# Object – Oriented Programming

Week 7

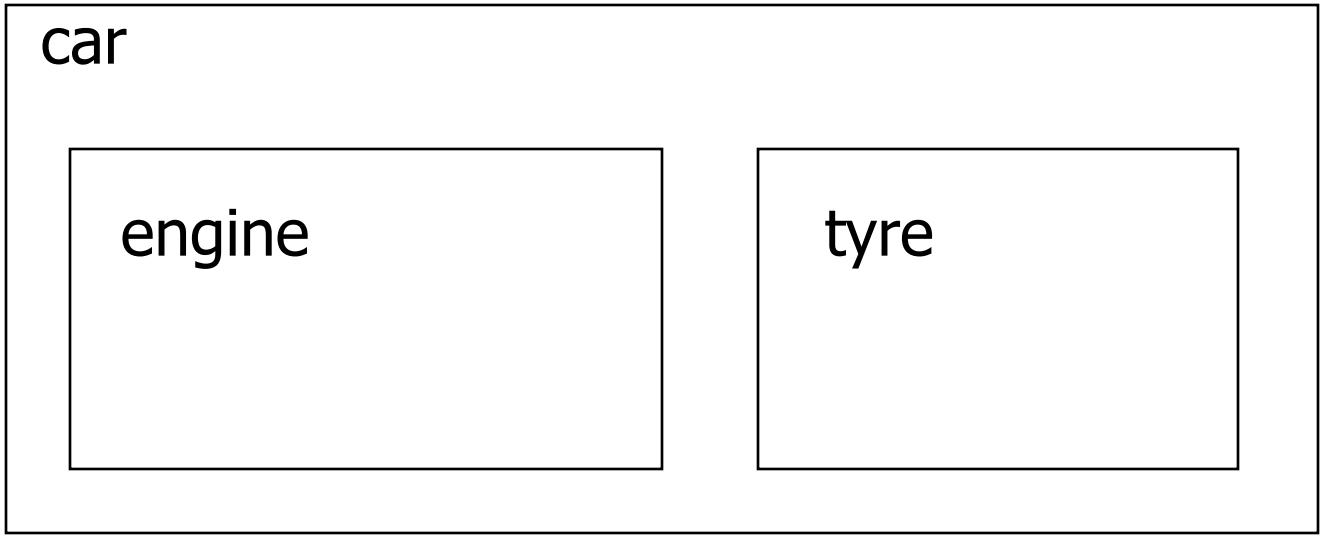
# Inheritance

Weng Kai

# Inheritance

# Reusing the implementation

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



# Composition

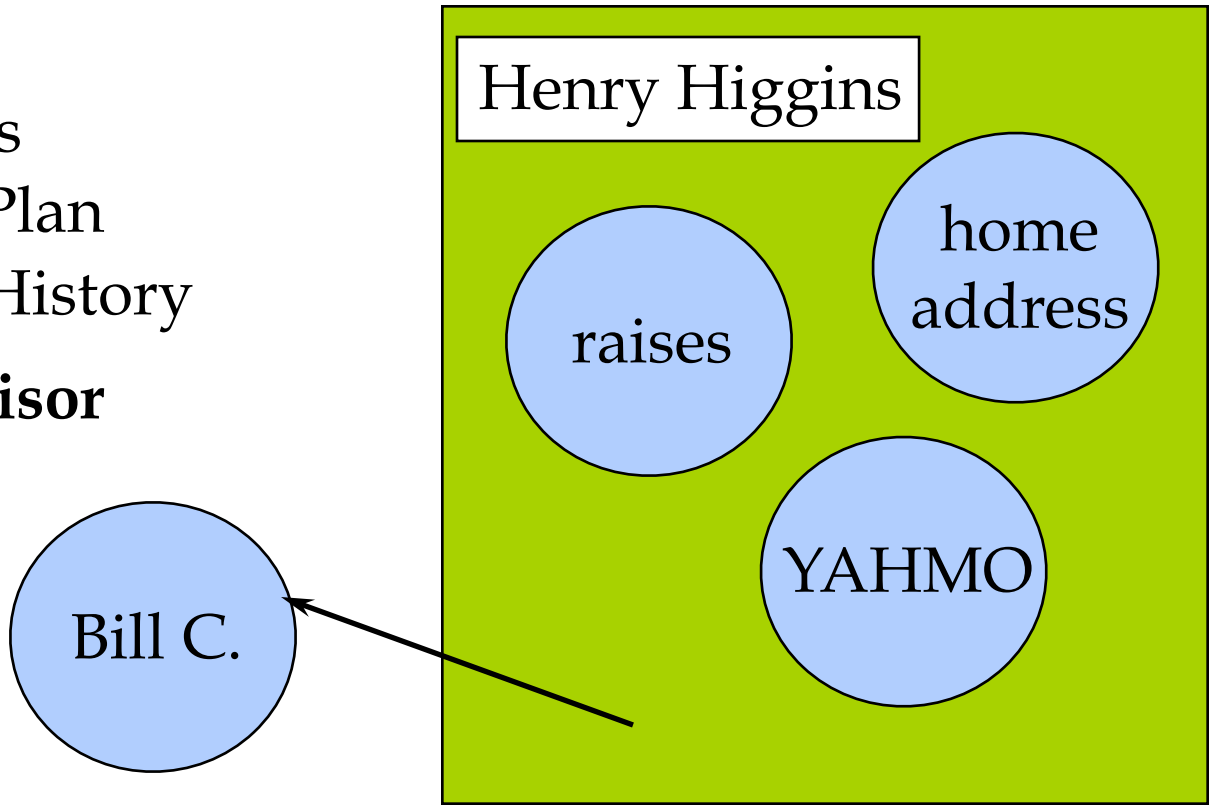
- Objects can be used to build up other objects
- Ways of inclusion
  - Fully
  - By reference
- Inclusion by reference allows sharing
- For example, an Employee has a
  - Name
  - Address
  - Health Plan
  - Salary History
    - Collection of Raise objects
  - Supervisor
    - Another Employee object!

# Composition in action

**Classes**

Employe
e
- Name
- Address
- HealthPlan
- Salary History
- **Supervisor**

**Instances**



# Example

```
class Person { ... };
class Currency { ... };
class SavingsAccount {
public:
   SavingsAccount( const char* name,
                   const char* address, int cents );
   ~SavingsAccount();
   void print();
private:
   Person m_saver;
   Currency m_balance;
};
```

# Example...

```
SavingsAccount::SavingsAccount ( const
  char* name, const char* address,
  int cents ) : m_saver(name, address),
  m_balance(0, cents) {}

void SavingsAccount::print() {
   m_saver.print();
   m_balance.print();
}
```

# Embedded objects

- All embedded objects are initialized
  - The default constructor is called if
    - you don't supply the arguments, and there is a default constructor (or one can be built)
- Constructors can have initialization list
  - any number of objects separated by commas
  - is optional
  - Provide arguments to sub-constructors
- Syntax:

```
name( args ) [':' init-list] '{'
```

# Question

- If we wrote the constructor as (assuming we have the set accessors for the sub-objects):

```
SavingsAccount::SavingsAccount ( const char* name,
   const char* address, int cents ) {
      m_saver.set_name( name );
      m_saver.set_address( address );
      m_balance.set_cents( cents );
}
```

