

Understanding the Fundamentals of Object-orientation



Gill Cleeren

CTO Xebia Microsoft Services Belgium

@gillcleeren

Overview



**Understanding object-oriented
programming in C#**

Applying encapsulation

Adding inheritance

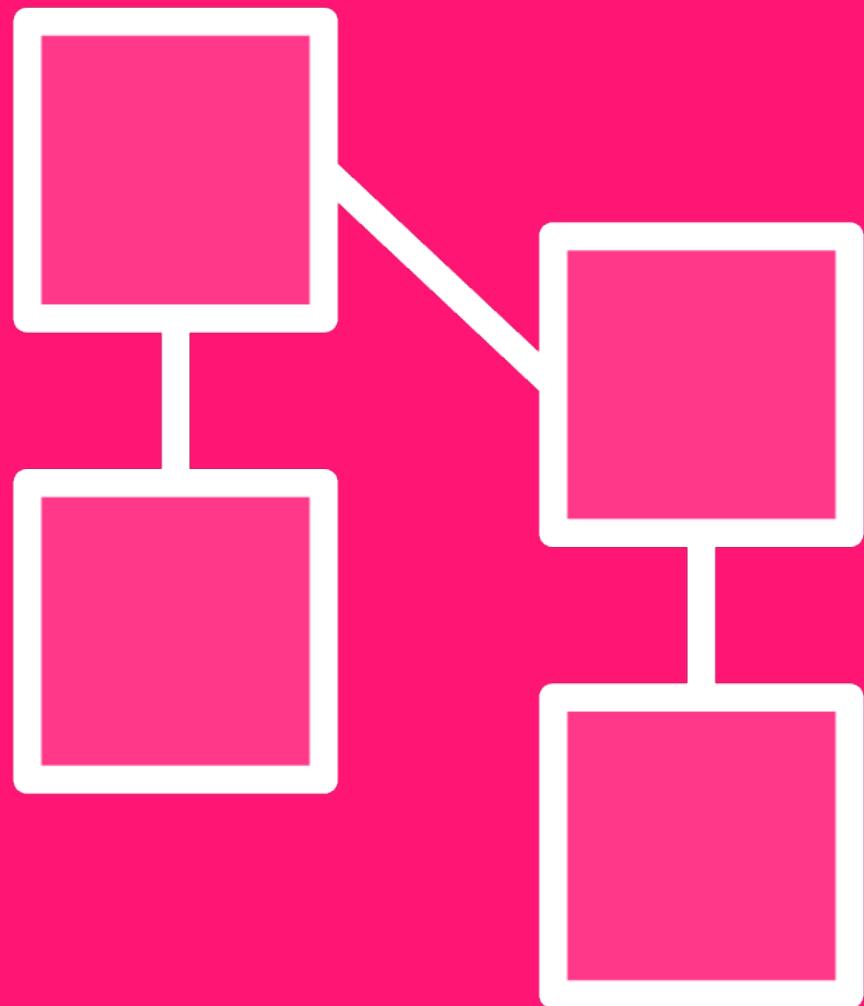
Using polymorphism

Introducing interfaces



Understanding Object-oriented Programming in C#





Object-oriented Programming in C#

Programming paradigm

C# is an OO language

Focus on classes and objects



Structure of Object-oriented Programming

Classes

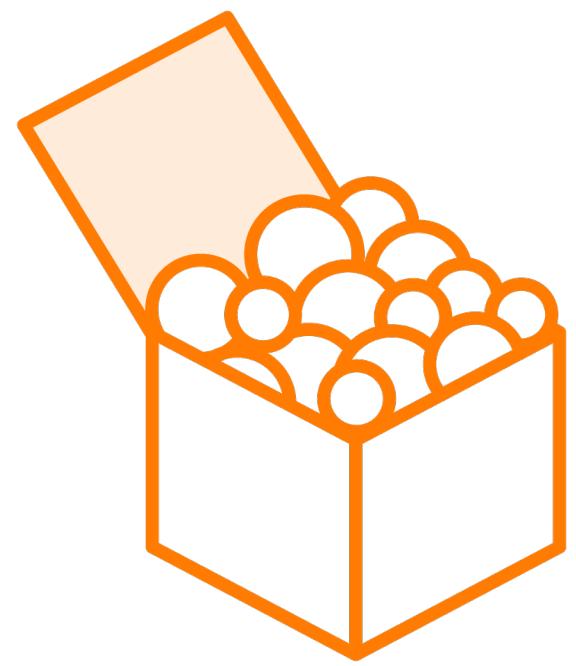
Objects

Methods

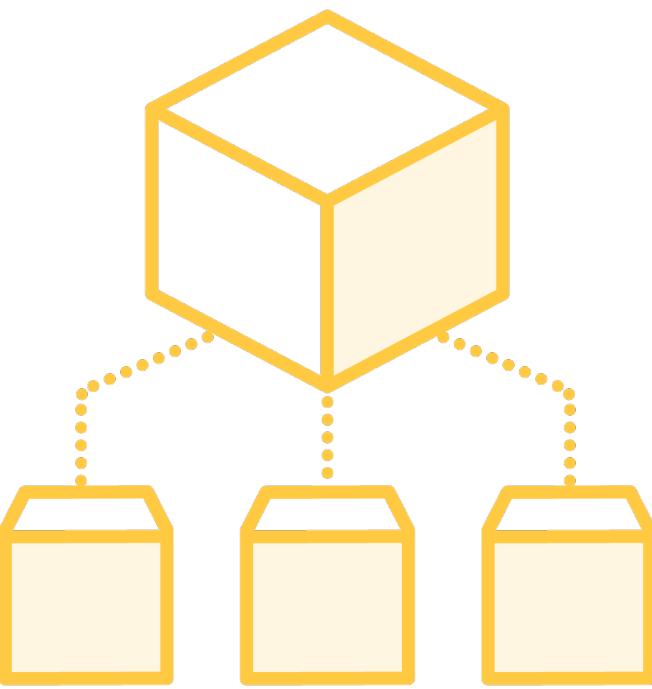
Properties



The Four Pillars of OO



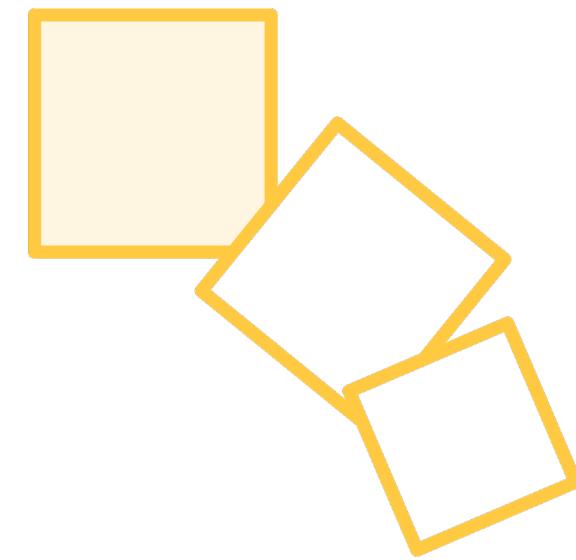
Encapsulation



Abstraction



Inheritance



Polymorphism



Encapsulation



Containing information inside object

Only certain information is exposed

Hides internal implementation and data

Avoid data corruption

Private & public



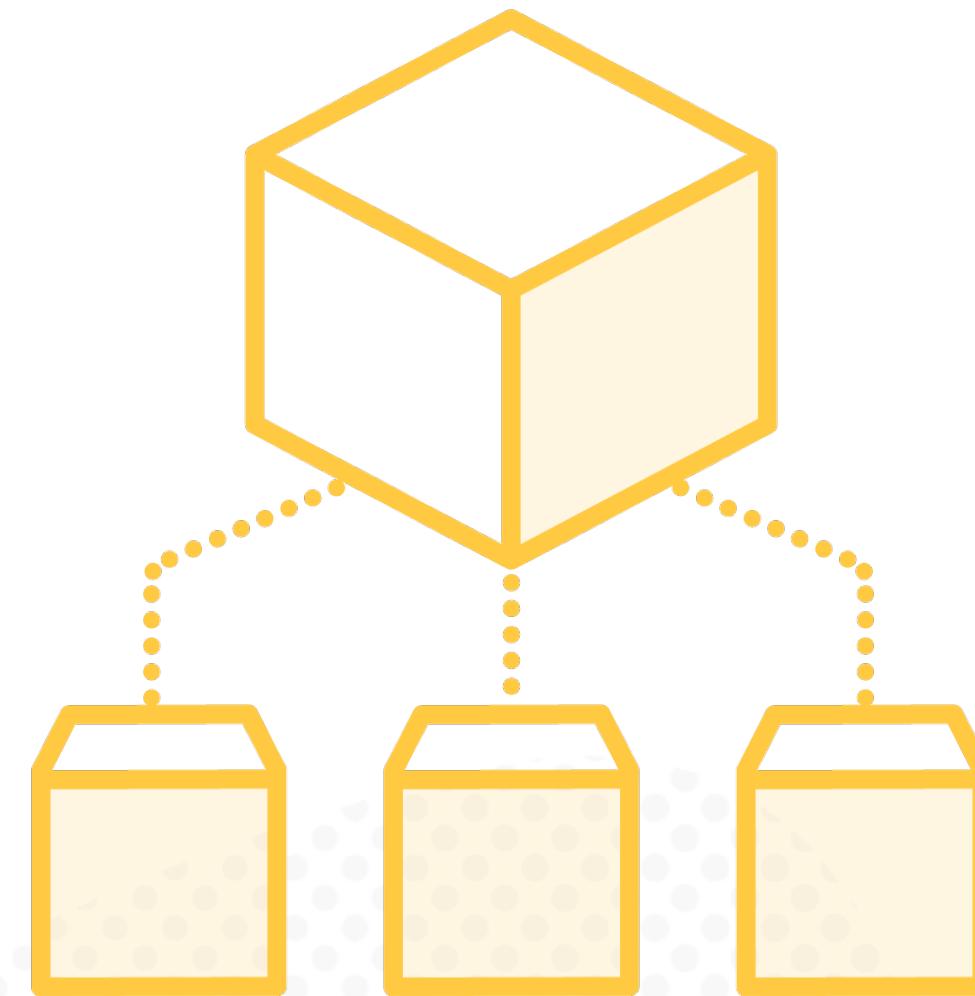
