

Inheritance

OOP Principles, Inheritance



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

sli.do

#cpp-oop

1. OOP Principles
2. Class Hierarchies
3. Inheritance
4. Accessing Members of the Base Class
5. Types of Class Reuse
 - Extension, Composition, Delegation
6. When to Use Inheritance



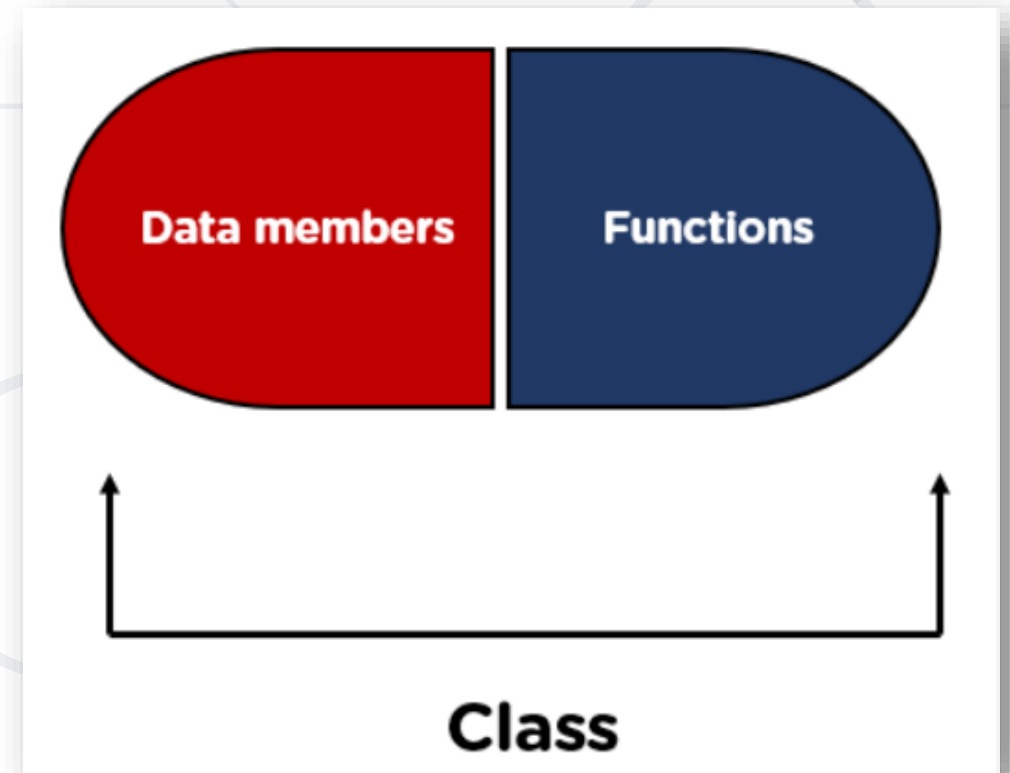
A background network diagram consisting of a grid of light gray lines intersecting at various points. Some of these intersections are marked with small, empty light gray circles. A larger, solid dark blue circle is centered in the upper half of the image, containing the text 'OOP' in white.

OOP

OOP Principles

Encapsulation, Inheritance, Polymorphism, Abstraction

- In general, encapsulation is a **process of wrapping similar code in one place**
- In OOP, encapsulation is **combining member functions and data members** in a single unit called a class and **restricting direct access to some components**



- Classes have internal state (**vector**'s capacity)
 - **private** / **protected** – state inaccessible to outside code
 - **public** – outside code interacts with object state through public members

```
class IntArray
{
private:
    int* data; int size;
public:
    IntArray(int size) : data(new int[size]), size(size) {}
    ~IntArray() { delete[] this->data; }
    ...
};
```

Can't be modified from the outside, so the class can:

- assume last index in data is **size-1**
- rename **size** to **length** without checking for outside usages

- Abstraction – using base virtual members
 - So allowing any class with **overrides** for them
- **ostream& operator<<(ostream& out, const Person& p)**
 - Allows any **ostream** – **ostringstream**, **ofstream**, **cout**

```
void stopIfOverLimit(Vehicle* v, double limit)
{
    if (v->getSpeed() > limit)
    {
        v->stop();
    }
}
```

- **Derived** classes **inherit** a **base** class to reuse its members

```
class Vehicle { private: double speed;  
public: Vehicle(double speed) : speed(speed) {}  
        void setSpeed(double speed) { this->speed = speed; }  
};
```

```
class Car : public Vehicle {  
private: bool parkingBrakeOn;  
public:  
Car(double spd, bool park) : Vehicle(spd), parkingBrakeOn(park) {}  
};
```

```
class Airplane : public Vehicle {  
private: double altitude;  
public:  
Airplane(double spd, double alt) : Vehicle(spd), altitude(alt) {}  
};
```


Inheritance

```
class Vehicle {  
    private:  
        double speed;  
    public:  
        Vehicle(double speed) : speed(speed) {}  
        void setSpeed(double speed) { this->speed = speed; }  
};
```

Base class

```
class Car : public Vehicle  
{  
    private:  
        bool parkingBrakeOn;  
    public:  
        Car(double spd, bool park)  
        : Vehicle(spd),  
        parkingBrakeOn(park) {}  
};
```

Derived class

```
class Airplane : public Vehicle  
{  
    private:  
        double altitude;  
    public:  
        Airplane(double spd, double alt) :  
        Vehicle(spd), altitude(alt) {}  
};
```

Derived class

- Base class can have **virtual** members
 - Derived classes **override** them to have different behavior

