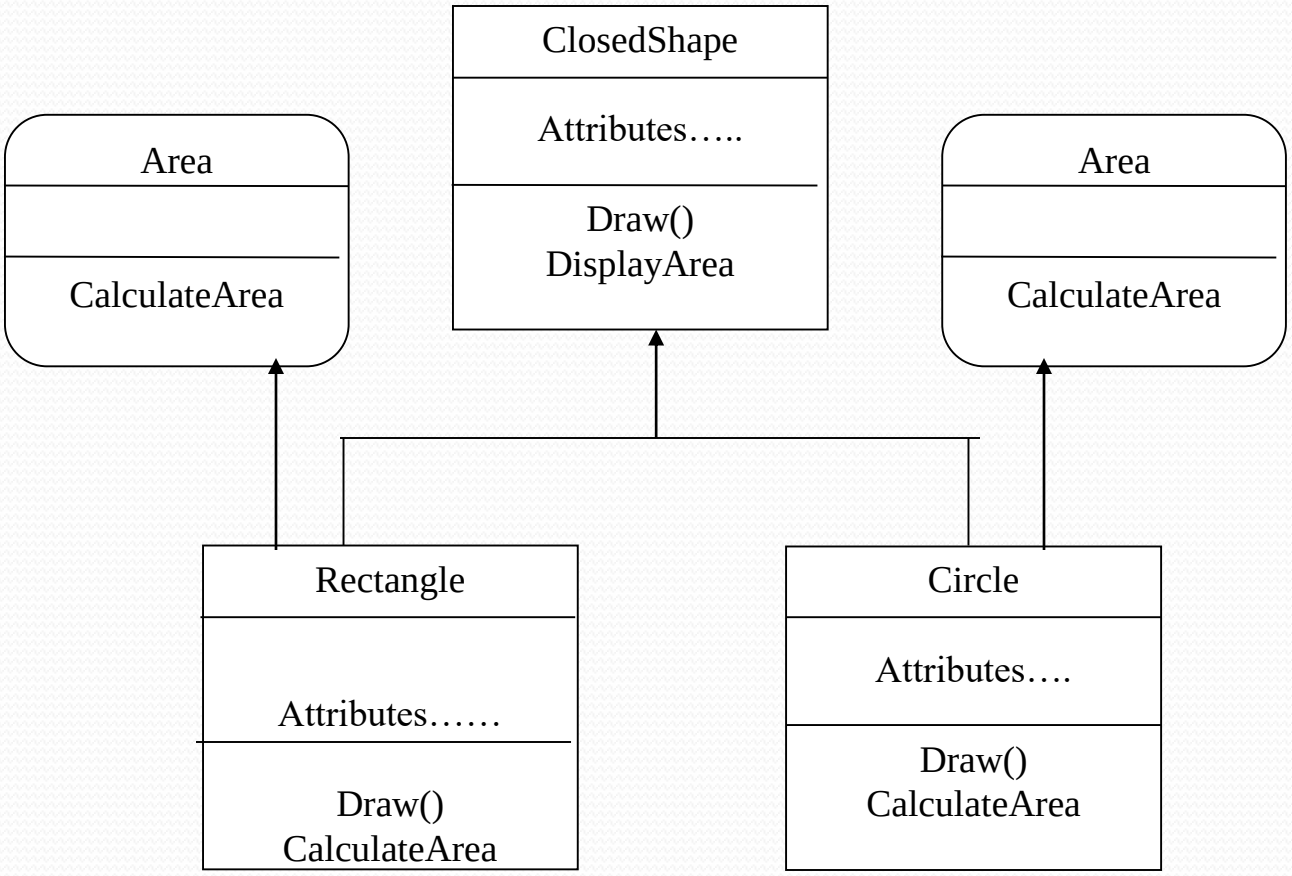
```
class DemoInterface
                                    {
public static void main(String args[])
{
  Circle c = new Circle(5);
  Rectangle s = new Rectangle(10,20);
  Area ref;
  ref = c;
  c.displayarea();
  ref = s;
  s.displayarea();
}
}
```

Multiple Inheritance



The process of deriving from more than base class is called multiple inheritance

How Multiple Inheritance is implemented in Java



Important Issues on Interfaces

When interface is implemented through a class, the instance of that class can be created and stored in a variable of that interface type.

```
interface Infa {     void print(); }
 class Clsb implements Infa
{
public void print()
       {   System.out.println("This is implemented in class B");
       }

}

class Test {
public static void main(String args[])   {
       Infa A=new Clsb();
        A.print();
       }
}
```

Important Issues on Interfaces

Interfaces can be extended. Therefore one interface can be derived from another interface. Interfaces behave same as classes in inheritance hierarchy.

