Interface

Review

What Does Abstraction Provide ?

Review

Abstraction allows us to add restrictions on child classes so they are bound to provide the required functionality.

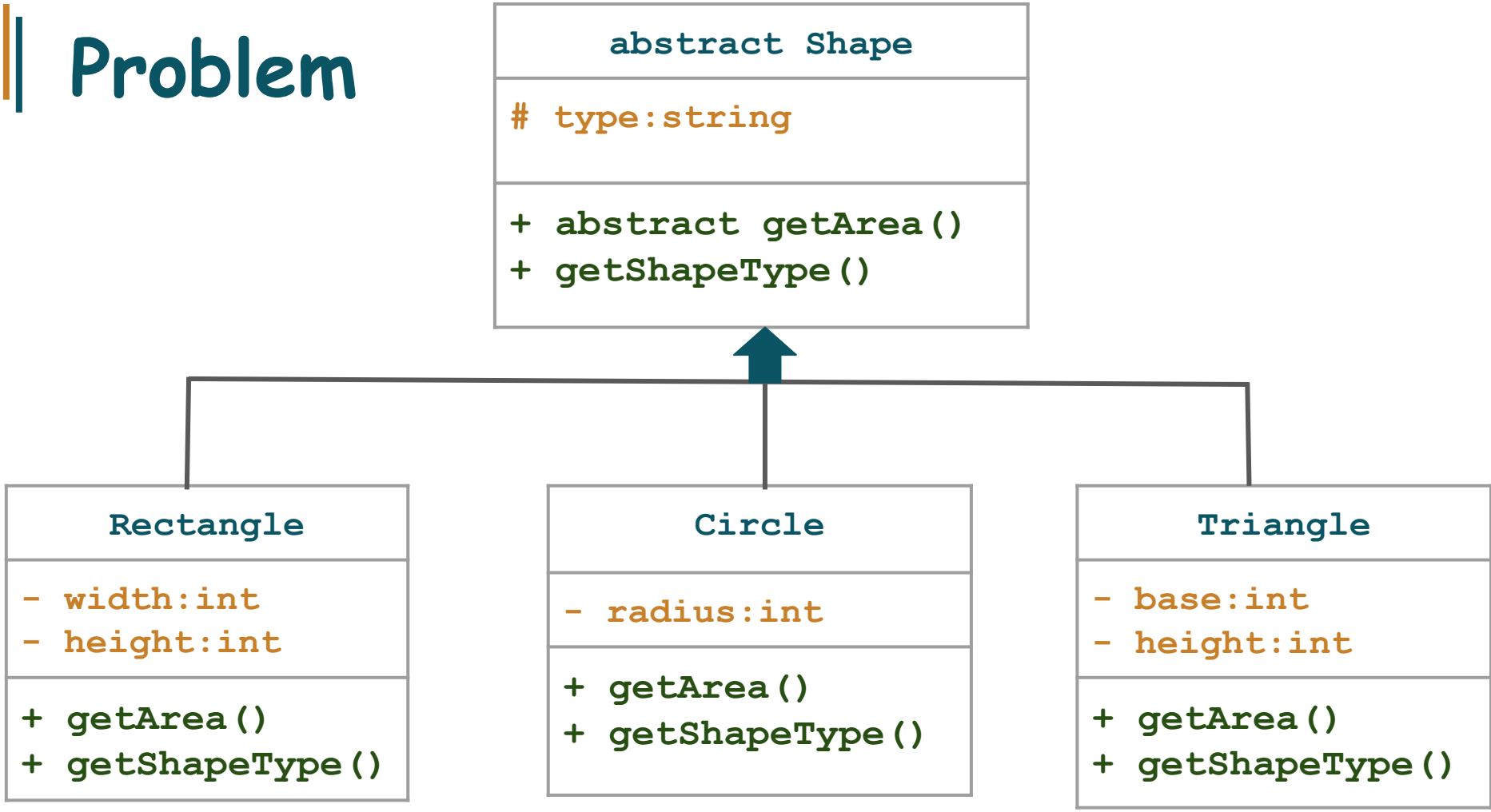
Review

It helps us to specify a must have behaviour of children classes of an abstract parent class.

Review

Also, when a child class says i am extending that abstract class, it is guaranteed that the child class shall provide the implementation of all abstract methods.

