Basic Inheritance and Encapsulation

Class Hierarchies

Table of Contents

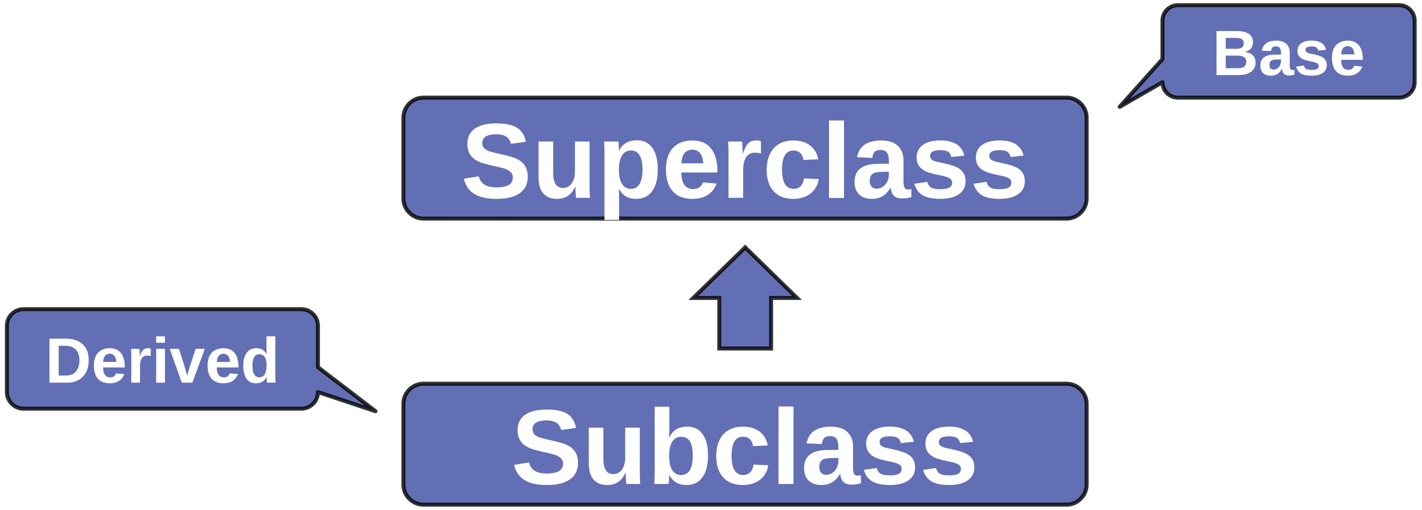
- Inheritance
- Class Hierarchies
- Inheritance in C#
- What is Encapsulation?
- Keyword this
- Access Modifiers