Abstraction

Abstract representation of the program

Only mechanisms useful for other objects are revealed

Implementation is hidden

Making changes becomes easier



Inheritance



Classes can reuse functionality from others

Relation between classes

**Lower development time because of
reusability**

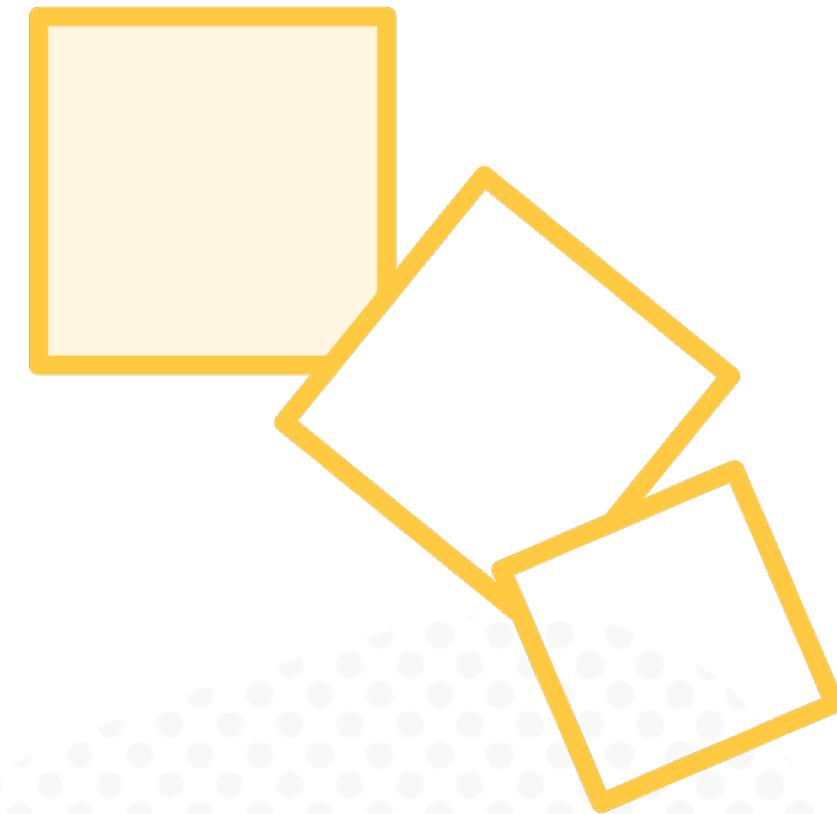


Polymorphism

**Share behaviors but can be
in more than one form**

Child can be used like its parent

**Correct method will be used based on
execution**



Adding Encapsulation



So Far, Our Data Is Stored in Fields

```
public class Employee
{
    public string firstName;
    public int age;

    public Employee(string name, int ageValue)
    {
        firstName = name;
        age = ageValue;
    }
}
```



```
Employee employee1 = new Employee();  
employee1.firstName = "Bethany";
```

Manipulating a Class's Data

Other classes can directly change the field data





Access to class data

