

Object Oriented Programming

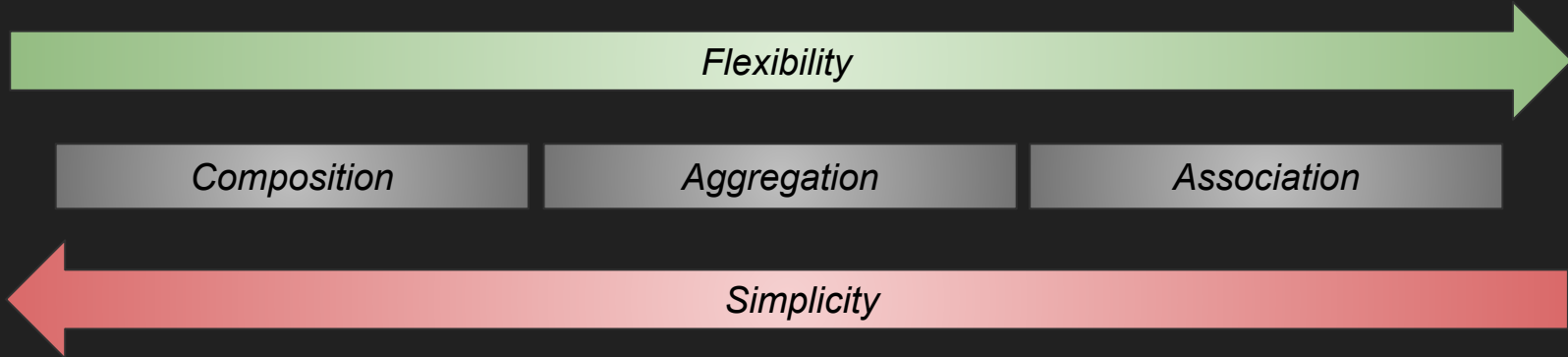
Object Relations & Interactions

Object relations

- Instance level relations
 - Composition
 - Aggregation
 - Association
- Class level relations
 - *Inheritance*

Instance level relations: Overview

- **Composition** is the strongest relation - *ownership (has-a)* with *lifecycle dependency*. Simplest relation, but no flexibility.
- **Aggregation** is weaker ownership (**has-a**) without *lifecycle dependency*.
- **Association** is the weakest relation of the three. **No** ownership, **no** *lifecycle dependency*, only knowledge / loose usage of the *associated* object(s).



Instance level relations: *Composition*

- **Composition** is the strongest ownership relation
- It is never **shared** - the owned object belongs exclusively and solely to the owner
- There is lifecycle dependency: destroying the owner in a *composition* relation typically means destroying the *owned object(s)*.
- However, usually the owned object cannot be changed, replaced, reused outside of its exclusive owner. Usually it cannot be retained after owner's death

```
class Cup
{
public:
    Cup();
    Cup(double, double, const Picture&);
    Cup(const Cup&);
    ~Cup();

    void fill(const double);
    double getQuantity() const;
private:
    double quantity;
    double capacity;
    Picture picture;
};
```

Instance level relations: *Aggregation*

- ***Aggregation*** is weaker ownership relation
- It can be shared (*but not always is*). If shared, the owned object potentially belongs to multiple owners.
- There is no *lifecycle dependency*: the owned object can outlive its current owner. Often the owner can be created *empty* and receive owned objects later.
- The programmer often needs to take care of *lifecycles* of both owner and owned objects separately. He must also prevent the risk of calling into already dead object. *Resource leaks* must be prevent too.
- The owned object typically can be *changed, replaced, reused* outside of the current owner.

```
class Rectangle
{
private:
```

```
    struct Point
    {
        double x;
        double y;
    };
```

```
    Point a;
    Point b;
```

```
};
```