EXTENDING CLASSES

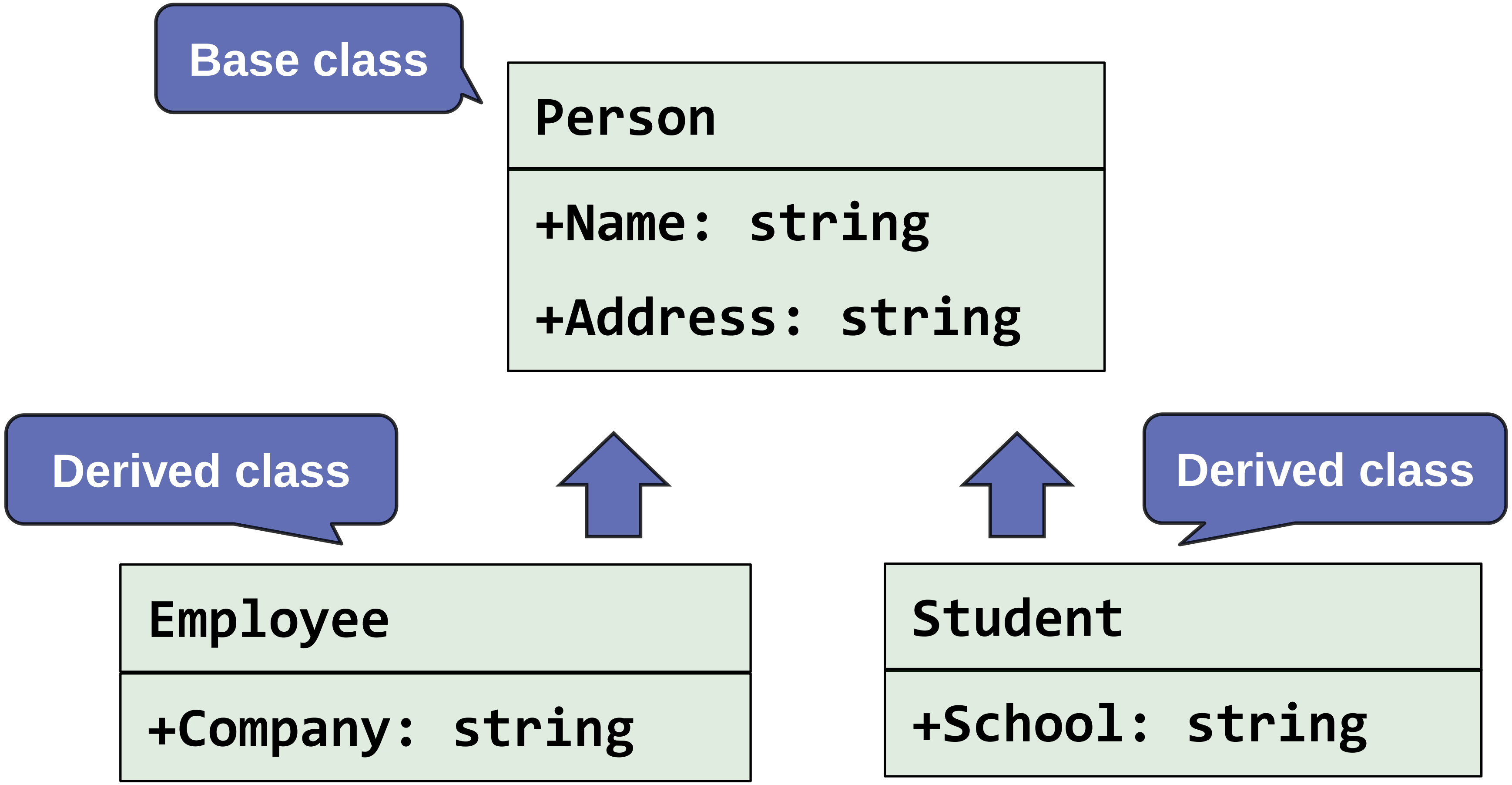
Inheritance

Inheritance

- Superclass - Parent class, Base Class
  - – The class giving its members to its child class
- Subclass - Child class, Derived class
  - – The class taking members from its base class