If data is public, everyone can change the data of an object



Adding Methods to Alter Data

```
public class Employee
{
    private string firstName;
    private int age;

    public int GetAge()
    {
        return age;
    }

    public void SetAge(int newAge)
    {
        age = newAge;
    }
}
```





Using methods

Syntax-wise, this is not ideal

**Solution in C# to enforce
encapsulation: Properties**



Introducing Properties

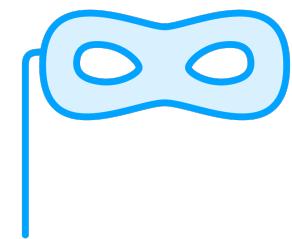
```
public class Employee
{
    private int age;
    public int Age
    {
        get { return age; }
        set
        {
            age = value;
        }
    }
}
```



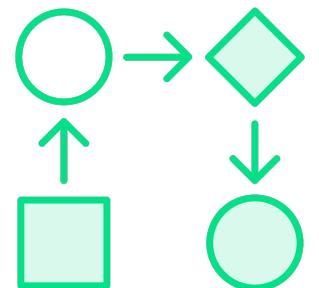
C# Properties



Wraps data (fields) of a class



Hide implementation



Define get and set implementation



```
Employee employee = new Employee();
```

◀ Instantiating the object

```
employee.FirstName = "Bethany";
```

◀ Setting a value through a property

```
int empFirstName = employee.FirstName;
```