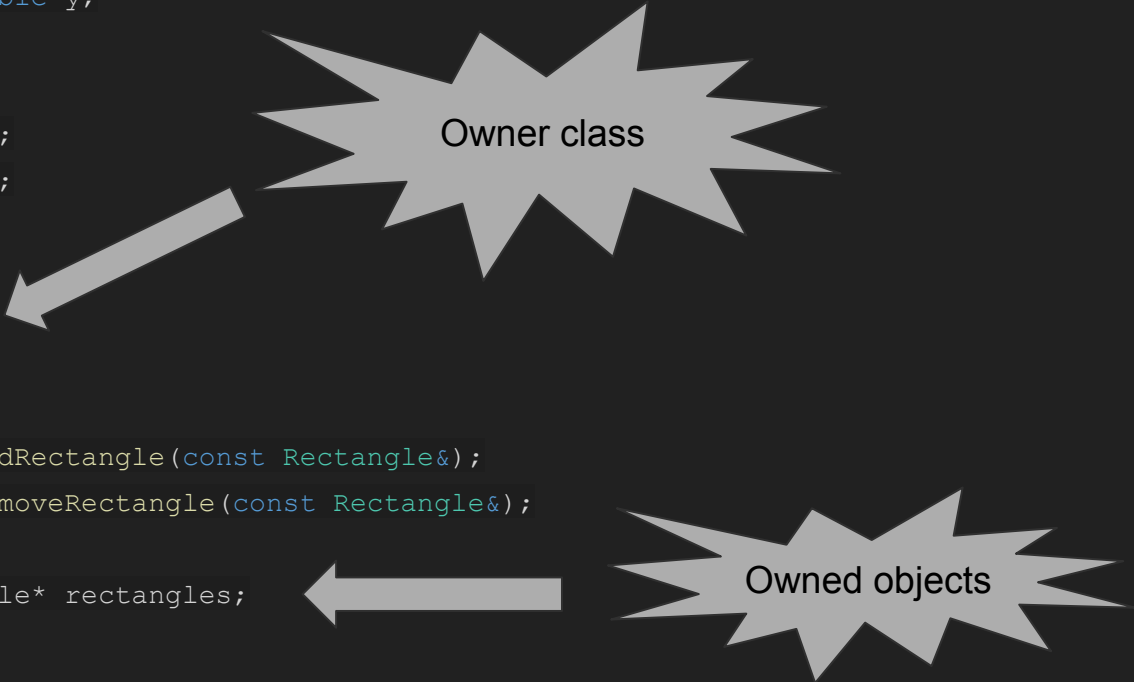
```
class SVG
```

```
{
public:
    void addRectangle(const Rectangle&);
    bool removeRectangle(const Rectangle&);
```

```
private:
```

```
    Rectangle* rectangles;
```

```
};
```



Owner class

Owned objects

```
class Rectangle
```

```
{
```

```
private:
```

```
    struct Point
```

```
    {
```

```
        double x;
```

```
        double y;
```

```
    };
```

```
    Point a;
```

```
    Point b;
```

```
};
```

```
class SVG
```

```
{
```

```
public:
```

```
    void addRectangle(const Rectangle&);
```

```
    bool removeRectangle(const Rectangle&);
```

```
private:
```

```
    Rectangle* rectangles;
```

```
};
```

?

Owner class

Owned objects

```
class Rectangle
{
private:
```

```
    struct Point
    {
        double x;
        double y;
    };
```

```
    Point a;
    Point b;
};
```

```
class SVG
```

```
{
public:
    void addRectangle(const Rectangle&);
    bool removeRectangle(const Rectangle&);
private:
    Rectangle* rectangles;
};
```

?

Owner class

Aggregating methods are usually provided in order to receive owned objects anytime.

Owned objects

Instance level relations: *Association*

- **Association** is weakest relation of the three. It is **no ownership** at all.
- There is no *lifecycle dependency*. An object just knows the type of another object and collaborates with it (request a service).
- Basically, an **objA** of class **A** receives a message (has a method call) with **objB** of class **B** passed as an argument. Then **objA** uses **objB** to perform a job and returns.

```
class Seller;  
  
class Customer  
{  
public:  
    void buyGoods(const Seller& seller);  
};
```

