Inheritance

OOP Principles, Inheritance

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

sli.do

#cpp-oop

Encapsulation

- Classes have internal state (**vector**'s capacity)

    - **private**/**protected** – state inaccessible to outside code

    - **public** members interact with outside code, keep state correct

```
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 **data** needs a **delete[]** in destructor
- assume last index in data is **size-1**
- rename **size** to **length** without checking for outside usages

Inheritance

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

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

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

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

Polymorphism – `virtual` Members

- Base class can have **virtual** members

    - Derived classes **override** them to have different behavior

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

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

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

Polymorphism – Base Class Pointers

- Base class pointers/references can point to any derived class

  - Normal members access **base** class member

  - **virtual** members access **override** member in **derived**

```cpp
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()
```
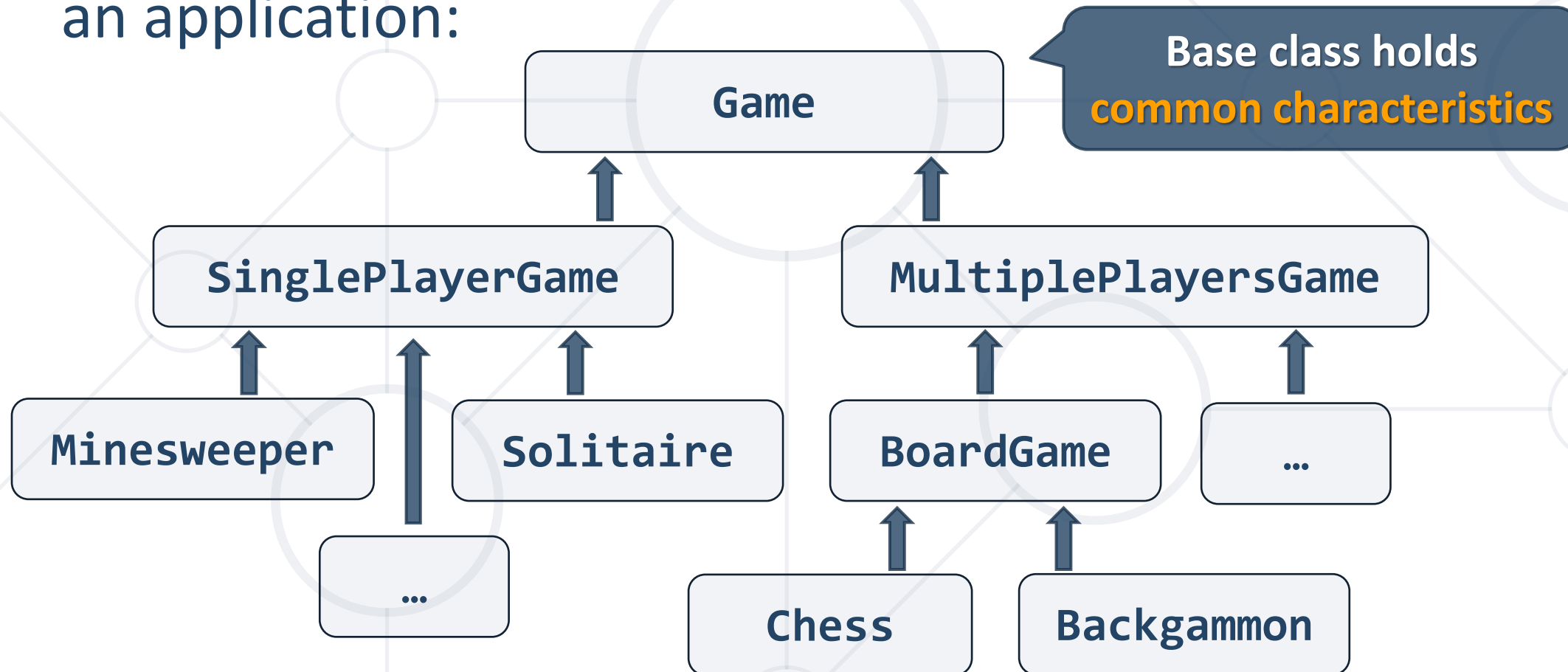
Abstraction

- Abstraction – using base virtual members

  - So allowing any class with **override**s 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();
    }
}
```
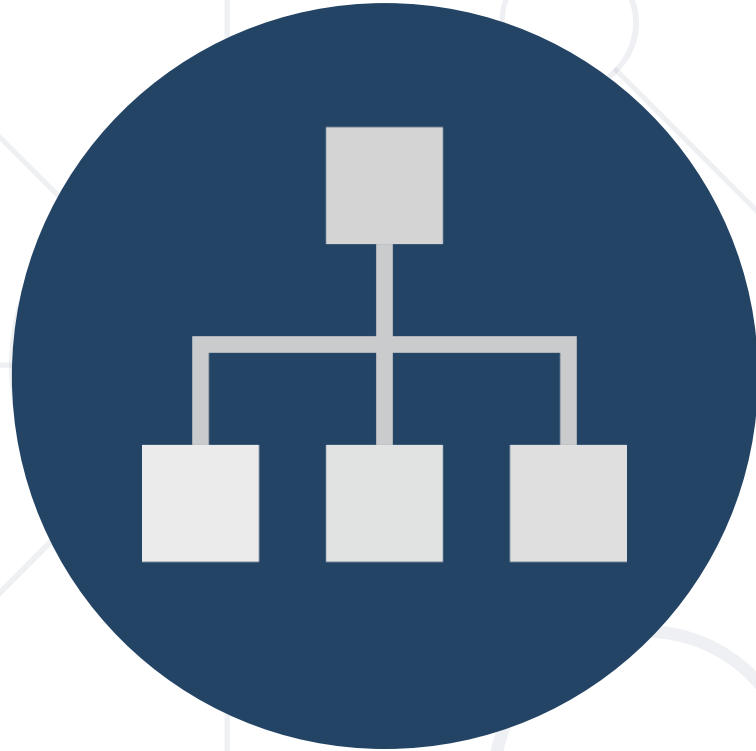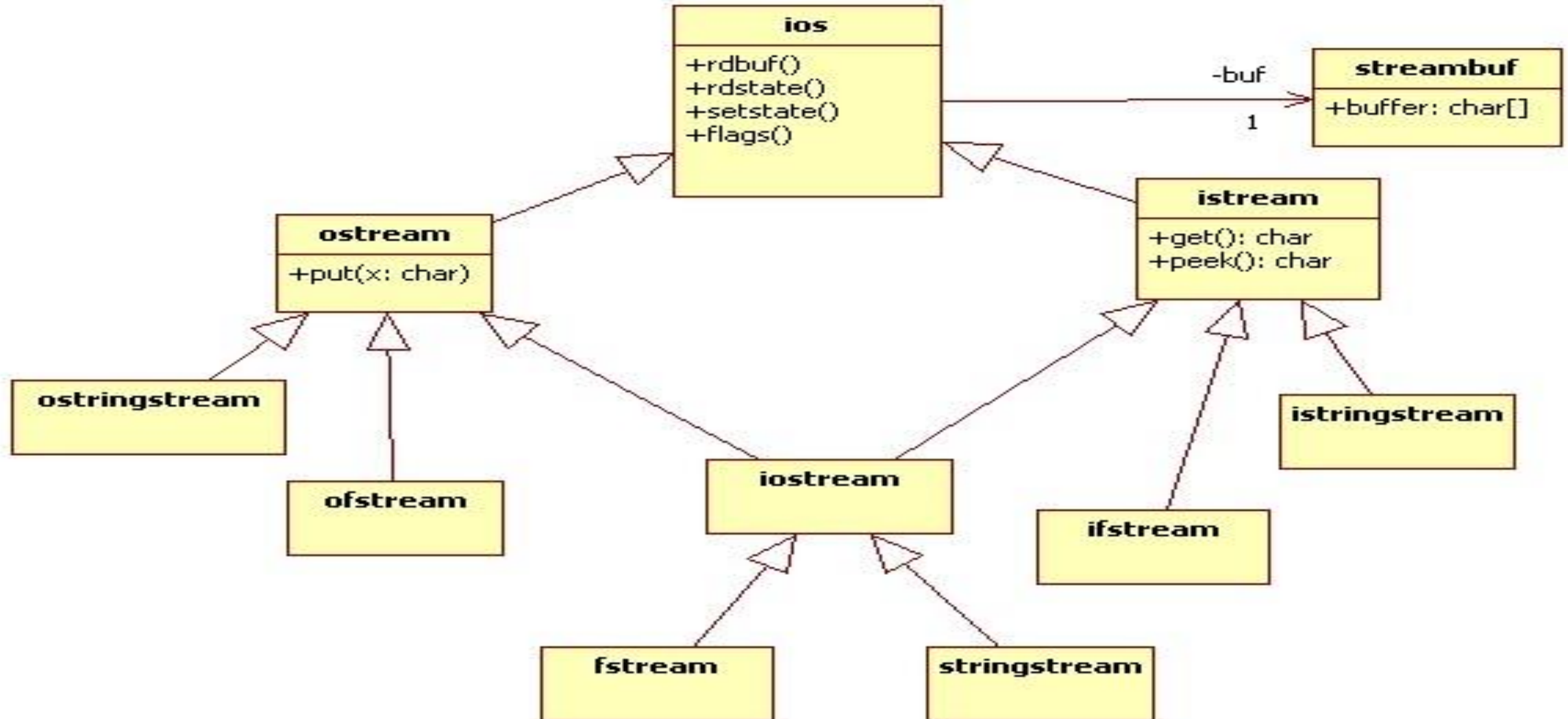
Class Hierarchies