◀ Getting the value through a property



Demo



Adding properties on our class

Using the properties instead of the fields

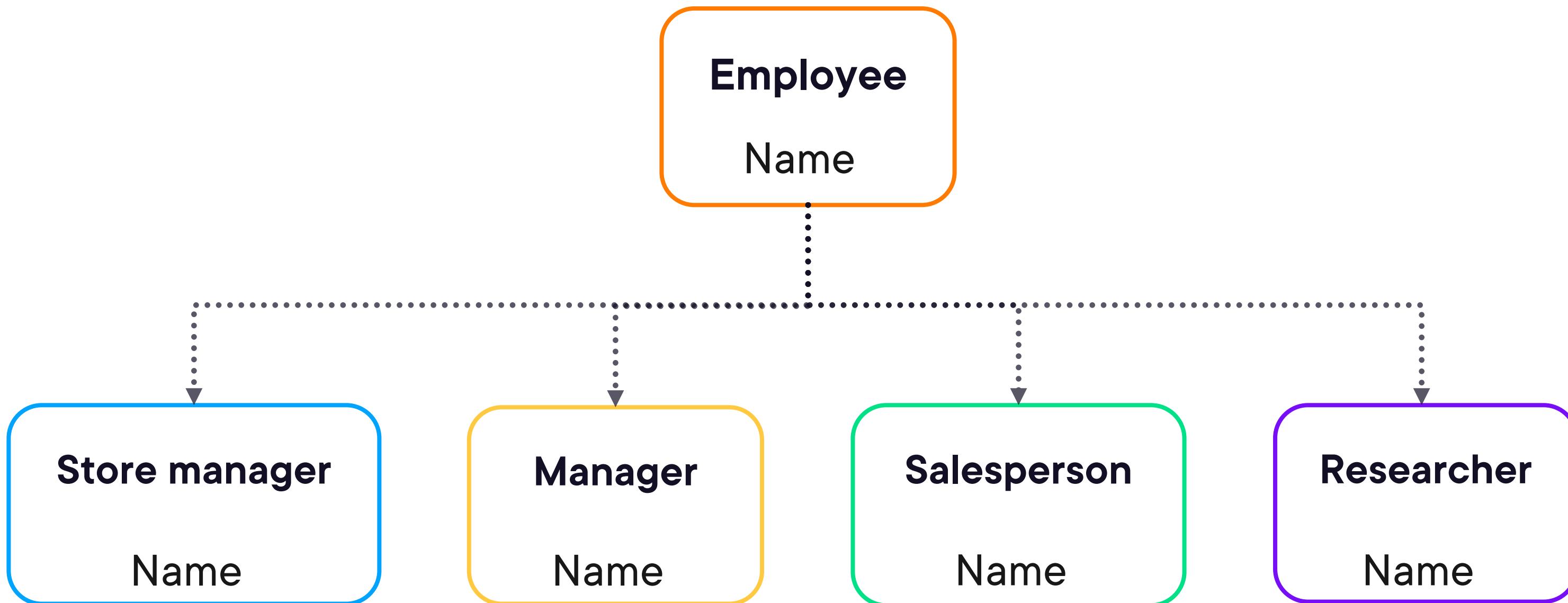
Protecting data on the Employee



Adding Inheritance



Different Types of Employees





Introducing inheritance

Important concept in object-oriented development

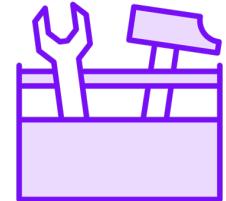
Class gets data and functionality from parent



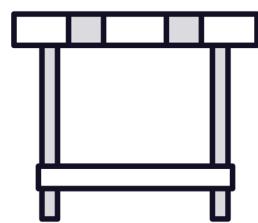
Using Inheritance in C#



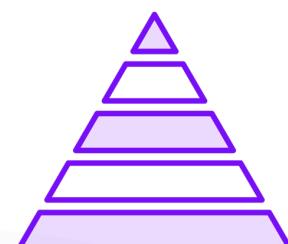
Parent (or base) and derived class



Reuse code



Easier to maintain



Can be one or more levels deep



Creating a Base and a Derived Type

```
public class BaseClass  
{  
}
```

```
public class DerivedClass: BaseClass  
{  
}
```

Base and Derived Classes



Creating the Base and Derived Class

Employee

```
public class Employee  
{  
}
```

Manager

```
public class Manager: Employee  
{  
}
```



Accessing the Base Class Members

```
public class Employee
{
    public string name;
    public void PerformWork()
    {
    }
}
```

```
public class Manager: Employee
{
    public void DisplayManagerData()
    {
        Console.WriteLine(name);
    }
}
```



Revisiting Access Modifiers

public

private

protected



Accessing the Base Class Members

```
public class Employee  
{  
    private string name;  
  
    public void PerformWork()  
    {  
    }  
}
```



```
public class Manager: Employee  
{  
    public void DisplayData()  
    {  
        Console.WriteLine(name); //error  
    }  
}
```



Accessing the Base Class Members

```
public class Employee
{
    protected string name;
    public void PerformWork()
    {
    }
}
```

```
public class Manager: Employee
{
    public void DisplayData()
    {
        Console.WriteLine(name); //ok
    }
}
```



Demo



Creating a base class

Building a derived class

Access the base class' members



The “Is-A” Relation



```
Manager m1 = new Manager(); //Manager derives from Employee  
Researcher r2 = new Researcher(); //Researcher derives from Employee  
m1.PerformWork(); //will call PerformWork() on the base Employee class  
r2.PerformWork(); //will call PerformWork() on the base Employee class
```