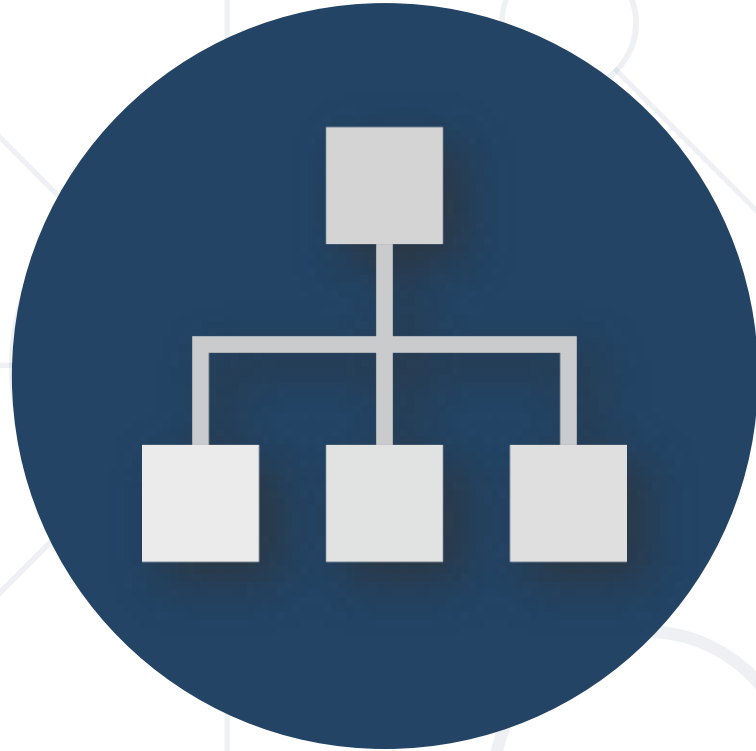
```
class Vehicle { ...  
    virtual void stop() { this->speed = 0; }  
};
```

```
class Car : public Vehicle override {  
    ...  
    virtual void stop() override {  
        Vehicle::stop();  
        this->parkingBrakeOn = true;  
    }  
};
```

```
class Airplane : public Vehicle override {  
    ...  
    virtual void stop() override {  
        Vehicle::stop();  
        this->altitude = 0;  
    }  
};
```

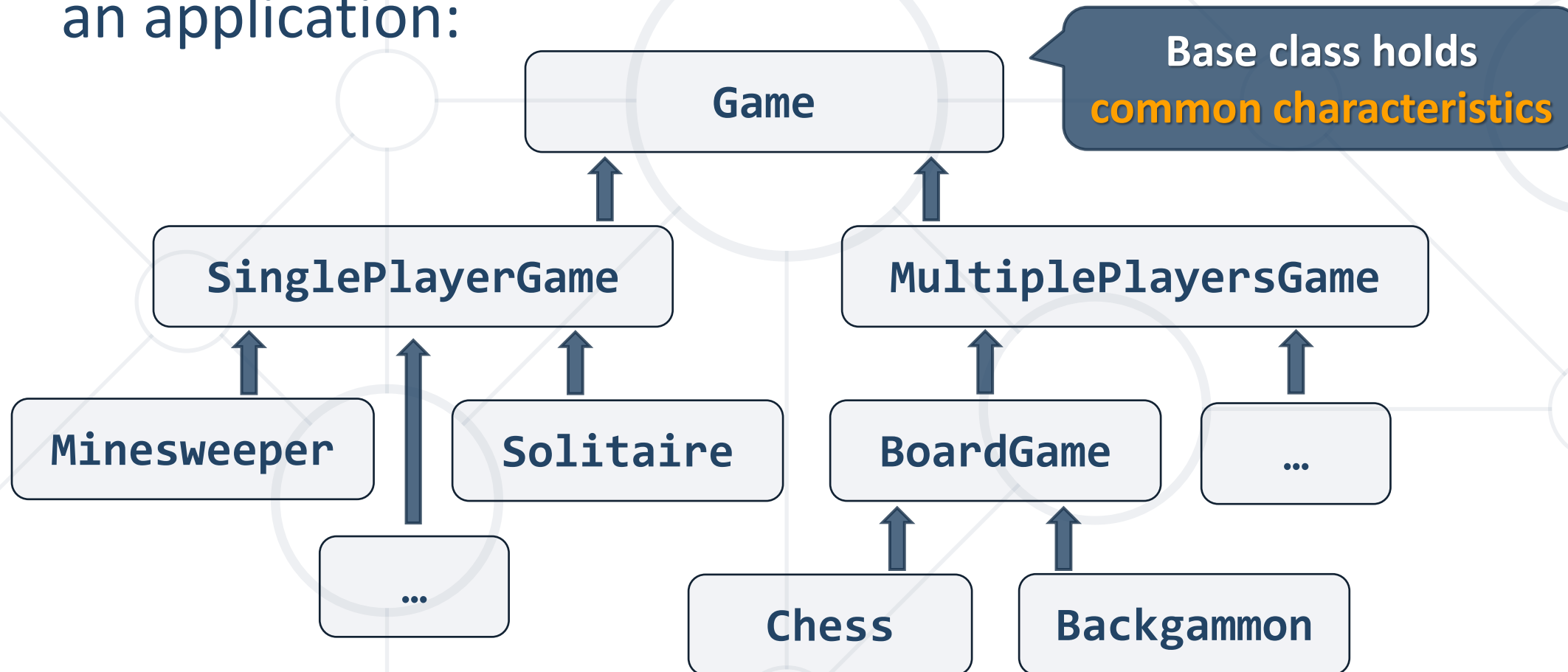
- Base class pointers / references can point to any derived class object
 - Normal members call **base** class member
 - **virtual** members call **override** member in **derived**

```
std::vector<Vehicle*> vehicles
{
    new Car(90, false),
    new Airplane(700, 10000, 242),
    new Car(0, true)
};
vehicles[0]->stop(); // calls Car::stop()
vehicles[1]->stop(); // calls Airplane::stop()
vehicles[2]->stop(); // calls Car::stop()
```