Class Hierarchies

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

Base class holds
common characteristics

```
                            Game

    SinglePlayerGame              MultiplePlayersGame

Minesweeper    Solitaire        BoardGame        …

        …                   Chess    Backgammon
```

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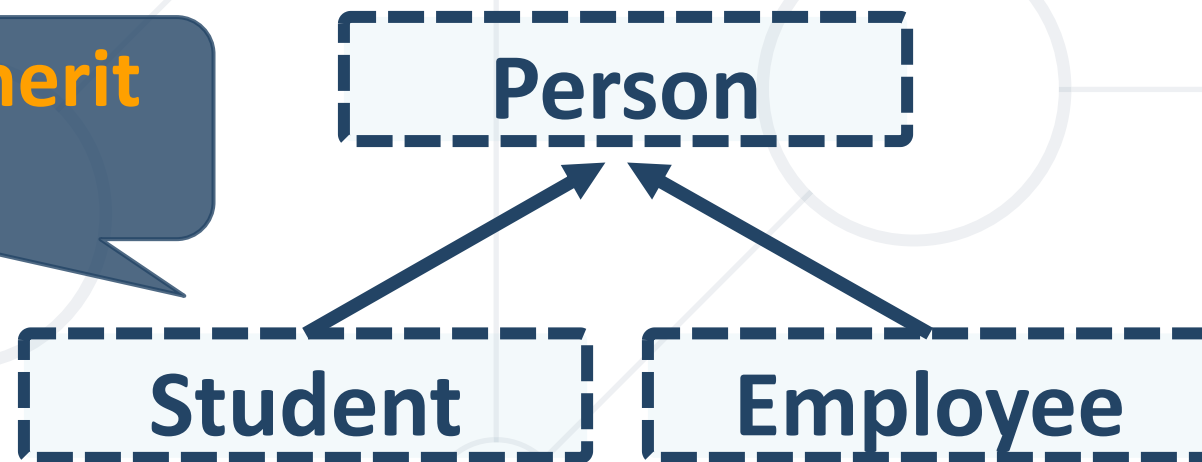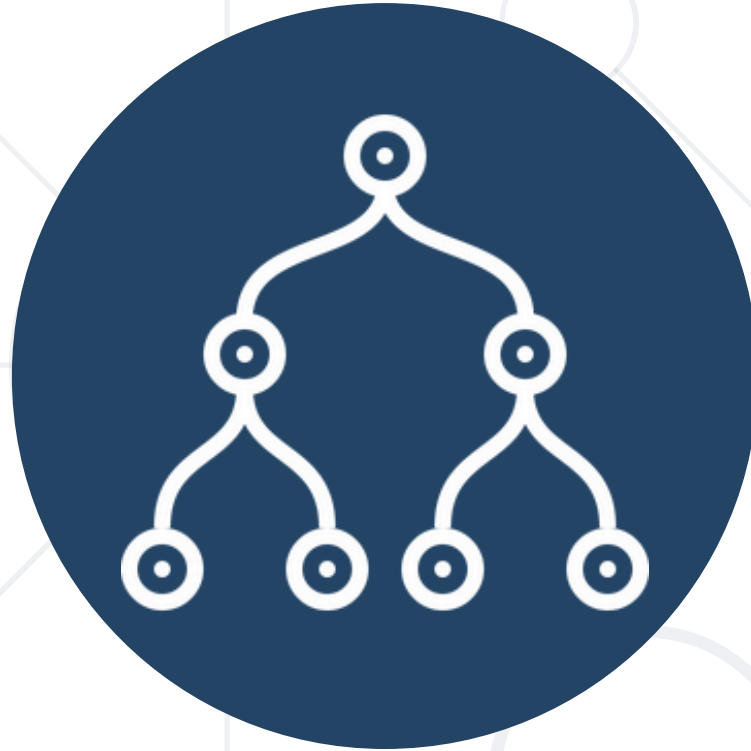
- C++ supports inheritance through **access modifier**

```
class Person { … }

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

Student inherit Person

Person

Student   Employee

Inheritance

Syntax, Protected Members, Accessing Base

Class Code Reuse

- 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

Inheritance

- **`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**

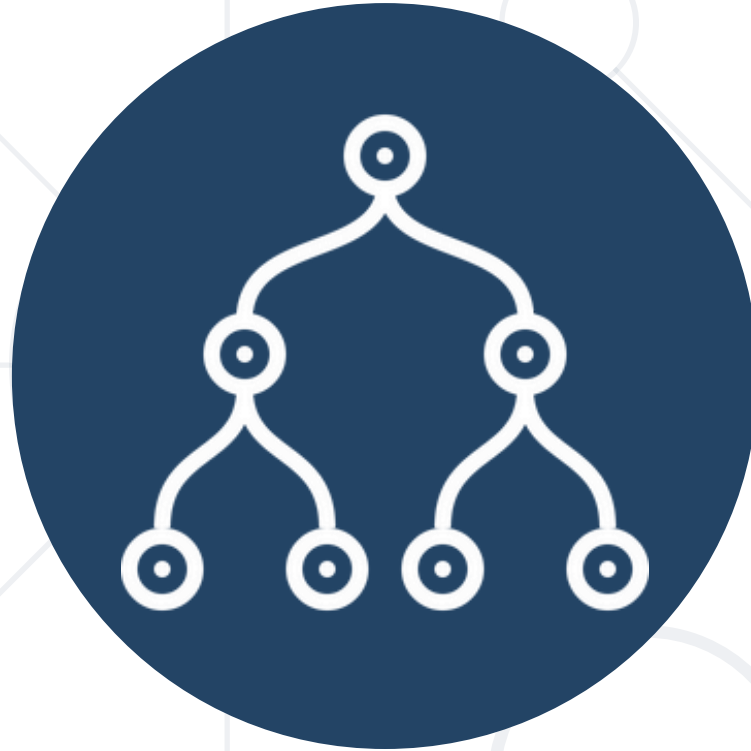# Inheritance – Extracting Base Class

- Extract common members into a base class

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

> Can't use initializer-list
> for base class field

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

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

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

Inheritance

LIVE DEMO

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
```

Protected Members

LIVE DEMO

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

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

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

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

Hiding Methods

- 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::**

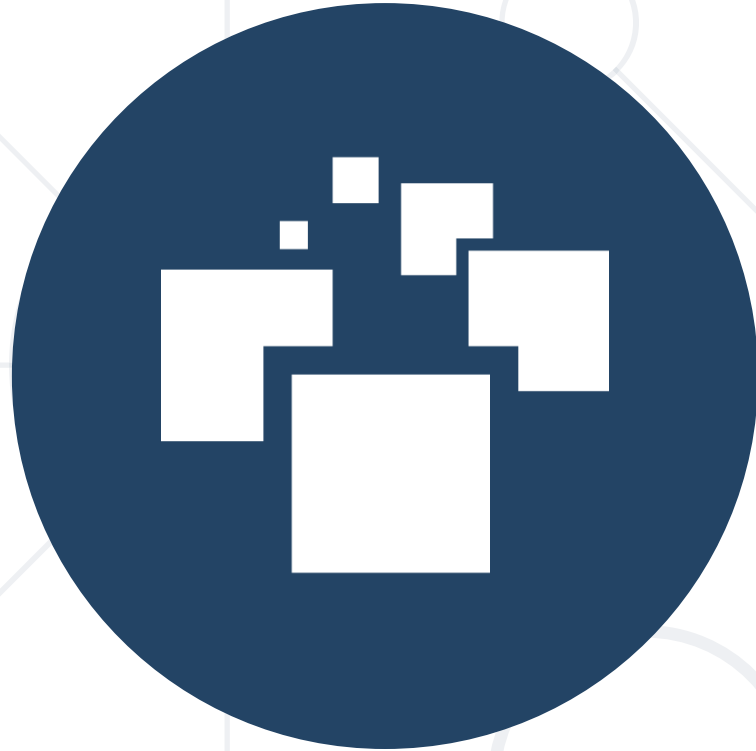 - E.g. **Base::f()** calls **f()** of inherited class **Base**