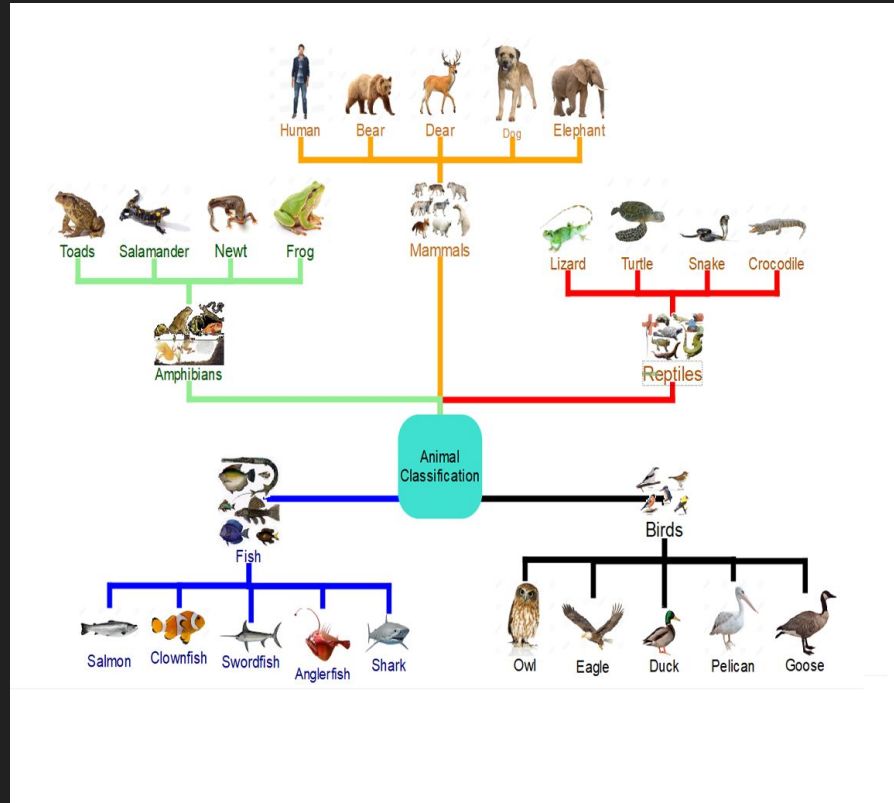
Inheritance & Generalization

Inheritance, Subtyping, Generalization - Basics

- ***class level*** type of relation (relation between ***entire classes***, not between instances/objects)
- Defines an “***is-a***” relationship between classes
- Used to *classify* classes into larger groups, which are also *classes* (*what ??*)
 - A Mamal “***is-an***” Animal (both Mamal and Animal are classes)
 - Deeper hierarchies are possible: a Human “***is-a***” Mamal which “***is-an***” Animal
- ***Each descendent (i.e. the Human class) receives members of its base class(es) (i.e. Mamal and Animal)***

Inheritance, Subtyping, Generalization contd.

Generalization



Inheritance

Inheritance/Generalization examples

- Which of the following examples correspond to *is-a* relationship, and which don't?

```
class Phone;  
// is-a  
class Display;
```

```
class Circle;  
class Rectangle;  
// is-a  
class Shape;
```

```
class Car;  
// is-a  
class Vehicle;
```

```
class SVG;  
// is-a  
class Rectangle;
```

Inheritance/Generalization examples

- Which of the following examples correspond to *is-a* relationship, and which don't?

```
class Phone;  
// has-a  
class Display;
```

```
class Circle;  
class Rectangle;  
// is-a  
class Shape;
```

```
class Car;  
// is-a  
class Vehicle;
```

```
class SVG;  
// has-a  
class Rectangle;
```

Inheritance/Generalization examples contd.

Consider the following example: What is not fine here?

```
class Vehicle {
public:
    Vehicle();
    Vehicle(const char*, const char*,
            const double&);
    Vehicle(const Vehicle&);
    Vehicle& operator=(const Vehicle&);
    ~Vehicle();
    void fuel(const double&);
private:
    char* model;
    char* registrationPlate;
    double fuelLiters;
};
```

```
class Car {
public:
    Car();
    Car(const char*, const char*, const double&, const unsigned&);
    Car(const Car&);
    Car& operator=(const Car&);
    ~Car();
    void fuel(const double&);
    unsigned getNumberOfSeats() const;
private:
    char* model;
    char* registrationPlate;
    double fuelLiters;
    unsigned numberOfSeats;
};
```

Inheritance/Generalization examples contd.

