Oriented Object Programming

Arnau Pascual - DAMv1

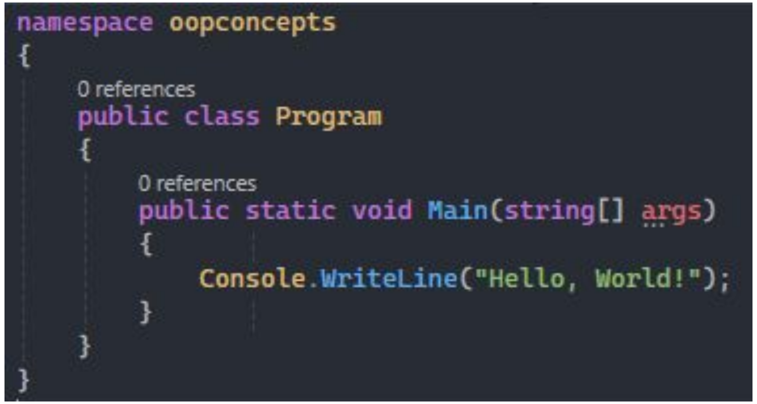
Object
Class
Instance

Pencil

- Color
- Hardness

Encapsulation - Access modifiers

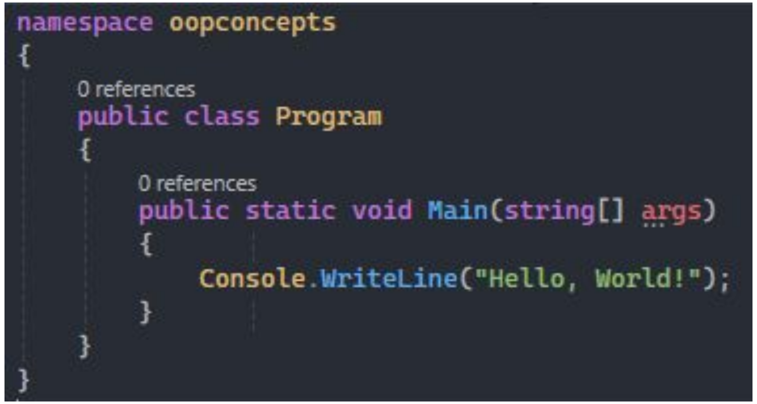
- **Private**: Members are visible only within the same class.
- **Public**: Members are accessible from outside the class
- **Protected**: Members are visible to the class itself and its subclasses.
- **Package**: Members are visible within the same package.



```
namespace oopconcepts
{
    0 references
    public class Program
    {
        0 references
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

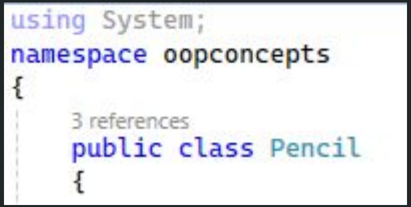
Encapsulation - Non-access modifiers

- **Static**: Makes attributes and methods belong to the class rather than the instance.
- **Abstract**: Defines classes or methods as abstract, meaning they are declared but not implemented.
- **Final**: Modifies a member so that it cannot be changed.

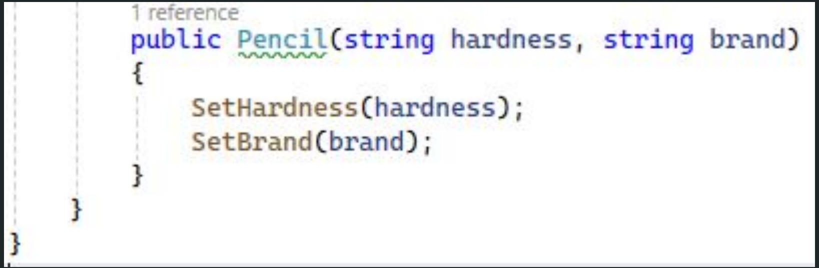


```
namespace oopconcepts
{
    0 references
    public class Program
    {
        0 references
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Constructor