- Default constructors would be called

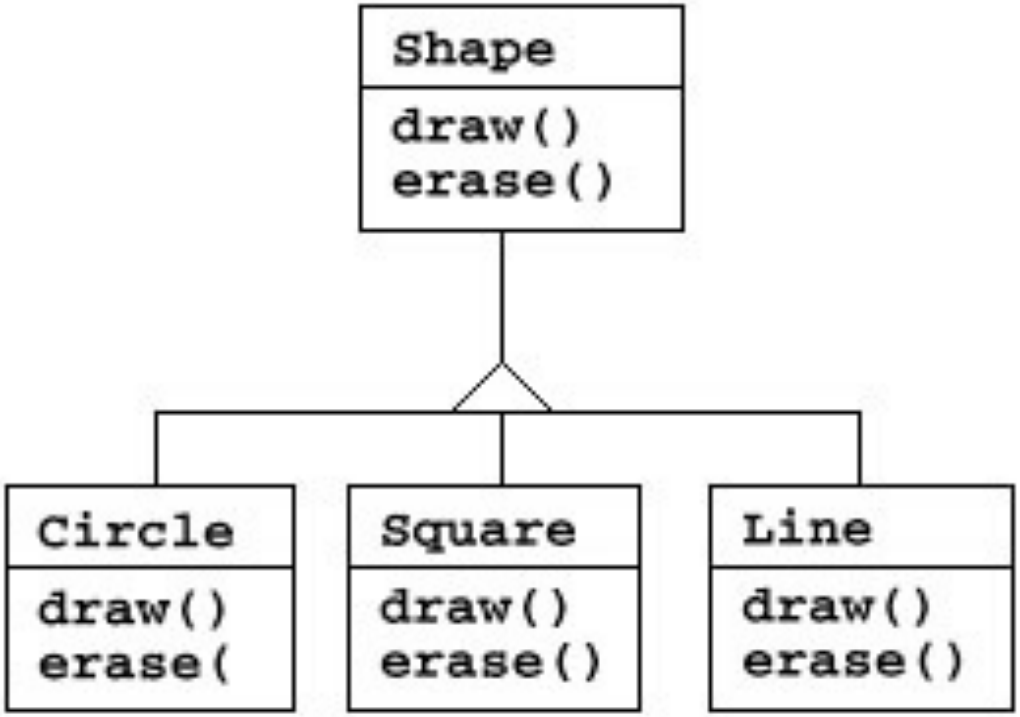
# Public vs. Private

- It is common to make embedded objects private:
  - they are part of the underlying implementation
  - the new class only has part of the public interface of the old class
- Can embed as a public object if you want to have the entire public interface of the subobject available in the new object:

```
class SavingsAccount {
 public:
         Person m_saver;  ... };     // assume
  Person class has set_name()
SavingsAccount  account;
account.m_saver.set_name("Fred" );
```

# Reusing the interface

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

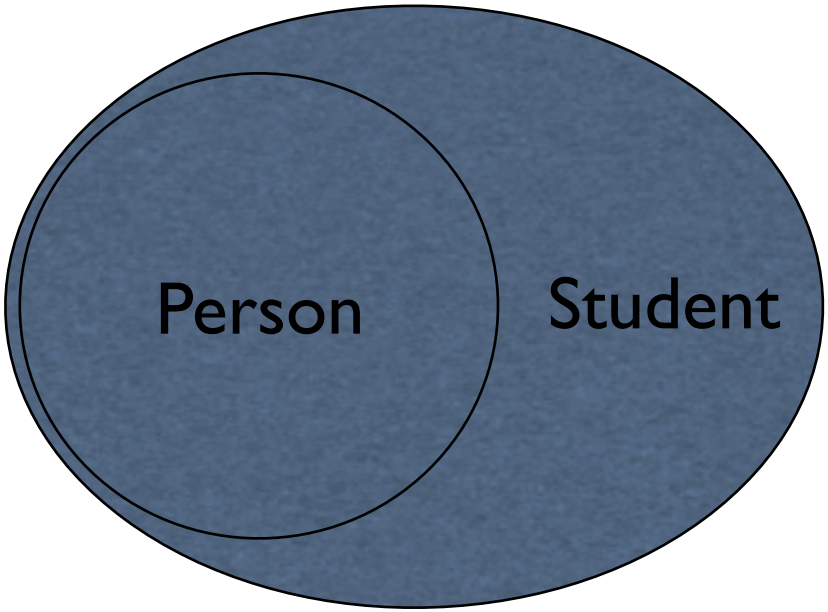


# Inheritance

- 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

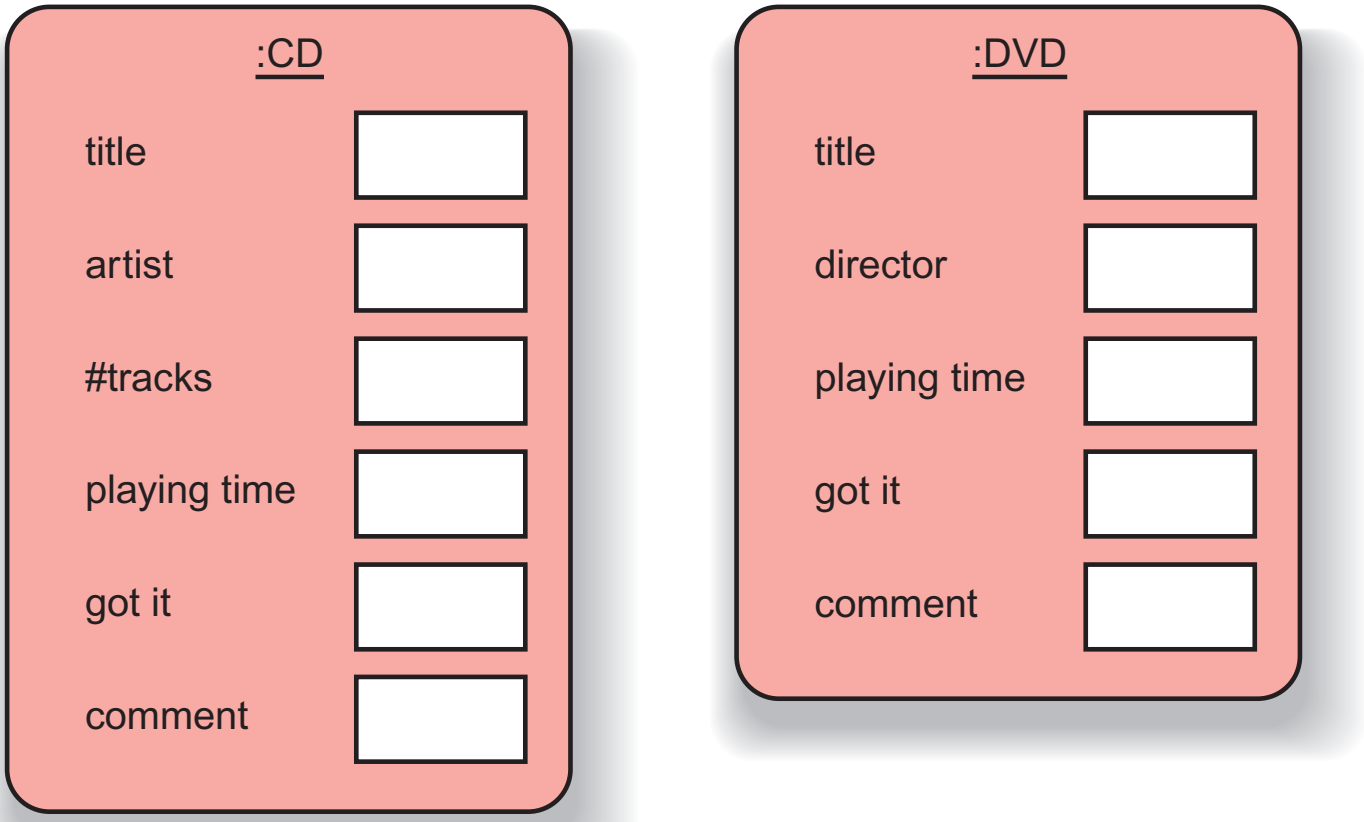
- the title of the DVD;
- the name of the director;
- the playing time (we define this as the playing time of the main feature);
- a ‘got it’ flag that indicates whether I own a copy of this DVD; and
- a comment (some arbitrary text)