Inheritance – Example



Base class

Person

+Name: string

+Address: string

Derived class

Employee

+Company: string

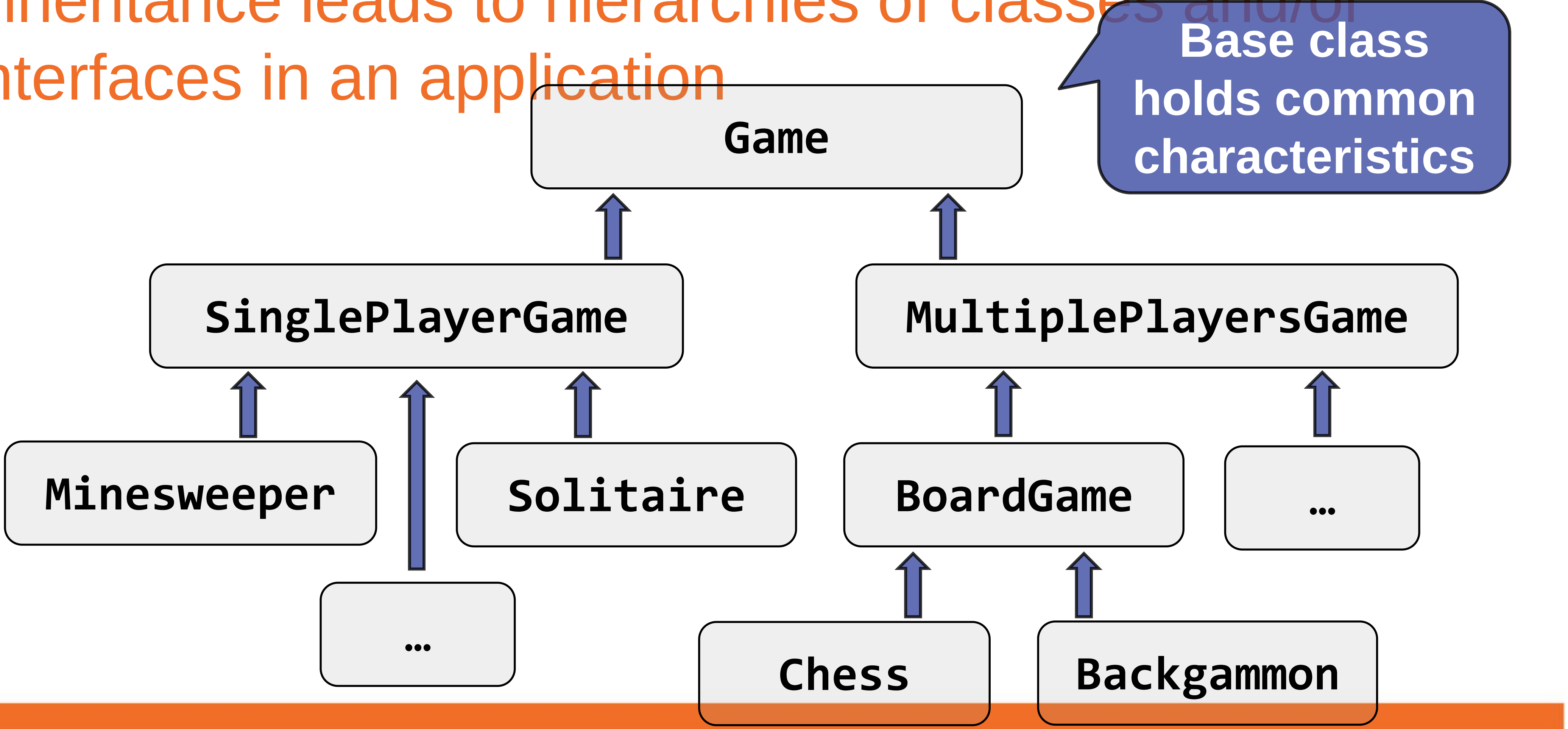
Derived class

Student

+School: string

Class Hierarchies

- Inheritance leads to hierarchies of classes and/or interfaces in an application

