

Ques 3.1. Implement, using C#, a dynamically-sized stack, which can store doubles. Using this stack, write an RPN calculator program. The operators \*, /, - and + should be supported. Use a “while” loop to capture user inputs.

- A scheme in which the operators *follow the operands* (postfix operators), resulting in the Reverse Polish Notation.
- This has the advantage that the operators appear in the *order required* for computation.
- RPN Models After Our Brains. (Mental Calculation)

$(1 + 2) * (3 + 4)$

1  
2  
+  
3  
4  
+  
\*

Answer: 21

Ques 3.1. Implement, using C#, a dynamically-sized stack, which can store doubles. Using this stack, write an RPN calculator program. The operators \*, /, - and + should be supported. Use a “while” loop to capture user inputs.

- We give solution when operators are arithmetic operators (operate on numbers: we convert them to “double” format).

$(1 + 2) * (3 + 4)$

1

2

+

3

4

+

\*

Answer: 21

# Stack in C#

.NET includes a Stack class inside the `System.Collections` namespace.

## Definition

A stack is a data structure that allows to add and remove objects at the same position. The last object placed on the stack is the first one to be removed following a Last In First Out (LIFO) data storing method.

## Common functions

- `Push(element)` – Adds a new object to the last position of the stack.
- `Pop()` – Returns and removes the last object of the stack.
- `Peek()` – Returns the last object without removing it.
- `IsEmpty()` – Validates if the stack is empty.

# Skeleton of program in C#

```
class rpnCalculator {
    static void Main(string[] args) {
        Stack stack01 = new Stack();
        string input="";
        double left=0.0, right=0.0;
        // input the string contents, i.e. string value
        while ( !input.Equals("e") ) {
            if ( input == "operator" {
                right = pop stack();
                left = pop stack();
                Push (left "operator" right) on stack;
            }
            else if (input == "r" { // reset and restart }
                else { // push input string as a number on stack }
            // input the string contents, i.e. string value
        }
        // print result from stack
    }
}
```

```
using System;  
using System.Collections; // to use stack class
```

```
namespace Tut3Q1 {  
    class rpnCalculator {  
        static void Main(string[] args) {  
            Stack stack01 = new Stack();  
            string input="";  
            double left=0.0, right=0.0;  
  
            Console.WriteLine("Please enter a value or an  
                operator one by one followed by enter key");  
            Console.WriteLine("Press 'r' to reset and 'e' to quit");  
  
            input = Console.ReadLine();
```

```
while ( !input.Equals("e") ) {
    if ( input == "+" || input == "-"
        || input == "*" || input == "/" ) {
        try {
            right = (double)stack01.Pop();
            left = (double)stack01.Pop();
        }
        catch(InvalidOperationException e) {
            Console.WriteLine(e);
        }
    }
}
```

```
switch (input) {
    case "+":
        stack01.Push(left + right);
        break;
    case "-":
        stack01.Push(left - right);
        break;
    case "*":
        stack01.Push(left * right);
        break;
}
```

```
        case "/":
            try {
                stack01.Push(left / right);
            }
            catch(Exception e) { Console.WriteLine(e); }
            break;
    }
}
else if ( input == "r" ) {
    stack01.Clear();
    Console.WriteLine("=== Reset ===");
}

else {
    try {
        stack01.Push(double.Parse(input));
    }
    catch(System.FormatException e)
        { Console.WriteLine(e); }
}
```

```
        input = Console.ReadLine();
    } //end of while (!input.Equals("e"))
printResult (stack01);
} // end of Main
```

```
public static void printResult(Stack s) {
    double answer = (double)s.Peek();
    Console.WriteLine(answer);
}
}
}
}
```

Ques 3.1. Implement, using C#, a dynamically-sized stack, which can store either integers, doubles or strings. Using this stack, write an RPN "calculator" program. The operators \*, /, - and + should be supported. Use a "while" loop to capture user inputs.

- We gave solution when operators are arithmetic operators.

