**Snippet 07 — Protected Method**

* **Code Recap**

public class MyClass

{

    protected void MyMethod()

    {

        Console.WriteLine("Hello");

    }

}

1. **Working Theory (keywords & concepts)**

**protected**

* Accessible **inside the same class** **and** in **derived (child) classes**.
* Not accessible from outside (unlike public).
* Use case: when you want a **base class** to provide functionality, but only child classes should use/extend it.

**How it differs**

* public → everyone can call it.
* private → only the class itself can call it.
* protected → class itself **and children** can call it.

**Practical Example:**

1. **Base class with protected method:**

* Create a file, name as “MyClass.cs”:

public class MyClass

{

    protected void MyMethod()

    {

        Console.WriteLine("Hello from protected method");

    }

}

1. **Derived class accessing protected method**

* Then, create another class name as “ChildClass.cs”:

public class ChildClass : MyClass

{

    public void CallParentMethod()

    {

        MyMethod(); // ✅ allowed (inherited)

    }

}

1. **Program.cs**

* **The Main Program class**

class Program

{

    static void Main()

    {

        var child = new ChildClass();

        child.CallParentMethod(); // ✅ works

        // child.MyMethod();      // ❌ Error: not accessible outside class hierarchy

    }

}

**Output**

Hello from protected method.

**🔧 Extras**

* **Why use protected?**
  + Encourages **inheritance** and controlled extensibility.
  + Lets you **hide implementation details** from outside code.
* **Protected internal**
  + Combination: accessible in same assembly **and** in derived classes.
* protected internal void Something() { }
* **Best practice**
  + Use protected for methods meant only for subclasses.
  + Keep your base class minimal — don’t overexpose members.

✅ snippet 07 explained: protected is about **controlled inheritance visibility** — subclasses can see/use it, outside code cannot.