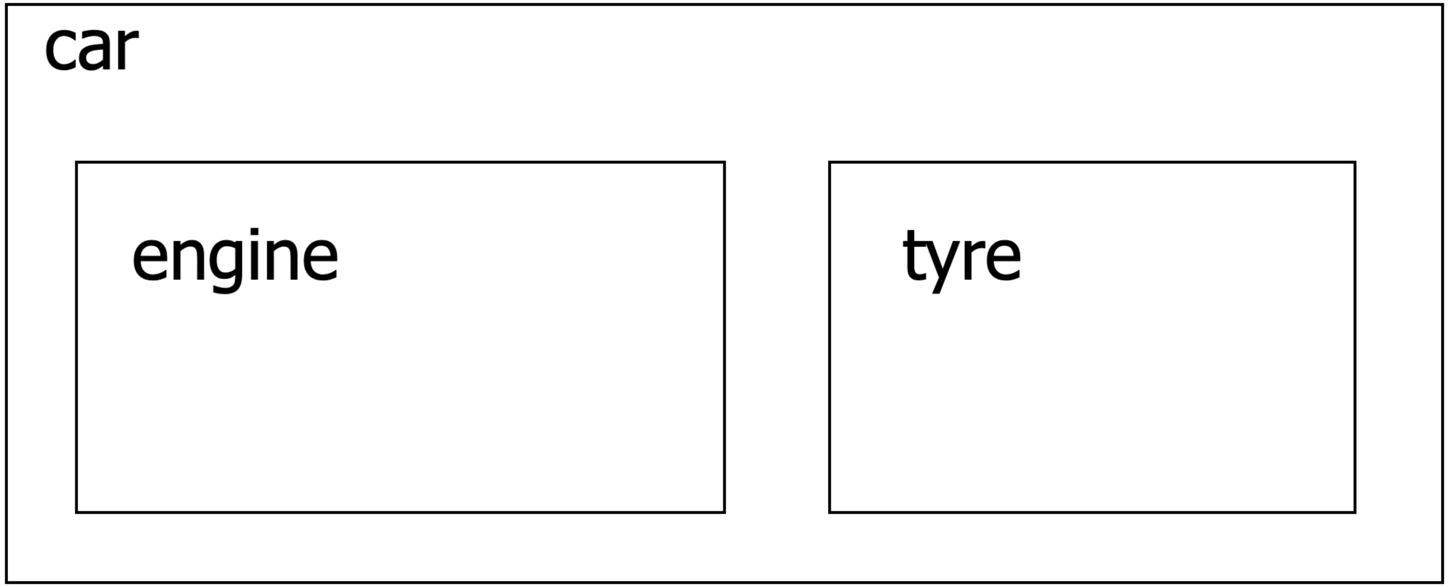
Inheritance

Reusing the implementation

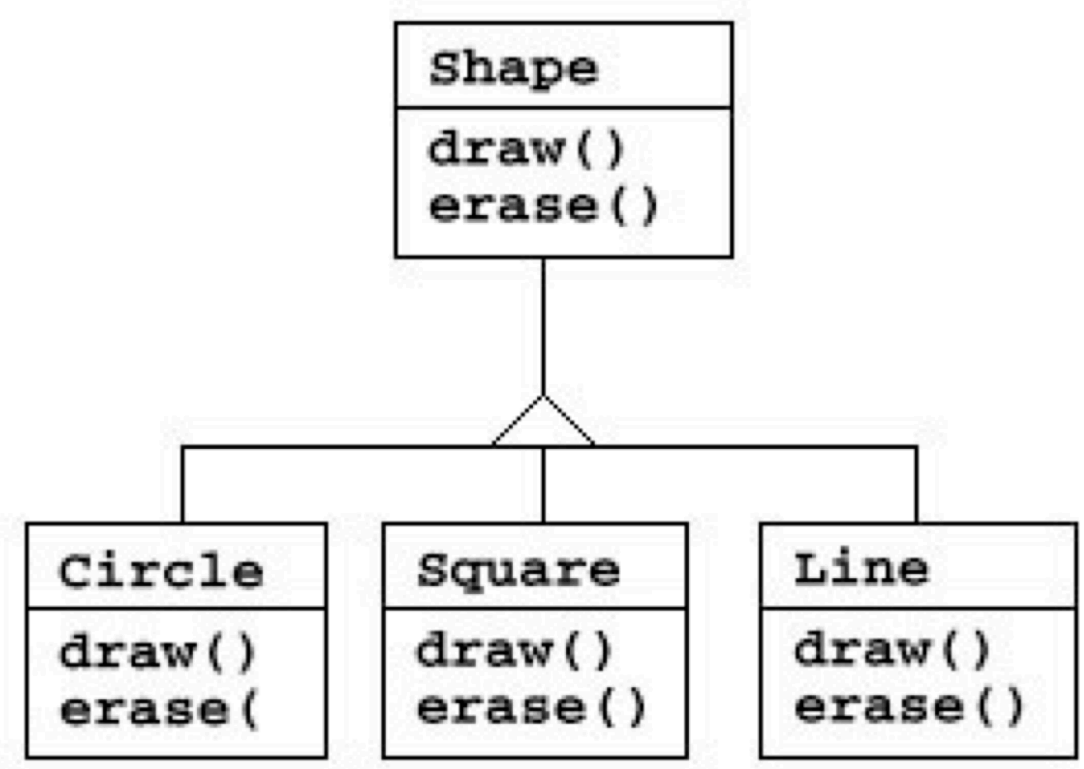
- Composition: construct new object with existing objects
- It is the relationship of “has-a”



Each object has its own memory consists of other objects. -- by Alan Kay

Reusing the interface

- Inheritance is to take the existing class, clone it, and then make additions and modifications to the clone.

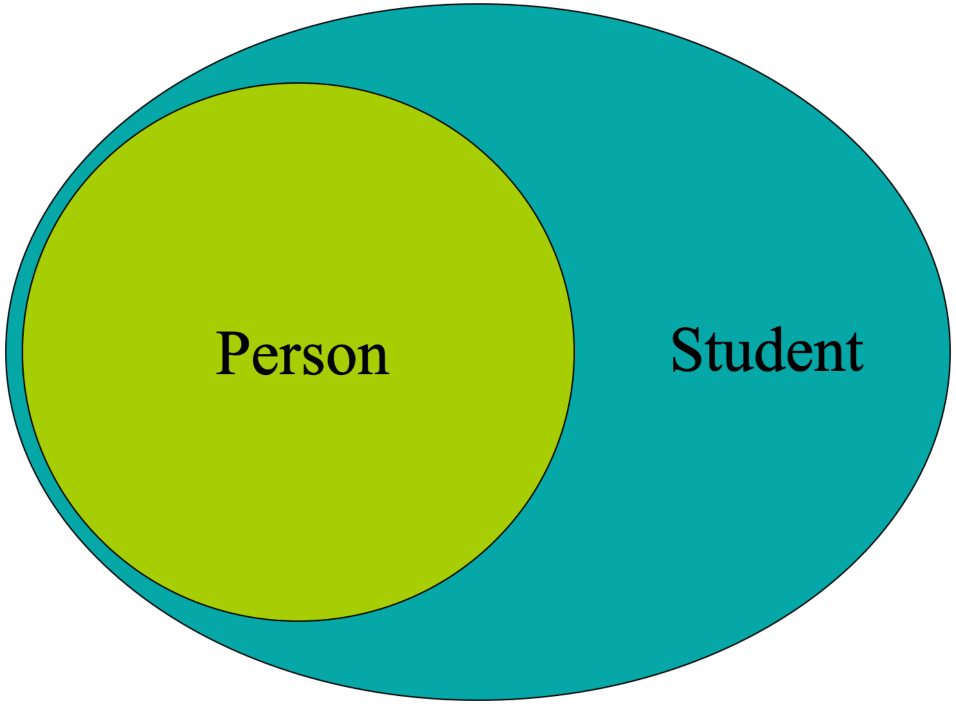


Inheritance

- Language implementation technique
- Also an important component of the OO design methodology
- Allows sharing of design for
    - Member data
    - Member functions
    - Interfaces
- Key technology in C++

Inheritance

- The ability to define the behavior or implementation of one class as a superset of another class



DoME

- is an application that lets us store information about CDs and DVDs.We can
  - enter information about CDs and DVDs
  - search, for example, all CDs in the database by a certain artist, or all DVDs by a given director

CD

- the title of the album;
- the artist (name of the band or singer);
- the number of tracks on the CD;
- the total playing time;
- a ‘got it’ flag that indicates whether I own a copy of this CD; and
- a comment (some arbitrary text).