- Example: Let's make a **toString()** for **Vehicle**

  - Reuse it in **Car**'s **toString()**

```cpp
class Vehicle {

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

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

Calling Base Constructors and Methods

LIVE DEMO

Practice

Live Exercise in Class

Problem 1 - ArrayTrouble

- Study and implement the missing functionality described in the "ArrayTrouble description and skeleton code"

C++ Object Slicing

- Base objects can be assigned with derived objects
    - Implicit cast, called **upcasting**
    - Fields from derived object are "sliced off"
    - *Should generally be avoided*
- **`Base x = Derived();`**
    - **x** can only access **Base** fields
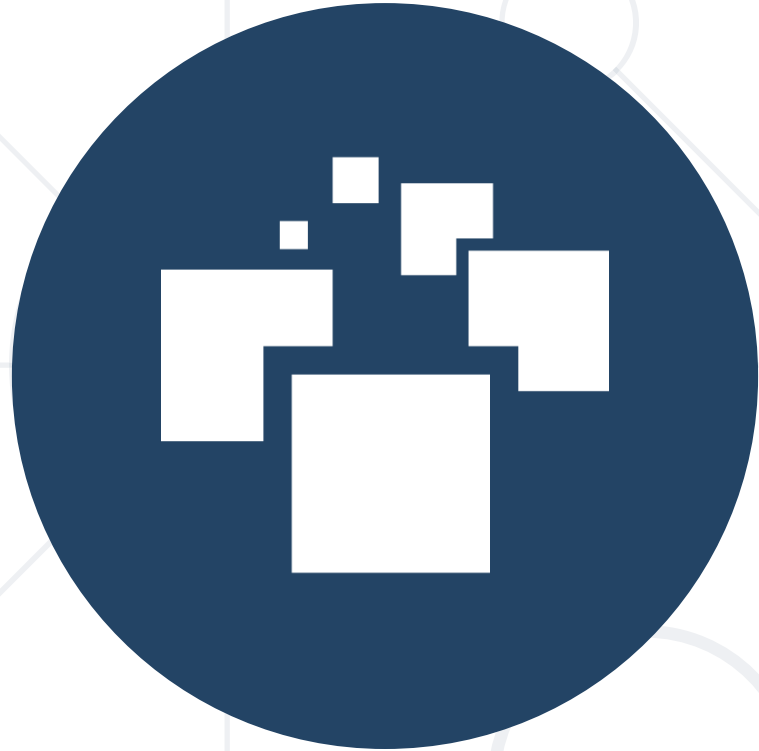
Object Slicing

LIVE DEMO

- What will this code do?

```
Vehicle v =
   Airplane(250, 10000);

cout << v.speed << endl;
```

```
struct Vehicle {
public:
   double speed;
   Vehicle() : speed(0) {}
};

class Airplane : public Vehicle {
public:
   double speed; double altitude;

