# Firefighter Training Simulation System (Case Study)

# Case Study

Fire Department has hired you to make a training and simulation system for them.

In this system they have Fire Trucks. Where each Fire Truck contains a Ladder and a Hose Pipe.Hose pipes are detachable from the truck. Hose pipes are either made of synthetic rubber or soft plastic and they can be either be cylindrical or circular in shape. They have specific diameter and water flow rate.

Ladder has a specific length and colour and they are built right into the truck (i.e., they cannot be separated from the truck).

Each FireTruck has a Firefighter as its Driver. FireFighter has a name. He can drive the fire truck and can extinguish fire as well.

They have a Fire Chief as well. The fire chief is just another firefighter. He can drive a truck. He can put out fires. But he can also delegate responsibility for putting out a fire to another firefighter.

# Solution

**Step 1:**
Identify the Classes which have no dependency on other Classes.
**Note:** We will only make the Classes for those who have distinctive Attributes.

# Case Study

Fire Department has hired you to make a training and simulation system for them.

In this system they have Fire Trucks. Where each Fire Truck contains a Ladder and a Hose Pipe. Ladder has a specific length and colour and they are built right into the truck (i.e., they cannot be separated from the truck).

Hose pipes are detachable from the truck. Hose pipes are either made of synthetic rubber or soft plastic and they can be either be cylindrical or circular in shape. They have specific diameter and water flow rate.

Each FireTruck has a Firefighter as its Driver. FireFighter has a name. He can drive the fire truck and can extinguish fire as well.

They have a Fire Chief as well. The fire chief is just another firefighter. He can drive a truck. He can put out fires. But he can also delegate responsibility for putting out a fire to another firefighter.

# Domain Model

Ladder

FireFighter

HosePipe

# Solution

**Step 2:**
Identify the Classes which have dependency on other Classes.
**Note:** We will only make the Classes for those who have distinctive Attributes.

# Case Study

Fire Department has hired you to make a training and simulation system for them.

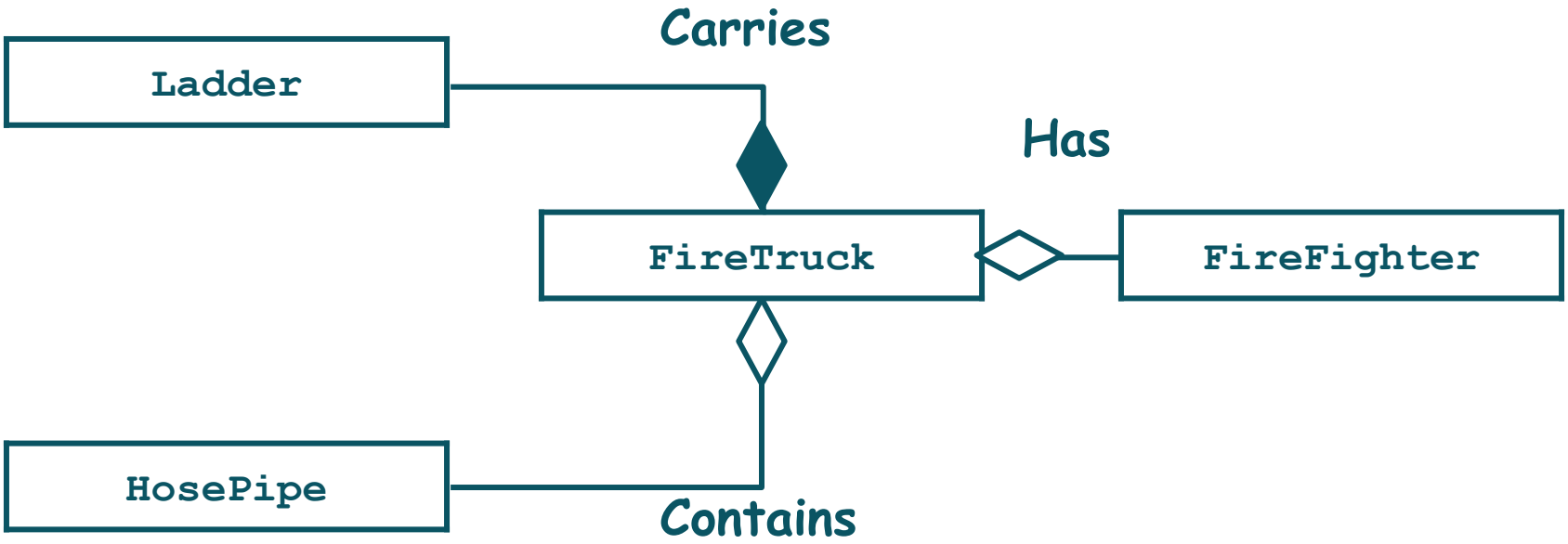
In this system they have Fire Trucks. Where each Fire Truck contains a Ladder and a Hose Pipe. Ladder has a specific length and colour and they are built right into the truck (i.e., they cannot be separated from the truck).