DVD

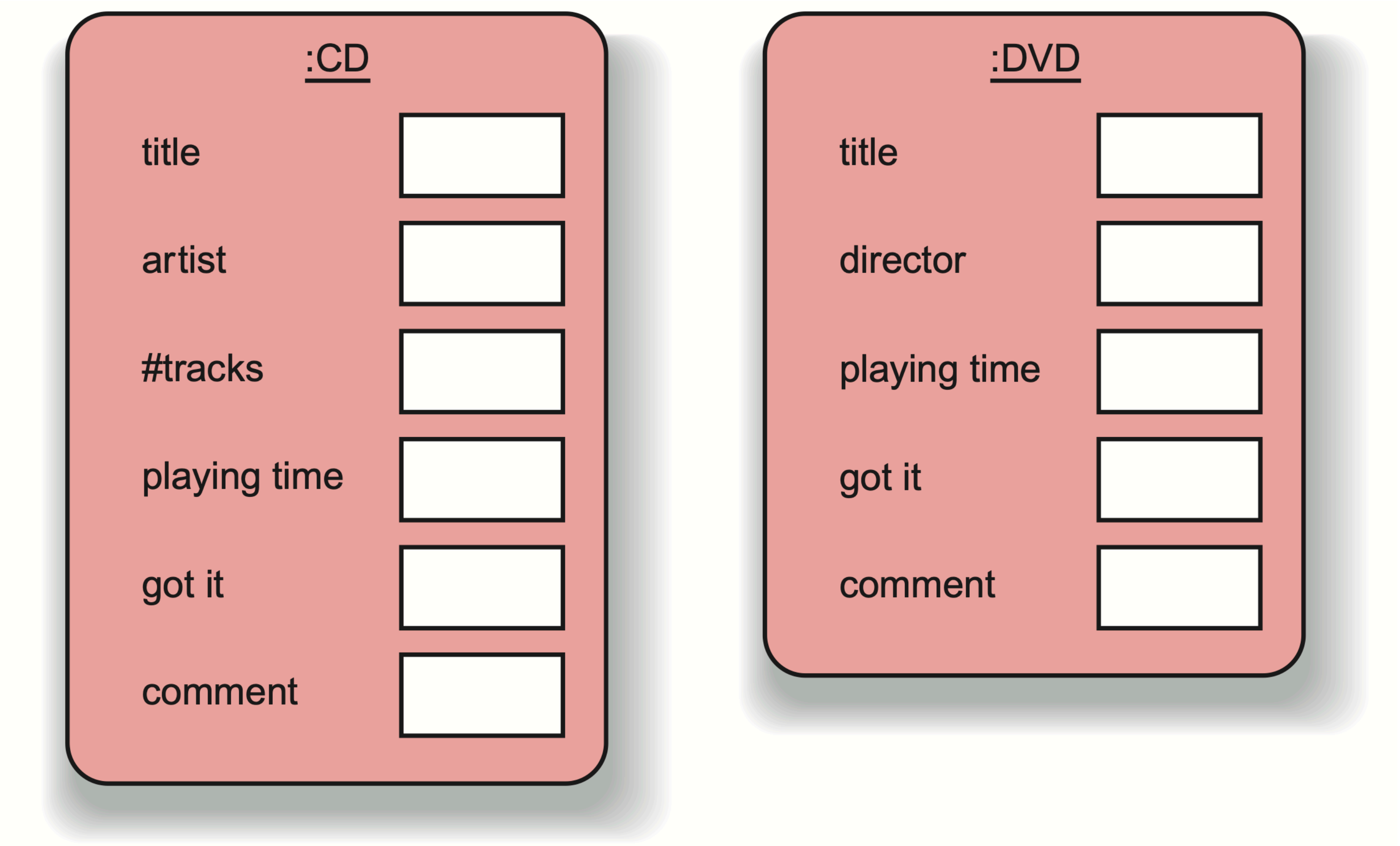
- the title of the DVD;
- the name of the director;
- the playing time (we define this as the playing time of the main feature);
- a ‘got it’ flag that indicates whether I own a copy of this DVD; and
- a comment (some arbitrary text)

The DoME example

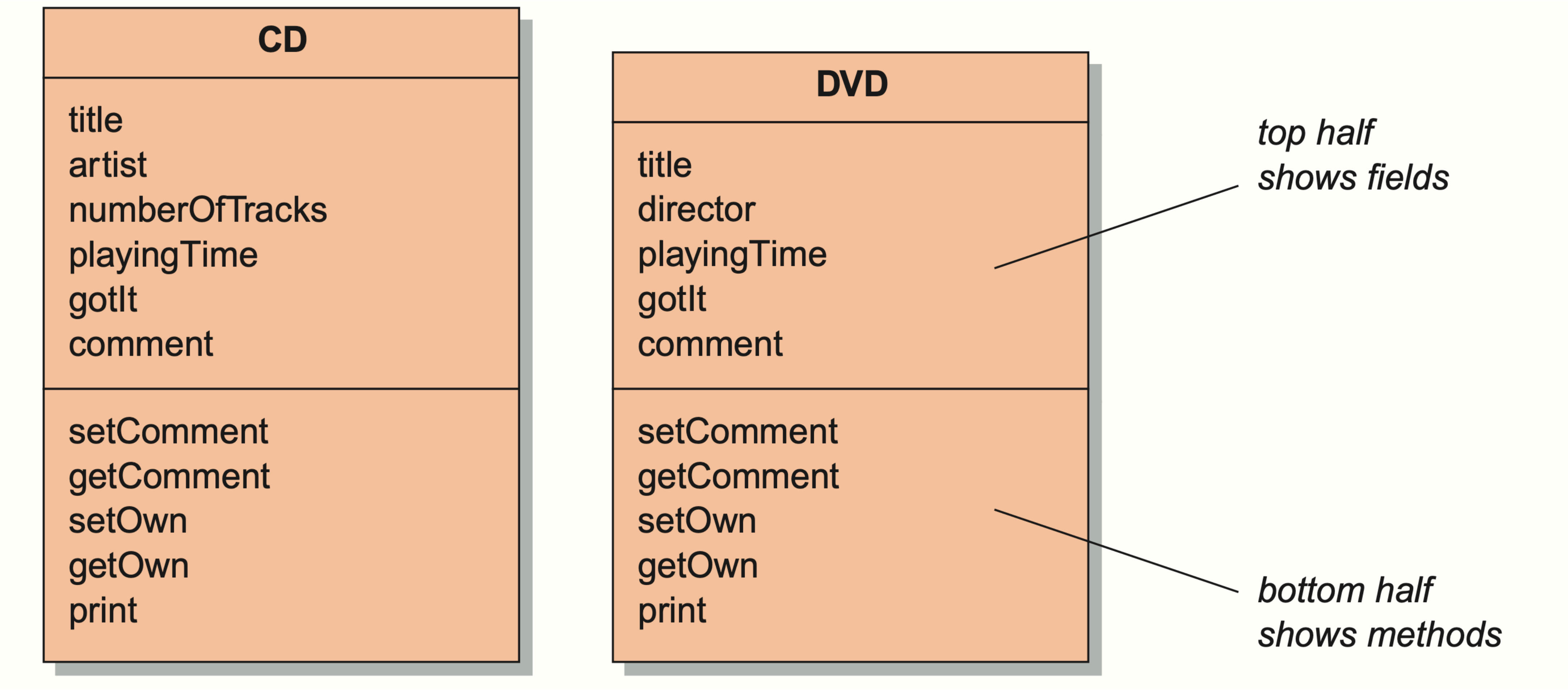
"Database of Multimedia Entertainment"