# The DoME example

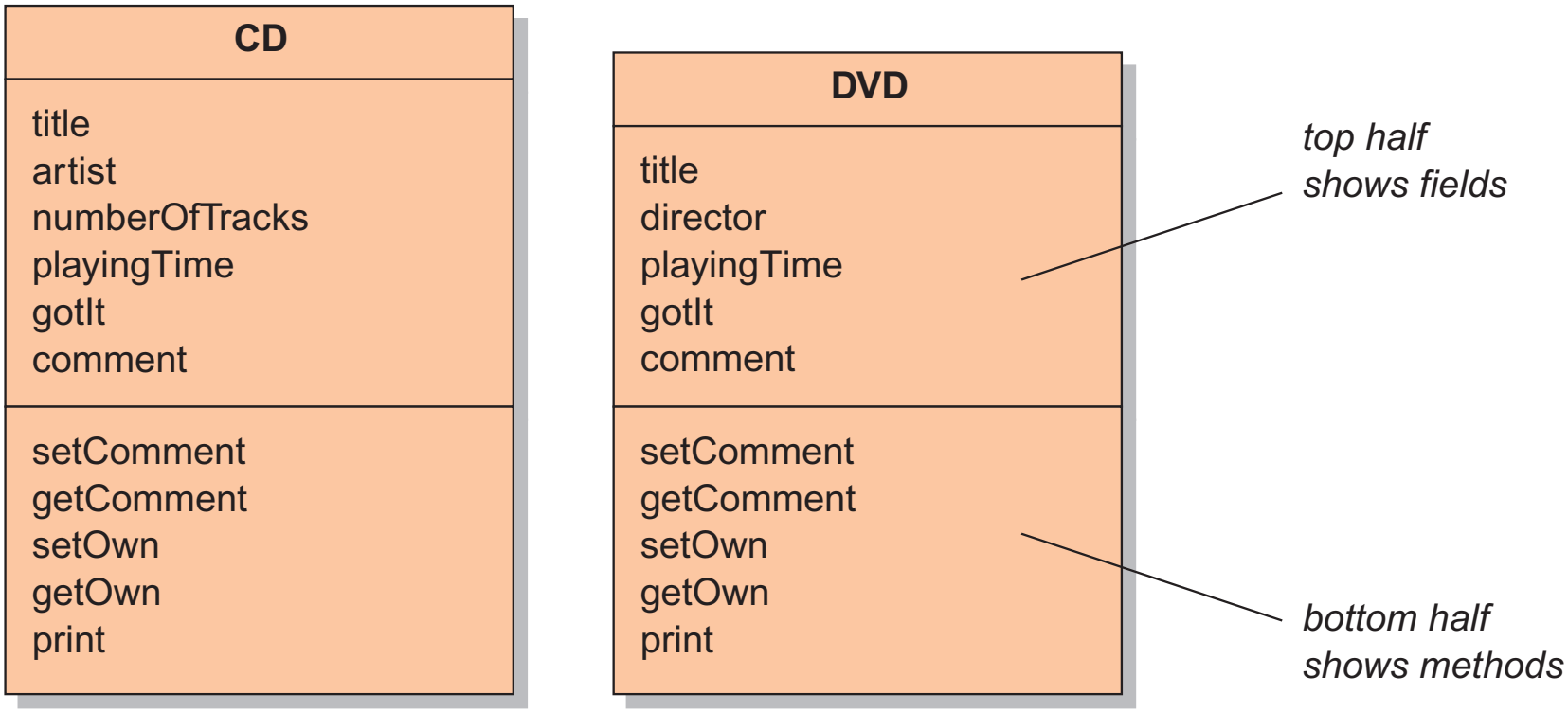
"Database of Multimedia Entertainment"