Hose pipes are detachable from the truck. Hose pipes are either made of synthetic rubber or soft plastic and they can be either be cylindrical or circular in shape. They have specific diameter and water flow rate.

Each FireTruck has a Firefighter as its Driver. FireFighter has a name. He can drive the fire truck and can extinguish fire as well.

They have a Fire Chief as well. The fire chief is just another firefighter. He can drive a truck. He can put out fires. But he can also delegate responsibility for putting out a fire to another firefighter.

# Domain Model



# Solution

**Step 3:**

Identify the Classes which are inherited from other Classes.

# Case Study

Fire Department has hired you to make a training and simulation system for them.

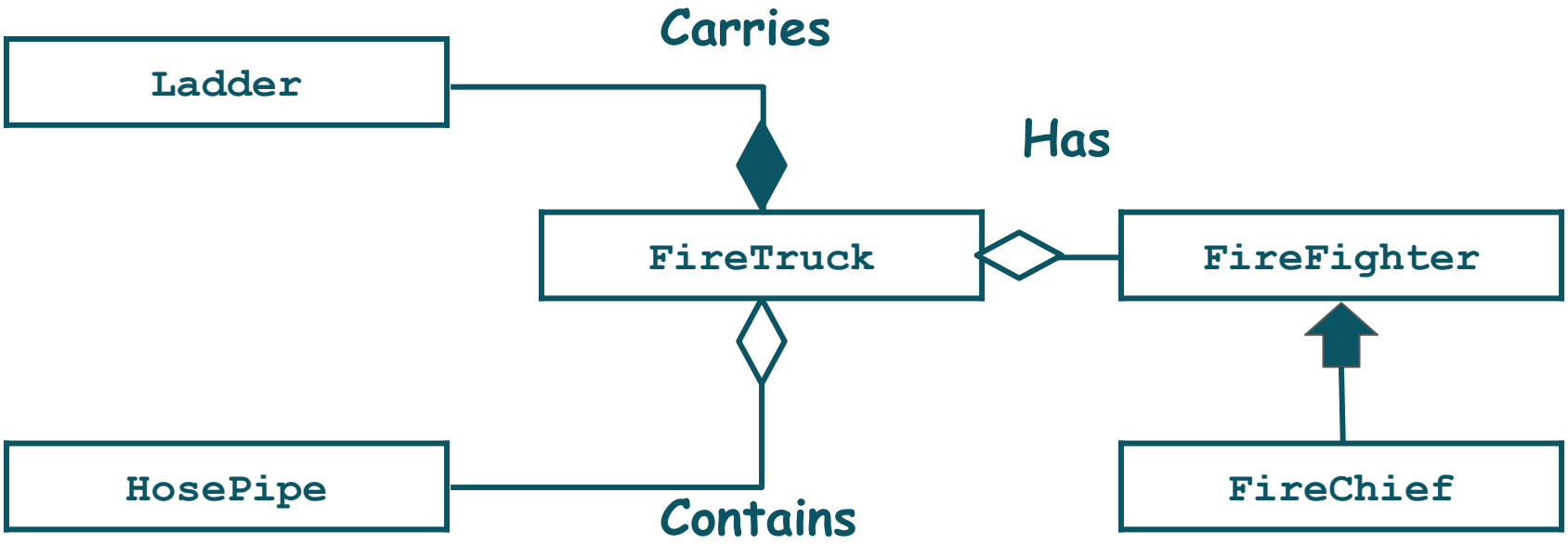
In this system they have Fire Trucks. Where each Fire Truck contains a Ladder and a Hose Pipe. Ladder has a specific length and colour and they are built right into the truck (i.e., they cannot be separated from the truck).