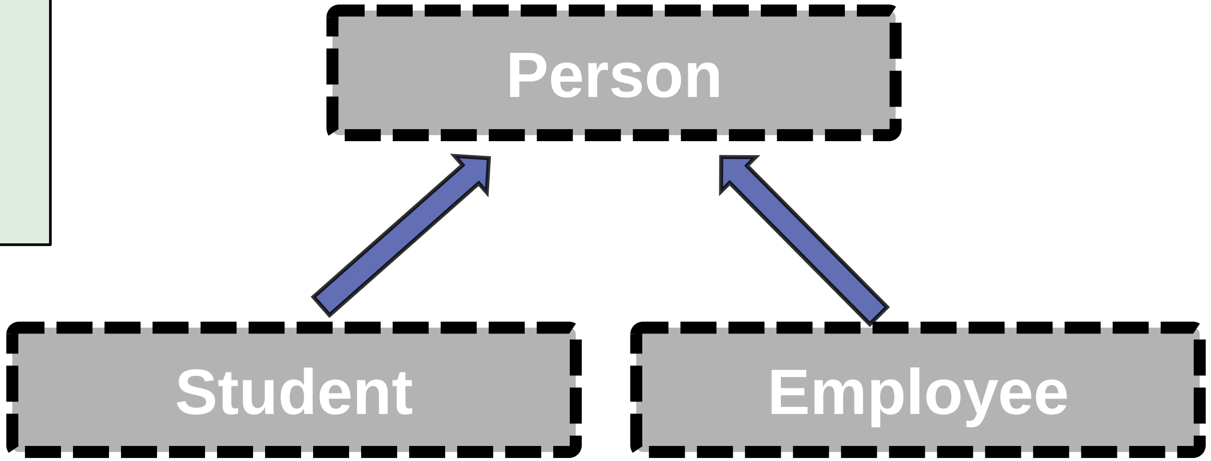


Base class holds common characteristics

Inheritance in C#

- In C# inheritance is defined by the : operator

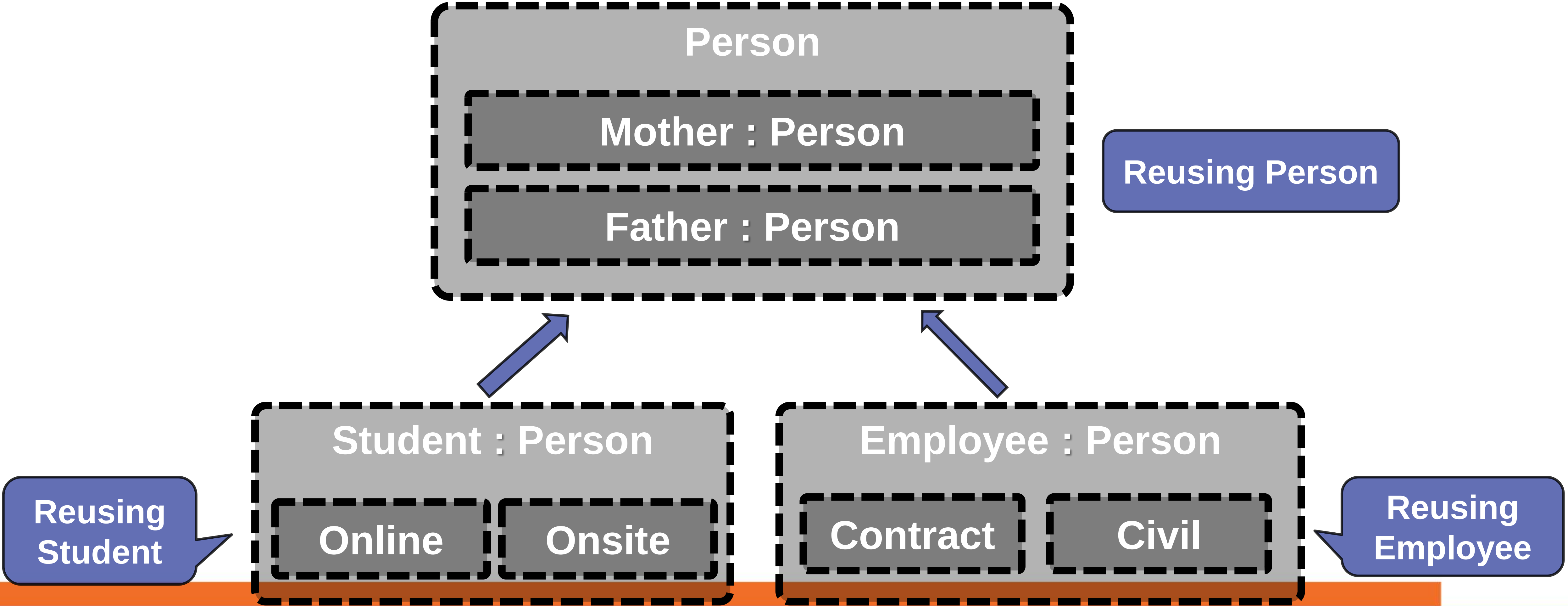
```
class Person { … }

class Student : Person { … }

class Employee : Person { … }
```



Student : Person

Inheritance – Derived Class

- Derived classes take all members from base classes



Using Inherited Members

- You can access inherited members as usual

```
class Person { public void Sleep() { … } }
class Student : Person { … }
class Employee : Person { … }
```

```
Student student = new Student();
student.Sleep();
Employee employee = new Employee();
employee.Sleep();
```