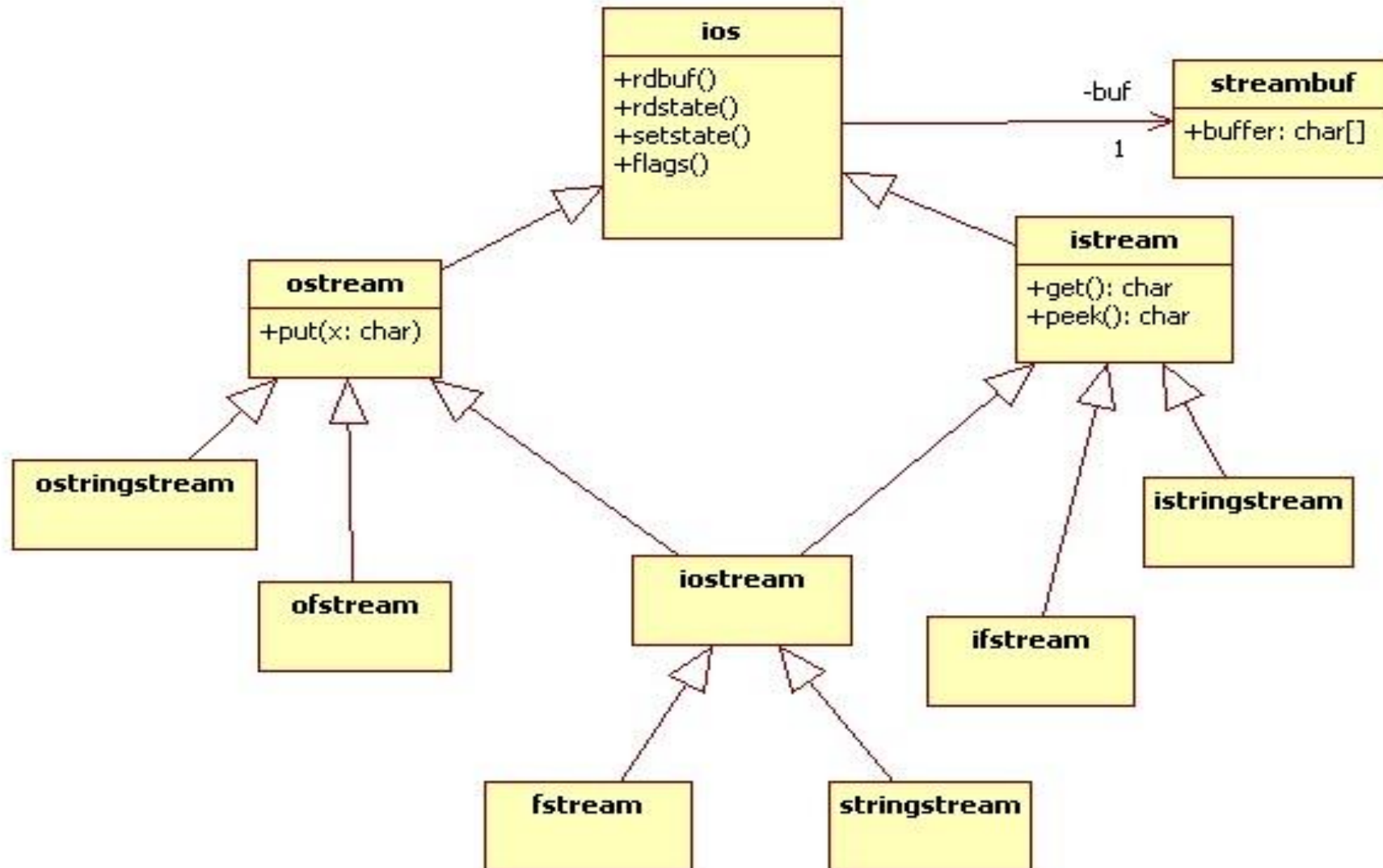


Class Hierarchies

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

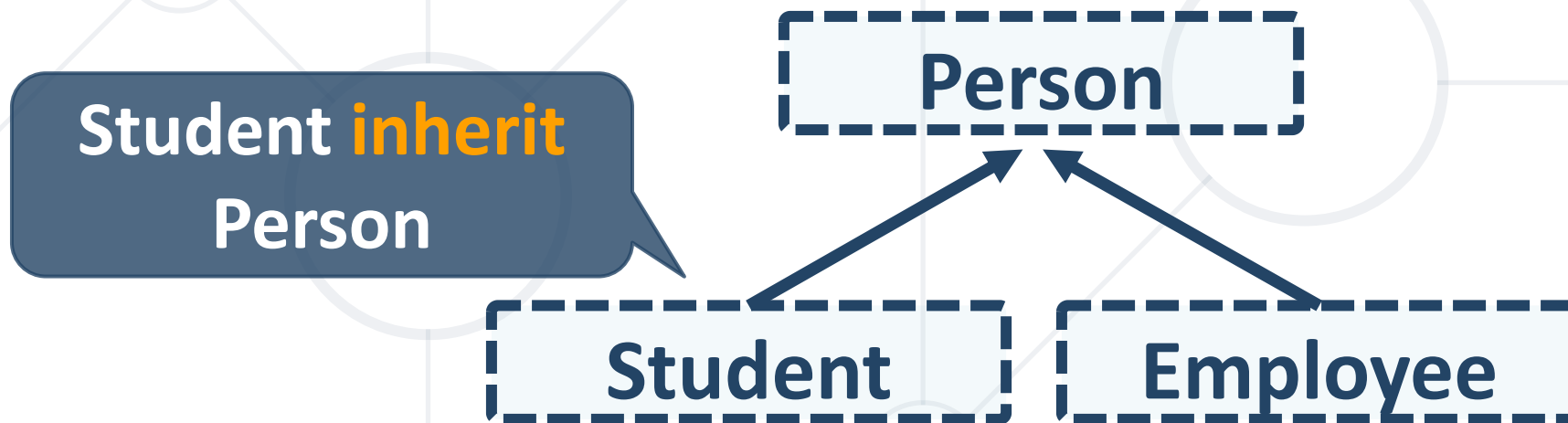


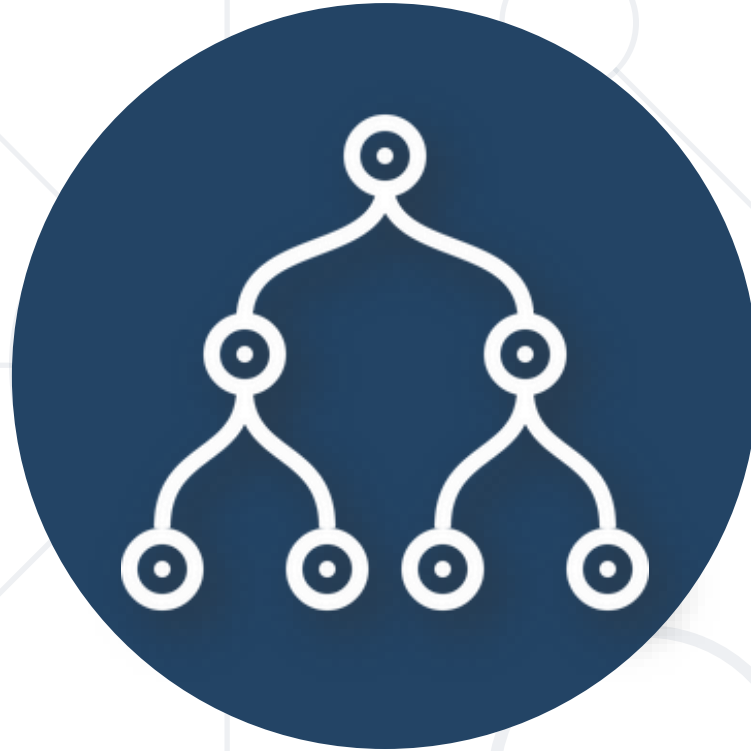
Class Hierarchies



- C++ supports inheritance through **access modifier**

```
class Person { ... }  
  
class Student : public Person { ... }  
class Employee : public Person { ... }
```





Inheritance

Syntax, Protected Members, Accessing Base

- Code **reuse patterns**:
 - Repeated code -> extract **function**
 - Functions using similar parameters / globals -> extract **class**
 - Repeated members in multiple classes -> extract **base class**
- Inheritance – sharing member definitions
 - A class **declares / defines** members
 - Other classes inherit it – get all members of inherited class

- `class Derived : access-modifier Base { ... }`
 - **access-modifier** – one of `public/protected/private`
- Members of **Base** class added to **Derived** class
 - Access limited to inheritance access-modifier
 - `public`: doesn't change **Base** modifiers
 - `protected`: `public` from **Base** -> `protected` in **Derived**
 - `private`: any from **Base** -> `private` in **Derived**

Inheritance – Extracting Base Class