Hose pipes are detachable from the truck. Hose pipes are either made of synthetic rubber or soft plastic and they can be either be cylindrical or circular in shape. They have specific diameter and water flow rate.

Each FireTruck has a Firefighter as its Driver. FireFighter has a name. He can drive the fire truck and can extinguish fire as well.

They have a Fire Chief as well. The fire chief is just another firefighter. He can drive a truck. He can put out fires. But he can also delegate responsibility for putting out a fire to another firefighter.

# Domain Model

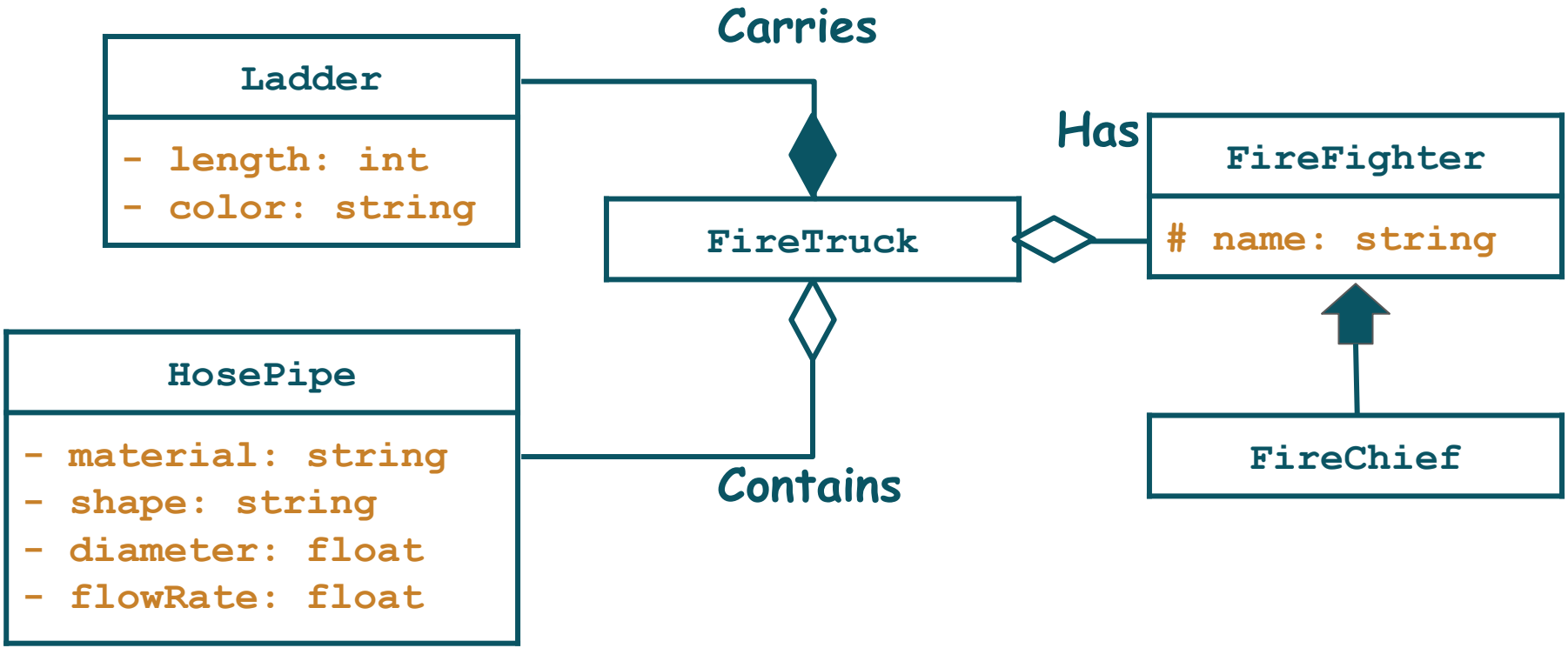


# Solution

**Step 4:**

Draw the Class Diagram with Attributes.