Problem


abstract Shape
type:string
+ abstract getArea()
+ getShapeType()
Rectangle
- width:int
- height:int
+ getArea()
+ getShapeType()
Circle
- radius:int
+ getArea()
+ getShapeType()
Triangle
- base:int
- height:int
+ getArea()
+ getShapeType()

Problem Scenario

Lets say, we want to add restriction on the Circle class that it should have getDiameter() method and Triangle class that is should have isRightAngle() method

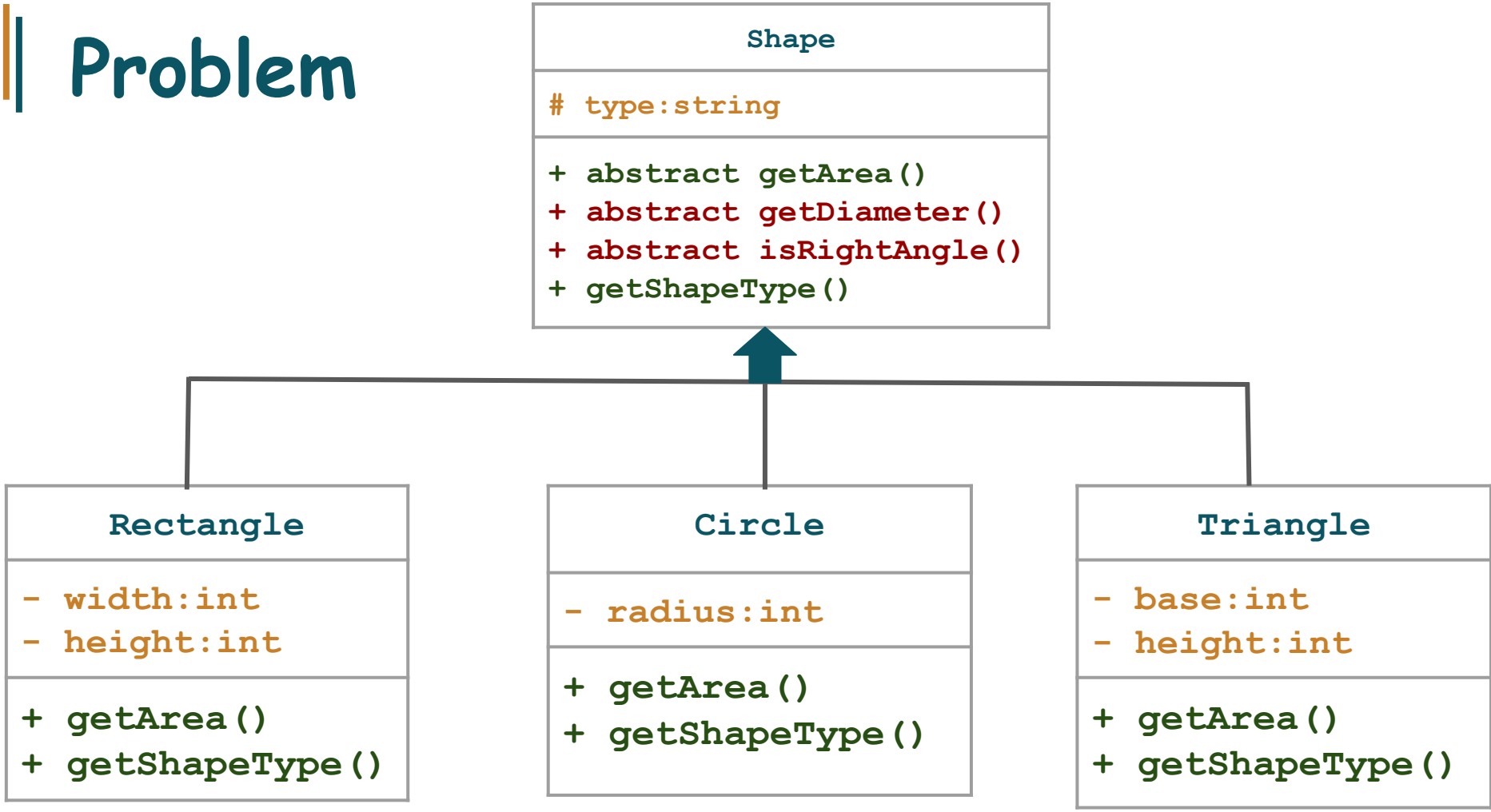
Problem Scenario

How can we make sure this requirement ?