- stores details about CDs and DVDs
  - CD: title, artist, # tracks, playing time, got-it, comment
  - DVD: title, director, playing time, got-it, comment
- allows (later) to search for information or print lists

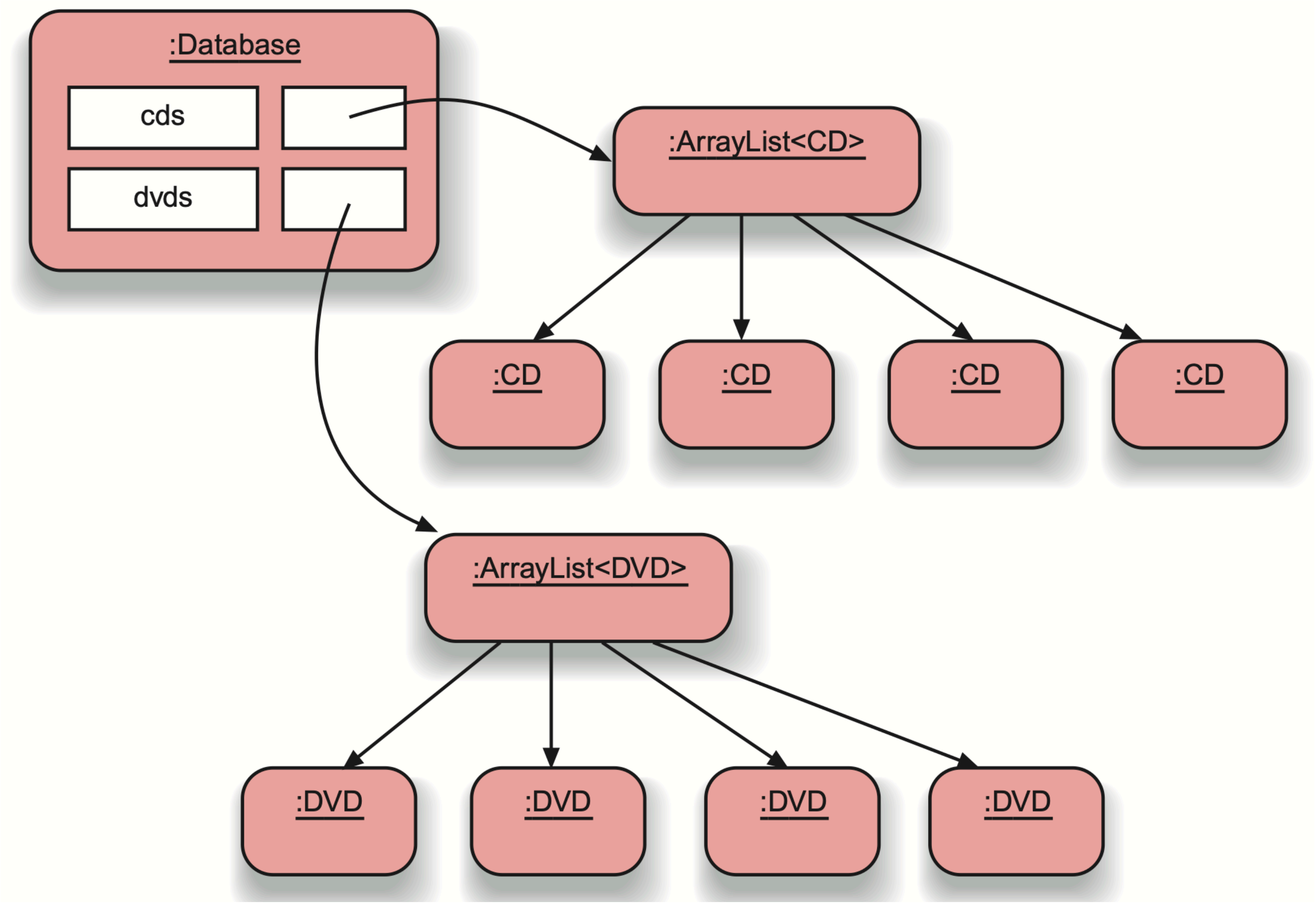
DoME classes



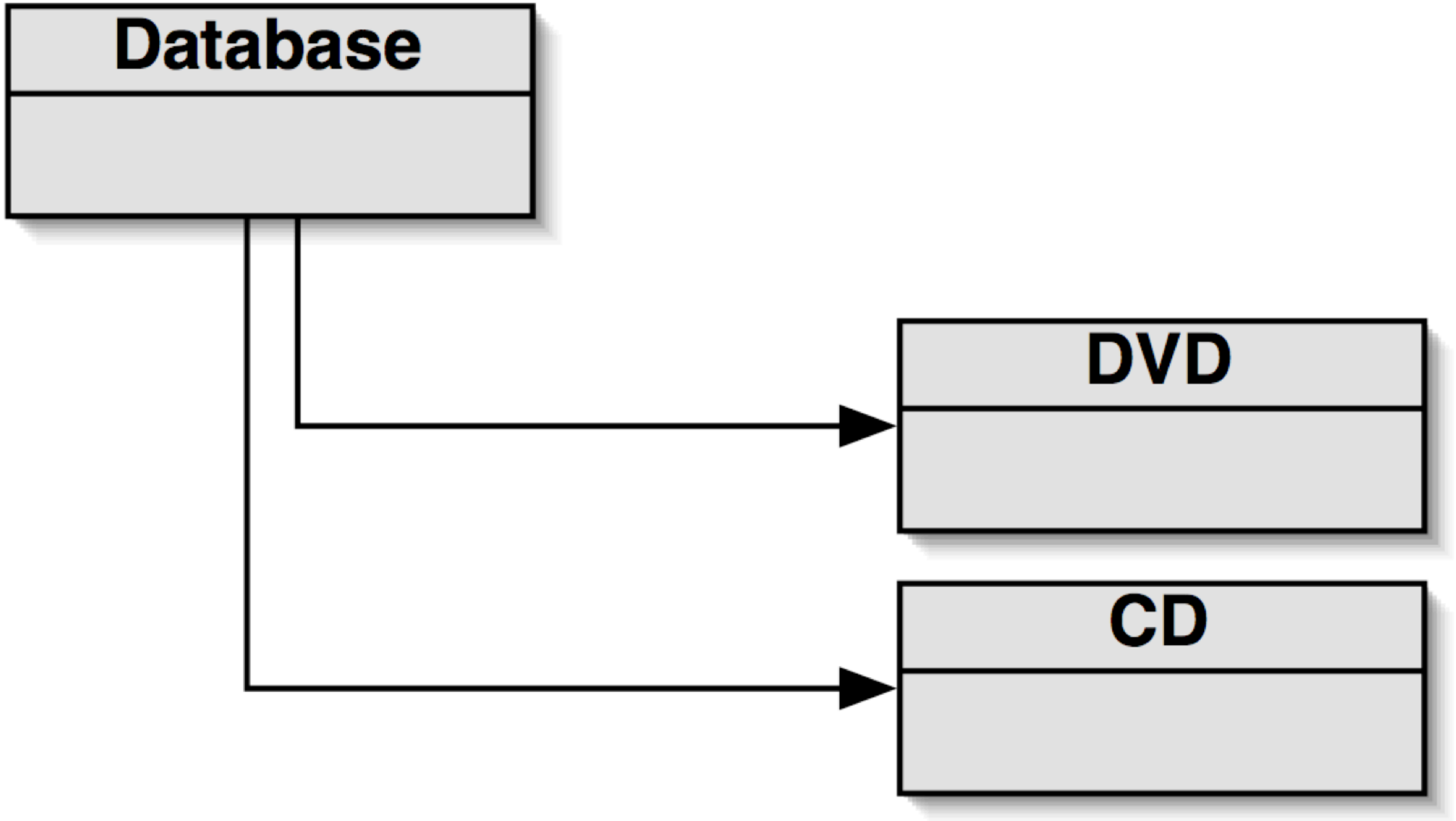
Class diagram



Object model



Class diagram



source code

```cpp
class Database {
    vector<CD> cds;
    vecrot<DVD> dvds;
public:
    void addCD(CD &aCD);
    void addDVD(DVD &aDVD);
    void list() {
        for (auto x:cds) { cd.print();}
        for (auto x:dvds) { dvd.print();}
    }
};
```