# Domain Model

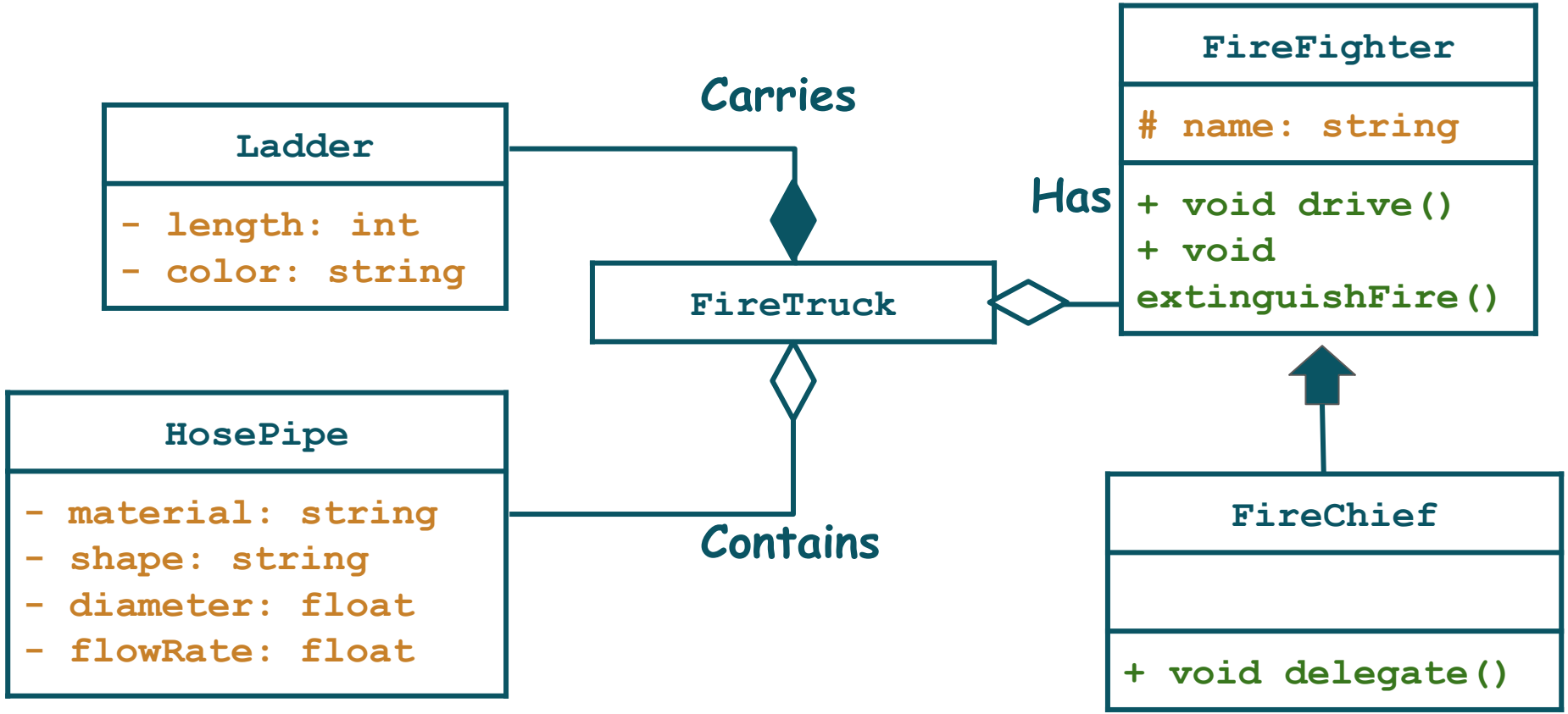


# Solution

**Step 5:**

Draw the Class Diagram with Attributes and Functions.

# Domain Model



# Solution

**Step 6:**

Write the C# code for the Classes.

# Classes Code: Ladder

```
class Ladder
{
    private int length;
    private string color;
    public Ladder(int length, string color)
    {
        this.length = length;
        this.color = color;
    }
}
```

# Classes Code: HosePipe

```
class HosePipe
{
    private string material;
    private string shape;
    private float diameter;
    private float flowRate;

    public HosePipe(string material, string shape, float diameter, float flowRate)
    {
        this.material = material;
        this.shape = shape;
        this.diameter = diameter;
        this.flowRate = flowRate;
    }
}
```

# Classes Code: FireFighter

```
class FireFighter
{
    private string name;
    public FireFighter(string name)
    {
        this.name = name;
    }
    public void drive()
    {
        Console.WriteLine(name + " is Driving the Truck");
    }
    public void extinguishFire()
    {
        Console.WriteLine(name + " is Extinguishing the Fire");
    }
}
```

# Classes Code: FireChief

```
class FireChief : FireFighter
{
    public FireChief(string name) : base(name)
    {
    }

    public void delegateResponsibility(string FirefighteName)
    {
        Console.WriteLine("Tell " + FirefighteName + " to extinguish fire");
    }
}
```

# Classes Code: FireTruck

```