Problem Scenario

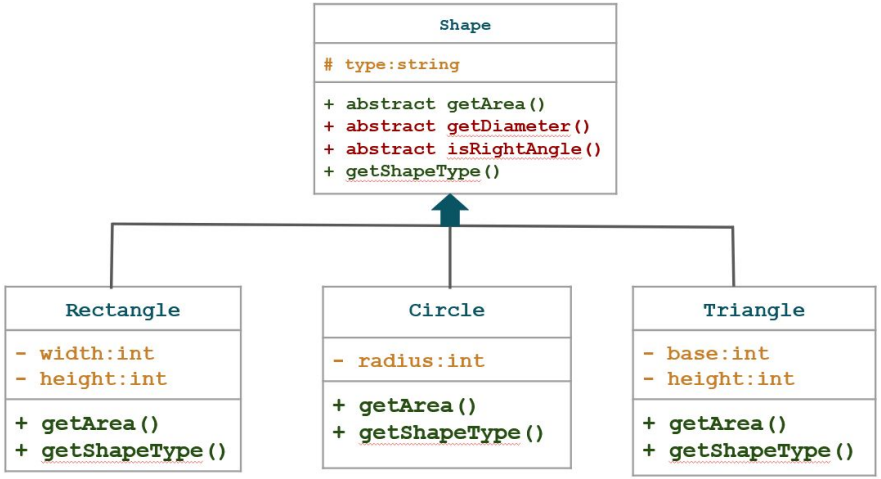
One way is to add both methods as abstract methods inside the Shape class

Problem


Shape
type:string
+ abstract getArea()
+ abstract getDiameter()
+ abstract isRightAngle()
+ getShapeType()
Rectangle
- width:int
- height:int
+ getArea()
+ getShapeType()
Circle
- radius:int
+ getArea()
+ getShapeType()
Triangle
- base:int
- height:int
+ getArea()
+ getShapeType()

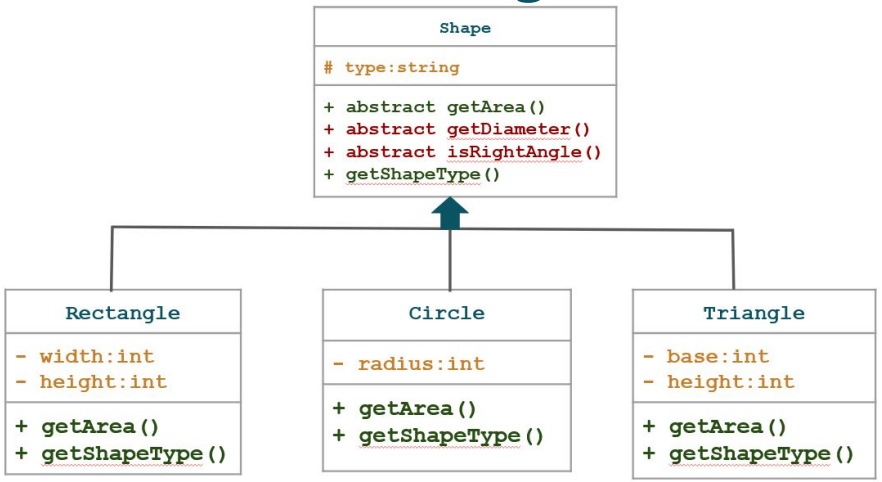
Problem Scenario

But this implementation has a problem.
Can you identify ?



Problem Scenario

Circle can not have the definition of isRightAngle() and similarly the Triangle can not define getDiameter() method but this design is forcing both classes to define the methods that have nothing to do with them.



Problem Scenario

Another common solution comes into mind to make the separate abstract classes for the Circle and Triangle.

Problem Scenario

Another common solution comes into mind to make the separate abstract classes for the Circle and Triangle.

but again it is not possible.

Problem Scenario

In that case, Triangle has to extend both Shape abstract class and another Triangle Abstract class. Same is true for Circle as well

Problem Scenario

Multiple inheritance is not possible in C# and Java.

And the languages that support Multiple Inheritance, that can generate diamond shape problem in multiple inheritance.

Problem Scenario

So Our problem is to find a way to take guarantee from the class that it will provide a specific type of behaviour.

Interface

OOP provides us interfaces to solve the problem

Interface

Interface is like an abstract class that have all the functions defined as abstract.

Interface

Interface can be declared with the interface keyword

```
interface ICircle
{
        float getDiameter();
}
```

Interface

Interface can be declared with the interface keyword. All the methods within the interface are public and abstract.

```
interface ICircle
{
        float getDiameter();
}
```

Interface VS Abstract

Difference between interface and abstract class is that abstract class can have both concrete and abstract methods but interface can only have abstract methods

```
interface ICircle
{
        float getDiameter();
}
```

```
abstract class Shape
{
    public abstract double getArea();
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

Interface

Similarly, we can declare a separate interface for the triangle class as well.