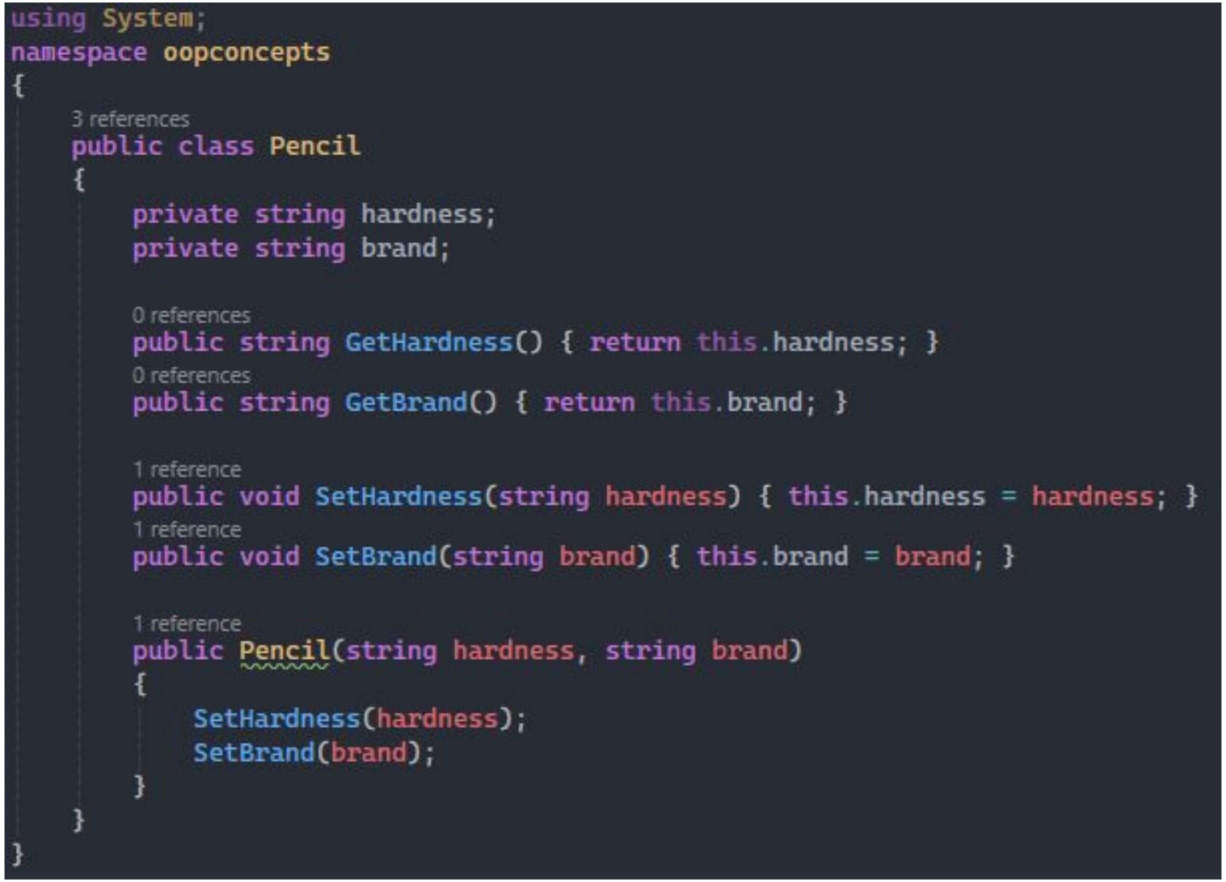


```
using System;
namespace oopconcepts
{
    3 references
    public class Pencil
    {
```