- Extract common members into a **base class**

```
class Vehicle
{
public: double speed;
};
```

Can't use initializer-list
for base class field


```
class Car : public Vehicle
{
    bool parkingBrakeOn;

public:
    Car(double speed, bool parked)
        : parkingBrakeOn(parked)
    {
        this->speed = speed;
    }
};
```

```
class Airplane : public Vehicle
{
    double altitude;
    double heading;
public:
    Airplane(double spd, double alt, double hdg)
        : altitude(alt), heading(hdg)
    {
        this->speed = spd;
    }
};
```

Share Access with Derived – protected

- **public speed** – breaking encapsulation
 - Can't use **private**, because we lose access to **speed**
- **protected** members – accessible to inheriting class



```
class Vehicle
{
protected:
    double speed;
};
```

```
class Car : public Vehicle
{ ...
public:
    Car(...) { this->speed = speed; }
};
```

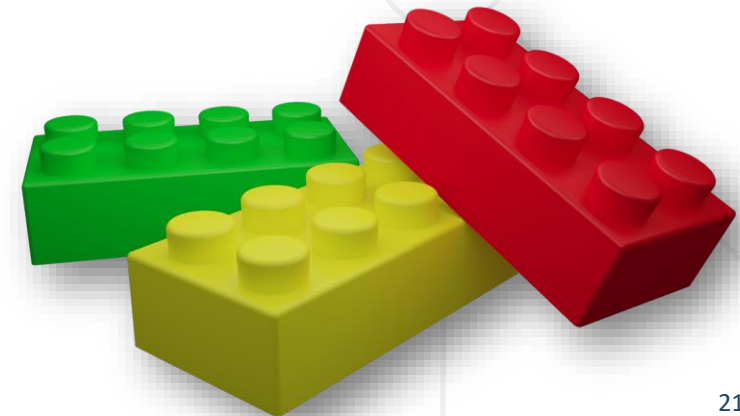
```
Car car(90, false);
cout << car.speed << std::endl;
// compilation error
```

Can't be used outside of
the class hierarchy

Using Base Constructors

- Inheriting class can call **base** constructor
 - In initializer list, like field, BUT with base class name
- **Syntax**

```
Derived(...) : Base(...), ... { ... }
```