   Airplane(double speed, double altitude)
   : speed(speed), altitude(altitude) {}
};
```

a) Print **250**

b) Print **0**

c) Compilation error

d) Behavior is undefined

I DON'T ALWAYS SLICE OBJECTS WITH HIDDEN MEMBERS...

C++ PITFALL: SLICING A HIDDEN FIELD

Fields can be hidden just like methods can.

The derived class in the example initializes its own field, not the base field. That field gets sliced-off.

The base class has a default constructor – it gets called on derived initialization, hence 0
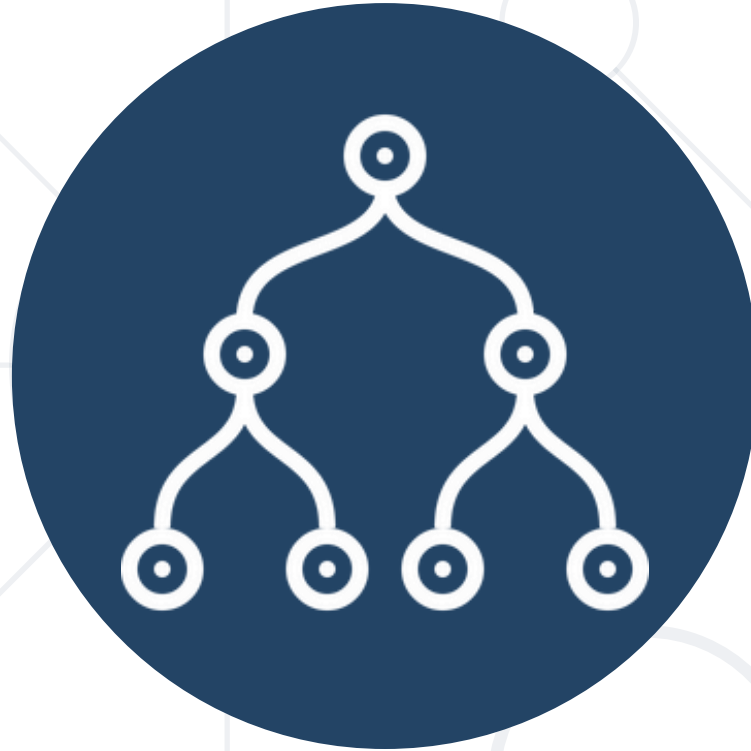
Constructors & Assignments

- 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 to Derived Objects

- **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 override**s