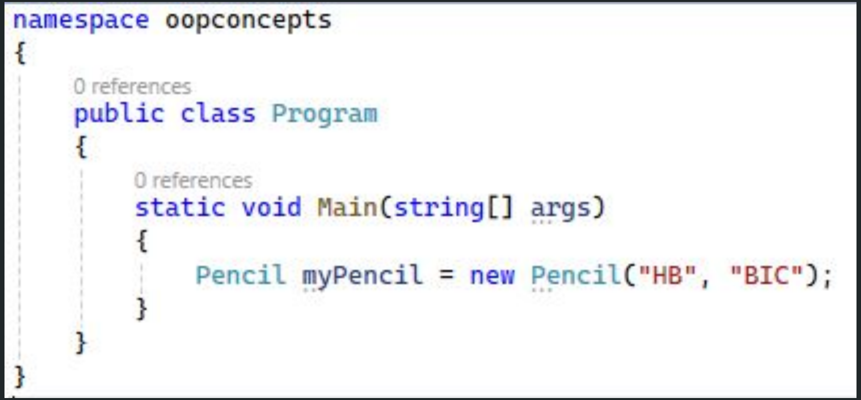
```
        1 reference
        public Pencil(string hardness, string brand)
        {
            SetHardness(hardness);
            SetBrand(brand);
        }
    }
}
```

Pencil Class Example



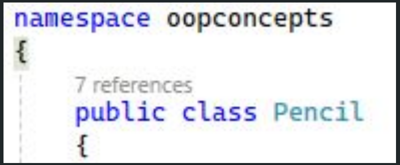
```
using System;
namespace oopconcepts
{
    3 references
    public class Pencil
    {
        private string hardness;
        private string brand;

        0 references
        public string GetHardness() { return this.hardness; }
        0 references
        public string GetBrand() { return this.brand; }

        1 reference
        public void SetHardness(string hardness) { this.hardness = hardness; }
        1 reference
        public void SetBrand(string brand) { this.brand = brand; }

        1 reference
        public Pencil(string hardness, string brand)
        {
            SetHardness(hardness);
            SetBrand(brand);
        }
    }
}
```

Instance

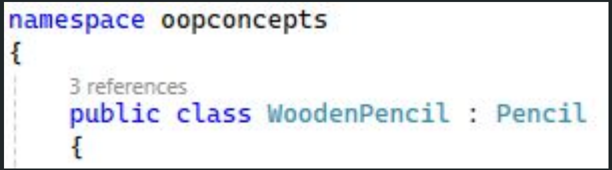


```
namespace oopconcepts
{
    0 references
    public class Program
    {
        0 references
        static void Main(string[] args)
        {
            Pencil myPencil = new Pencil("HB", "BIC");
        }
    }
}
```

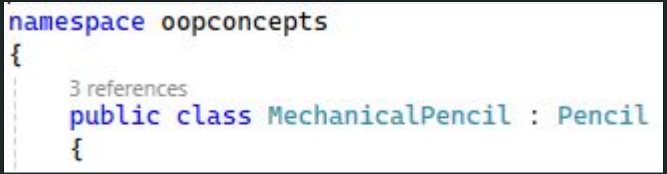
Inheritance



```
namespace oopconcepts
{
    7 references
    public class Pencil
    {
```



```
namespace oopconcepts
{
    3 references
    public class WoodenPencil : Pencil
    {
```

