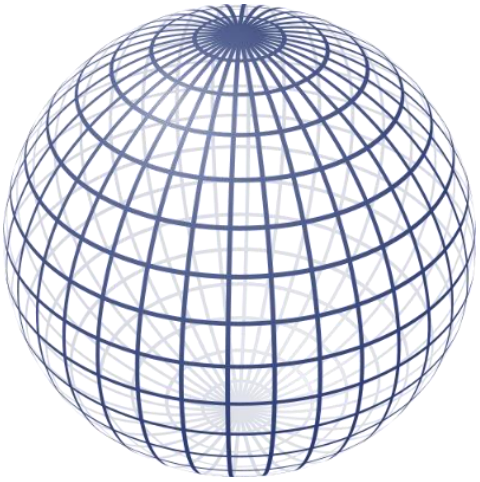
Assignment: Overriding & Overloading Methods – 3D Point

Object-Oriented Programming (OOP) is not just about concepts—Abstraction, Encapsulation, Inheritance, Polymorphism—but also methodology. A software developer must master a bewildering set of programming skills: implementing interfaces; overriding inherited methods; overloading constructors, methods, and operators; defining and coding static and instance methods.

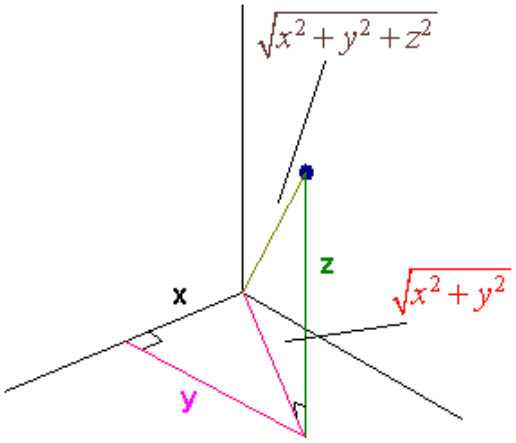
In this assignment, you will create a new type, a three-dimensional point, to practice these skills. Please submit only the completed program file (Program […].cs) to the drop box.

Overview

Our universe, in fact, our world, is three-dimensional; every object you see has height, depth, and width. A globe, for example, is a three-dimensional object.



A three-dimensional point is a location in three-dimensional space; it has three coordinates: x, y, and z.



From the origin, the distance to point (x, y, z) is calculated using the distance formula: $\sqrt{x^2 + y^2 + z^2}$.

Instructions

Setup

1. Download the assignment document (PDF) and the "Program […].cs" starter file.
2. Create a new C# Console application.
3. Delete the existing "Program.cs" file.
4. Add the "Program […].cs" file provided with the assignment.
5. Enter all code into the "Program […].cs" file.
6. Submit only the completed program file (Program […].cs) to the drop box.

Steps to Follow

Section 1: Defining the Point3D Type

1. Add a **struct** type named Point3D having three integer fields—X, Y, and Z—to your code module.
2. Insert a parameterized constructor initializing X, Y, and Z.

```
namespace UCLAExtension.ObjectOrientedProgramming.Inheritance.Point3D
{
    struct Point3D
    {
        public int X, Y, Z;
        public Point3D(int x, int y, int z)
        {
            X = x;
            Y = y;
            Z = z;
        }

    }
```

The distance between two points in the xyz-plane—(x_1 ,y_1 ,z_1) and (x_2, y_2, z_2)—is calculated by using the distance formula.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

3. Define a Length property returning the distance of a point from the origin. Note that the origin is at point (0, 0, 0).

```
public double Length
{
    get
    {
        return Math.Sqrt(X * X + Y * Y + Z * Z);
    }
}
```

4. Override the ToString method to display the X, Y, and Z values in this format: "(<x>,<y>,<z>)." ToString is inherited from System.Object. By default, it simply returns the fully-qualified type name (GetType().ToString()).

```
public override string ToString()
{
    return string.Format("({0},{1},{2})", X, Y, Z);
}
```

5. Override the Equals method to support equality testing. Equals is also inherited from System.Object.

```
public override bool Equals(object obj)
{
    Point3D other = (Point3D)obj;
    return (X == other.X) && (Y == other.Y) && (Z == other.Z);
}
```

6. Implement the generic IEquatable<T> interface to provide strongly-typed equality testing. This interface was added to the .NET Framework to make boxing unnecessary when testing for equality (The inherited Equals method takes an object parameter).

```
struct Point3D : IEquatable<Point3D>
{

public bool Equals(Point3D other)
{
    return other.X == this.X && other.Y == this.Y && other.Z == this.Z;
}
```