Consider the following example:

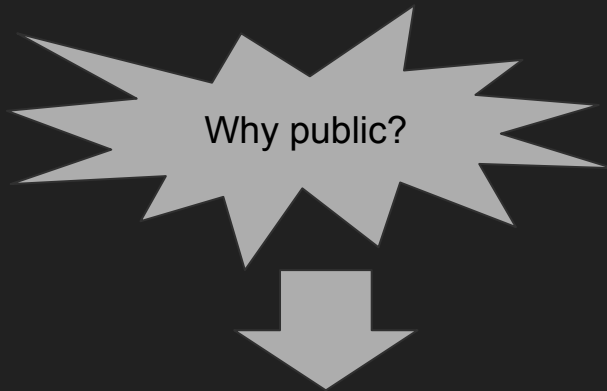
```
class Vehicle {
public:
    Vehicle();
    Vehicle(const char*, const char*,
            const double&);
    Vehicle(const Vehicle&);
    Vehicle& operator=(const Vehicle&);
    ~Vehicle();
    void fuel(const double&);
private:
    char* model;
    char* registrationPlate;
    double fuelLiters;
};
```

```
class Car : public Vehicle
{
public:
    unsigned getNumberOfSeats() const;
private:
    unsigned numberOfSeats;
};
```


Inheritance/Generalization examples contd.

Consider the following example:

```
class Vehicle {  
public:  
    Vehicle();  
    Vehicle(const char*, const char*,  
            const double&);  
    Vehicle(const Vehicle&);  
    Vehicle& operator=(const Vehicle&);  
    ~Vehicle();  
    void fuel(const double&);  
private:  
    char* model;  
    char* registrationPlate;  
    double fuelLiters;  
};
```

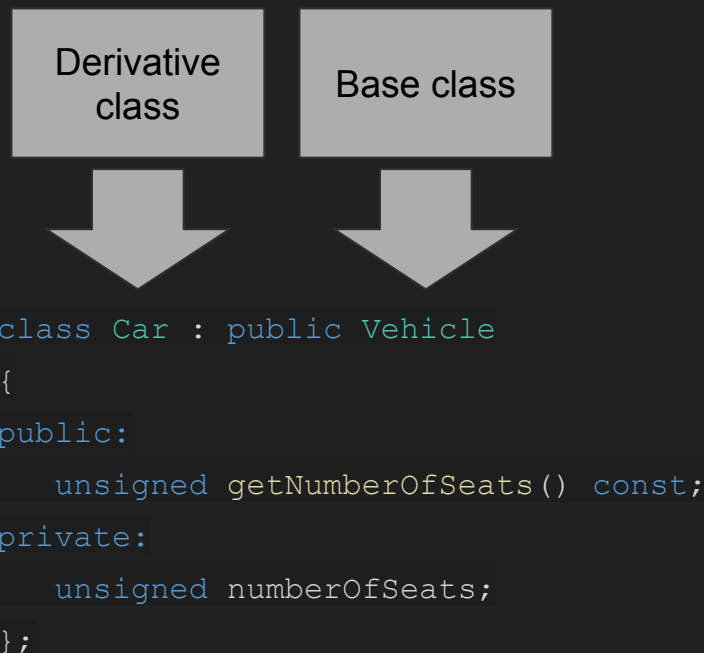


```
class Car : public Vehicle  
{  
public:  
    unsigned getNumberOfSeats() const;  
private:  
    unsigned numberOfSeats;  
};
```

Inheritance/Generalization examples contd.

Consider the following example:

```
class Vehicle {  
public:  
    Vehicle();  
    Vehicle(const char*, const char*,  
            const double&);  
    Vehicle(const Vehicle&);  
    Vehicle& operator=(const Vehicle&);  
    ~Vehicle();  
    void fuel(const double&);  
private:  
    char* model;  
    char* registrationPlate;  
    double fuelLiters;  
};
```



Inheritance types and access modifiers

Access modifier Inheritance type	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Inherited objects construction & destruction

- By default the base class **default constructor** is called every time a derivative object is created.
- If we want to call another (*non-default*) *Base class* constructor from the derivative class, we must do it explicitly with *Base::Base(params | Base&);*
- Calling *Base class' operator=* from the derivative class:
Base::operator=(derivativeObj);
- Destruction of derived class object - by default calls *Base class destructor*
- ***Construction & destruction order of derived classes objects explanation***