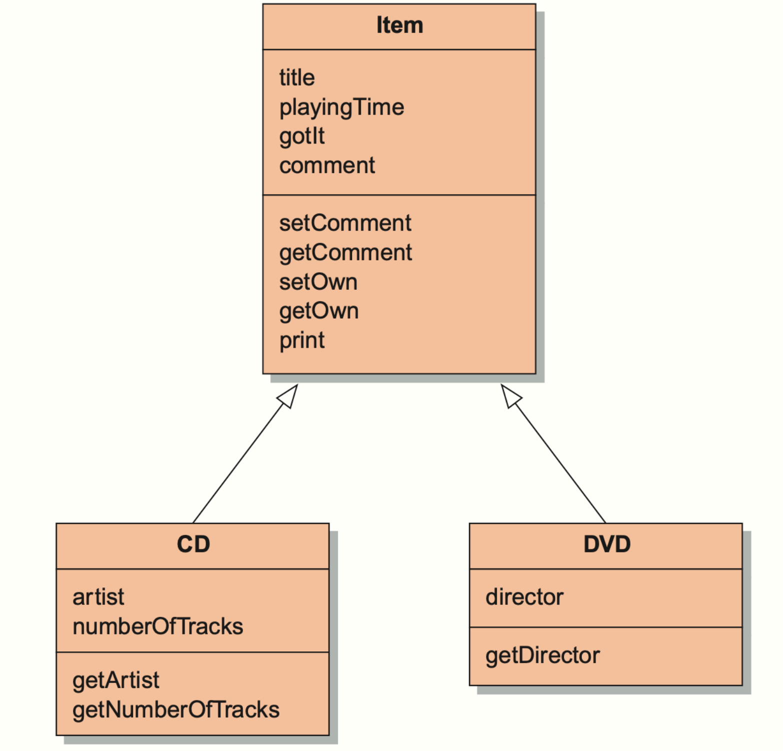
Critique of DoME

- code duplication
  - CD and DVD classes very similar (large part are identical)
  - makes maintenance difficult/more work
  - introduces danger of bugs through incorrect maintenance
- code duplication also in Database class

Discuss

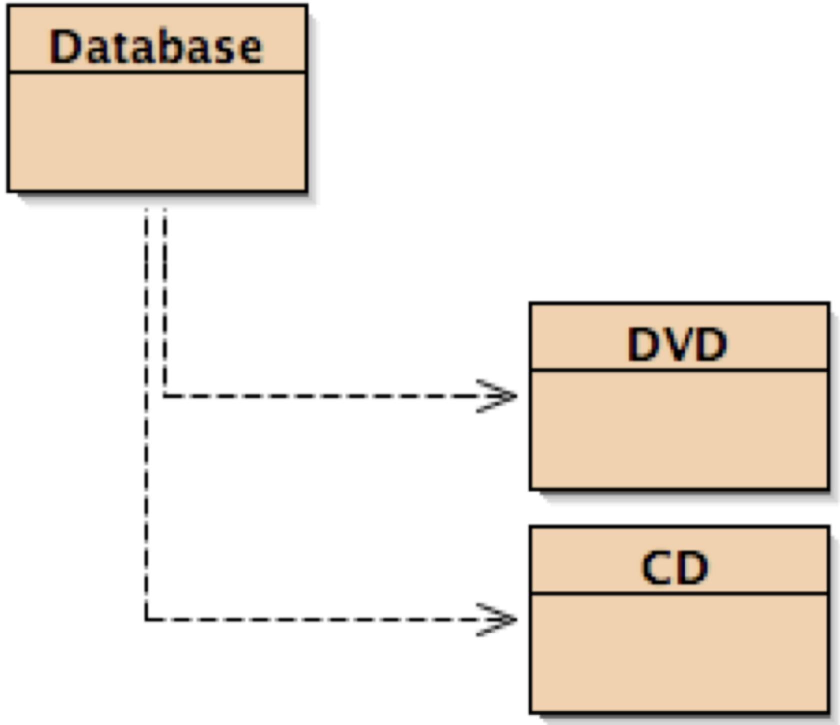
- The CD and DVD classes are very similar. In fact, the majority of the classes' source code is identical, with only a few differences
- In the Database class.We can see that everything in that class is done twice – once for CDs and once for DVDs
- What if we'd add new types of media?

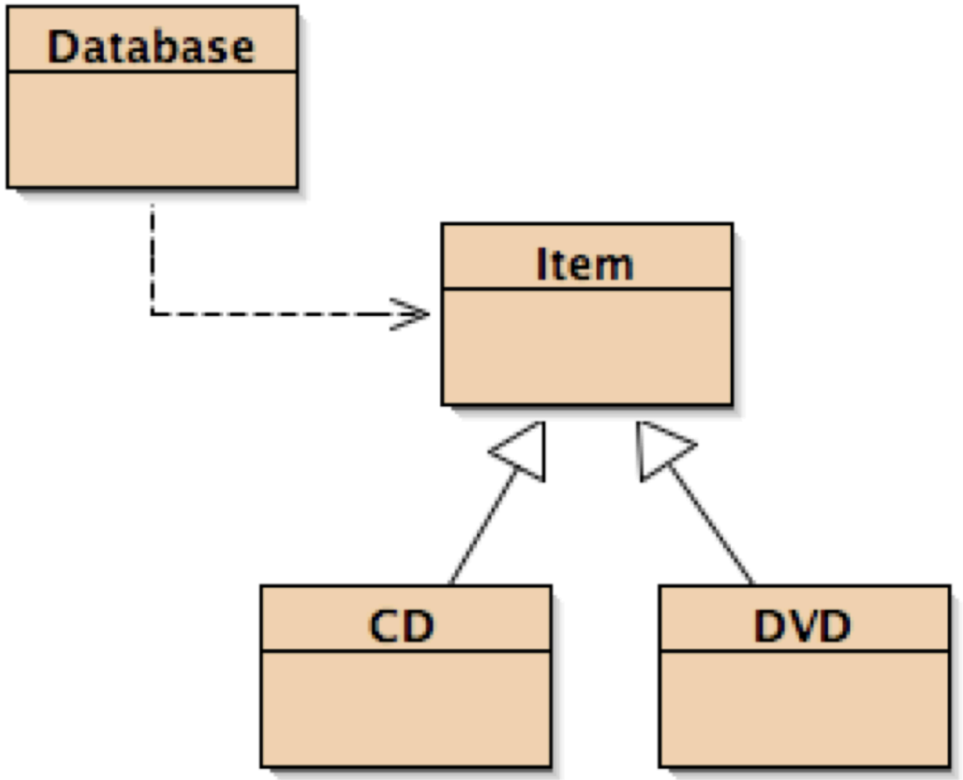
Solution -- Inheritance



- Inheritance allows us to define one class as an extension of another.