Using the Base Type

Using the Is-A relation



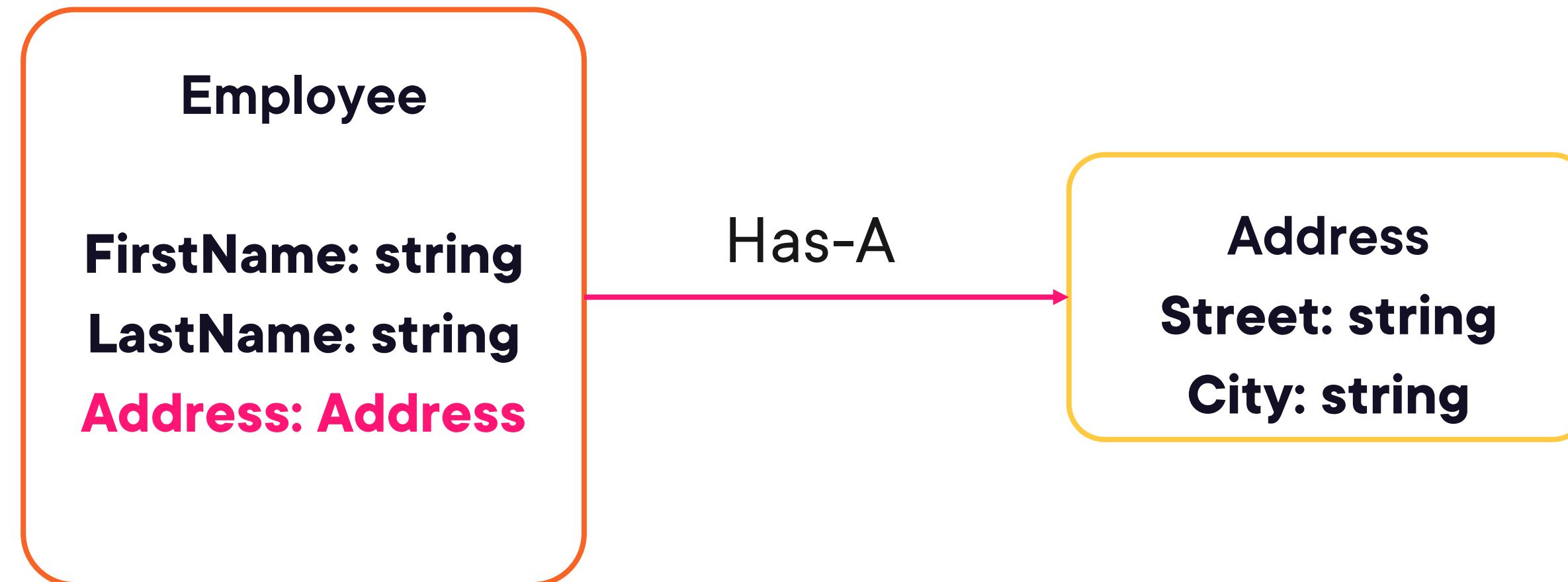
Demo



Using the “Is-A” relation



Understanding Composition



Demo

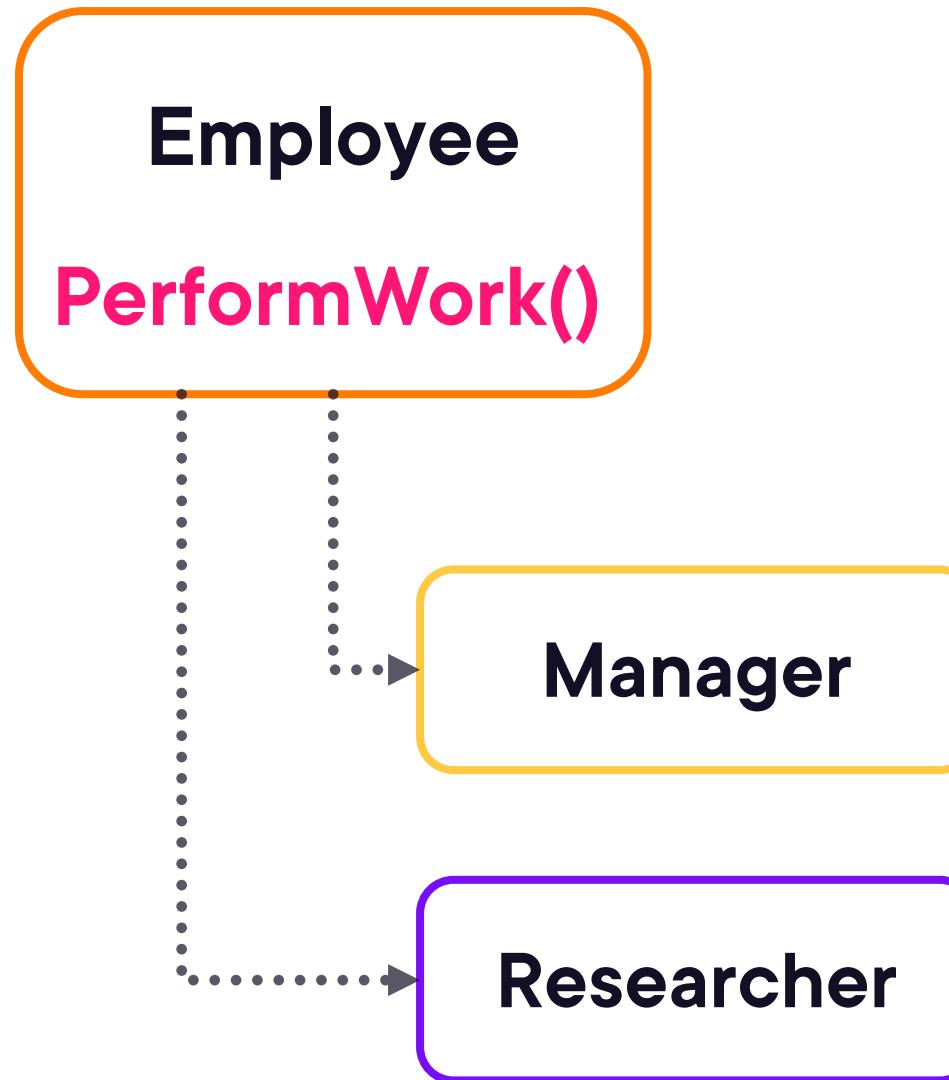


Adding a class to model the Address



Using Polymorphism

Using a Base Method



```
public class Employee  
{  
    public void PerformWork()  
    { ... }  
}
```

```
public class Manager: Employee  
{ }
```

```
public class Researcher: Employee  
{ }
```

```
Manager m1 = new Manager();  
m1.PerformWork();  
Researcher r1 = new Researcher();  
r1.PerformWork();
```