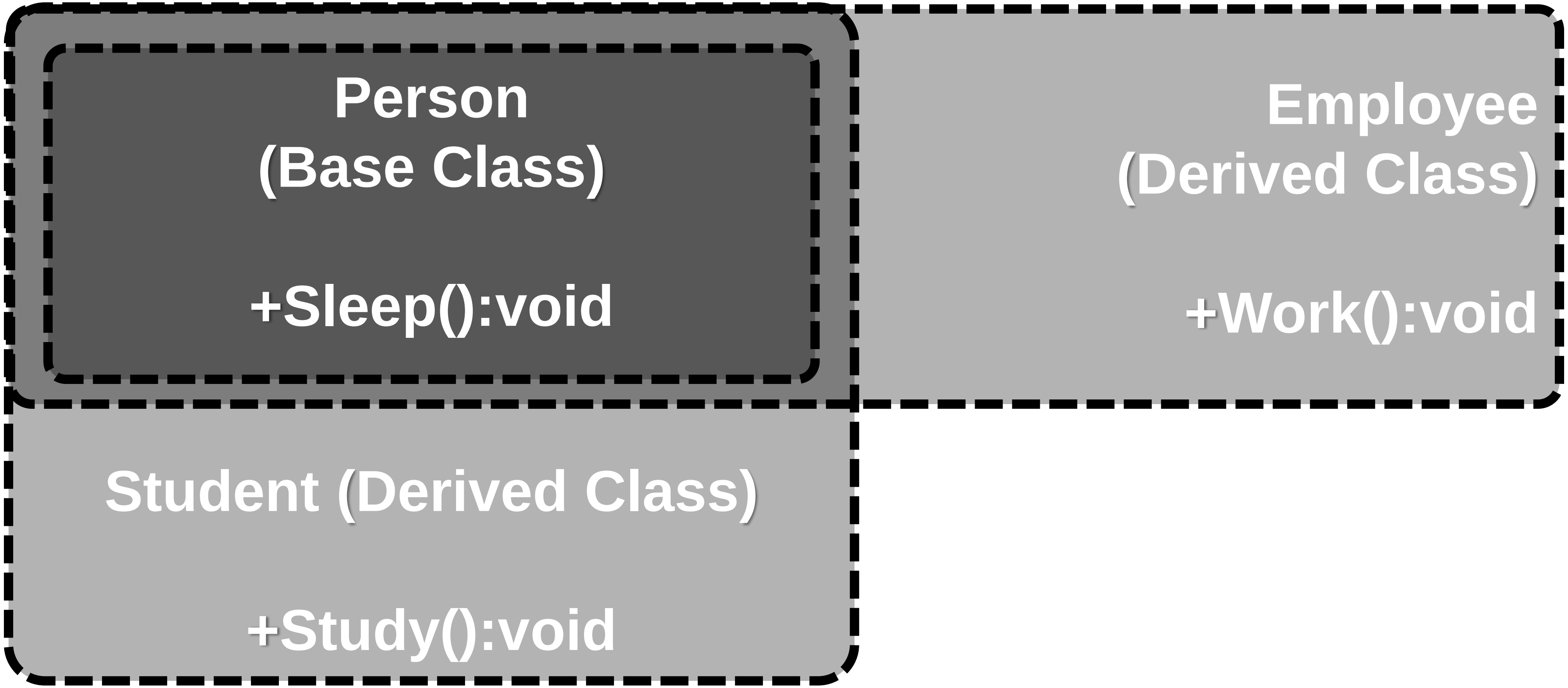
Reusing Constructors

- Constructors are not inherited
- They can be reused by the child classes

```
class Student : Person {
private School school;
  public Student(string name, School school)
    :base(name) {this.school = school;}
}
```

Thinking about Inheritance – Extends

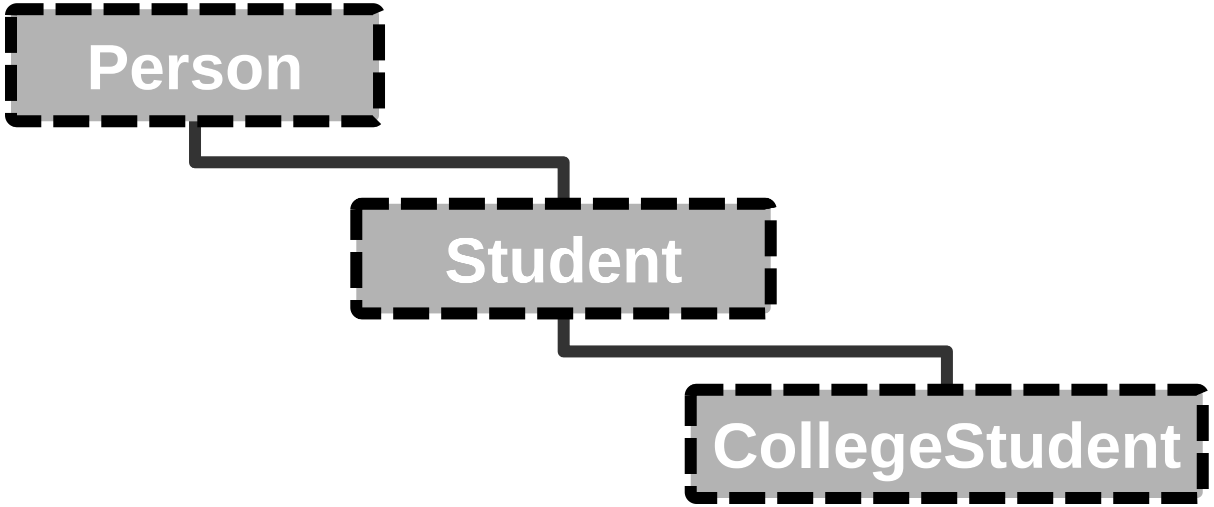
- Derived class instance contains instance of its base class