- 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

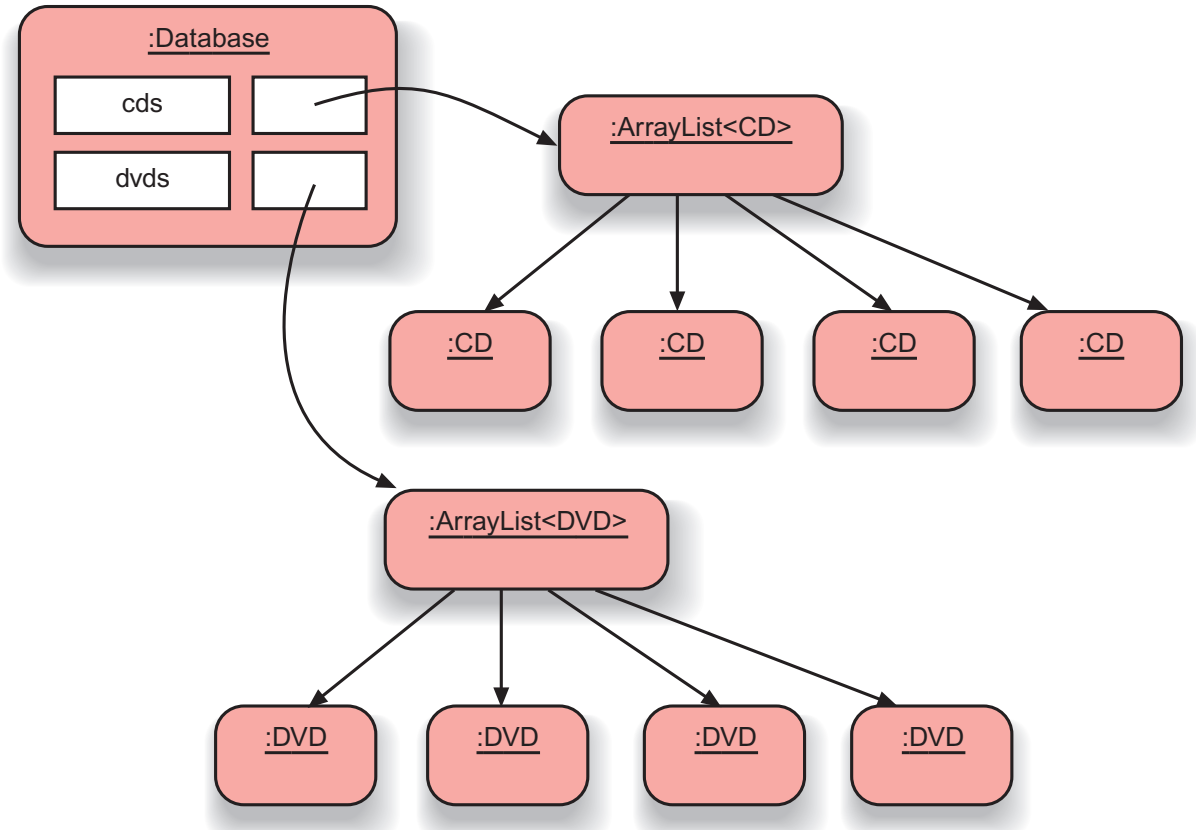
# DoME classes



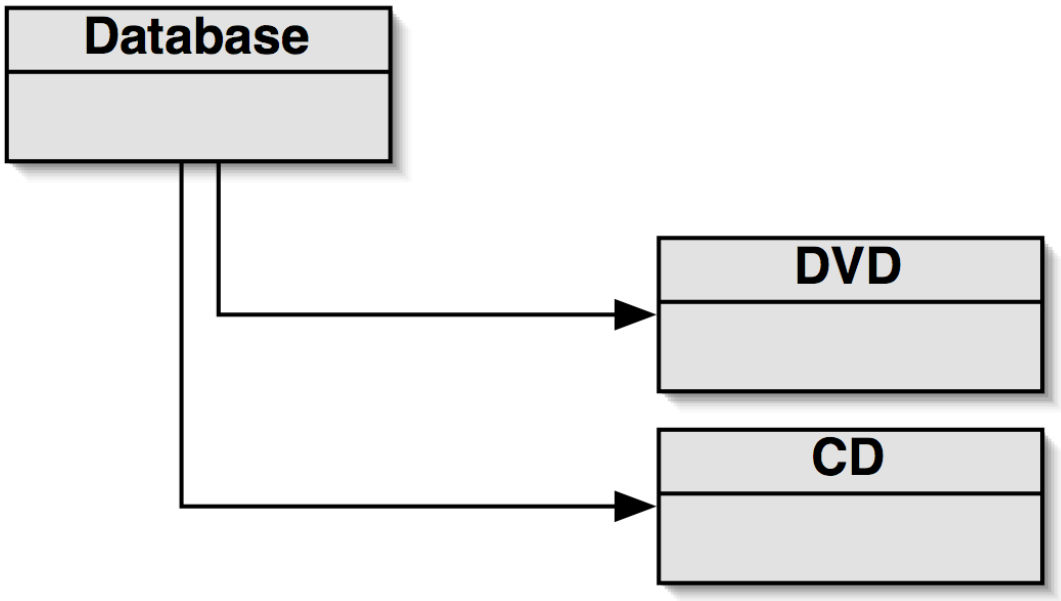
# Class diagram



# Object Model



# Class diagram



# source code

```
public class Database
{
    private ArrayList<CD> c
    private ArrayList<DVD>
```

```
public void addCD(CD theCD)
{
    cds.add(theCD);
}
```

```
public void addDVD(DVD theDVD)
{
    dvds.add(theDVD);
}
```

```
public void list()
{
    // print list of CDs
    for(CD cd : cds) {
        cd.print();
        System.out.println();
    }

    // print list of DVDs
    for(DVD dvd : dvds) {
        dvd.print();
        System.out.println();
    }
}
```

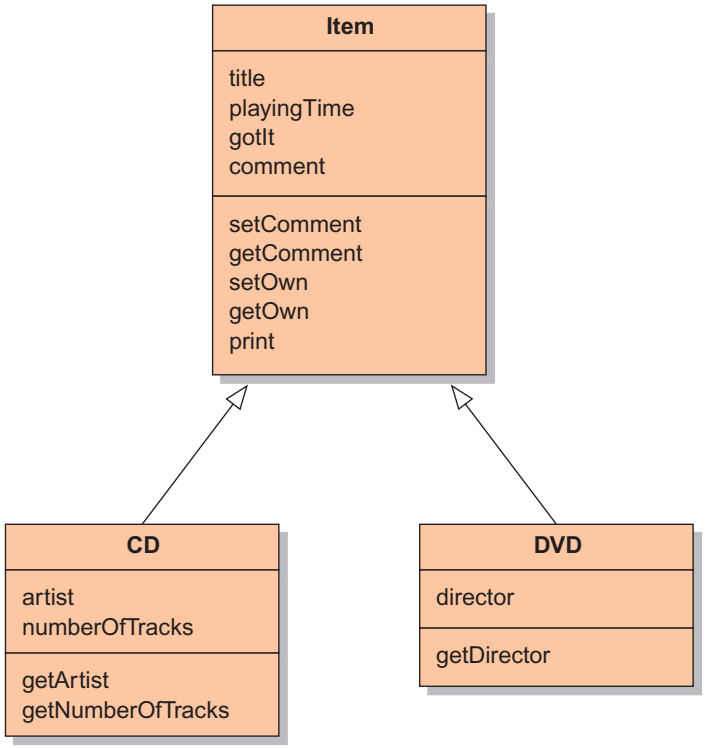
# Critique of DoME

- 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

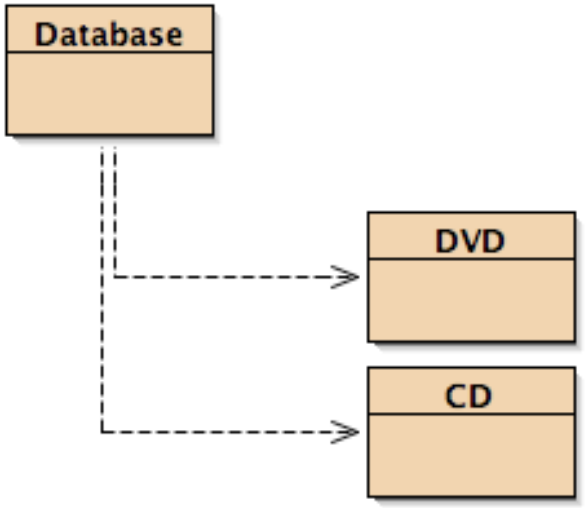
- The CD and DVD classes are very similar. In fact, the majority of the classes' source code is identical, with only a few differences
- In the Database class. We can see that everything in that class is done twice – once for CDs and once for DVDs
  - What if we'd add new types of media?