Class diagram

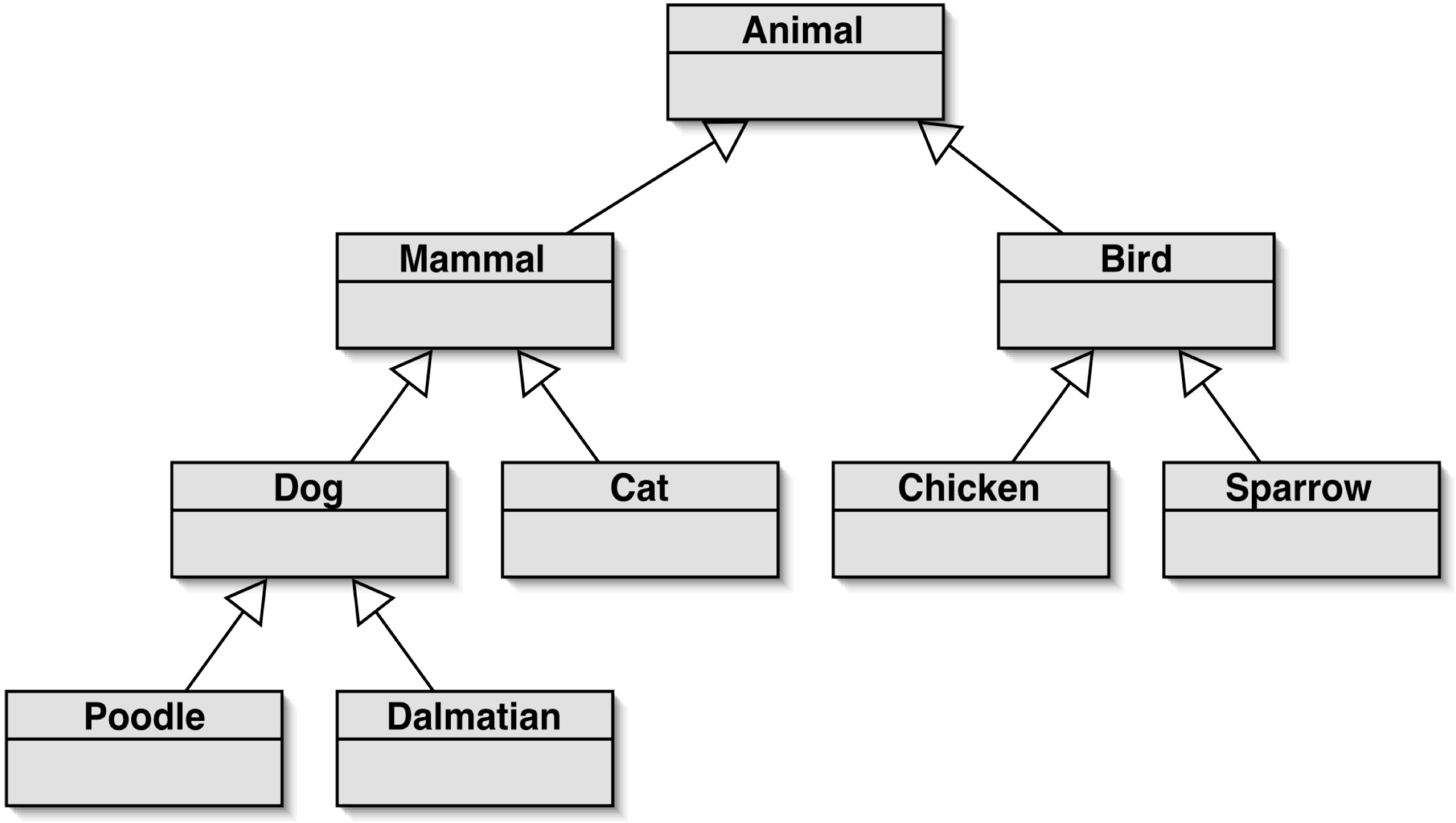




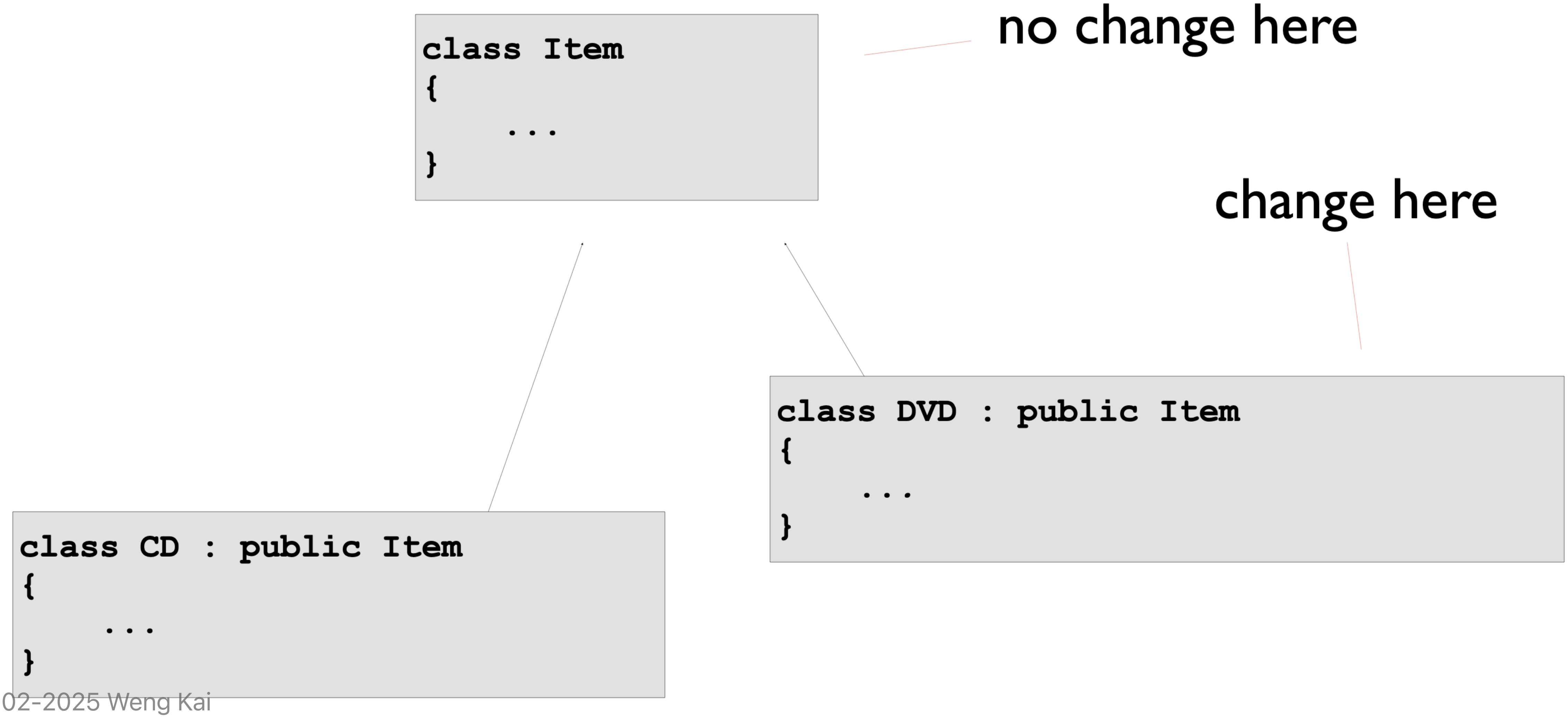
Using inheritance

- define one superclass : `Item`
- define subclasses for `CD` and `DVD`
- the superclass defines common attributes
- the subclasses inherit the superclass attributes the subclasses add own attributes