Note: + is string operator as well:

```
Robert  
ran  
+  
1.5  
2  
*  
Kilometers.  
+
```

```
Some Features: (i) + on strings  
allowed  
(ii) Multiplication result shown  
as 3.0  
(iii) Output shown on single line.
```

Expected Answer: Robert ran 3.0 Kilometers.

Ques 3.1. Implement, using C#, a dynamically-sized stack, which can store either integers, doubles or strings. Using this stack, write an RPN "calculator" program. The operators \*, /, - and + should be supported. Use a "while" loop to capture user inputs.

- Additional Exercise: Formulate "reasonable" rules for:
  - (a) applying operators to operands of various type (write your own "overloading" methods if need arises for your formulation);
  - (b) detemining the type (class) of the result; and,
  - (c) how result would be displayed;and solve problem to implement your rules.

```
Robert  
ran  
+  
1.5  
2  
*  
Kilometers.  
+
```

**Some Features:** (i) + on strings allowed  
(ii) Multiplication result is not shown as 3.0 but shown as integer 3 (as the result is integer).  
(iii) Output shown on single line.

Expected Answer: Robert ran 3 Kilometers.

## Ques 3.2

- Given the following specification and implementation of the Point class in C++, rewrite the code in C#.
- Implement Circle class using Point class.
- Implement Cylinder class using one of the above classes.
- Draw Class Hierarchy.
- Create a test program to test the above classes.
- Was it a good choice to use Point class as base class? Suggest alternatives.

## point.h (C++)

```
#ifndef POINT_H // define POINT_H only if it is not defined earlier
#define POINT_H
class Point {
    friend ostream &operator<<( ostream &, const Point & );
public:
    Point( int = 0, int = 0 ); // default constructor
    void setPoint( int, int ); // set coordinates
    int getX() { return x; } // get x coordinate
    int getY() { return y; } // get y coordinate
protected: // accessible to derived classes
    int x, y; // coordinates of the point
};
#endif
```

## point.cpp (C++)

```
#include <iostream.h>
#include "point.h"
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set the x and y coordinates
void Point::setPoint( int a, int b ){
    x = a;
    y = b;
}

// Output the Point
ostream &operator<<( ostream &output, const Point &p ){
    output << '[' << p.x << ", " << p.y << ']';
    return output;          // enables cascading
}
```

## point.cs

```
using System;
```

```
namespace Tut3Q2 {  
    public class Point {  
        protected int x, y;  
  
        public int X { get{return x;} }  
        public int Y { get{return y;} }  
  
        public Point() { x = 0; y = 0; }  
        public Point(int a, int b) { x = a; y = b; }  
  
        public void setPoint(int a, int b) { x = a; y = b; }  
  
        public override string ToString(){  
            return "[" + x + "," + y + "];"  
        }  
    }  
}
```

## circle.cs

```
using System;
```

```
namespace Tut3Q2 {  
    public class Circle : Point {  
        protected double radius;
```

```
        public Circle() { radius = 1; }  
        public Circle(double radius) { this.radius = radius; }  
        public Circle(double radius, int a, int b):base(a,b) {  
            this.radius = radius;  
        }  
    }  
}
```

```
        public double getRadius() { return radius; }  
        public void setRadius(double radius) { this.radius = radius; }
```

```
        public double area() { //OR public virtual double area()  
            return Math.PI * Math.Pow(radius,2);  
        }  
    }  
}
```

```
public double Radius {  
    get{ return radius; }  
    set{ this.radius = value; }  
}
```

```
public override string ToString(){
    return "Circle of radius " + this.radius + " at point [" + x
+      ", " + y + " ]";
}
}
}
```