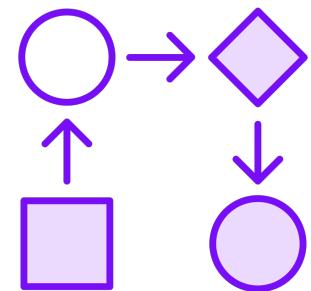
The invoked method will be
the same for all inheriting
types.



Introducing Polymorphism



Override a base class method with a new implementation



“Poly” & “morph”



Uses virtual and override keywords



Introducing Polymorphism

Employee

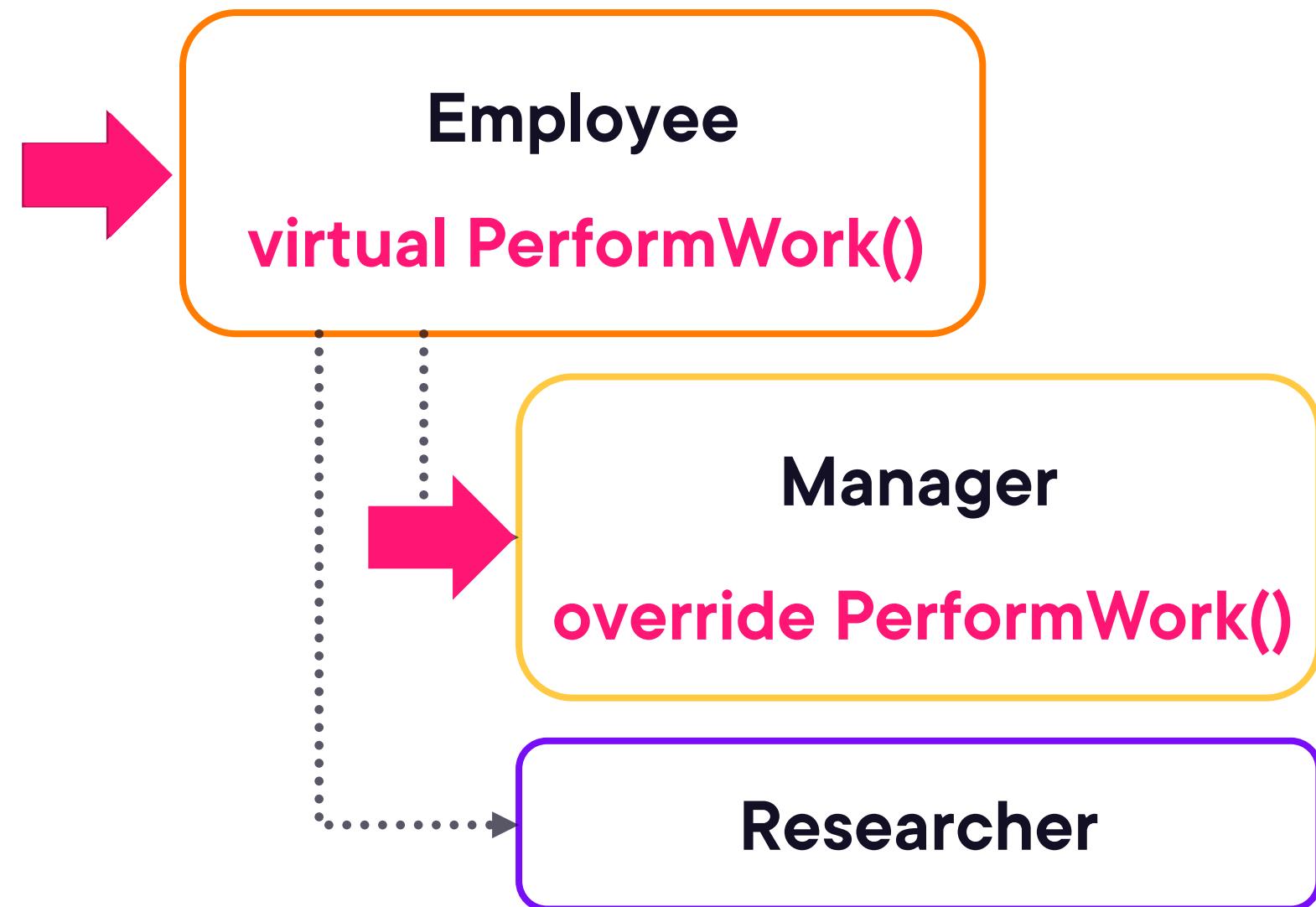
```
public class Employee
{
    public virtual void PerformWork()
    { ... }
}
```

Manager

```
public class Manager : Employee
{
    public override void PerformWork()
    { ... }
}
```