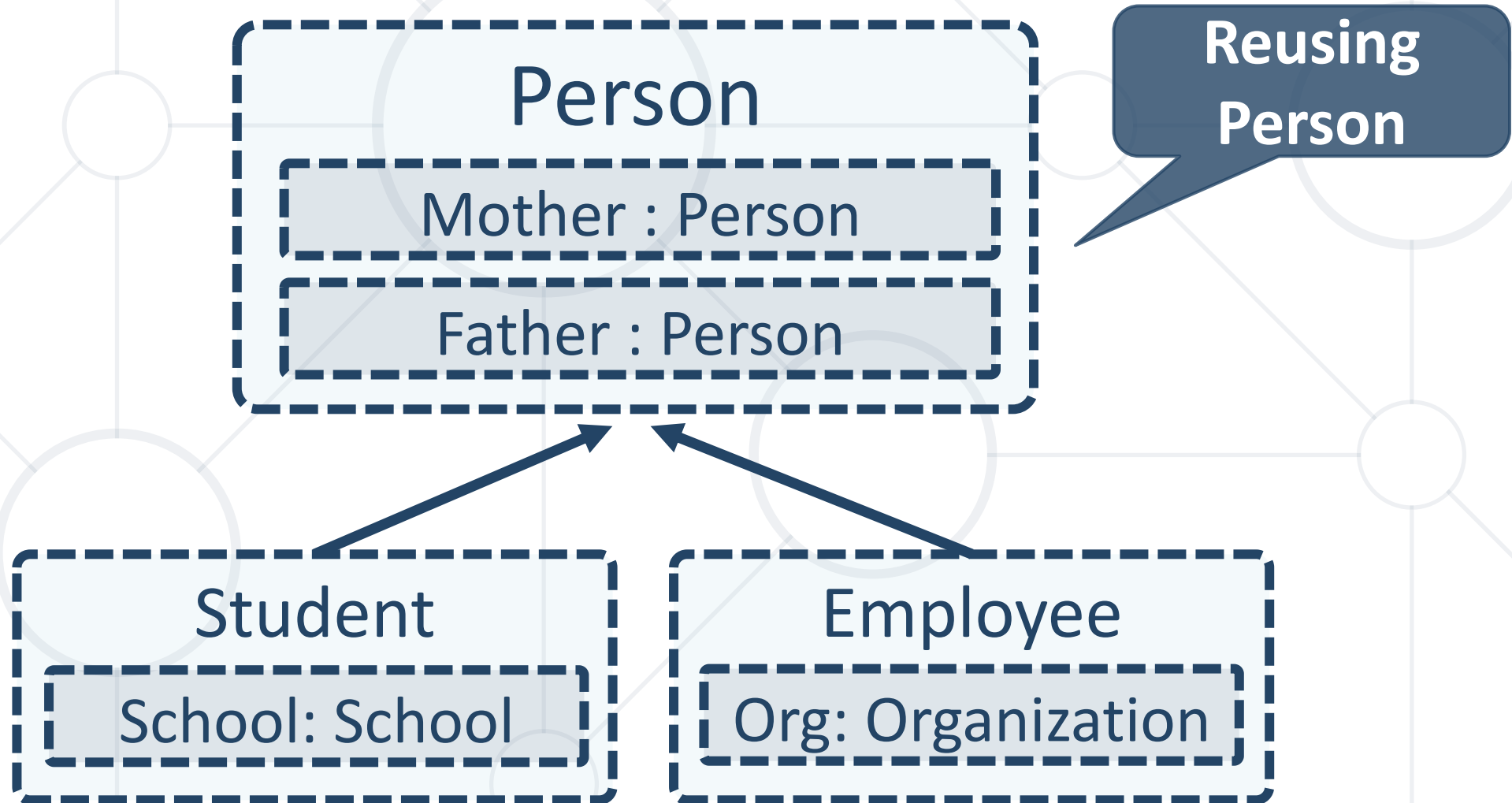   (will be covered in the next lecture - Polymorphism)

Base Pointer to Derived Class

LIVE DEMO

Inheritance – Derived Class

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

Reusing Person

Person

Mother : Person

Father : Person

Student

School: School

Employee

Org: Organization

Using Inherited Members

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

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

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

Person
(Base Class)

+sleep():void

Employee
(Derived Class)

+work():void

Student (Derived Class)

+study():void
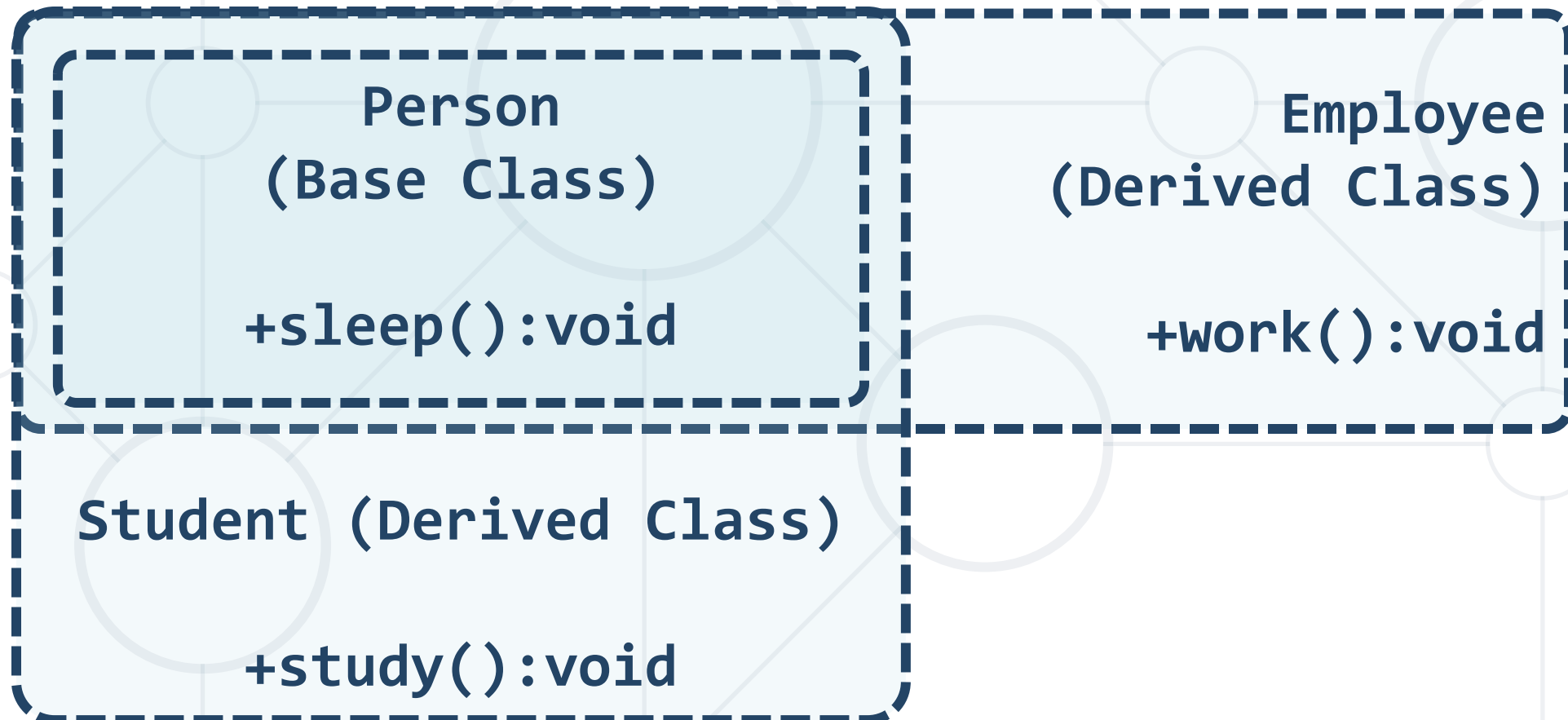
Inheritance

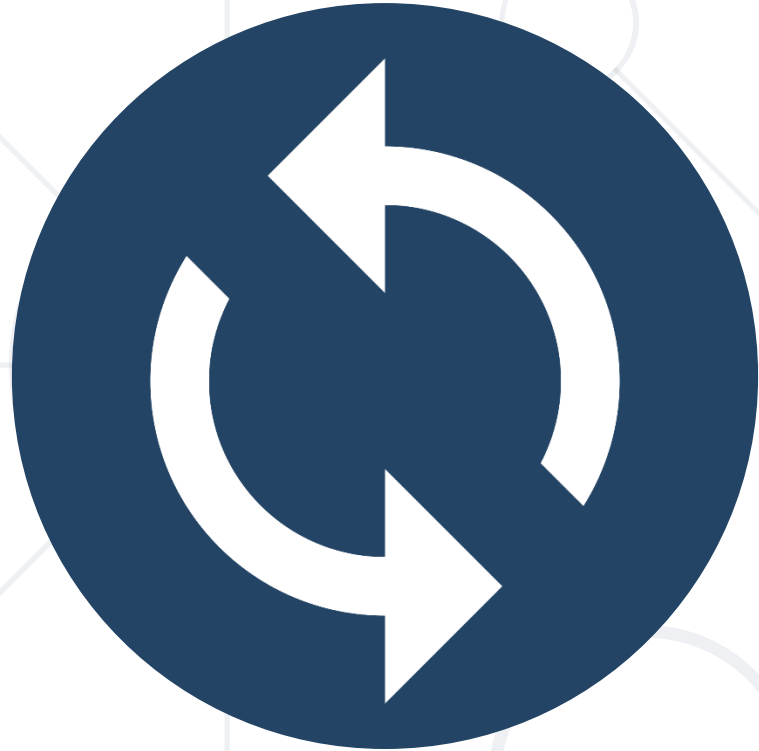- Inheritance has a **transitive relation**