## cylinder.cs

```
using System;
```

```
namespace Tut3Q2 {
```

```
    public class Cylinder : Circle {  
        protected double height;
```

```
        public Cylinder() { height = 1; }
```

```
        public Cylinder(double h) { height = h; }
```

```
        public Cylinder(double h, double r) : base(r) { height = h; }
```

```
        public Cylinder(double h, double r, int a, int b) : base(r,a,b) {  
            height = h;  
        }
```

```
public double Height {  
    get { return height; }  
    set { this.height = value; }
```

```
public double getHeight() { return height; }
```

```
public void setHeight(double height) { this.height = height; }
```

```
public new double area() {  
    // OR public override double area() if Circle.area() declared virtual  
    return 2 * (base.area() +  
        Math.PI * base.getRadius() * height);  
}
```

```
public double volume() { return base.area() * height; }
```

```
public override string ToString() {  
    return "Cylinder of height " + height + ", radius " +  
        radius + " at point [" + x + ", " + y + "];  
}
```

```
}
```

```
}
```

## test.cs

```
using System;
```

```
namespace Tut3Q2{
```

```
    public class Test{
```

```
        public static void Main(string args[]){
```

```
            Circle testCircle1 = new Circle();
```

```
            Circle testCircle2 = new Circle(35, 2, 4);
```

```
            Cylinder testCylinder1 = new Cylinder();
```

```
            Cylinder testCylinder2 = new Cylinder(8, 20, 3, 3);
```

```
            Console.WriteLine("{0} created", testCircle1);
```

```
            Console.WriteLine("{0} created", testCircle2);
```

```
            Console.WriteLine("{0} created", testCylinder1);
```

```
            Console.WriteLine("{0} created", testCylinder2);
```

```
            Console.WriteLine("Area of Circle 1 = {0}",  
                testCircle1.area());
```

```
            Console.WriteLine("Area of Circle 2 = {0}",  
                testCircle2.area());
```

```
Console.WriteLine("Surface Area of Cylinder 1 = {0}",  
    testCylinder1.area());
```

```
Console.WriteLine("Surface Area of Cylinder 2 = {0}",  
    testCylinder2.area());
```

```
Console.WriteLine("Volume of Cylinder 1 = {0}",  
    testCylinder1.volume());
```

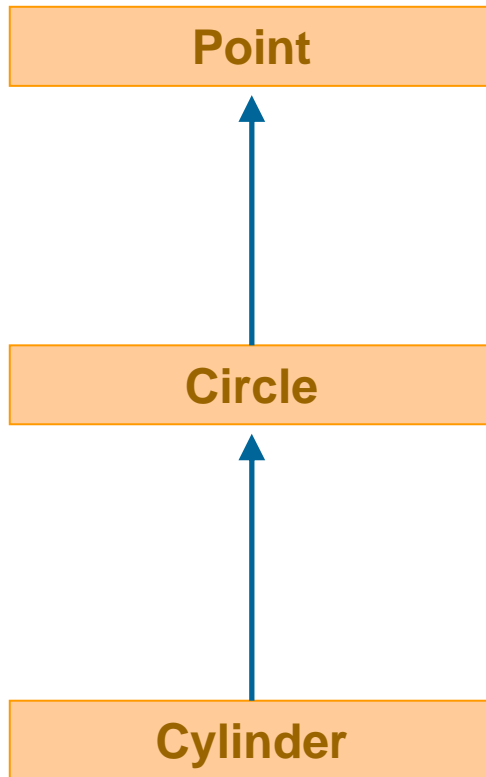
```
Console.WriteLine("Volume of Cylinder 2 = {0}",  
    testCylinder2.volume());
```

```
}
```

```
}
```