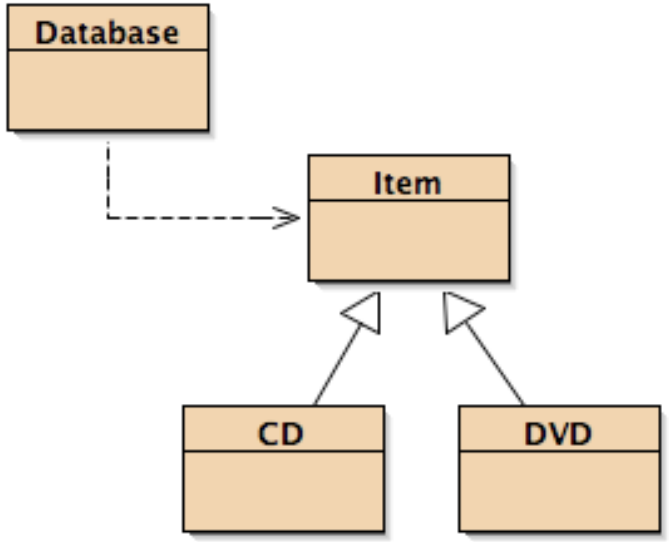
# Solution - inheritance




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

# Class diagram

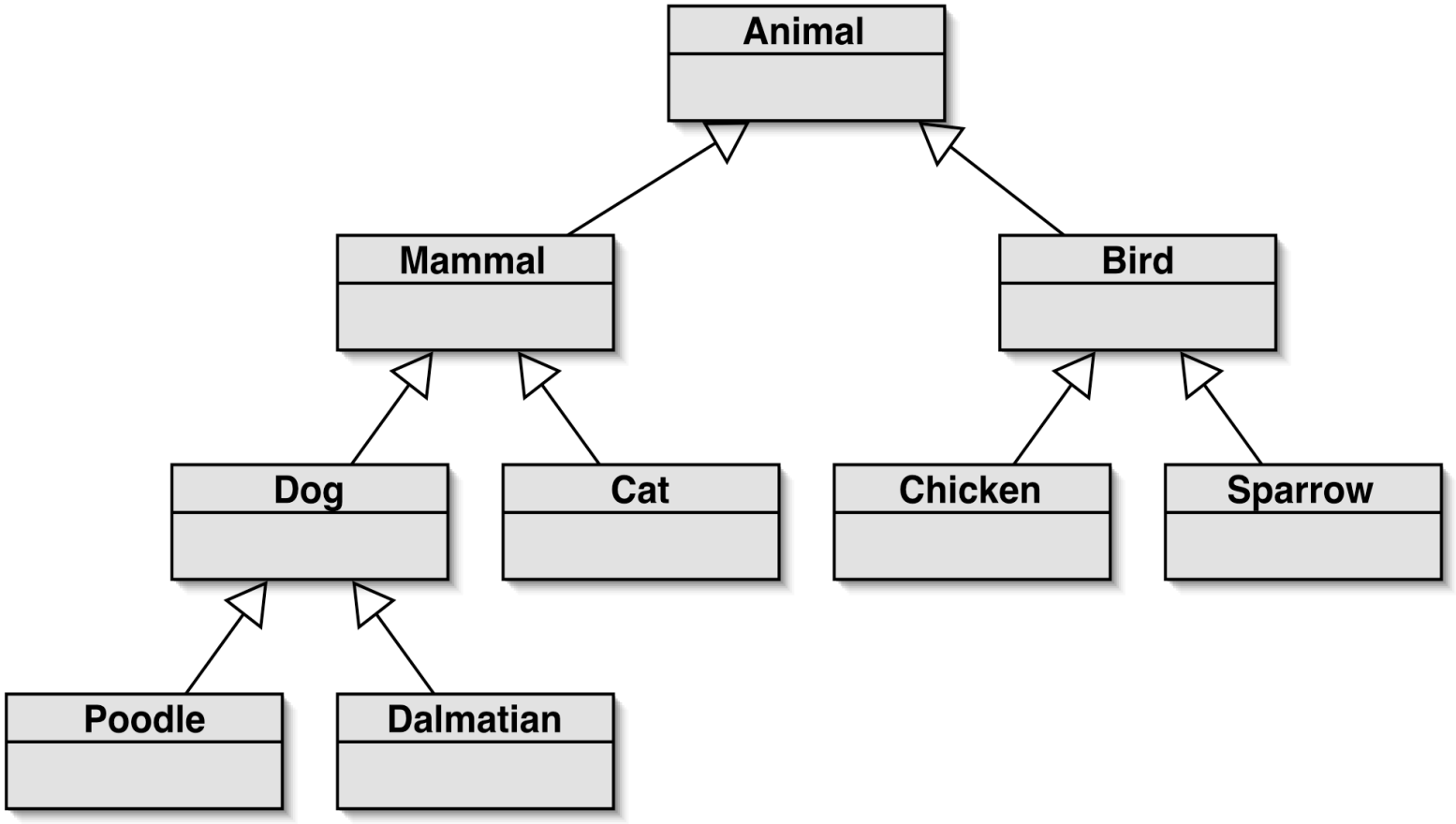


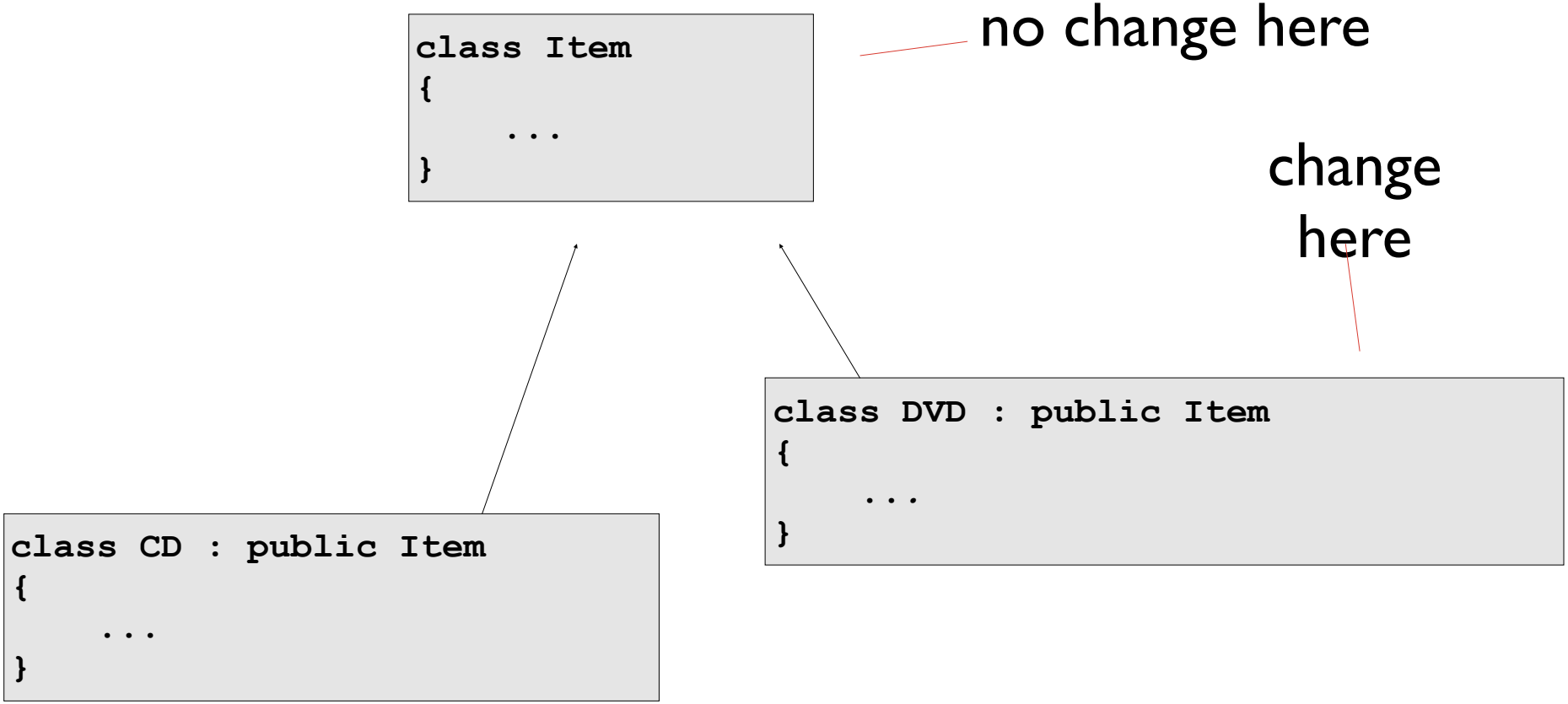
# Using inheritance

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

# Inheritance hierarchies



# Inheritance



# 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(Item item : items) {
        item.print();
        System.out.println();    // empty line between items
    }
}
```