Inheritance hierarchies



Inheritance



```
class Item
{
     ...
}
```

no change here

change here

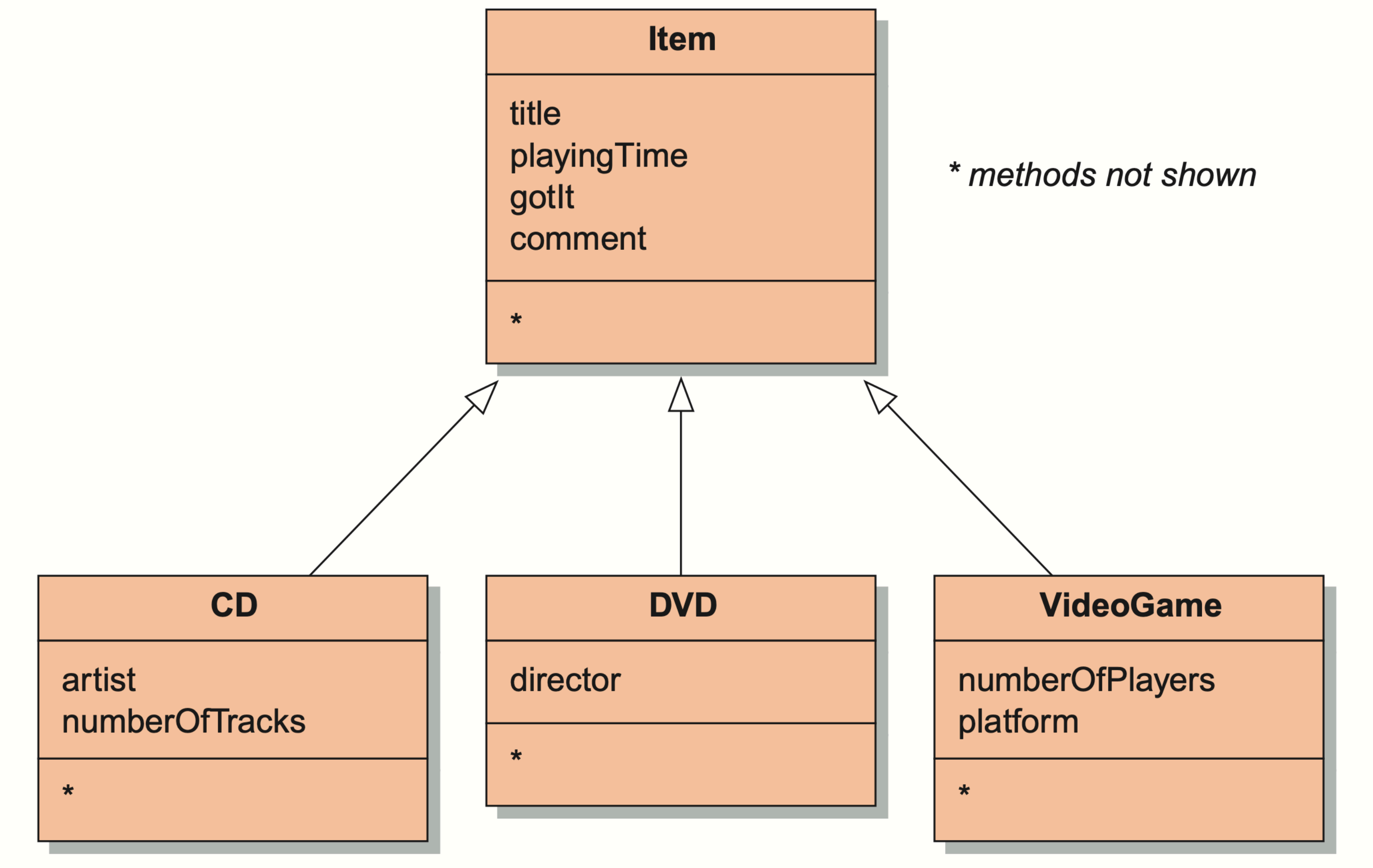
```
class DVD : public Item
{
     ...
}
```

```
class CD : public Item
{
     ...
}
```

Database v2.0

```
    ...
    public void addItem(Item theItem) {
        items.add(theItem);
    }
    /**
    * Print a list of all currently stored items to the text terminal.
    */
    public void list() {
        for(auto item : items) {
            item.print();
    }
}
```