```
class Person { … }
class Student          :public Person { … }
class CollegeStudent :public Student { … }
```
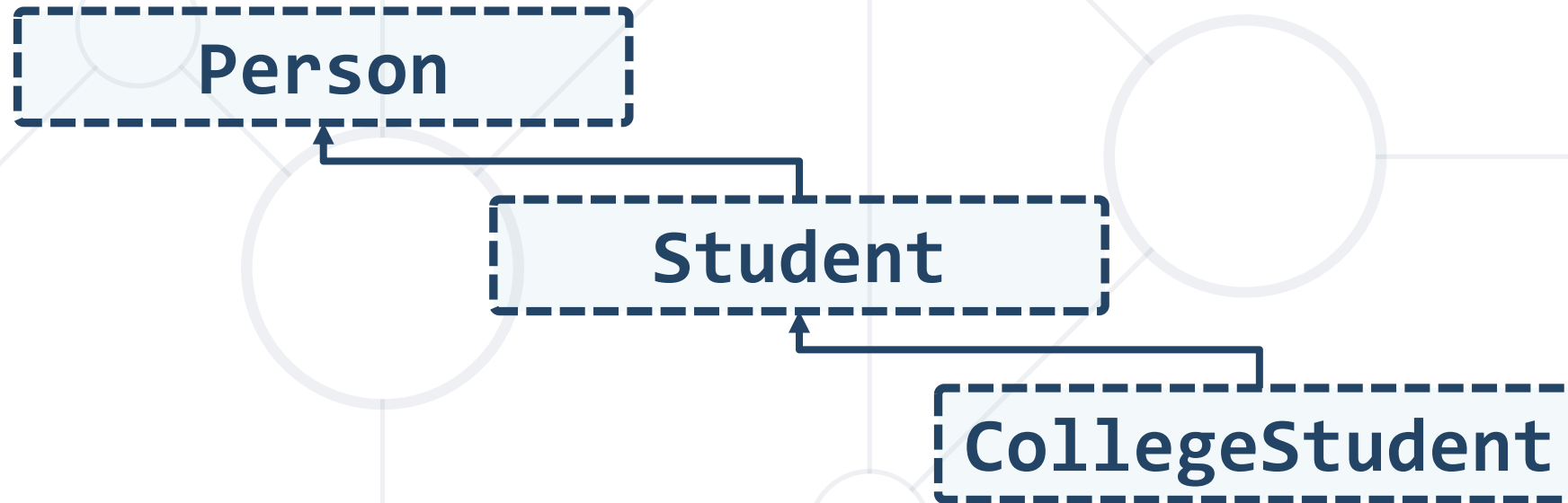
Final Classes

- Inheriting from a final classes is forbidden