Using Base Constructors - Example

```
class Vehicle { protected:  
    double speed;  
    Vehicle(double speed) : speed(speed) {}
```

```
class Car : public Vehicle {  
    ...  
    Car(double speed, bool park)  
        : Vehicle(speed), parkingBrakeOn(park) {}
```

```
class Airplane : public Vehicle {  
    ...  
    Airplane(double s, double a, double h)  
        : Vehicle(s), altitude(a), heading(h) {}
```

- Methods are inherited just like any member
- Hiding – using same signature in **derived** as in **base**
 - E.g. **base** has **void f()**, **derived** hides with **int f()**
 - calling **f()** in **derived** calls **derived** version (same for objects)
- Explicit access to base member (field/method/...)
 - Prefix member with **base** class name and **operator::**
 - E.g. **Base::f()** calls **f()** of inherited class **Base**

Example: Hiding & Calling Base Methods

- Example: Let's make a **toString()** for **Vehicle**
 - Reuse it in **Car's toString()**

```
class Vehicle {  
  
    ...  
    string toString() const {  
        ostreamstream stream;  
        stream << "speed: "  
            << this->speed;  
        return stream.str();  
    }  
}
```

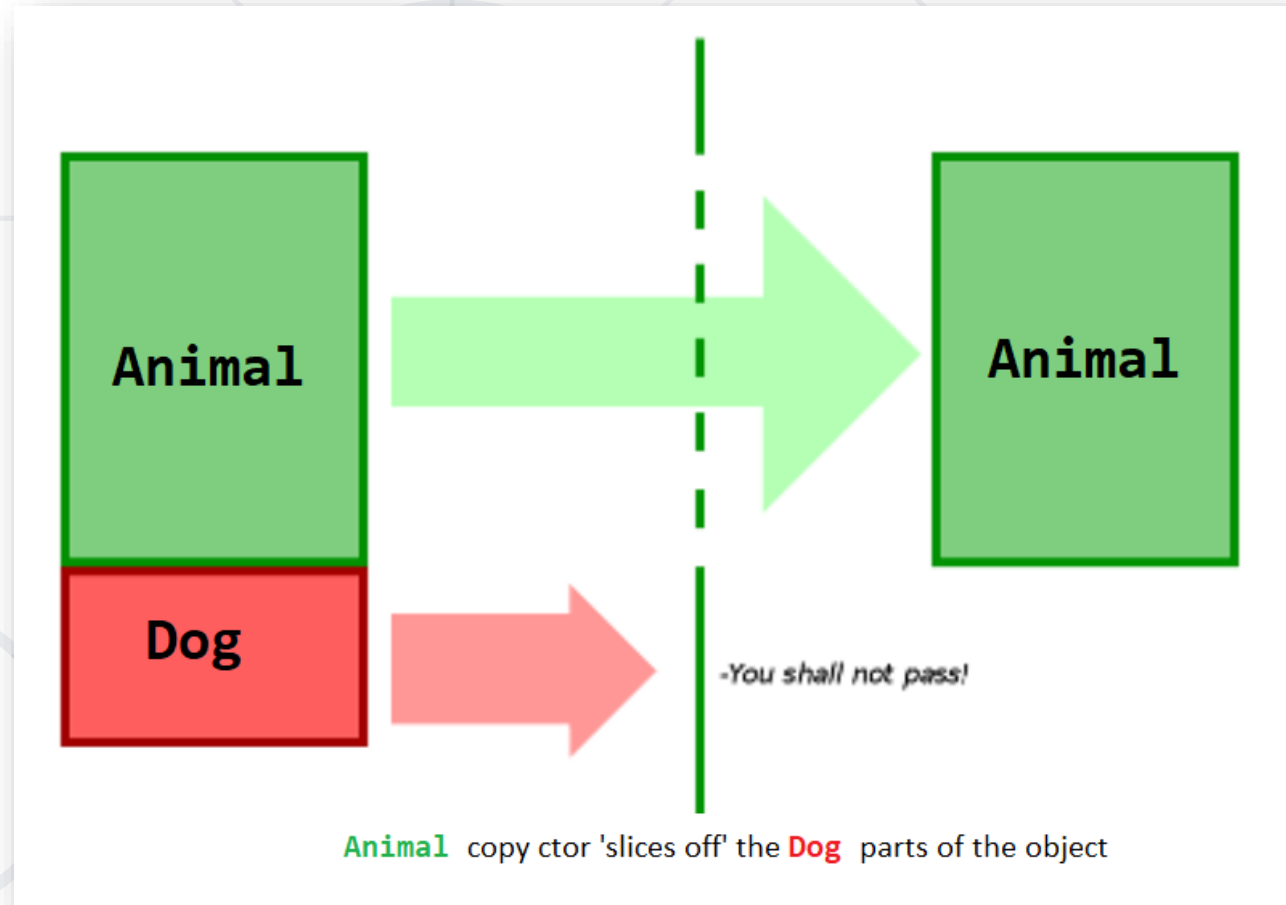
```
class Car {  
    ...  
    string toString() const {  
        ostreamstream stream;  
        stream << Vehicle::toString()  
            << " parking brake: "  
            << this->parkingBrakeOn;  
        return stream.str();  
    }  
}
```


Object Slicing

- Happens when we assign an **object of a derived class** to an **object of a base class**
 - Implicit cast, called **upcasting**
 - Fields from derived object are "sliced off"
 - Should generally be avoided
- **Base x = Derived();**
 - **Base** class copy constructor is called
 - **Derived** part of the object is lost
 - **x** contains only **Base** fields



C++ Object Slicing



- If a **base** has no default constructor
 - **Derived** must define constructor calling the **base** constructor
- Assignment operator is always hidden in a **derived** class
 - Signature not the same, but implicitly the same as base (**upcast**)
- Constructors aren't inherited – can't be used externally
 - Only used internally in initializer list
 - This also applies to copy/move constructors