Transitive Relation

- Inheritance has a transitive relation

```
class Person { … }
class Student : Person { … }
class CollegeStudent : Student { … }
```



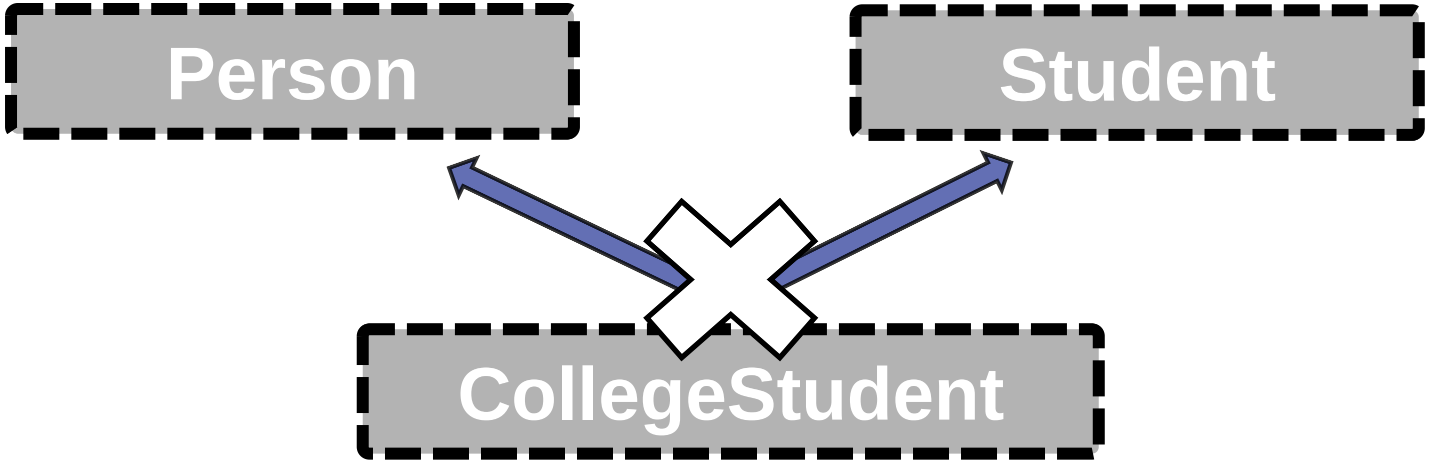
Person

Student

CollegeStudent

Multiple Inheritance

- In C# there is no multiple inheritance
- Only multiple interfaces can be implemented