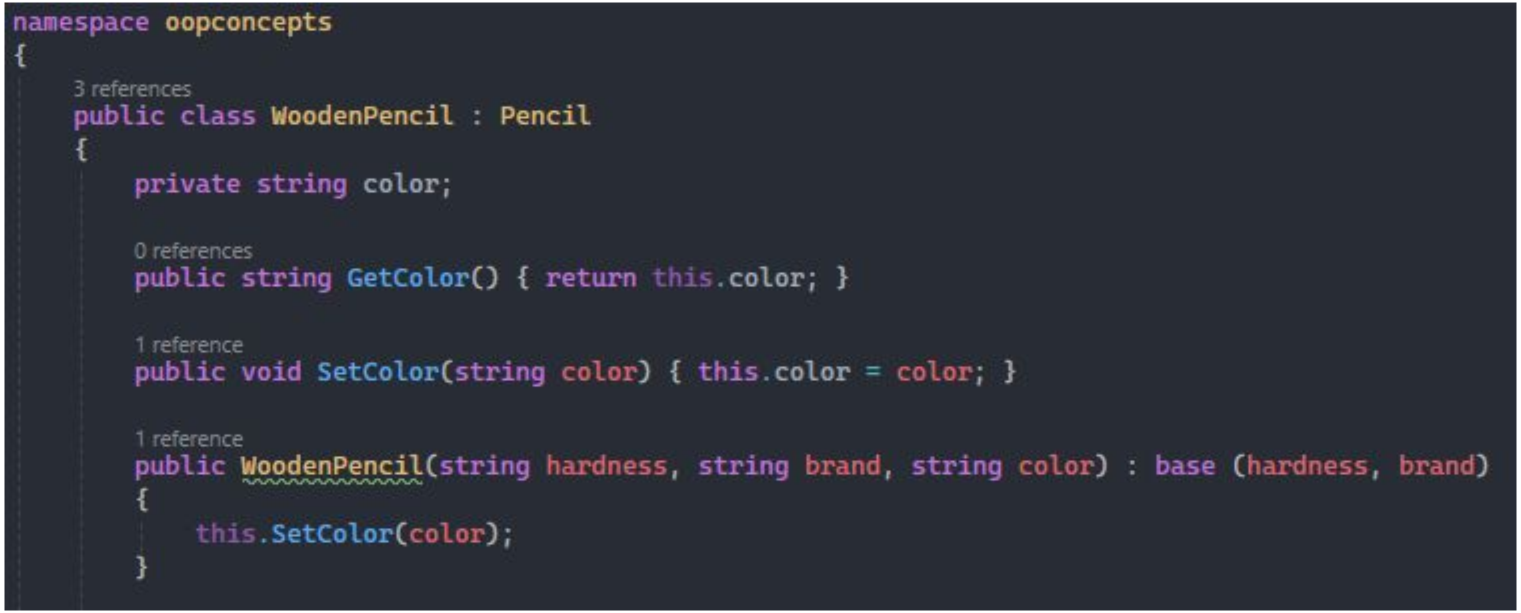


```
namespace oopconcepts
{
    3 references
    public class MechanicalPencil : Pencil
    {
```

Polymorphism

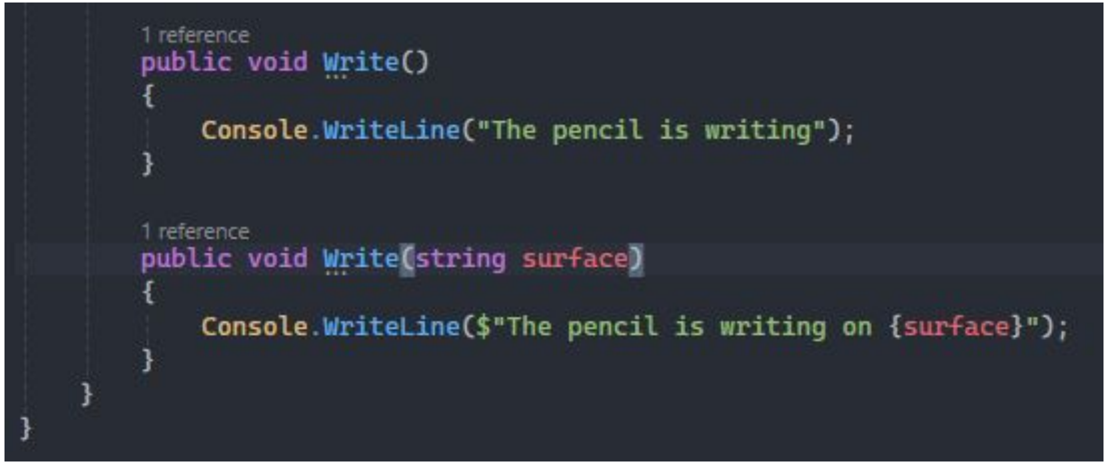
- **Overloading** involves having multiple methods with the same name but different parameters.
- **Overriding** occurs when a method in the parent class is inherited by the child class.