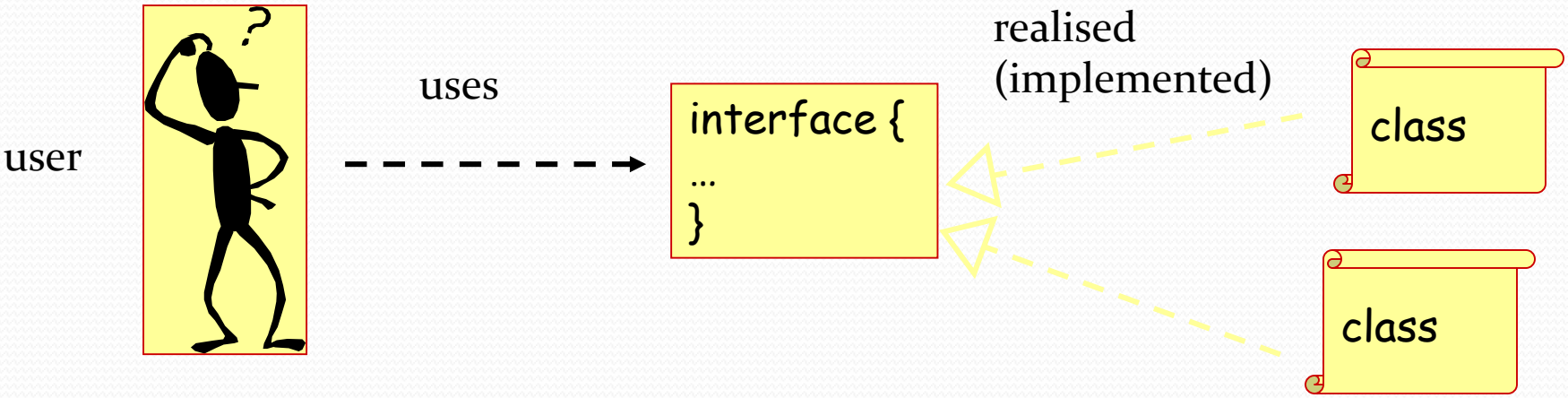
```
interface infA {
        void printA();
}
interface infB extends infA {
        void printB();
}
 class ClsTest implements infB

{
public void printA() {
  System.out.println("This is declared in interface A");
}
```

Important Issues on Interfaces

```
public void printB()
   {
    System.out.println("This is declared in interface B");
   }
}
class TestExtend
{
public static void main(String args[])
   {
    ClsTest A=new ClsTest();
    A.printA();
    A.printB();
   }
}
```

Why use Interfaces?

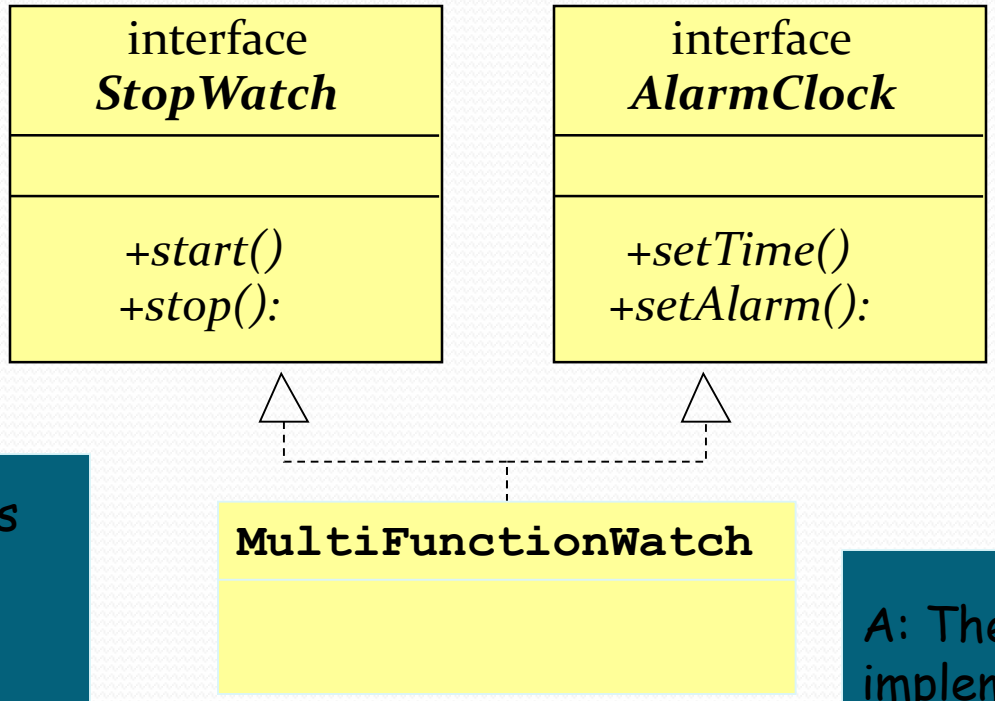
- To *separate (decouple)* the specification available to the user from implementation
  - I can use any class that implements the interface through the interface type (i.e. polymorphism)