UNIVERSITY of GREENWICH

Alliance with FPT Education

ENCAPSULATION

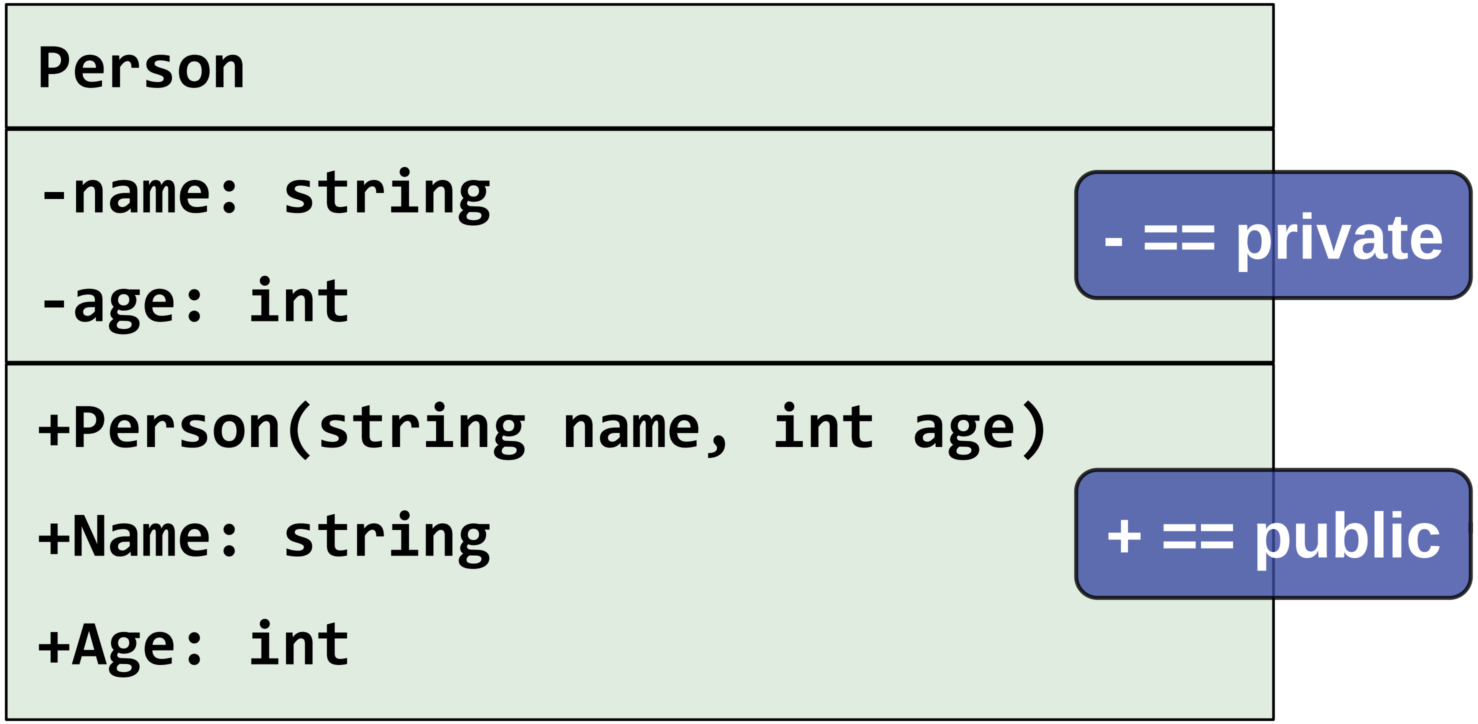
Hiding Implementation

Encapsulation

- Process of wrapping code and data together into a single unit
- Flexibility and extensibility of the code
- Reduces complexity
- Structural changes remain local
- Allows validation and data binding

Encapsulation – Example

- Fields should be



Person

-name: string

-age: int

+Person(string name, int age)

+Name: string

+Age: int

- == private

+ == public

- Properties should be

Keyword This

- Reference to the current object
- Refers to the current instance of the class
- Can be passed as a parameter to other methods
- Can be returned from method
- Can invoke current class methods

VISIBILITY OF CLASS MEMBERS

Access Modifiers

Private Access Modifier

- It's the main way to perform encapsulation and hide data from the outside world

```
private string name;
Person (string name) {
  this.name = name;
}
```

- The default field and method modifier is
- Avoid declaring private classes and interfaces
  - accessible only within the declared class itself

Public Access Modifier

- The most permissive access level
- There are no restrictions on accessing public members

```
public class Person {
  public string Name { get; set; }
  public int Age { get; set; }
}
```

- To access class directly from a namespace
- use the using keyword to include the namespace

Internal Access Modifier

- internal is the default class access modifier

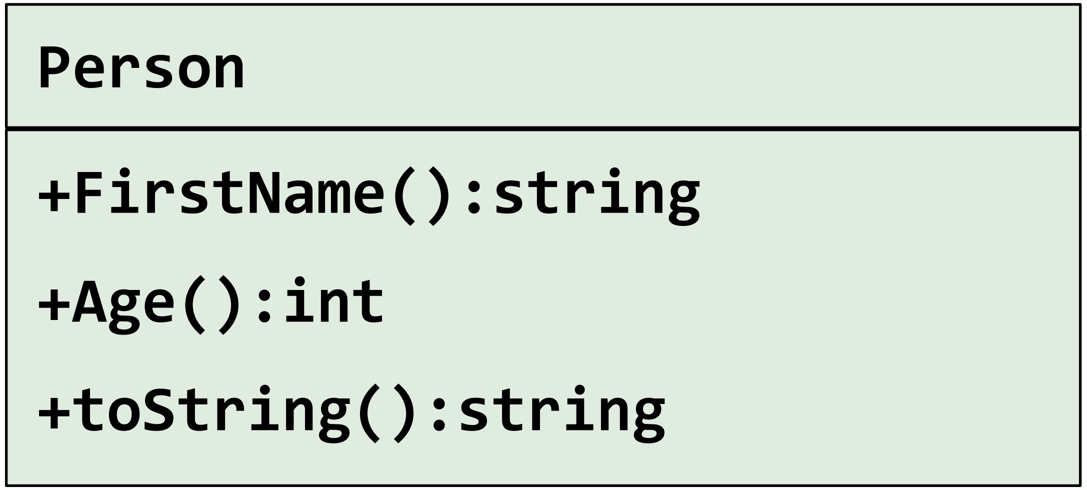
```
class Person {
   internal string Name { get; set; }
   internal int Age { get; set; }
}
```