Adding other item types

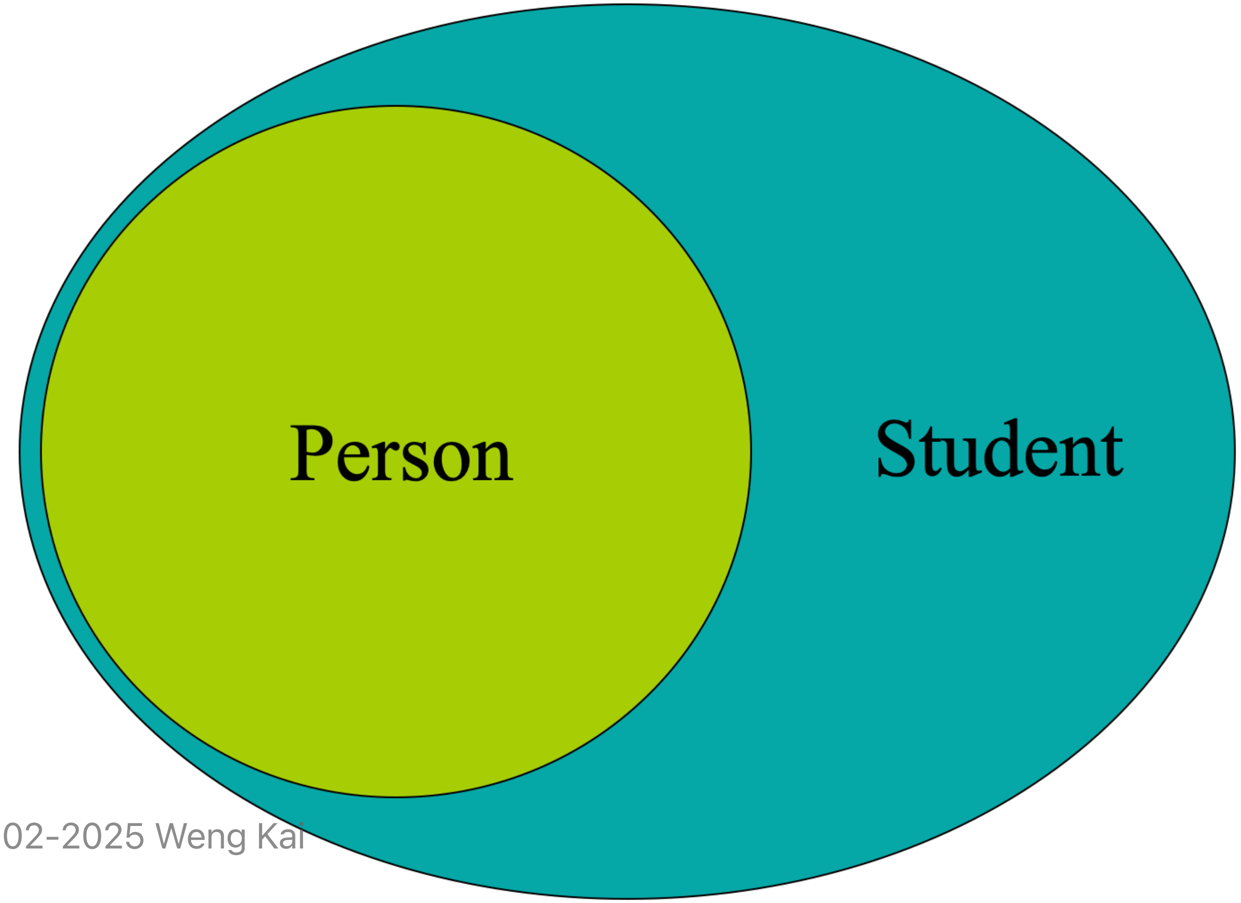


Advantages of inheritance

- Avoiding code duplication
- Code reuse
- Easier maintenance
- Extendibility

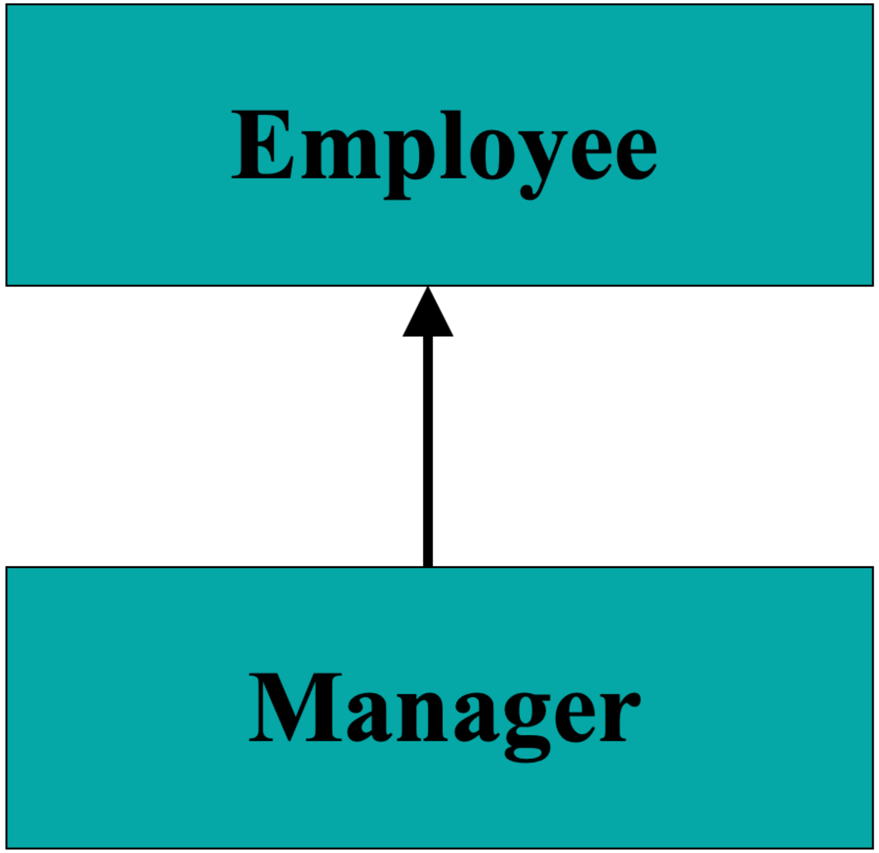
Inheritance

The ability to define the behavior or implementation of one class as a superset of another class



Inheritance

- Class relationship: Is-A



Employee

Manager

Base Class
Super
Parent

Derived Class
Sub
Child

What does it inherited?

- (private) member variables
- public member functions
- private member functions
- protected members
- static members

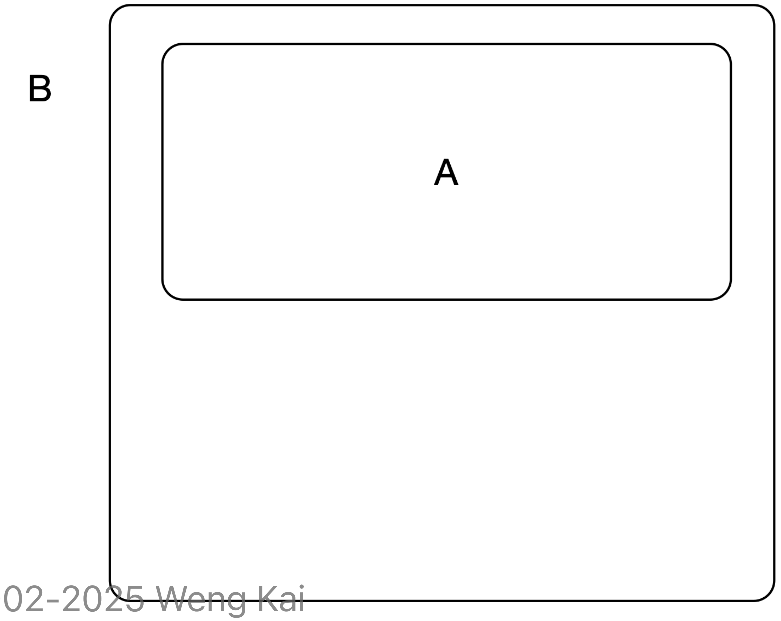
Private Member Variables

- There is the object of the super class right there inside the object of the derived class
- with all the member variables in
- but the derived one does NOT have access to those variables
- have to use those via member functions of the super class
- If the derived one has a varible as the same name, it is an isolated new one

Derived-Class Objects and the Derived-to-Base Conversion

- A derived object contains multiple parts: a subobject containing the (nonstatic) members defined in the derived class itself, plus subobjects corresponding to each base class from which the derived class inherits

```
class A...
class B:public A...
```



Public Member Functions

- They are public member functions of the derived class
- They defined the interface of the class

All objects of a peticular class can receive the same messages. -- by Alan Kay

Private Member Functions

- They are NOT accessible in the derived class

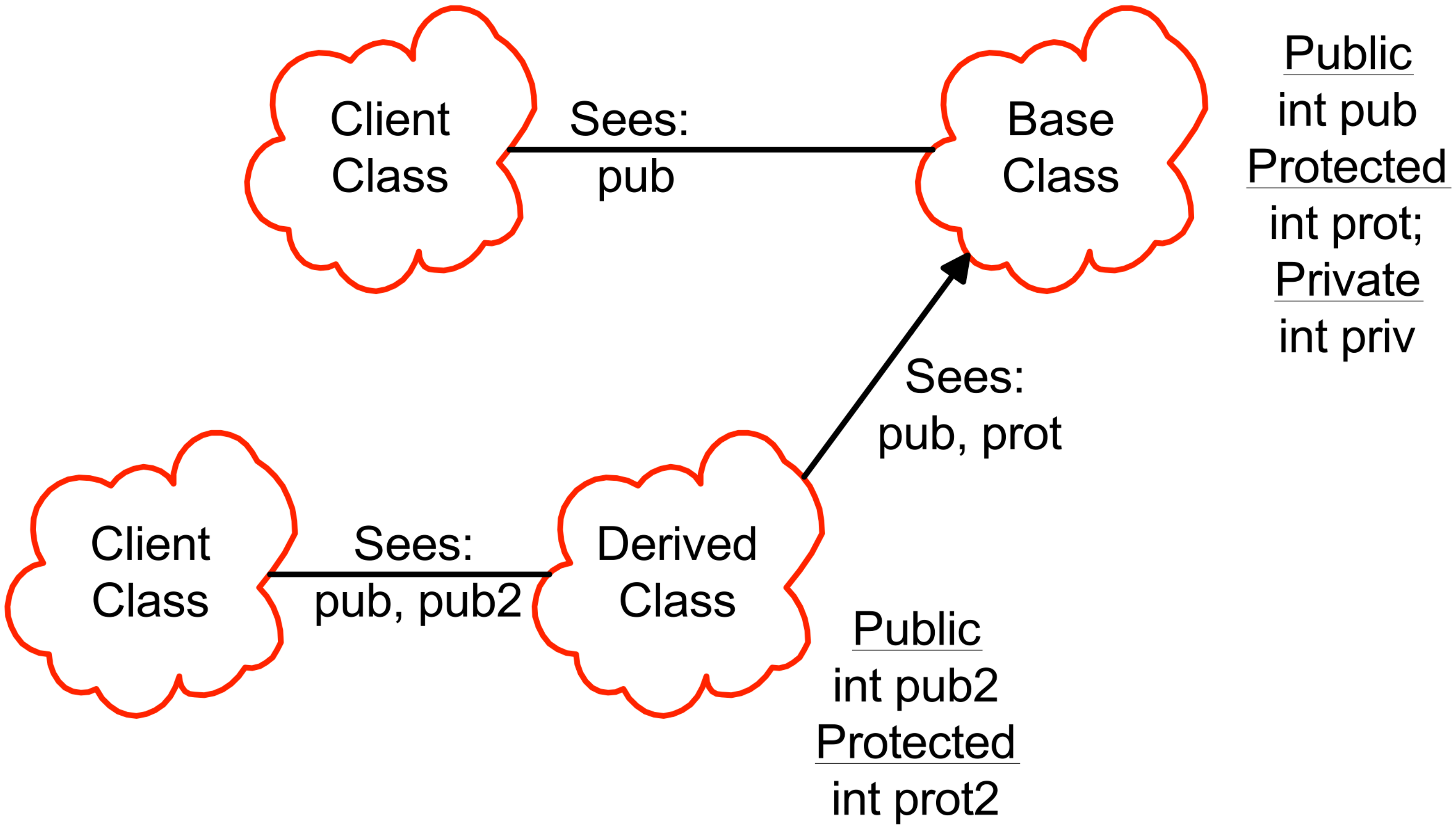
Protected Members

- They are fully accessible in the derived class

Static Members

- They are still class-wide members

Scopes and access in C++



Client Class

Sees: pub

Base Class

Public
int pub
Protected
int prot;
Private
int priv

Sees: pub, prot

Client Class

Sees: pub, pub2

Derived Class

Public
int pub2
Protected
int prot2

Declare an `Employee` class

```
class Employee {
public:
    Employee( const std::string& name, const std::string& ssn );
    const std::string& get_name() const;
    void print(std::ostream& out) const;
    void print(std::ostream& out, const std::string& msg) const;
protected:
    std::string m_name;
    std::string m_ssn;
};
```