class FireTruck
{
    private Ladder l1;
    private HosePipe h1;
    private FireFighter driver;

    public FireTruck(HosePipe h1, FireFighter driver)
    {
        l1 = new Ladder(34, "Black");
        this.h1 = h1;
        this.driver = driver;
    }
}
```

# Driver Program Code

```
static void Main(string[] args)
{
    HosePipe h = new HosePipe("plastic", "circular", 2.5F, 3);
    FireFighter f = new FireFighter("Harry");
    FireChief fc = new FireChief("Potter");
    FireTruck t = new FireTruck(h, fc);
    fc.drive();
    fc.extinguishFire();
    fc.delegateResponsibility("Harry");
    Console.ReadKey();
}
```

# Driver Program Code

Here we are assigning FireChief to the FireFighter Object? How is it Possible?

```
static void Main(string[] args)
{
    HosePipe h = new HosePipe("plastic", "circular", 2.5F, 3);
    FireFighter f = new FireFighter("Harry");
    FireChief fc = new FireChief("Potter");
    FireTruck t = new FireTruck(h, fc);
    fc.drive();
    fc.extinguishFire();
    fc.delegateResponsibility("Harry");
    Console.ReadKey();
}
```

# Driver Program Code

Lets see the next topic to understand it completely.

```
static void Main(string[] args)
{
    HosePipe h = new HosePipe("plastic", "circular", 2.5F, 3);
    FireFighter f = new FireFighter("Harry");
    FireChief fc = new FireChief("Potter");
    FireTruck t = new FireTruck(h, fc);
    fc.drive();
    fc.extinguishFire();
    fc.delegateResponsibility("Harry");
    Console.ReadKey();
}
```