```
public void addCD(CD theCD)
{
    cds.add(theCD);
}
```

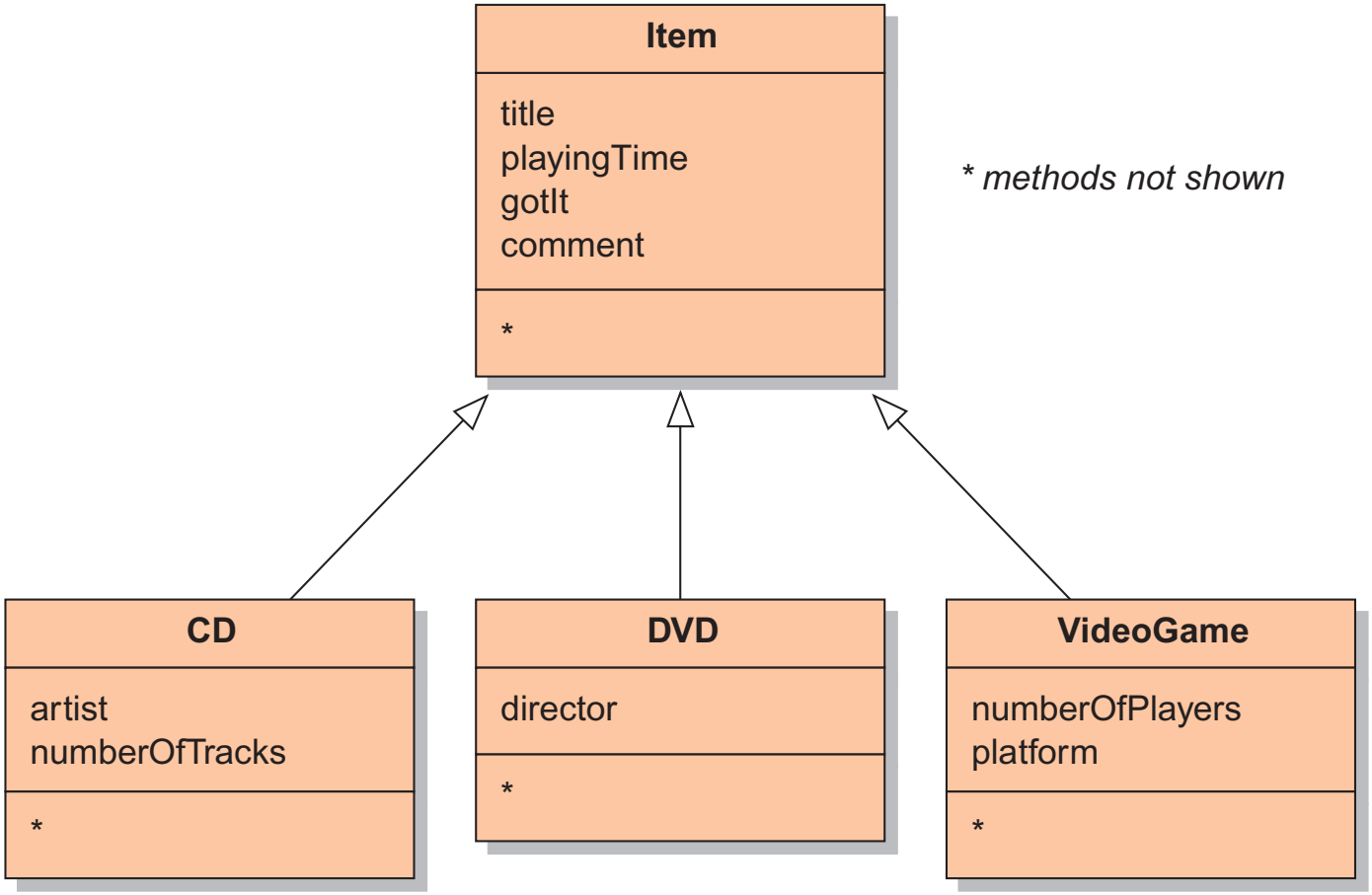
```
public void addItem(Item theItem)
{
    items.add(theItem);
}
```

```
public void addDVD(DVD theDVD)
{
    dvds.add(theDVD);
}
```

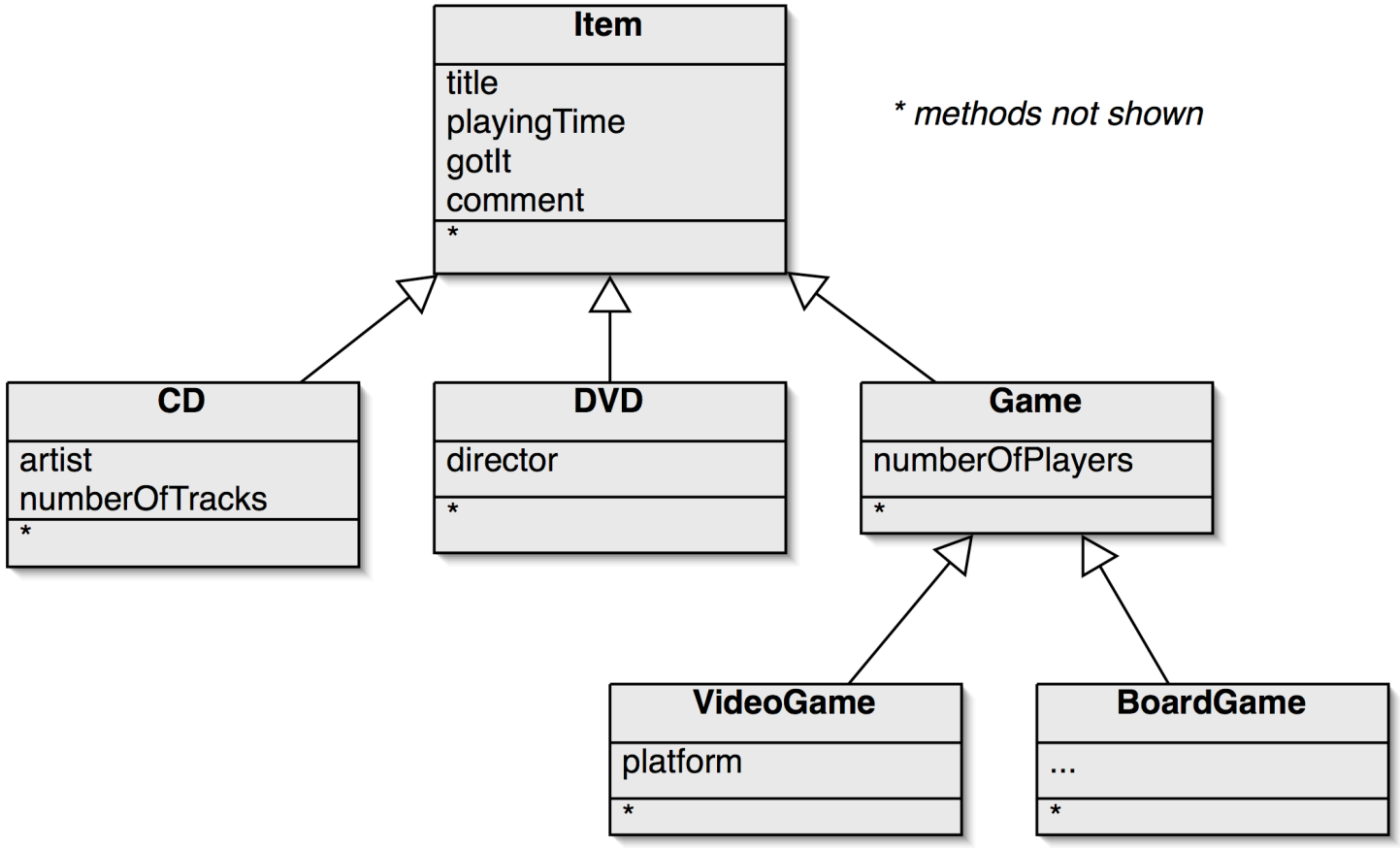
```
public void list()
{
    // print list of CDs
    for(CD cd : cds) {
        cd.print();
        System.out.pri
    }