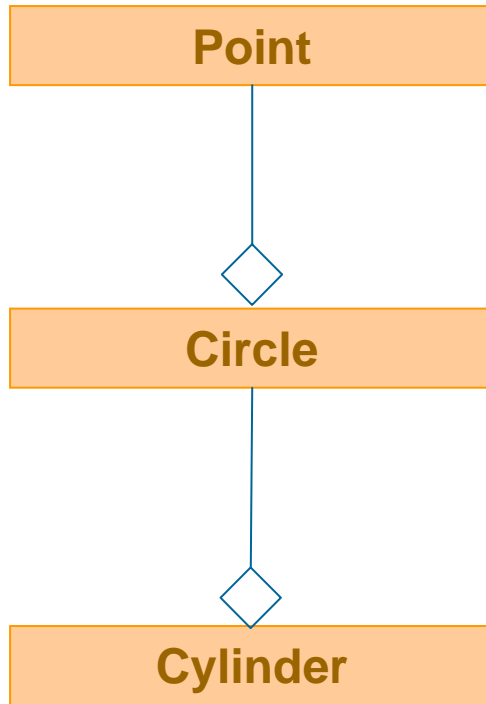
```
}
```

# Class hierarchy : Inheritance Modelling



- We model Point as a base class.
- Circle has all attributes of “Point” , & additional attribute “radius”.
- So, Circle is “Subclass” or “derived Class” of the class “Point”
- Cylinder has all attributes of “Circle” , & additional attribute “height”.
- So, Cylinder is “Subclass” or “derived Class” of the class “Circle”

# Alternate class structure



- Exhibits “has-a” relationship
- Circle has a Point
- Cylinder has a Circle

e.g.

```
public class Circle {
    protected Point centre = new Point();
    protected radius;
    :
}
```

Ques 3.3. Figure Q3 lists the C++ code for a **Polygon** class. Two subclasses, **Rectangle** and **Triangle**, are derived from the **Polygon** class.

```
#include <string.h>
#include <iostream.h>

typedef enum {POLY_PLAIN, POLY_RECT, POLY_TRIANG}
    KindofPolygon;

class Polygon{
public:
    Polygon(char * theName, float theWidth, float theHeight);
    float calArea() const;
    void printWidthHeigth() ;

protected:
    char name[30];
    float width;
    float height;
};
```

```
Polygon::Polygon(char *theName, float theWidth, float theHeight)
{
    strcpy(name , theName);
    width = theWidth;
    height = theHeight;
}
```

```
float Polygon::calArea() const{ return 0 }
```

```
void Polygon::printWidthHeigth() {
    cout << "Width= " << width << " Height= " << height <<
    endl;
}
```

```
void printArea(. . .){
    float area;
    . . .
    cout << area << endl;
}
```

```
int main(int argc, char* argv[]) {  
    Rectangle rect1 ("Rect1", 3.0, 4.0);  
    printArea(rect1);  
    rect1.printWidthHeight();  
  
    Triangle triang1 ("Triang1", 3.0, 4.0);  
    printArea (triang1);  
    triang1.printWidthHeight();  
  
    return 0;  
}
```

## (i) Rewrite the code in C#

- *enum* does not use *typedef* declaration for declaring synonyms. In C# it is given by:

```
[attributes] [modifiers] enum identifier [: base-type]  
{ enumerator-list } [;]
```

- No global functions or variables are allowed in C#. All functions and variables must belong to a class.
- Data type *string* used in place of character arrays
- Passing float values require suffix *f*

## Polygon.cs

```
using System;
```

```
namespace Tut3Q3 {
```

```
    // using enum without the typedef declaration
```

```
    public enum KindofPolygon { POLY_PLAIN, POLY_RECT,  
        POLY_TRIANG};
```

```
    public class Polygon {
```

```
        protected string name; //use string data type instead
```

```
        protected float width;
```

```
        protected float height;
```

```
        protected KindofPolygon polytype;
```

```
public Polygon(string theName, float theWidth, float
    theHeight) {
    name = theName;    width = theWidth;
    height = theHeight;
    polytype = KindofPolygon.POLY_PLAIN;
}
```

```
public KindofPolygon Polytype {
    get { return polytype; }
    set { polytype = value; }
}
```

```
public float calArea() { return 0; }
public void printWidthHeight( ) {
    Console.WriteLine("Width = {0} Height = {1}",
        width, height);
}
```

```
}
```