- As a partial solution to Java's lack of multiple inheritance

Multiple Interfaces

- Classes are allowed to implement multiple interfaces



Q: Why is this not the same as multiple inheritance?

A: There is no implementation to inherit

Review-Interfaces with C#

- An interface is a pure abstract base class.
- It can contain only abstract methods, and no method implementation.
- A class that implements a particular interface must implement the members listed by that interface.

```
public interface IPict
{
    int DeleteImage();
    void DisplayImage();
}
```

Interfaces (2)

```
public class MyImages : IPict
{
public int DeleteImage()
      {
      System.Console.WriteLine("DeleteImage Implementation!");
      return(0);
      }

public void DisplayImage()
      {
      System.Console.WriteLine("DisplayImage Implementation!");
      }
}
class Test
{
static void Main()
      {
      MyImages m = new MyImages();

      m.DisplayImage();
      int t = m.DeleteImage();
      }
}
```

Interfaces

- If we merge the last two codes and compile them, we will get the following output:

```
DisplayImage Implementation!
DeleteImage Implementation!
```

- Take another example:

```
public class BaseIO
{
public void Open()
      {
      System.Console.WriteLine("This is the Open method of
BaseIO");
      }
}
```

Interfaces (4)

- Now, if we need to inherit a class, **MyImages**……

```
public class MyImages : BaseIO, IPict
{
public int DeleteImage()
      {
      System.Console.WriteLine("DeleteImage Implementation!");
      return(0);
      }

public void DisplayImage()
      {
      System.Console.WriteLine("DisplayImage Implementation!");
      }
}
class Test
{
static void Main()
      {
      MyImages m = new MyImages();

      m.DisplayImage();
      int t = m.DeleteImage();

      m.Open();
      }
}
```

Interfaces (5)

- The output of the example is:

```
DisplayImage Implementation!
DeleteImage Implementation!
This is the Open method of BaseIO
```

Multiple Interface Implementation (1)

- C# allows multiple interface implementations.

```
public interface IPictManip
{
   void ApplyAlpha();
}
```

Multiple Interface Implementation (2)

```
public class MyImages : BaseIO, IPict, IPictManip
{
public int DeleteImage()
      {
      System.Console.WriteLine("DeleteImage Implementation!");
      return(0);
      }

public void DisplayImage()
      {
      System.Console.WriteLine("DisplayImage Implementation!");
      }

public void ApplyAlpha()
      {
      System.Console.WriteLine("ApplyAlpha Implementation!");
      }
}
class Test
{
static void Main()
      {
      MyImages m = new MyImages();

      m.DisplayImage();
      int t = m.DeleteImage();

      m.Open();

      m.ApplyAlpha();
      }
}
```

Output

```
DisplayImage Implementation!
DeleteImage Implementation!
This is the Open method of BaseIO
ApplyAlpha Implementation!
```

Explicit Interface Implementation

- Explicit interface implementation can be used when a method with same name is available in 2 interfaces.

```
public interface IPict
{
    int DeleteImage();
    void DisplayImage();
}

public interface IPictManip
{
    void ApplyBlending();
    void DisplayImage();
}

public class MyImages : BaseIO, IPict, IPictManip
{
...
void IPict.DisplayImage()
        {
        System.Console.WriteLine("IPict Implementation of
DisplayImage");
        }

void IPictManip.DisplayImage()
        {
        System.Console.WriteLine("IPictManip  Implementation of
DisplayImage");
        }
...
}
```

Interface Inheritance

- New Interfaces can be created by combining together other interfaces.

```
interface IPictAll : IPict, IPictManip
{
//More operations can be added if necessary (apart from that of
IPict & IManip)
}
```

Abstract classes vs. Interfaces

- Can have data fields
- Methods may have an implementation
- Classes and abstract classes extend abstract classes.
- Class cannot extend multiple abstract classes

- Can only have constants
- Methods have **no** implementation
- Classes and abstract classes implement interfaces
- Interfaces can extend multiple interfaces
- A class can implement multiple interfaces