Constructor for Employee

```
Employee::Employee( const string& name, const string& ssn )
    : m_name(name), m_ssn( ssn) {
    // initializer list sets up the values!
}
```

Employee member functions

```cpp
inline const std::string& Employee::get_name() const {
    return m_name;
}
inline void Employee::print( std::ostream& out ) const {
    out << m_name << endl;
    out << m_ssn << endl;
}
inline void Employee::print(std::ostream& out, const std::string& msg) const {
    out << msg << endl;
    print(out);
}
```

Now add Manager

```cpp
class Manager : public Employee {
public:
    Manager(const std::string& name, const std::string& ssn, const std::string& title);
    const std::string title_name() const;
    const std::string& get_title() const;
    void print(std::ostream& out) const;
private:
    std::string m_title;
};
```

Inheritance and constructors

- Think of inherited traits as an embedded object
- Base class is mentioned by class name

```
Manager::Manager( const string& name, const string& ssn, const string& title = "" )
    :Employee(name, ssn), m_title( title ) {
}
```

More on constructors

- Base class is always constructed first
- If no explicit arguments are passed to base class
  - Default constructor will be called
- Destructors are called in exactly the reverse order of the constructors.

继承构造函数

- 类具有可派生性，派生类自动获得基类的成员变量和接口（虚函数和纯虚函数）
- 基类的构造函数也没有被继承，因此：

```
class A {
public:
    A(int i) {}
};

class B : public A {
public:
    B(int i): A(i), d(i) {}
private:
    int d;
};
```

- B的构造函数起到了传递参数给A的构造函数的作用：透传
- 如果A具有不只一个构造函数，B往往需要设计对应的多个透传

`using`声明

- 派生类用`using`声明来使基类的成员函数成为自己的
    - 解决name hiding问题：非虚函数被`using`后成为派生类的函数
    - 解决构造函数重载问题

```cpp
class Base {
public:
    void f(double ) {
        cout << "double\n";
    }
};

class Derived : Base { //不是public继承
public:
    using Base::f;
    void f(int ) {
        cout << "int\n";
    }
};

int main()
{
    Derived d;
    d.f(4);
    d.f(4.5);
}
```

3-3.cpp

```cpp
class A {
public:
    A(int i) { cout << "int\n"; }
    A(double d, int i) {}
    A(float f, char *s) {}
};

class B : A {
public:
    using A::A;
};

int main()
{
    B b(2);
}
```

- 继承构造函数是隐式声明的，如果没有用到就不产生代码

```
g++ 3-4.cpp --std=c++11
```

- 如果基类的函数具有默认参数值，`using` 的派生类无法得到默认参数值，就必须转为多个重载的函数

```
class A {
public:
    A(int a=3, double b=2.4) {}
};
```

- 实际上可以被看作是：

```
A(int, double);
A(int);
A();
```

- 那么，被 `using` 之后就会产生相应的多个函数

Manager member functions

```cpp
inline void Manager::print( std::ostream& out ) const {
    Employee::print( out );      // call the base class print
    out << m_title << endl;
}
inline const std::string& Manager::get_title() const {
    return m_title;
}
inline const std::string Manager::title_name() const {
    return string( m_title + ": " + m_name );  // access base m_name
}
```

Uses

```
int main () {
    Employee bob( "Bob Jones", "555-44-0000" );
    Manager bill( "Bill Smith", "666-55-1234", "Important Person" );

    string name = bill.get_name();      // okay Manager inherits Employee
    //string title = bob.get_title();   // Error -- bob is an Employee!
    cout << bill.title_name() << '\n' << endl;
    bill.print(cout);
    bob.print(cout);
    bob.print(cout, "Employee:");
    //bill.print(cout, "Employee:");    // Error hidden!
```

Name Hiding

- If you redefine a member function in the derived class, all other overloaded functions in the base class are inaccessible.
- We'll see how the keyword `virtual` affects function overloading next time.

What is not inherited?

- Constructors
  - synthesized constructors use memberwise initialization
  - In explicit copy ctor, explicity call base-class copy ctor or the default ctor will be called instead.
- Destructors
- Assignment operation
  - synthesized operator= uses memberwise assignment
  - explicit operator= be sure to explicity call the base class version of operator=
- Private data is hidden, but still present

Access protection

- Members
  - Public: visible to all clients
  - Protected: visible to classes derived from self (and to friends)
- Private: visible only to self and to friends!
- Inheritance
  - Public: `class Derived : public Base ...`
  - Protected: `class Derived : protected Base ...`
  - Private: `class Derived : private Base ...`
    - default

How inheritance affects access

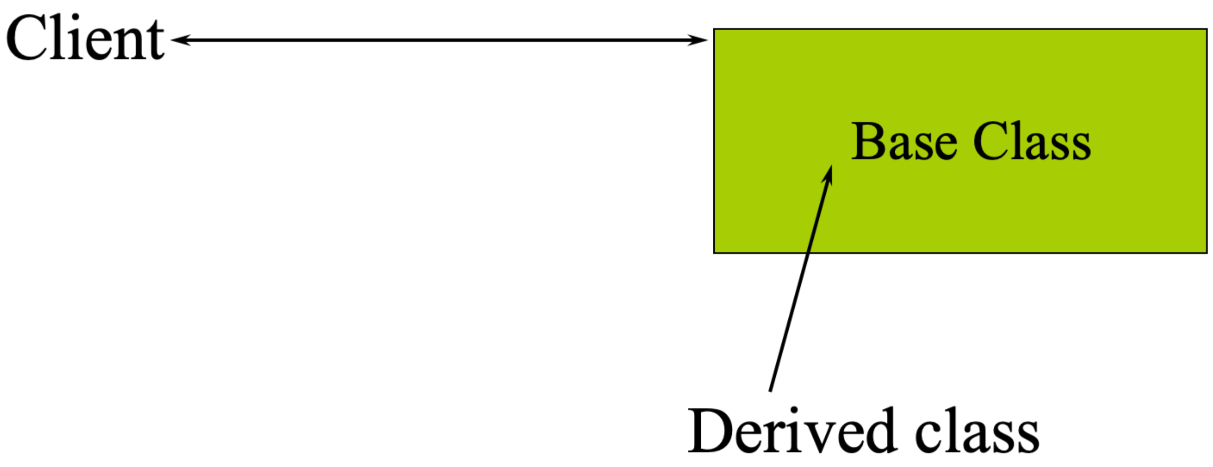
- Suppose class `B` is derived from `A`. Then

Base class member access specifier

Inheritance Type (B is)	**`public`**	**`protected`**	**`private`**
`public A`	public in B	protected in B	hidden
`private A`	private in B	private in B	hidden
`protected A`	protected in B	protected in B	hidden

When is protected not protected?

- When your derived classes are ill-behaved!
- Protected is public to all derived classes
- For this reason
  - make member functions protected
  - keep member variables private



What we've learned?

- inheritance