Using Polymorphism in C#

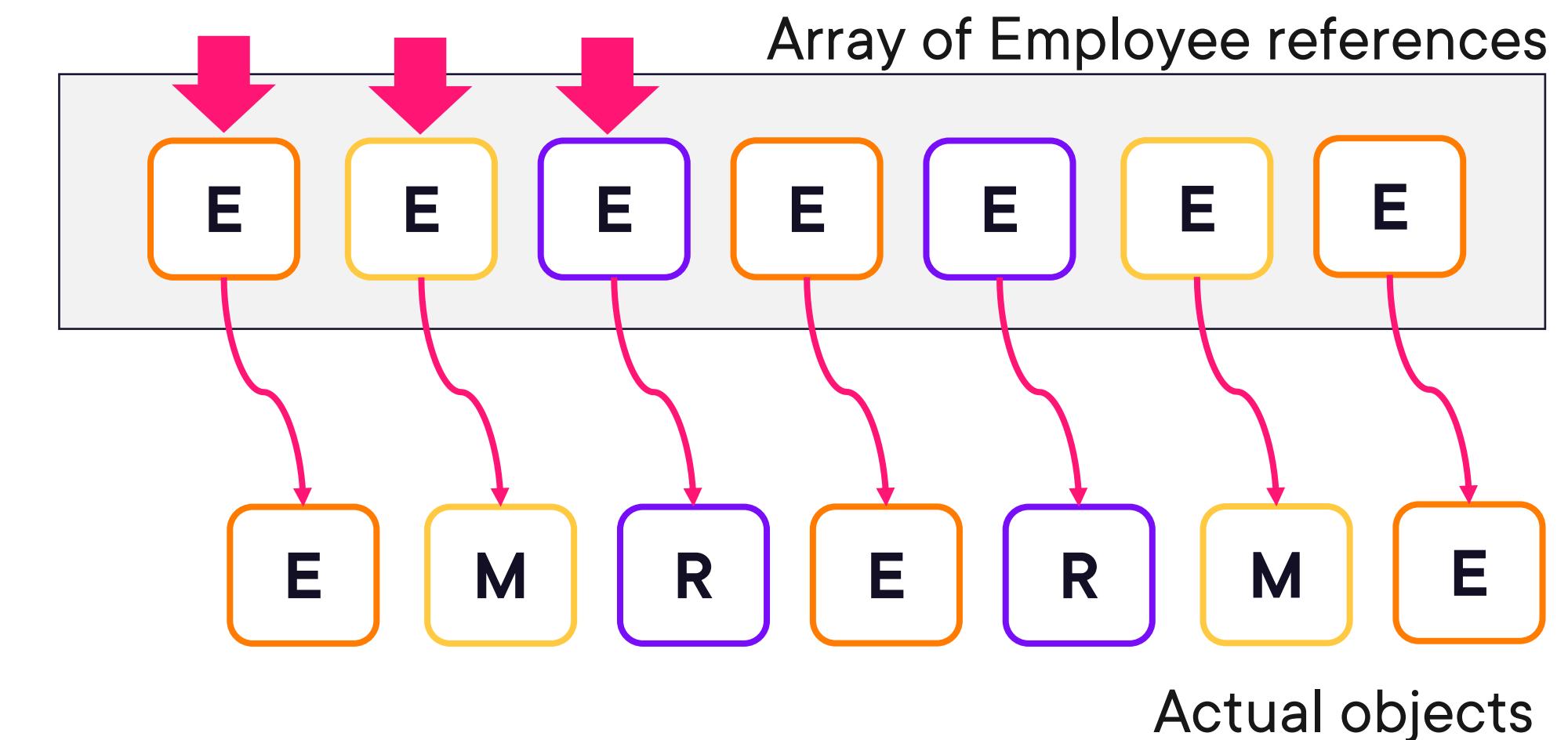
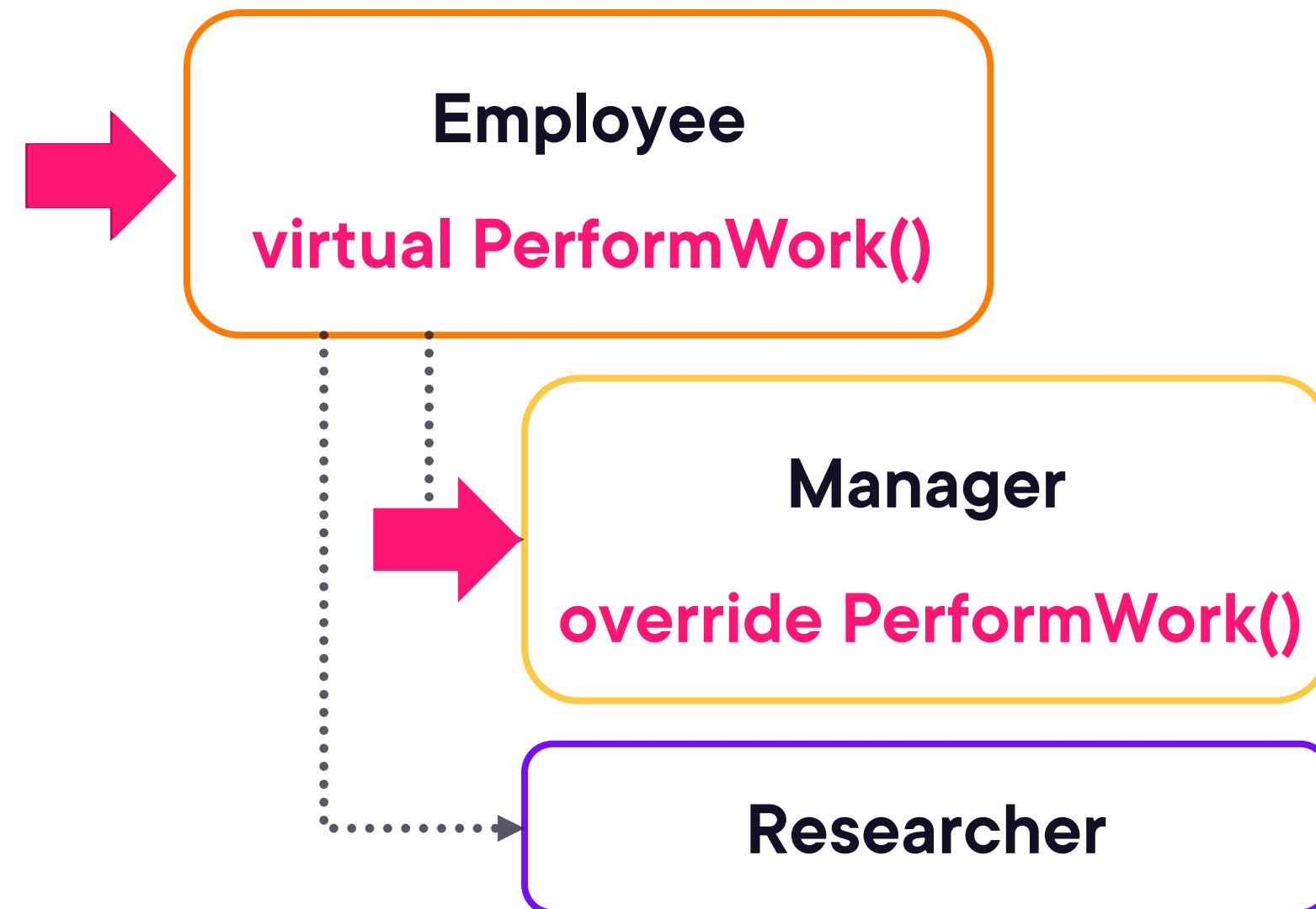


```
Employee e1 = new Manager();
Employee e2 = new Researcher();
e1.PerformWork(); //will call the most specific version, so the one on Manager
e2.PerformWork(); //will call the most specific version, so the one on Researcher
e1.AttendManagementMeeting(); //error if defined on Manager derived type
```