- Accessible to any other class in the same project

```
Team rm = new Team("Real");
rm.Name = "Real Madrid";
```

Problem: Sort Persons by Name and Age

- Create a class Person



Person

+FirstName():string

+Age():int

+toString():string

Solution: Sort Persons by Name and Age

```
public class Person {
  // TODO: Add a constructor
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public int Age { get; private set; }
  public override string ToString() {
    return $"{FirstName} {LastName} is {Age} years old.";
  }
}
```

Solution: Sort Persons by Name and Age (2)

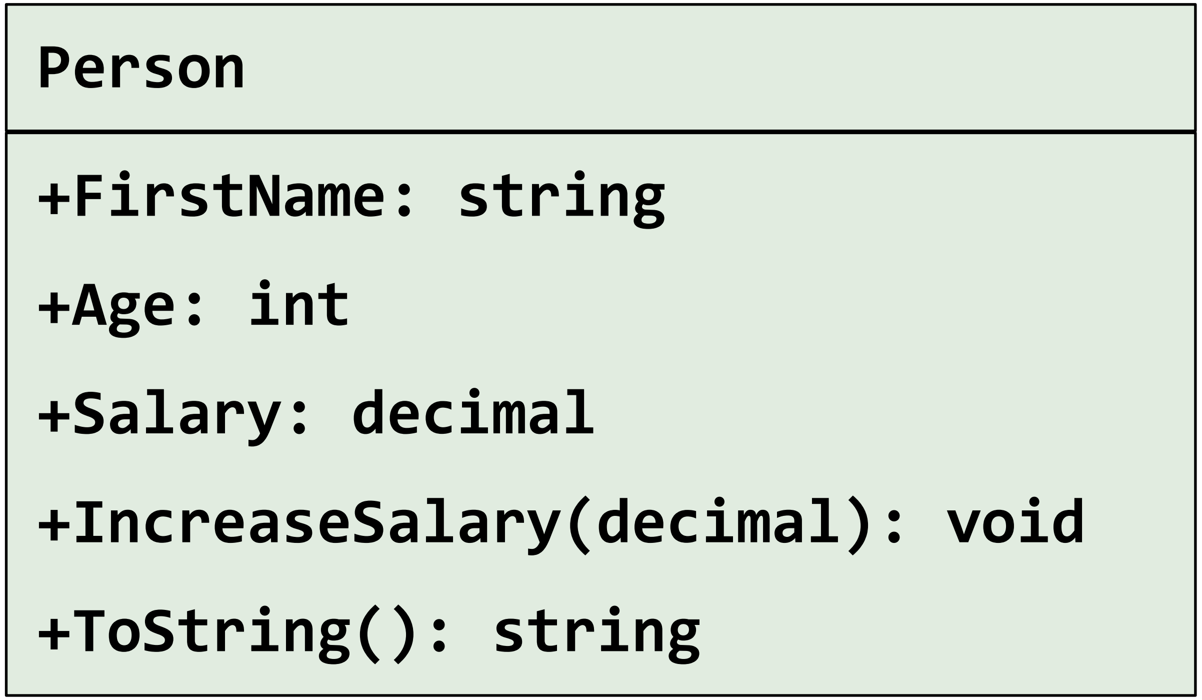
```
var lines = int.Parse(Console.ReadLine());
var people = new List<Person>();
for (int i = 0; i < lines; i++) {
  var cmdArgs = Console.ReadLine().Split();
  // Create variables for constructor parameters
  // Initialize a Person
  // Add it to the list
}
```

Solution: Sort Persons by Name and Age (3)

```
var sorted = people.OrderBy(p => p.FirstName)
  .ThenBy(p => p.Age).ToList();

Console.WriteLine(string.Join(
  Environment.NewLine, sorted));
```

Problem: Salary Increase

- Expand Person with salary
- Add getter for salary
- Add a method, which updates salary with a given percent
- Persons younger than 30 get half of the normal increase



Person

+FirstName: string

+Age: int

+Salary: decimal

+IncreaseSalary(decimal): void

+ToString(): string

Solution: Salary Increase

```
public decimal Salary { get; private set; }
public void IncreaseSalary(decimal percentage)
{
   if (this.Age > 30)
     this.Salary += this.Salary * percentage / 100;
   else
     this.Salary += this.Salary * percentage / 200;
}
```