Inheritance Example



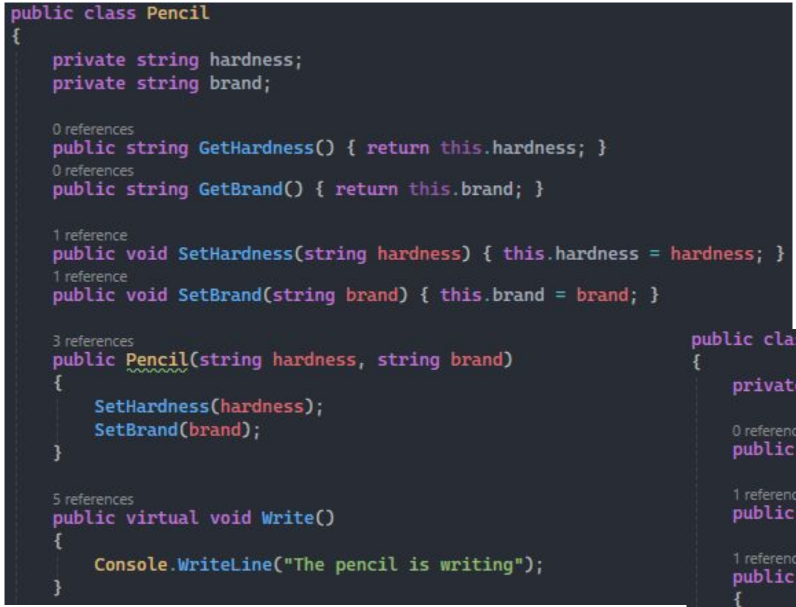
```
namespace oopconcepts
{
    3 references
    public class WoodenPencil : Pencil
    {
        private string color;

        0 references
        public string GetColor() { return this.color; }

        1 reference
        public void SetColor(string color) { this.color = color; }

        1 reference
        public WoodenPencil(string hardness, string brand, string color) : base (hardness, brand)
        {
            this.SetColor(color);
        }
```

Overloading Example

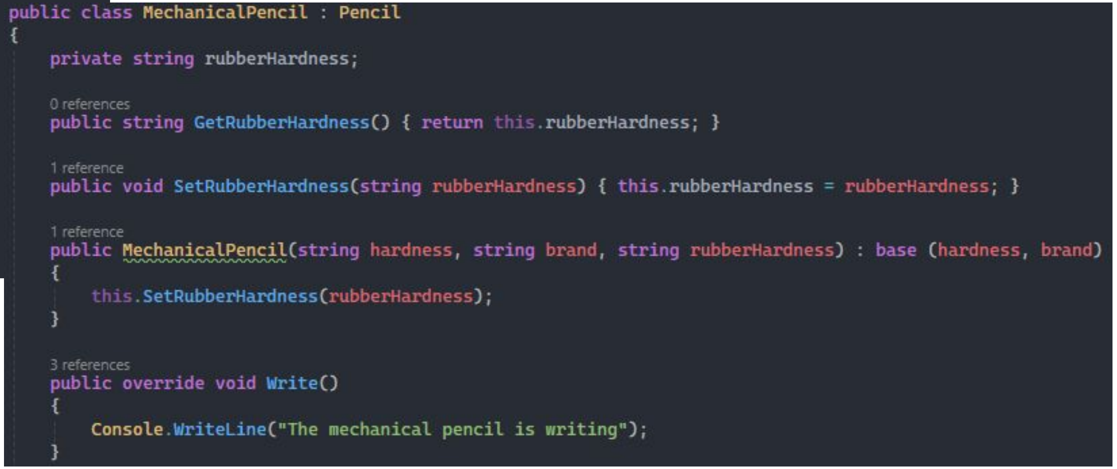


```
        1 reference
        public void Write()
        {
            Console.WriteLine("The pencil is writing");
        }

        1 reference
        public void Write(string surface)
        {
            Console.WriteLine($"The pencil is writing on {surface}");
        }
    }
}
```