```cpp
class Animal final {
    …
}
```
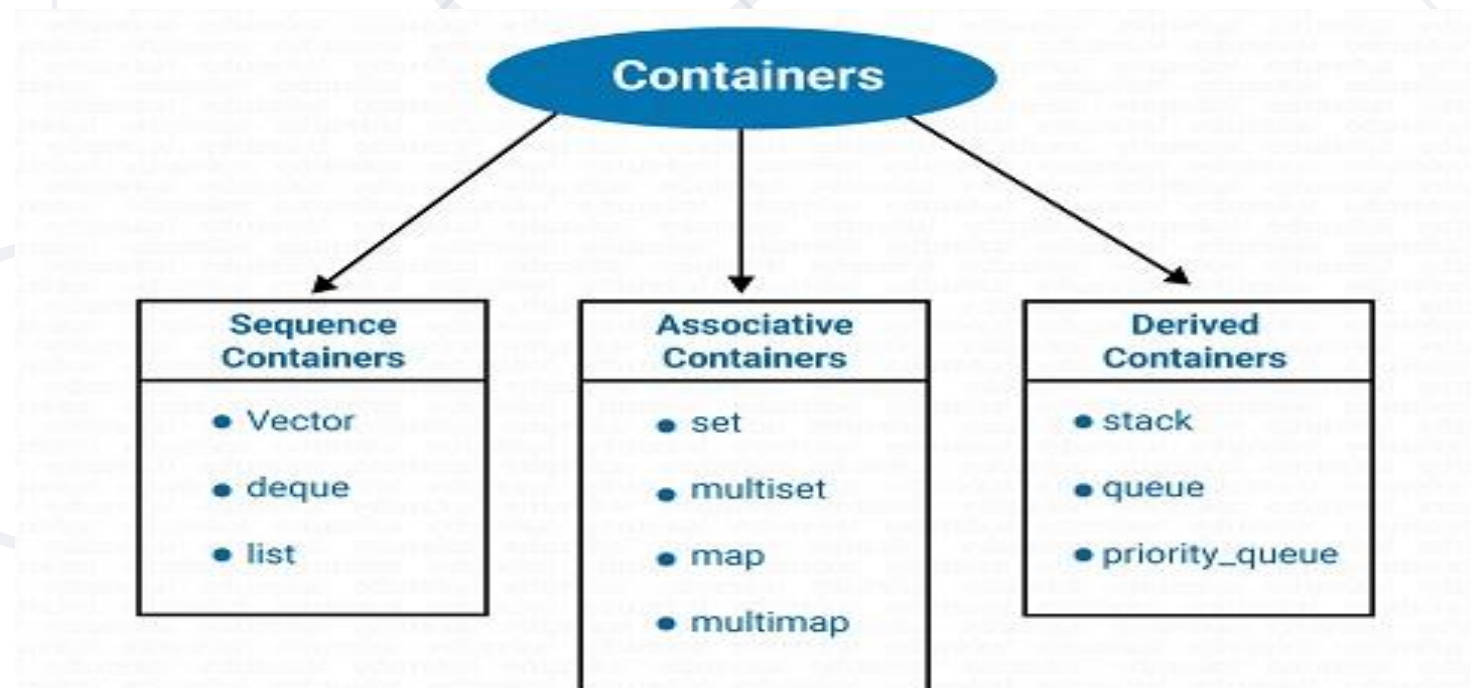
```cpp
public class Dog :public Animal { }        // Error…
```

Reusing Classes

Extension

- **Duplicate code** is error prone

- **Reuse classes** through an **extension**

- Sometimes the only way

Composition
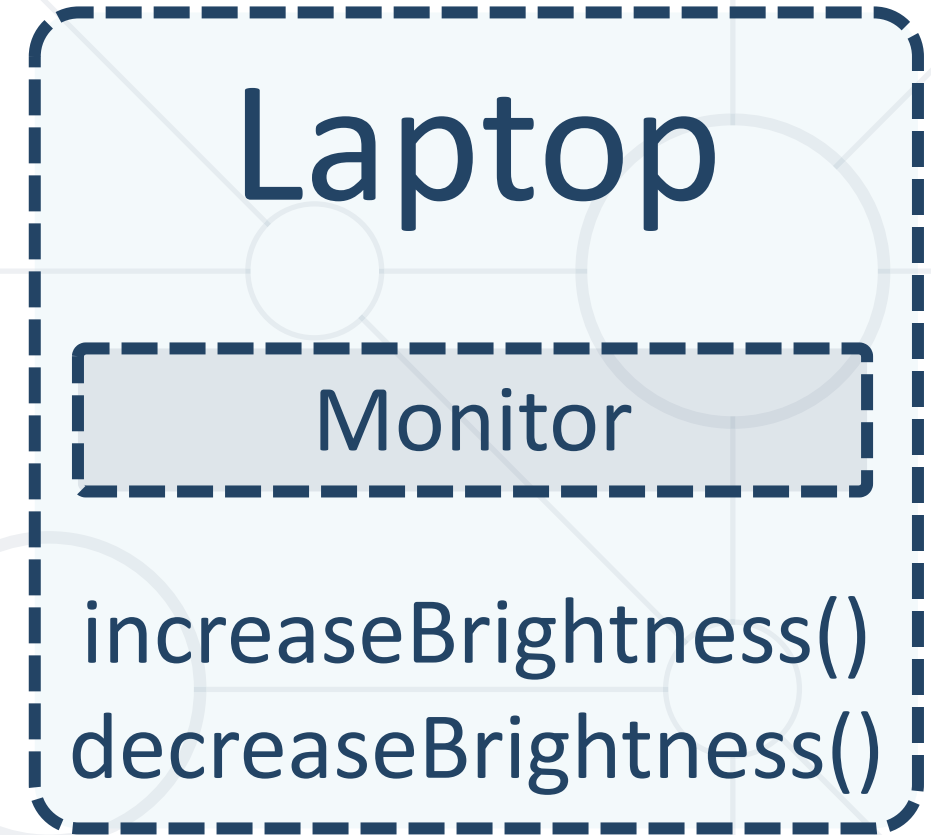
- Using classes to **define classes**

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

Reusing classes

Laptop

Monitor

Touchpad

Keyboard

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

  void decrBrightness() {
    monitor.dim();
  }
}
```

Laptop

Monitor

increaseBrightness()
decreaseBrightness()

Practice

Live Exercise in Class

- Study and implement the missing functionality described in the "Hardware Store description and skeleton code"