# Parent Reaction when Child Overrides

# Function Overriding

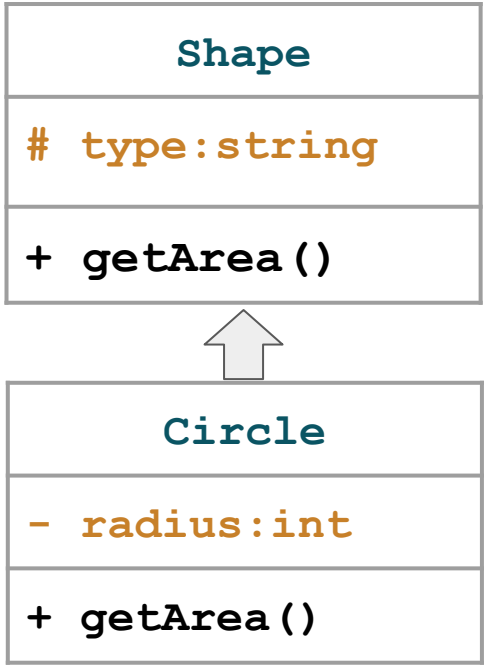
Object Oriented Programming offers us a way to extend the functionality of the parent class through function overriding.

# Function Overriding

Lets say we have class Shape with the function getArea that currently returns zero and we want another class Circle to extend the class Shape.

# Function Overriding

Circle Extends the Shape and overrides getArea.



# Function Overriding

## Circle Extends the Shape and overrides getArea.

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

# Function Overriding: Output?

## Circle Extends the Shape and overrides getArea.

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Console.WriteLine(c.getArea());
Console.ReadKey();
```

# Function Overriding: Output?

## Circle Extends the Shape and overrides getArea.

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Console.WriteLine(c.getArea());
Console.ReadKey();
```

```
25.1327412287183
```

# Function Overriding: Output?

## What if we do This?

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

# Function Overriding: Output?

## getArea function of Circle or Shape will be called ?

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

# Function Overriding: Output?

In current scenario, the getArea will print 0 irrespective of child class implementation.

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

```
0
```

# Function Overriding

But, We want the parent class function (getArea) behaves according to child class overridden functions.

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

# Function Overriding

## How can we make getArea function to behave according to the relevant child behaviour ?

```
class Shape
{
    protected string type;
    public double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public new double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

# Virtual and Override Keyword

When we assign child class object to parent object and we want the functionality of parent object functions to be replaced with the child functionality, we add virtual keyword in the parent class and override keyword in child class.

# Function Overriding

Declare getArea as virtual in Parent Class and add override with function declaration inside child class.

```
class Shape
{
    protected string type;
    public virtual double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public override double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

# Function Overriding

Now the s.getArea() will print the area of circle.

```
class Shape
{
    protected string type;
    public virtual double getArea()
    {
        return 0;
    }
    public string getShapeType()
    {
        return "undefined.";
    }
}
```

```
class Circle : Shape
{
    private int radius;
    public Circle(int radius)
    {
        this.radius = radius;
    }
    public override double getArea()
    {
        double area; area = 2 * Math.Pow(radius, 2) * Math.PI;
        return area;
    }
}
```

```
Circle c = new Circle(2);
Shape s = c;
Console.WriteLine(s.getArea());
Console.ReadKey();
```

```
25.1327412287183
```

# Virtual and Override Keyword

Object Oriented Programming allows Parent Class to use the functionality of its Child Class.
This functionality will be assigned at runtime.

# Virtual and Override Keyword: Advantage

Parent Class could have dynamic behaviour at run time.

# Conclusion

- If we want parent class to use the child overridden behaviour we need to add
  - virtual keyword with parent method
  - override keyword with the child class function declaration.

# Learning Objective

Make Parent to Use the child Behaviour when child overrides any of its method

# Self Assessment: Write Output

```
class Lightsaber
{
    public virtual string getColor()
    {
        return "Green";
    }
}

class SithLightsaber : Lightsaber
{
    public override string getColor()
    {
        return "Red";
    }
}
```

```
public static void Main()
{
    var lightsaber = new Lightsaber();
    Console.WriteLine(lightsaber.getColor());

    var sithLightsaber = new SithLightsaber();
    Console.WriteLine(sithLightsaber.getColor());
    Console.ReadKey();
}
```