    // print list of DV
    for(DVD dvd : dvds)
        dvd.print();
        System.out.println();
    }
}
```

```
public void list()
{
    for(Item item : items) {
        item.print();
        System.out.println();
    }
}
```

# Adding other item types



# Deeper hierarchies



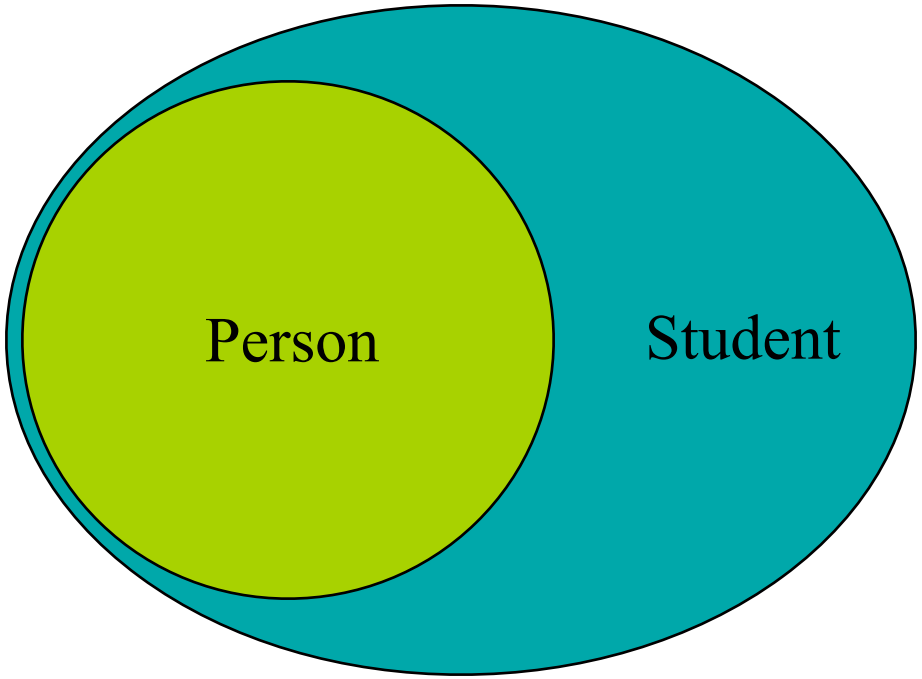
# Advantages of inheritance

- Avoiding code duplication
- Code reuse
- Easier maintenance
- Extensibility 可扩展性

# Inheritance

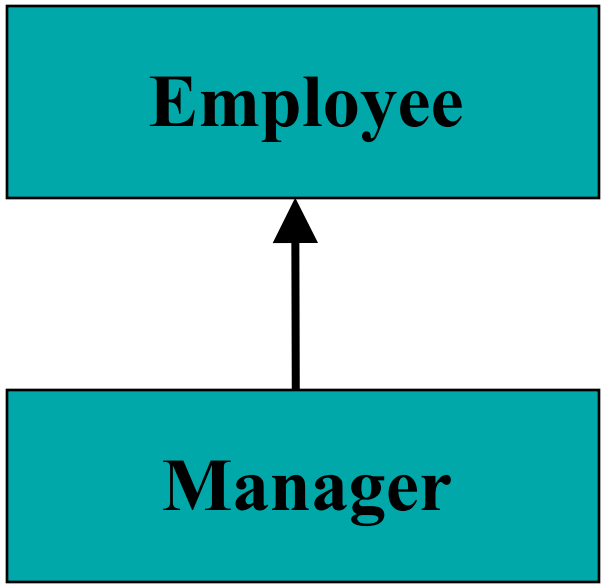
- The ability to define the behavior or implementation of one class as a ***superset*** of another class

将一个类的行为或实现定义为另一个类的超集的能力



# Inheritance

- **Class relationship: Is-A**



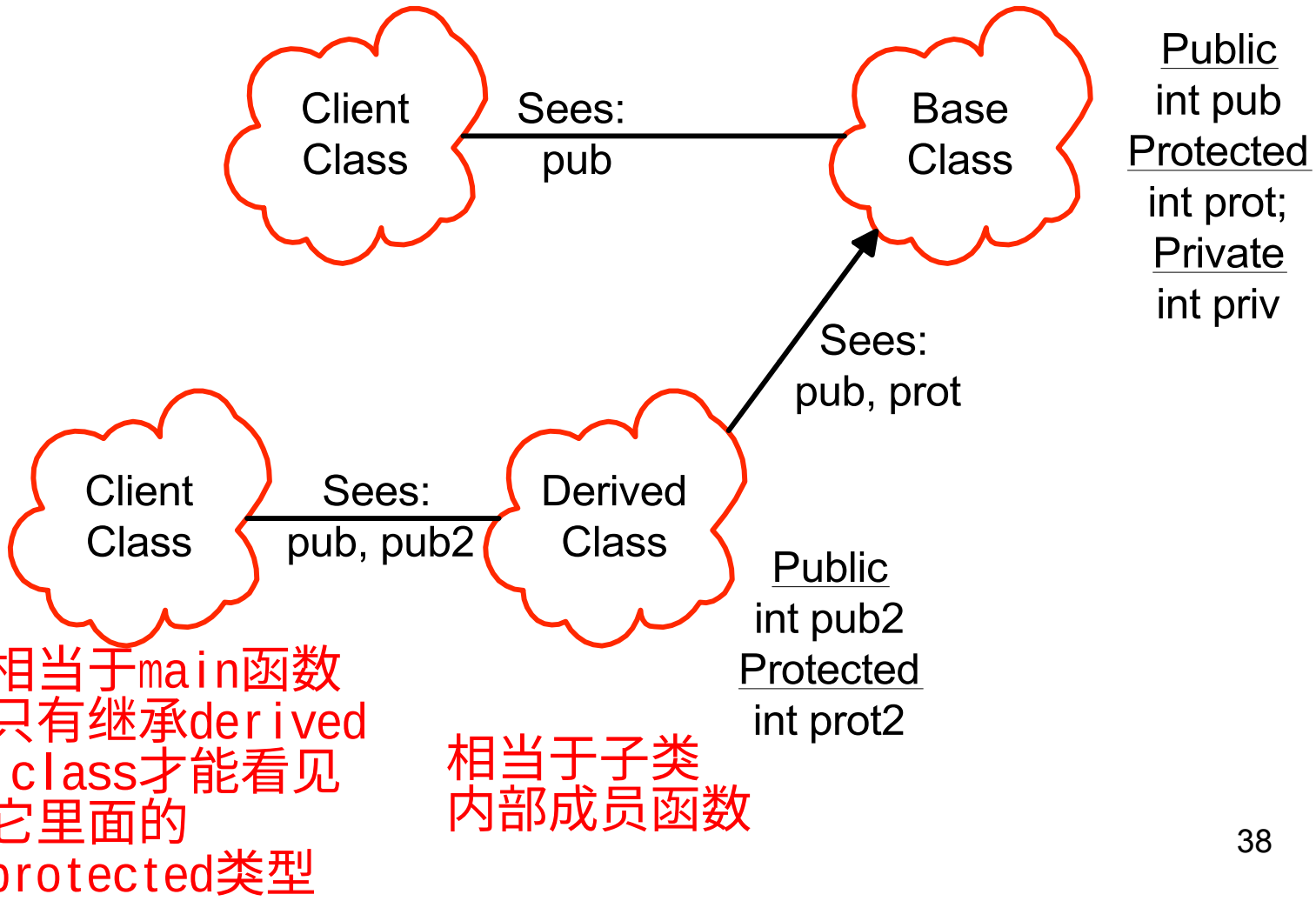

**Base Class** 基类 超类 父类
**Super**
**Parent**

**Derived Class** 派生类 子类
**Sub**
**Child**

manager is an Employee

范围

# Scopes and access in C++



# 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