Overriding Example

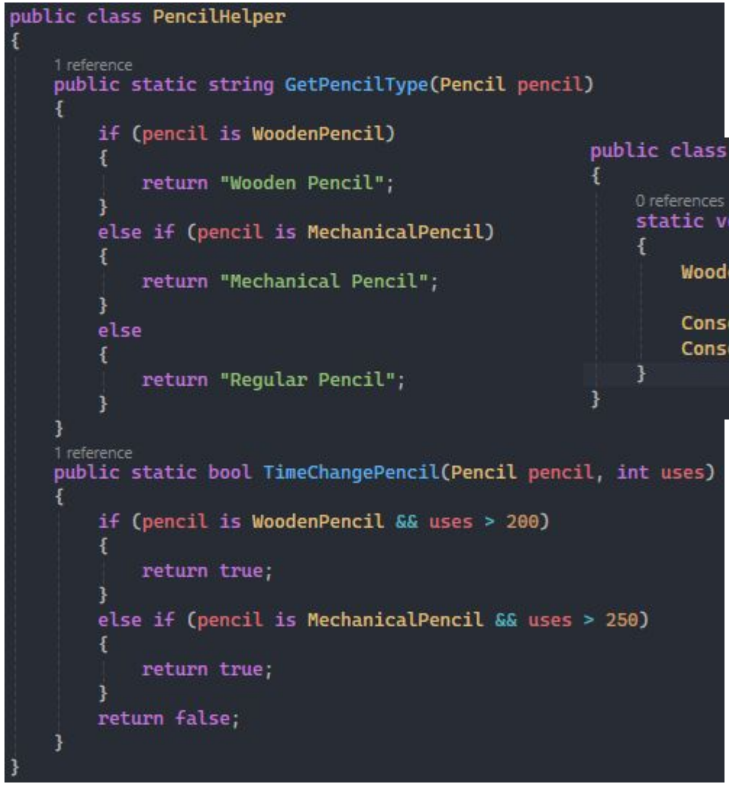


```
public class Pencil
{
    private string hardness;
    private string brand;

    0 references
    public string GetHardness() { return this.hardness; }
    0 references
    public string GetBrand() { return this.brand; }

    1 reference
    public void SetHardness(string hardness) { this.hardness = hardness; }
    1 reference
    public void SetBrand(string brand) { this.brand = brand; }

    3 references
    public Pencil(string hardness, string brand)
    {
        SetHardness(hardness);
        SetBrand(brand);
    }

    5 references
    public virtual void Write()
    {
        Console.WriteLine("The pencil is writing");
    }
```

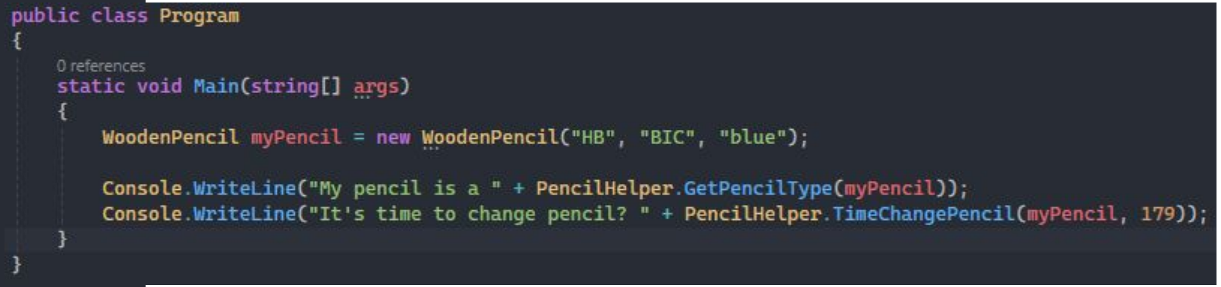


```
public class MechanicalPencil : Pencil
{
    private string rubberHardness;

    0 references
    public string GetRubberHardness() { return this.rubberHardness; }

    1 reference
    public void SetRubberHardness(string rubberHardness) { this.rubberHardness = rubberHardness; }

    1 reference
    public MechanicalPencil(string hardness, string brand, string rubberHardness) : base (hardness, brand)
    {
        this.SetRubberHardness(rubberHardness);
    }

    3 references
    public override void Write()
    {
        Console.WriteLine("The mechanical pencil is writing");
    }
```