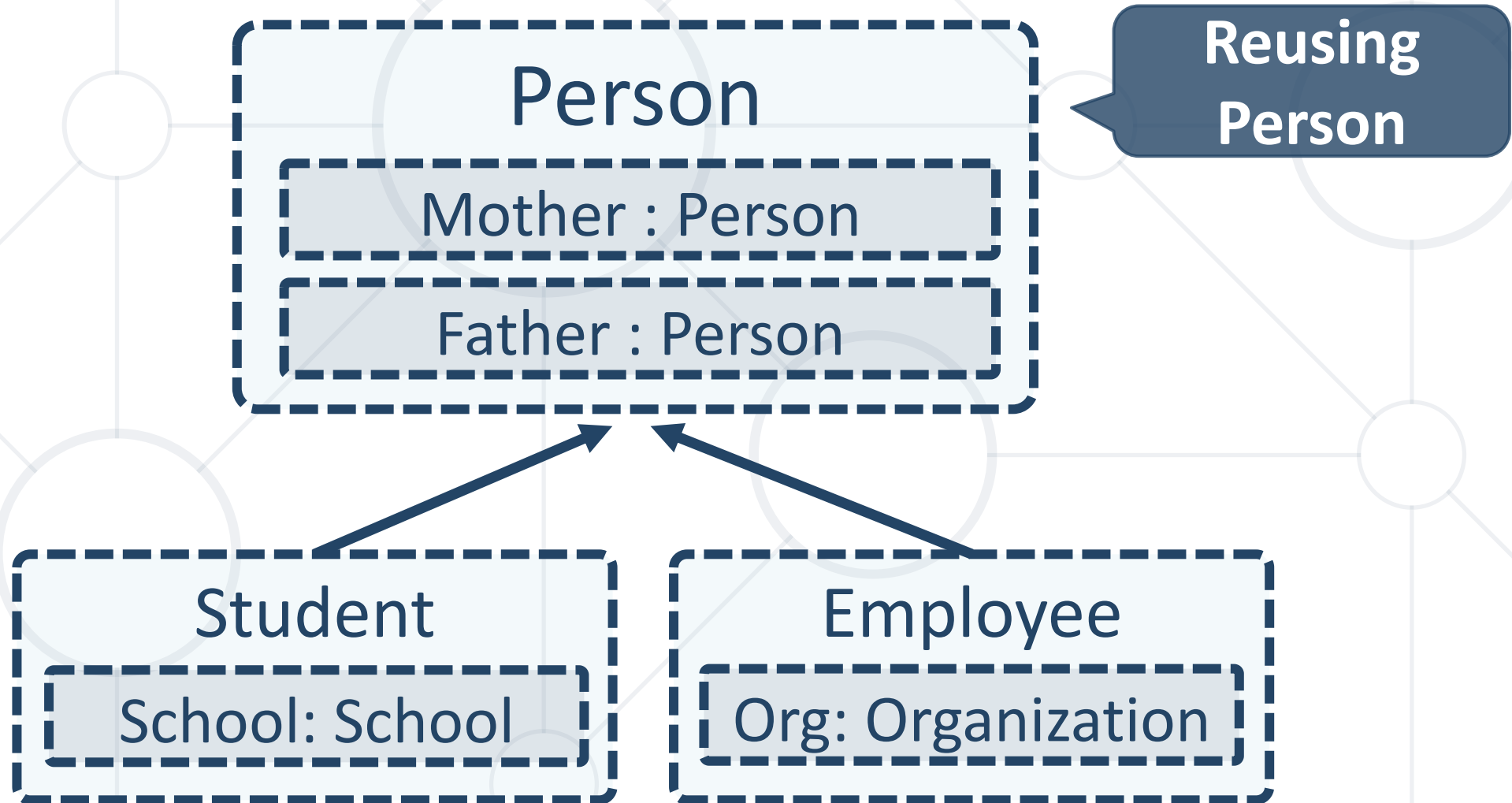
- **Base** pointers / references can point to **derived** objects
 - **upcast**, NO slicing – not fitting larger into smaller object
 - **Derived d; Base* p = &d;**
 - **Base* p = new Derived(); ...**
- Accesses base members, regardless of hiding

```
Airplane plane(510, 2400, 90);  
Vehicle* v = &plane;  
cout << v->toString() << endl; // calls Vehicle::toString()
```

- Unless members are **virtual overrides**
(will be covered in the next lecture - Polymorphism)