Inheritance chaining

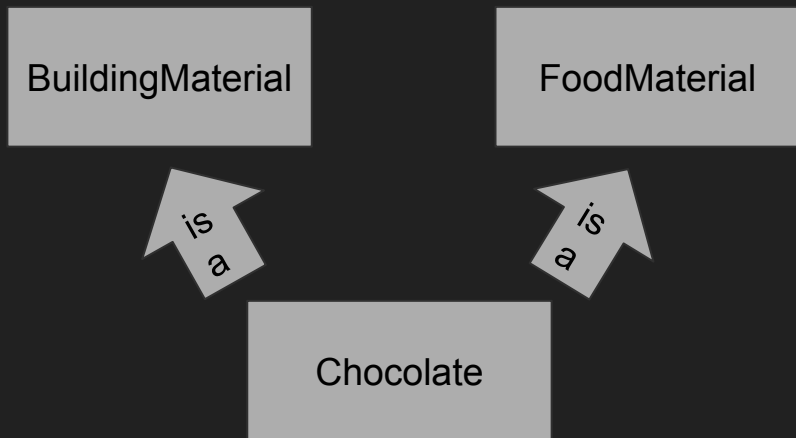
- Deeper hierarchies is possible in c++
 - **A** inherits from **B** inherits from **C** ... inherits from **Z**
- Dealing with more than 3 levels of inheritance becomes hard to understand, hard to maintain, less flexible and so on
- When there is **two** possible relations between **two** classes (*objects*), ***always favour composition / aggregation over inheritance***
 - However, we rarely consider more than one logical relation between classes(objects)

Brainstorming

- ❑ Create a (part of) car racing game. You have the following requirements:
- ❑ There is a Garage in the game; it provides maintenance to all Cars
- ❑ The player is a Driver who owns Car(s) and goes to the Garage from time to time to change the oil, etc.
- ❑ The Garage keeps information about what cars have been served there, but doesn't need that information until an Inspector comes and requests to check it

Multiple inheritance

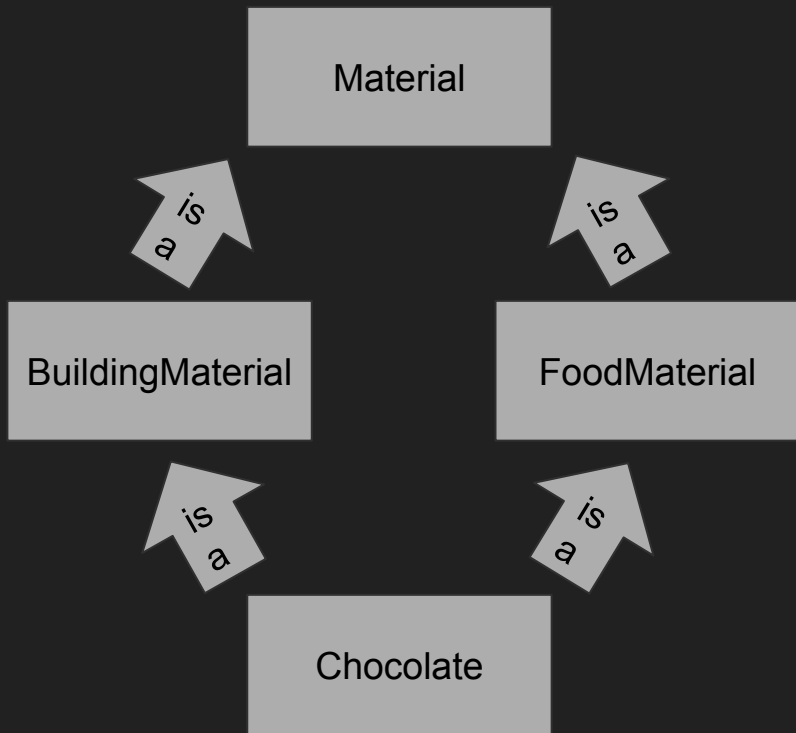
- **Multiple inheritance (MI)** - a scenario where a *descendant* has more than one *base* classes (**not to be confused with multilevel inheritance**)
 - A throwback to std streams hierarchy



- The descendant (i.e. *Chocolate*) inherits base classes' "features" "is a" relation to all base classes (i.e. *BuildingMaterial* and *FoodMaterial*)
- Ambiguities may occur (having members with the same name) - using scope operator to resolve that (i.e. *Base1::x* vs. *Base2::x*).

Multiple inheritance - Dangers

- Dreaded diamond



- Let that *Material* has attribute *density(double)*
- Let that *BuildingMaterial* has attribute *strength(int)*
- Let that *FoodMaterial* has attribute *taste(enum)*
- At the *Chocolate* class what attributes are inherited?
- How many copies of *density*?