When to Use Inheritance

- 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
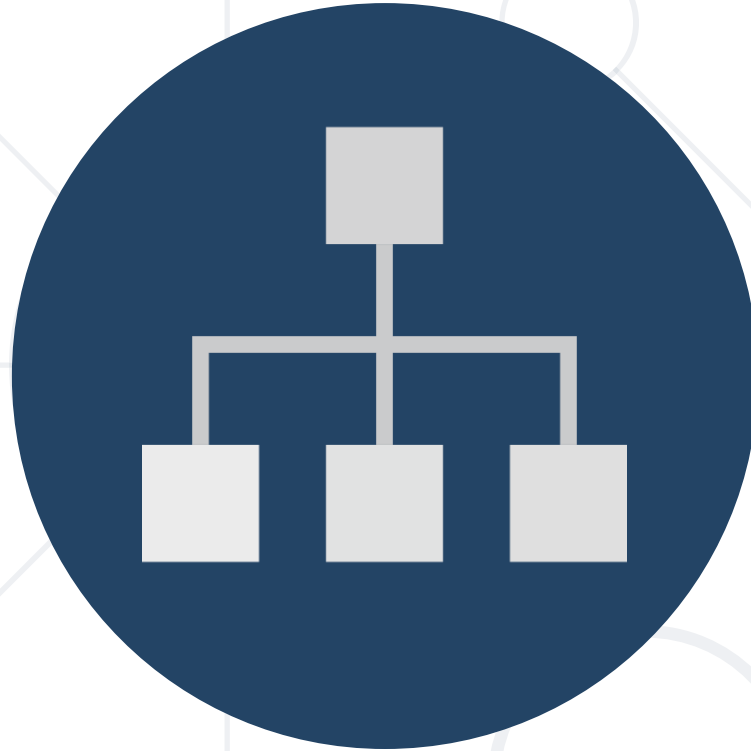
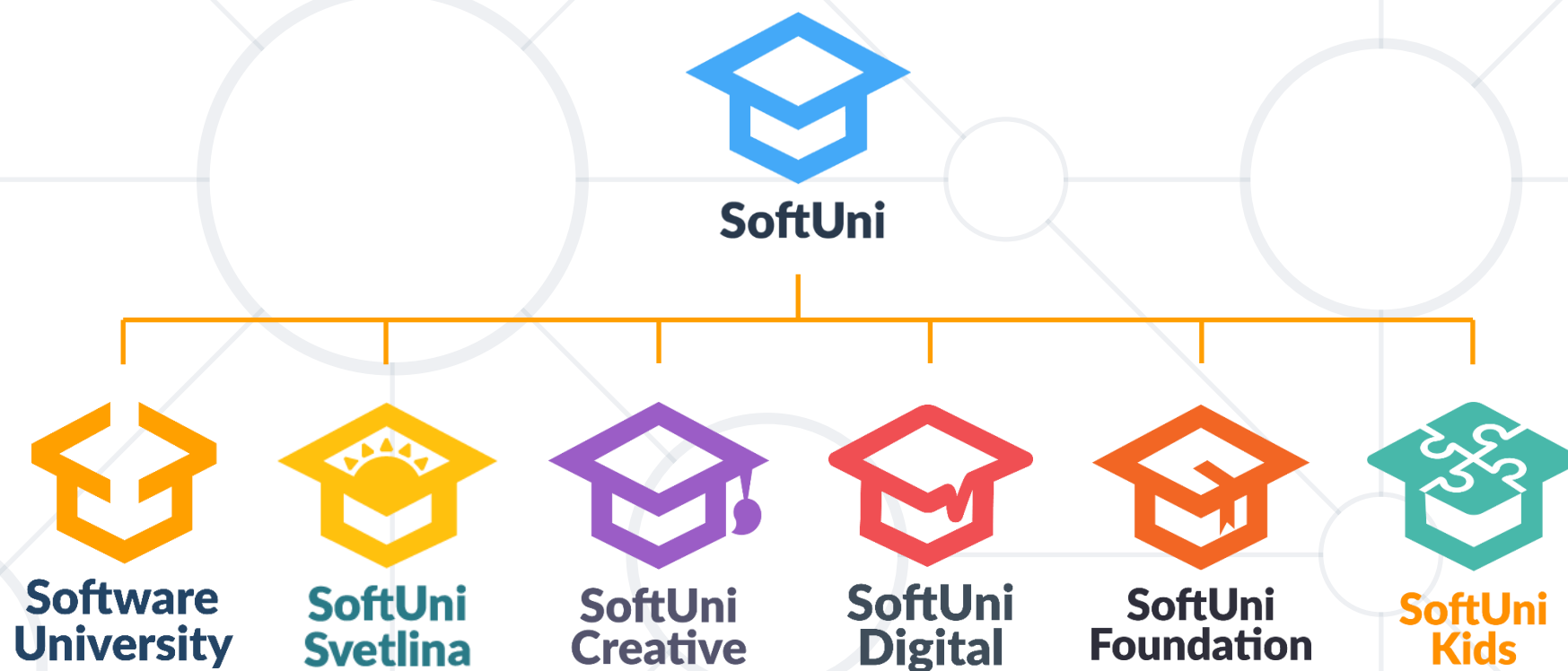# Signaling from Composition Classes to Owner Class

LIVE DEMO

Summary

■ 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

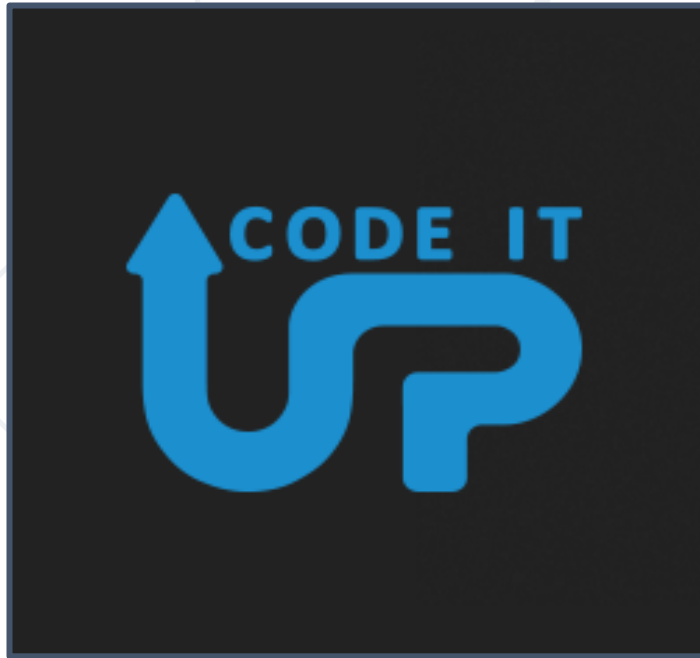■ Extract multiple-usage code into a base class

Questions?

SoftUni

Software University SoftUni Svetlina SoftUni Creative SoftUni Digital SoftUni Foundation SoftUni Kids

SoftUni Diamond Partners

Educational Partners

License

- 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

Trainings @ Software University (SoftUni)

- 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
- Software University Forums
  - forum.softuni.bg