Inheritance – Derived Class

- Class **taking all members** from another class

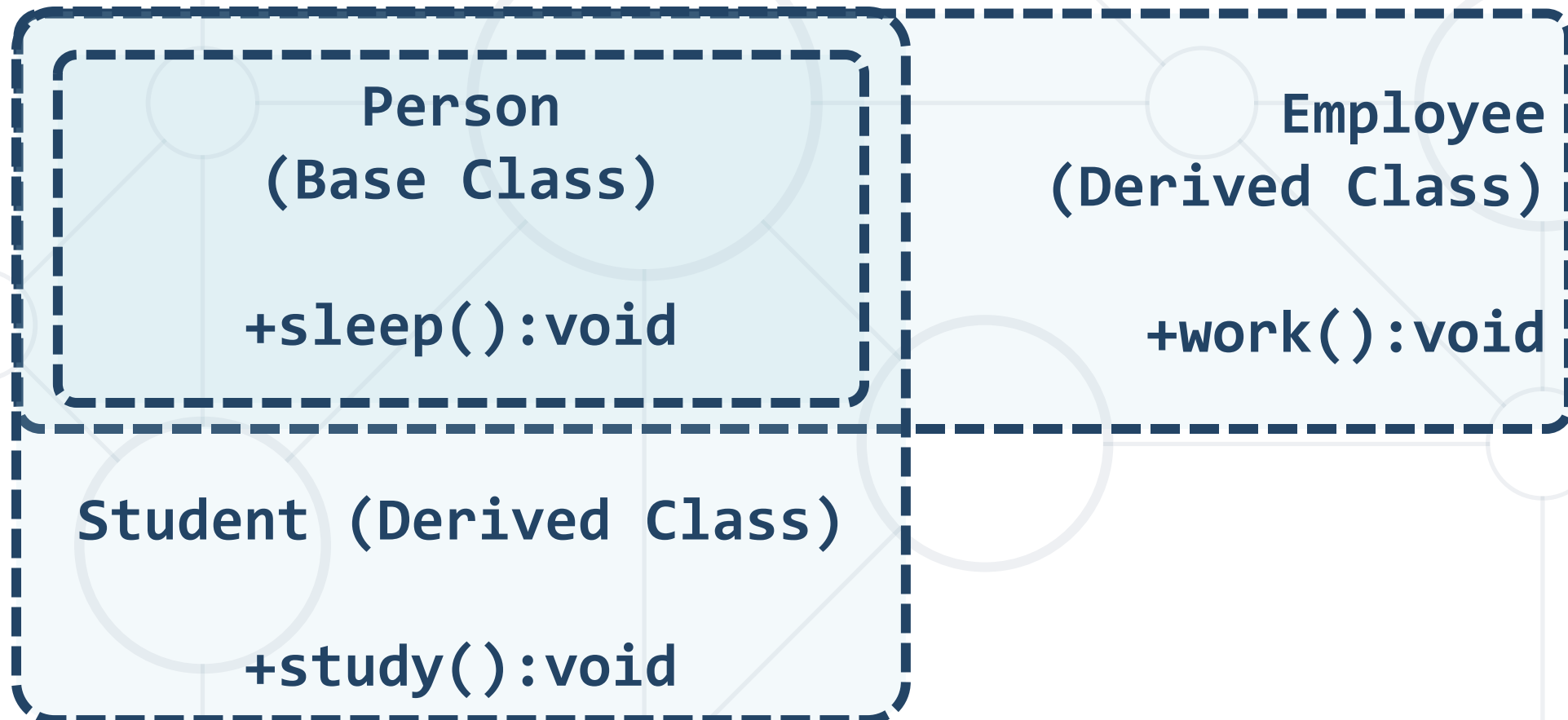


Using Inherited Members

```
class Person { public void sleep() { ... } }  
class Student :public Person { ... }  
class Employee :public Person { ... }
```

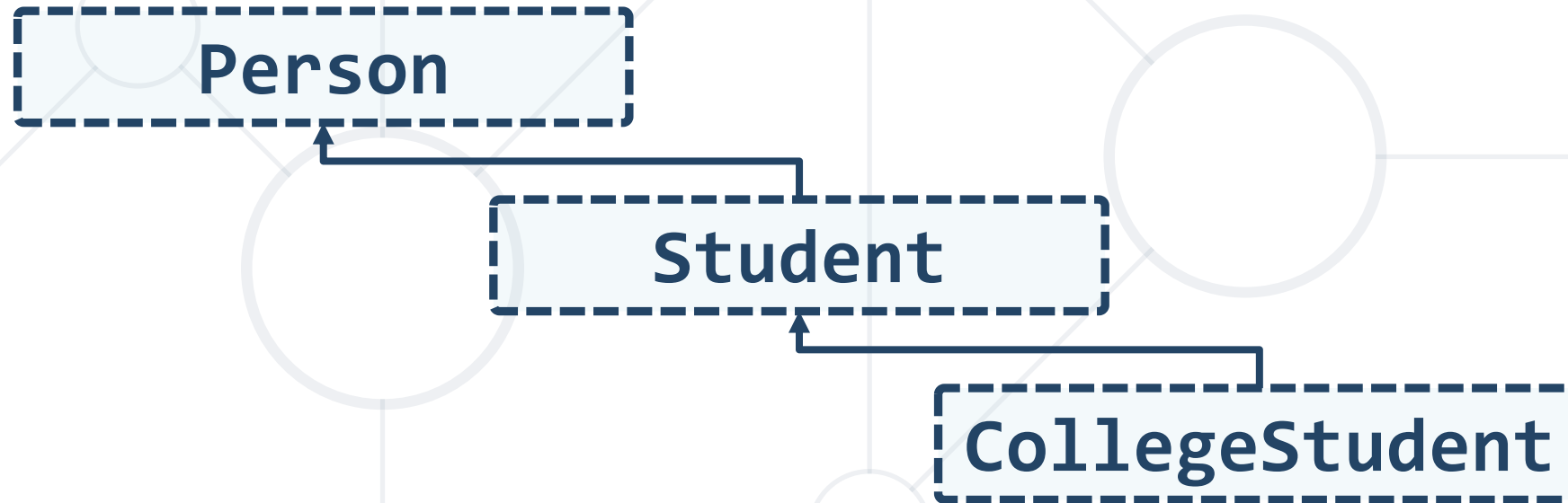
```
Student student;  
student.sleep();  
Employee* employee = new Employee();  
employee->sleep();
```

- A derived class instance **contains** an instance of its base class



- Inheritance has a **transitive relation**

```
class Person { ... }  
class Student :public Person { ... }  
class CollegeStudent :public Student { ... }
```