Helper class



```
public class PencilHelper
{
    1 reference
    public static string GetPencilType(Pencil pencil)
    {
        if (pencil is WoodenPencil)
        {
            return "Wooden Pencil";
        }
        else if (pencil is MechanicalPencil)
        {
            return "Mechanical Pencil";
        }
        else
        {
            return "Regular Pencil";
        }
    }
    1 reference
    public static bool TimeChangePencil(Pencil pencil, int uses)
    {
        if (pencil is WoodenPencil && uses > 200)
        {
            return true;
        }
        else if (pencil is MechanicalPencil && uses > 250)
        {
            return true;
        }
        return false;
    }
}
```

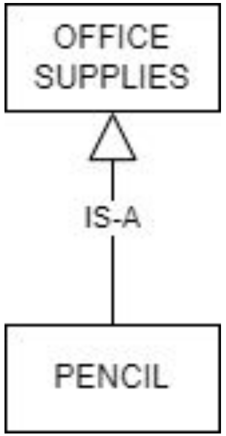


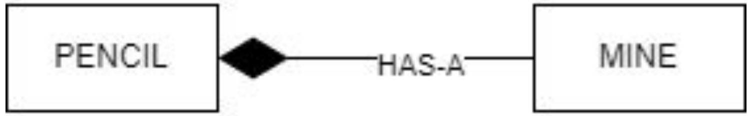
```
public class Program
{
    0 references
    static void Main(string[] args)
    {
        WoodenPencil myPencil = new WoodenPencil("HB", "BIC", "blue");

        Console.WriteLine("My pencil is a " + PencilHelper.GetPencilType(myPencil));
        Console.WriteLine("It's time to change pencil? " + PencilHelper.TimeChangePencil(myPencil, 179));
    }
}
```

Relationships

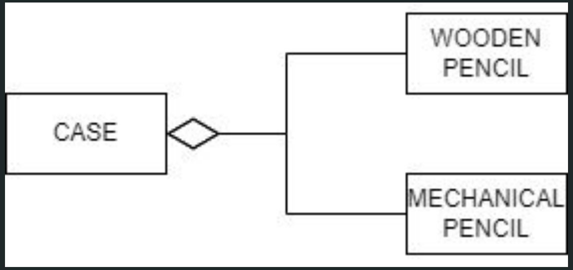
The **IS-A** relationship is a concept of inheritance, and it is unidirectional.

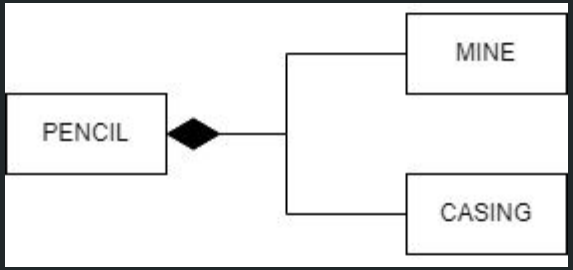
The **HAS-A** relationship is a concept where one class has another class as a component or member.

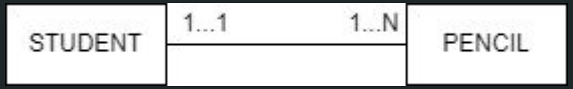




HAS-A relationship


CASE
WOODEN PENCIL
MECHANICAL PENCIL


PENCIL
MINE
CASING


STUDENT
1...1
1...N
PENCIL