7. There are two overloaded Equals methods now; some code is duplicated, however. Have the non-generic Equals method delegate to the generic version.

```
public override bool Equals(object obj)
{
    return Equals((Point3D)obj);
}

public bool Equals(Point3D other)
{
    return other.X == this.X && other.Y == this.Y && other.Z == this.Z;
}
```

8. Implement the IComparable<T> and IComparable interfaces. The first interface, IComparable<T>, defines a generic CompareTo method; the second interface, IComparable, a non-generic CompareTo method. Note that as with the generic and non-generic Equals methods, the non-generic method should delegate to the generic version.

```
public int CompareTo(Point3D other)
{
    return Length.CompareTo(other.Length);
}

public int CompareTo(object obj)
{
    return CompareTo((Point3D)obj);
}
```

9. Override GetHashCode returning the result from calling GetHashCode on Length.

Overriding GetHashCode is recommended when overriding Equals in order to preserve equality semantics. When two points are equal, they must return the same hash code; unfortunately, this does not happen automatically. Where is the hash code used?

```
public override int GetHashCode()
{
    return Length.GetHashCode();
}
```

10. Create five points in Main; load them into a list of points.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Title = "Overriding and Interfaces: 3D Point";
        Console.WindowHeight = (int)(Console.LargestWindowHeight * 0.75);
        Console.BufferHeight = 5000;

        // Create five 3D points
        Point3D pt1 = new Point3D(100, 200, 400);
        Point3D pt2 = new Point3D(200, 100, 300);
        Point3D pt3 = new Point3D(150, 50, 450);
        Point3D pt4 = new Point3D(200, 100, 200);
        Point3D pt5 = new Point3D(300, 150, 50);

        // Add the points to a list of points
        List<Point3D> points = new List<Point3D>() { pt1, pt2, pt3, pt4, pt5 };
```

11. Display the points before sorting them.

```
// Display the points
Console.WriteLine("\n Display unsorted 3D points.\n");
foreach (Point3D point in points)
    Console.WriteLine(" {0}", point);
Console.WriteLine();
```

```
Display unsorted 3D points.

(100,200,400)
(200,100,300)
(150,50,450)
(200,100,200)
(300,150,50)
```

12. Display the points after sorting them.

```
// Display the sorted points
Console.WriteLine("\n Display sorted 3D points.\n");
foreach (Point3D point in points)
    Console.WriteLine(" {0}", point);
Console.WriteLine();
```

```
Display sorted 3D points.

(200,100,200)
(300,150,50)
(200,100,300)
(100,200,400)
(150,50,450)
```

13. Create a sixth point; set it equal to the fifth point. Test for equality using the Equals method.

```
// Create a new point; set it equal to point 5
Point3D pt6 = pt5;

// Test whether the points 5 and 6 are equal
Console.Write("\n Test for equality using Equals method: ");
if (pt5.Equals(pt6))
    Console.WriteLine(" Points 5 and 6 are equal\n");
else
    Console.WriteLine(" Points 5 and 6 are not equal\n");
```

```
Test for equality using Equals method:  Points 5 and 6 are equal
```

14. Add a static method named Distance that takes two points as parameters and returns the distance between them. Modify the Length property to call this method, instead.

```
public double Length
{
    get
    {
        return Distance(this, new Point3D(0, 0, 0));
    }
}

public static double Distance(Point3D pt1, Point3D pt2)
{
    return Math.Sqrt(Math.Pow((pt1.X - pt2.X), 2) +
                     Math.Pow((pt1.Y - pt2.Y), 2) +
                         Math.Pow((pt1.Z - pt2.Z), 2));
}
```

15. Add a static, read-only field named Origin that returns a Point3D object initialized to (0, 0, 0).

```
struct Point3D : IEquatable<Point3D>, IComparable<Point3D>, IComparable
{
    public static readonly Point3D Origin = new Point3D(0,0,0);
```

16. Modify the Length property to use this static field.

```
public double Length
{
    get
    {
        return Distance(this, Origin);
    }
}
```

Section 2: Overloading Operators and Other Things…

1. It would be nice to check for equality between points using the "==" operator instead of the Equals method. Add the following code to the Point3D class to overload the "==" and "!=" operators, respectively. Operator overloading is not used often, but here it makes sense.