Using Polymorphism



Looping over an Array of Employee References



Demo



**Adding virtual and override
Using polymorphism**





Introducing Interfaces



Recap: The Different Custom Categories of Types

Enum

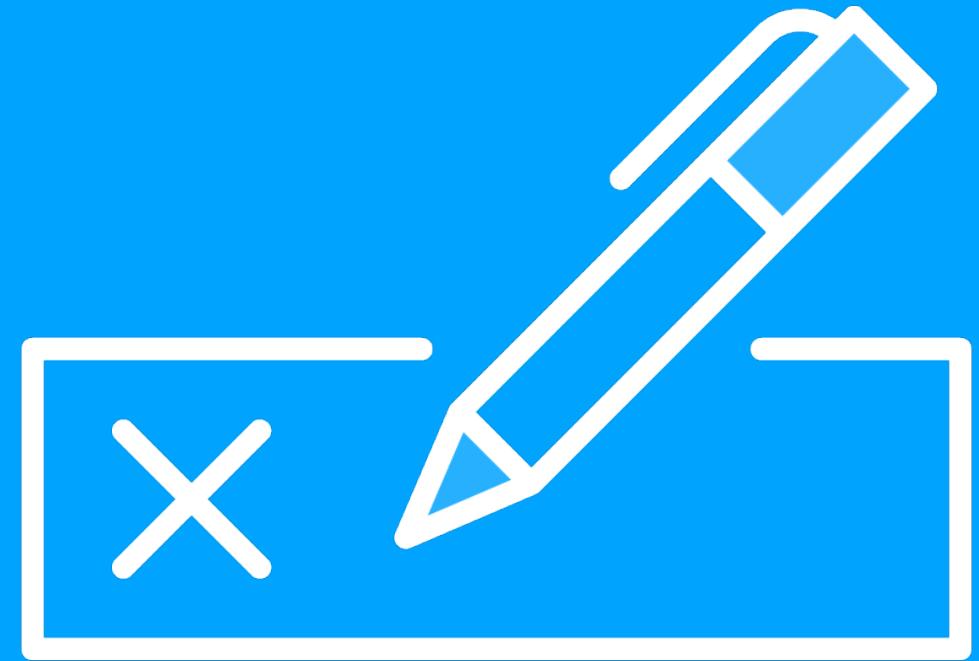
Struct

Class

Interface

Delegate





Understanding C# Interfaces

Define a contract that must be
implemented by classes that use it



```
public interface IEmployee
{
    void PerformWork();
    int ReceiveWage();
}
```

A Sample Interface



Implementing an Interface

```
public void Manager: IEmployee
{
    public void PerformWork()
    {
        ...
    }

    public int ReceiveWage()
    {
        ...
    }
}
```



```
IEmployee e1 = new Manager();  
e1.PerformWork();
```

Using Polymorphism with Interfaces



Demo



Creating a custom interface

Implementing an interface

Using an interface



Summary



C# fully supports object-oriented programming

Properties help with encapsulation

Inheritance helps with code reuse

Polymorphism allows giving a specific implementation

Interfaces define a contract all implementing types need to adhere to



Up Next:

Testing our code