## (ii) Show the code for the Rectangle subclass

- Inherits from the superclass *Polygon*.
- Operator ":" denotes inheritance in C#.
- Constructors of the base-class (super-class) can be called using the *base* keyword.
- *new* is used to hide the functions of the base-class (super-class)

## Rectangle.cs

```
using system;

namespace Tut3Q3 {
public class Rectangle : Polygon {
    public Rectangle(String theName, float theWidth,
        float theHeight) : base(theName, theWidth, theHeight) {
        this.polytype = KindofPolygon.POLY_RECT;
    }

    public new float calArea() { return width * height; }
}
}
```

## Triangle.cs

```
using system;

namespace Tut3Q3 {
    public class Triangle : Polygon {
        public Triangle(String theName, float theWidth,
            float theHeight) : base(theName, theWidth, theHeight){
            this.polytype = KindofPolygon.POLY_TRIANG;
        }

        public new float calArea () {
            return 0.5f * width * height;    }
        }
    }
```

(iii) Complete the codes for the **printArea( )** function, and the **Rectangle** and the **Polygon** classes for static binding of all functions.

## Static binding

- resolving a function at compile time
- used when all information needed to make a function call is known at compile time

# TestPolygon.cs

```
using system;
namespace Tut3Q3 {
    // class to test other classes replace the function main
    public class TestPolygon {
        public static void printArea(Rectangle rect) {
            float area = rect.calArea( );

            Console.WriteLine("The area of the {0} is {1}.",
                rect.Polytype, area);
        }

        public static void printArea(Triangle tri) {
            float area = tri.calArea( );

            Console.WriteLine("The area of the {0} is {1}.",
                tri.Polytype, area);
        }
    }
}
```

**Function Overloading  
(Static Binding)**



```
// Main is always declared public static  
public static void Main( ) {  
    // use f as suffix for floating point numbers  
    Rectangle rect1 = new Rectangle("Rect1", 3.0f, 4.0f);  
    printArea(rect1);  
    rect1.printWidthHeight();  
  
    Triangle triang1 = new Triangle("Triang1", 3.0f, 4.0f);  
    printArea(triang1);  
    triang1.printWidthHeight();  
    }  
    }  
    }
```

## Impact on program when new subclass of Polygon is introduced

There is a need to define another overloaded function for *printArea()*.

For example, If you have another polygon called pentagon, you'll need to define:

```
public static void printArea(Pentagon thePentagon) {...}
```

Note: to determine the area of pentagon, two parameters, namely: height and width, are not sufficient. So this subclass needs to make use of additional information stored in the form of its own variables to compute the area.

(v) Modify the **Polygon** code so that any of its subclasses must include a **calArea()** member function. Suggest reason(s) why this requirement would be appropriate in this case.

- Superclass Polygon is declared *abstract*.
- calArea() member function in Polygon class is declared *abstract*.
- The calArea() is defined in the subclasses Rectangle and Triangle using the keyword *override*.

## Polygon.cs

```
using System;
public abstract class Polygon{
....
    protected abstract float calArea();
}
```

## Rectangle.cs

```
using System;
public class Rectangle: Polygon {
...
    protected override float calArea() {
        return width * height;
    }
};
```

## Reasons why this requirement would be appropriate

- General polygons don't have an area formula, so having an implementation of `calArea()` in the Polygon class is useless.
- Using the ***abstract*** keyword will enable `calArea()` to be implemented only in the subclasses of polygon.

(iv) Repeat part (iii) for dynamic binding of printArea()

## Dynamic binding

- resolving a function at run time
- used when some of the information needed to make a function call is not known until run time

## TestPolygon.cs

```
using system;
namespace Tut3Q3 {

    public class TestPolygon {
        public static void printArea(Polygon theShape) {
            float area;

            area = theShape.calArea();

            Console.WriteLine("The area of the {0} is {1}.",
                theShape.Polytype, area);
        }

        public static void Main( )
            { ... ... }
    }
}
```

Finished Tut 3 ...