```
interface ICircle
{
        float getDiameter();
}
```

```
interface ITriangle
{
        bool isRightAngle();
}
```

Interface

Now, the class that want to behave like circle it will extend the ICircle interface.

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface

Also, all these functions should have public access.

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface

Have we solved the Original Problem ? Now the circle is not like shape any more ?

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface

This is the magic of interface, a class can extend multiple interfaces or even can extend a class and interfaces both.

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: Shape, ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface

This is the magic of interface, a class can extend multiple interfaces or even can extend a class and interfaces both.

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: Shape, ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface

In Case of inheriting both class and interfaces, we first have to write the name of the class then the interfaces.

```
interface ICircle
{
        float getDiameter();
}
```

```
class Circle: Shape, ICircle
{
  private int radius;
  public Circle (int radius)
  {
     this.radius = radius;
  }
  public float getDiameter()
  {
     float d = this.radius*2;
     return  d;
  }
}
```

Interface: Advantage

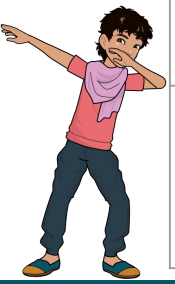
By implementing an interface, a class gives guarantee that it will provide the behaviour that its interface is promising.

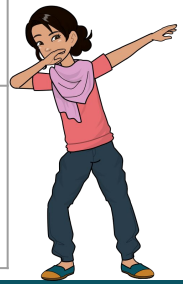
Interface: When do we Use?

- **Security:**
  When we have to simply hide some features/functions and have to use those later. It is essential to hide a few operations while only showing the related Operations to the user.
- **Multiple Inheritance:**
  In c#, one class can inherit from a simple parent class, inheriting all its features. Multiple Inheritance is not supported in C#. But with the use of an interface, multiple interfaces can be implemented into a single class.

Conclusion

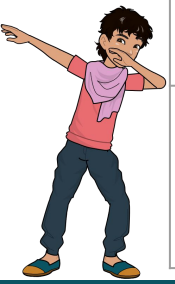
Interface	Abstraction
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does not contain data members	Abstract class can contain data members
Interface does not contain Constructors	Abstract class can contains Constructors
Interface only have abstract methods	Abstract class can have both concrete and abstract methods
Interface can not have access modifiers. By default everything is assumed public	Abstract class uses access modifiers to declare data members and member functions.

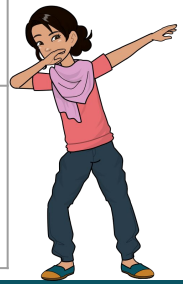




Conclusion

Interface	Abstraction
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does not contain data members	Abstract class can contain data members
Interface does not contain Constructors	Abstract class can contains Constructors
Interface only have abstract methods	Abstract class can have both concrete and abstract methods
Interface can not have access modifiers. By default everything is assumed public	Abstract class uses access modifiers to declare data members and member functions.





Food for Thought

We know that abstract class can never be instantiated. Which means we can never have an object of an abstract class.

Then how are we supposed to call a constructor when we can't even create an object of an abstract class?

Learning Objective

Add multiple and varying restrictions on the classes so they are bound to provide the required behaviour through Interfaces.

Self Assessment: Output?

```
public abstract class Fruit
{
    public Fruit()
    {
        Console.WriteLine("1. Base class: Fruit");
    }
    public abstract Name();
}
```

```
class Apple : Fruit
{
    public Apple()
    {
        Console.WriteLine("2. Derived class: Apple");
    }
    public string Name();
    {
        return "Apple";
    }
}
```

```
public static void main(string[] args)
{
    Apple a;
    a = new Apple();
    a.Name();
    Console.ReadKey();
}
```

Self Assessment: Output?

```
abstract class Animal
{
    public int legs;
    public void speak()
    {
        Console.WriteLine("hmmm");
    }
    public abstract void move();

}

interface Flyable
{
    void fly();
}
```

```
class Bird : Animal, Flyable
{
    public void fly()
    {
        Console.WriteLine("With Wings");
    }
    public override void move()
    {
        Console.WriteLine("2 Steps");
    }
}
```

```
public static void main(string[] args)
{
    Bird b = new Bird();
    b.speak();
    b.move();
    b.fly();
    Console.ReadKey();
}
```