```
public static bool operator ==(Point3D a, Point3D b)
{
    // If both are null, or both are same instance, return true.
    if (System.Object.ReferenceEquals(a, b))
    {
        return true;
    }

    // If one is null, but not both, return false.
    if (((object)a == null) || ((object)b == null))
    {
        return false;
    }

    // Return true if the fields match:
    return a.X == b.X && a.Y == b.Y && a.Z == b.Z;
}

public static bool operator !=(Point3D a, Point3D b)
{
    return !(a == b);
}
```

2. Test for equality using "=="this time.

```
// Test whether the points 5 and 6 are equal
Console.Write("\n Test for equality using '==' operator: ");
if (pt5 == pt6)
    Console.WriteLine(" Points 5 and 6 are equal\n");
else
    Console.WriteLine(" Points 5 and 6 are not equal\n");
```

```
Test for equality using '==' operator:  Points 5 and 6 are equal
```

3. Overload the greater than (>), greater than or equal (>=), less than (<), and less than or equal (<=) operators.

```
public static bool operator >=(Point3D a, Point3D b)
{
    return a.Length >= b.Length;
}

public static bool operator >(Point3D a, Point3D b)
{
    return a.Length > b.Length;
}

public static bool operator <=(Point3D a, Point3D b)
{
    return a.Length <= b.Length;
}

public static bool operator <(Point3D a, Point3D b)
{
    return a.Length < b.Length;
}
```

4. Provide addition (+) and subtraction (-) operators.

```
public static Point3D operator +(Point3D a, Point3D b)
{
    return new Point3D(a.X + b.X, a.Y + b.Y, a.Z + b.Z);
}

public static Point3D operator -(Point3D a, Point3D b)
{
    return new Point3D(a.X - b.X, a.Y - b.Y, a.Z - b.Z);
}
```

5. Add points four, five, and six together. Store the result in point seven.

```
// Add points 4, 5, and 6 together
Point3D pt7 = pt4 + pt5 + pt6;
```

6. Display points 4, 5, 6, and 7.

```
// Add points 4, 5, and 6 together
Point3D pt7 = pt4 + pt5 + pt6;

// Display points 4, 5, 6, and 7
Console.WriteLine("\n Display points 4, 5, 6, and 7.\n");
Console.WriteLine(" pt4: {0}", pt4);
Console.WriteLine(" pt5: {0}", pt5);
Console.WriteLine(" pt6: {0}", pt6);
Console.WriteLine(" pt7: {0}", pt7);
```

```
Display points 4, 5, 6, and 7.

pt4: (200,100,200)
pt5: (300,150,50)
pt6: (300,150,50)
pt7: (800,400,300)
```

Did the addition operator work correctly? Test the subtraction operator, too, if you like.

Some languages don't support operator overloading, so it's a good idea to supply alternative methods that perform the same operations, for example, Add and Subtract.

7. Provide alternative Add and Subtract instance methods. Have the plus and minus operators delegate to these methods. Reusing code is a good habit.

```
public Point3D Add(Point3D p)
{
    return new Point3D(X + p.X, Y + p.Y, Z + p.Z);
}

public Point3D Subtract(Point3D p)
{
    return new Point3D(X - p.X, Y - p.Y, Z - p.Z);
}

public static Point3D operator +(Point3D a, Point3D b)
{
    return a.Add(b);
}

public static Point3D operator -(Point3D a, Point3D b)
{
    return a.Subtract(b);
}
```

8. Add an overloaded method named “Offset” that takes three integer parameters—x, y, and, z—or a Point3D object and adds them to the existing point. One method should call the other.

```
public void Offset(int x, int y, int z)
{
    X += x;
    Y += y;
    Z += z;
}

public void Offset(Point3D p)
{
    Offset(p.X, p.Y, p.Z);
}
```

9. Try the Offset method out

```
// Offset point 7 by (50, 20, 100)
Console.WriteLine("\n Offset point 7 by (50, 20, 100)\n");

Console.WriteLine("\n Before Offset: {0}.\n", pt7);

pt7.Offset(50, 20, 100);

Console.WriteLine("\n After Offset: {0}.\n", pt7);
```

```
Offset point 7 by (50, 20, 100)

Before Offset: (800,400,300).

After Offset: (850,420,400).
```

Summary

This assignment offered an excellent opportunity to exercise all sorts of object-oriented techniques on a new type: implementing interfaces, overriding and overloading methods, overloading operators, defining static and instance members. These are essential skills for the object-oriented programmer.

A lot of discussion was devoted to demonstrating best practices. One good practice is to **reuse code**; another is to **keep it simple**…