- Inheriting from a final classes is forbidden

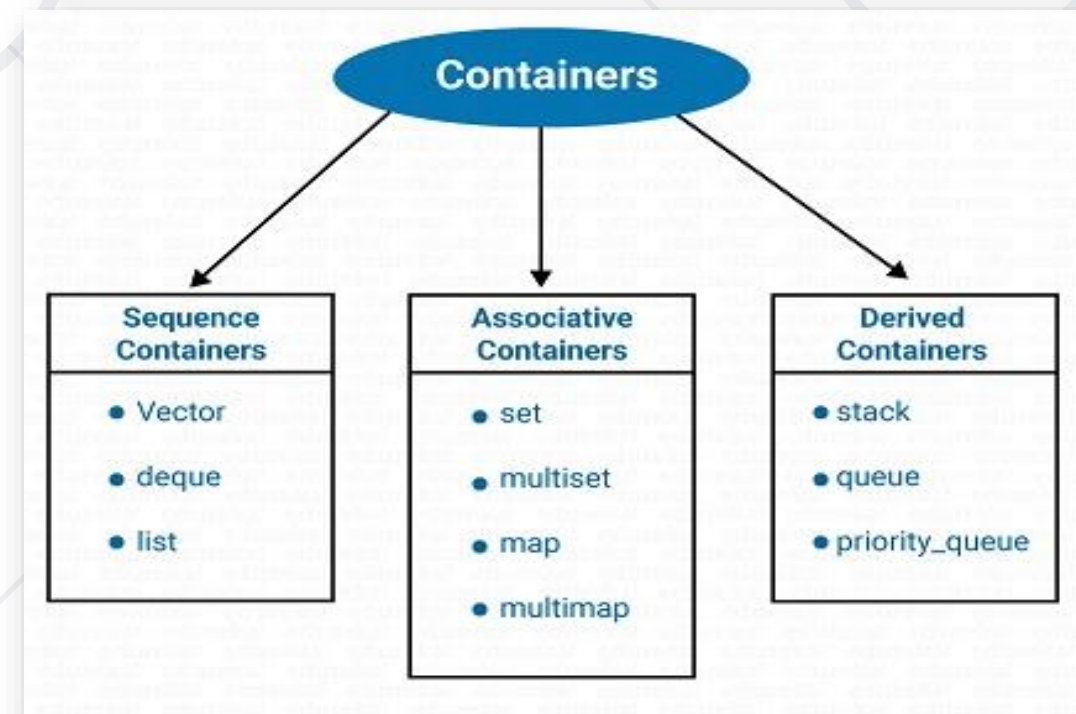
```
class Animal final {  
    ...  
}
```

```
public class Dog :public Animal { } // Error...
```



Reusing Classes

- **Duplicate code** is error prone
- **Reuse classes** through an **extension**
- Sometimes the only way



- Using classes to **define classes**

```
class Laptop {  
    Monitor monitor;  
    Touchpad touchpad;  
    Keyboard keyboard;  
    ...  
}
```

Reusing classes




```
class Laptop {  
    Monitor monitor;  
    void incrBrightness() {  
        monitor.brighten();  
    }  
  
    void decrBrightness() {  
        monitor.dim();  
    }  
}
```

Laptop

Monitor

increaseBrightness()
decreaseBrightness()

- Classes share **IS-A** relationship  Too simplistic
 - A car "is a" vehicle, a person "is a" mammal
- Derived class **IS-A-SUBSTITUTE** for the base class
- Share the **same role**
- Derived class is the **same as the base class** but adds a **little bit more functionality**
- Composition **HAS-A** relationship
 - A car "has an" engine, a person "has a" name

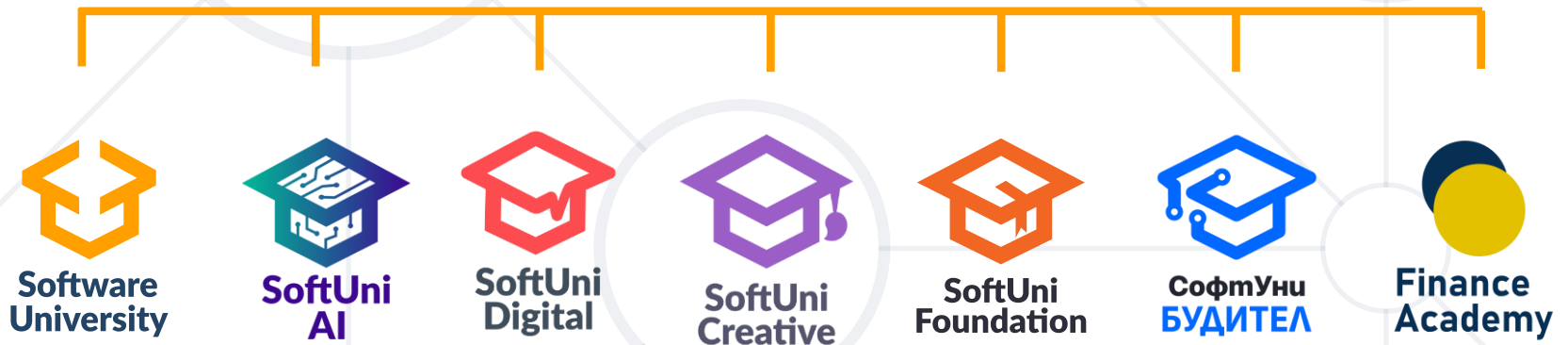
- Inheritance is a powerful tool for **code reuse**
- **Subclass inherits** members from **Superclass**
- Subclass can **override** methods
- Look for classes with the **same role**
- Look for **IS-A** and **IS-A-SUBSTITUTE** for relationship
- Consider **Composition** and **Delegation** instead
- Extract multiple-usage code into a base class



Questions?



SoftUni



SoftUni Diamond Partners



**SUPER
HOSTING
.BG**



INDEAVR
Serving the high achievers



VIVACOM

- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