```
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

```
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.

# Manager member functions

```
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!
}
```

子类没有重新定义父类中全部重载的函数。重新定义的可以用，但是其他没有重新定义的父类函数都被隐藏，子类都不能用。

# 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.

当子类没有显式定义构造函数时，编译器会自动生成子类的合成构造函数。这个合成构造函数会使用成员逐一初始化的方式，对子类中的成员属性进行初始化。它会按照继承关系，先调用基类的合成构造函数来初始化基类的成员，然后再对子类的成员进行逐一初始化。

# What is not inherited?

- Constructors 在显式复制构造函数中，必须显式调用基类的复制构造函数，否则将会基类调用默认构造函数
  - synthesized constructors use memberwise initialization
  - In explicit copy ctor, explicity call base-class copy ctor or the default ctor will be called instead.
- Destructors 赋值运算
- 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 默认是private

# How inheritance affects access

Suppose class B is derived from A. Then

基类成员访问限定符（访问范围类型）

Base class member access specifier

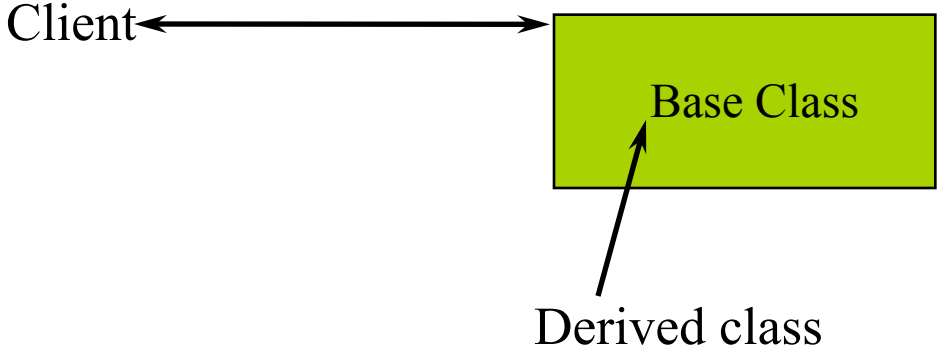
| Inheritance Type (B is) | public | protected | private |
|---|---|---|---|
| public A | public in B | protected in B | hidden |
| private A | private in B | private in B | hidden |
| protected A | protected in B | protected in B | hidden |

父类Publicle类型被子类继承，成为子类中的私有成员，子类的用户，不能访问子类的私有成员

子类不能访问父类私有成员

# 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



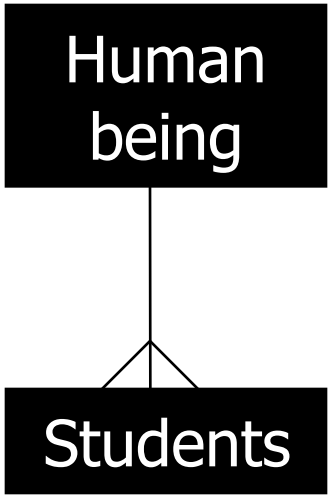
# Conversions

- Public Inheritance should imply substitution
  - If B *isa* A, you can use a B anywhere an A can be used.
    - if B isa A, then everything that is true for A is also true of B.
  - Be careful if the substitution is not valid!

| D is derived from B | | |
|---|---|---|
| D | ⇒ | B |
| D* | ⇒ | B* |
| D& | ⇒ | B& |

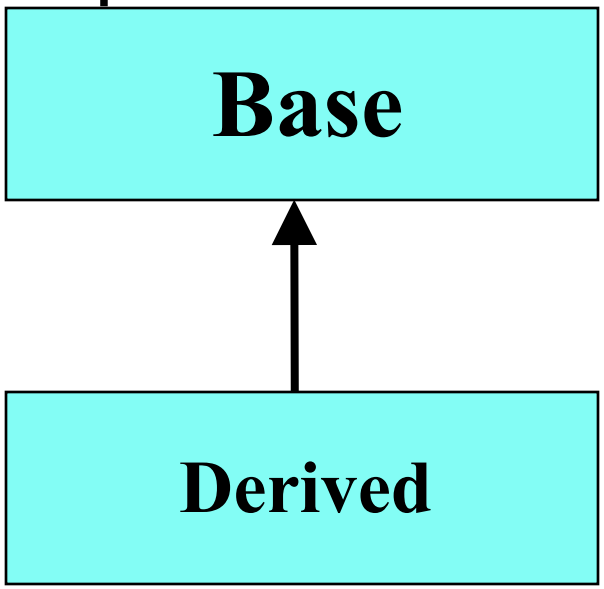
# Up-casting

- Is to regard an object of the derived class as an object of the base class.
- It is to say: Students are human beings. You are students.  So you are human being.



# Upcasting

- Upcasting is the act of converting from a Derived reference or pointer to a base class reference or pointer.



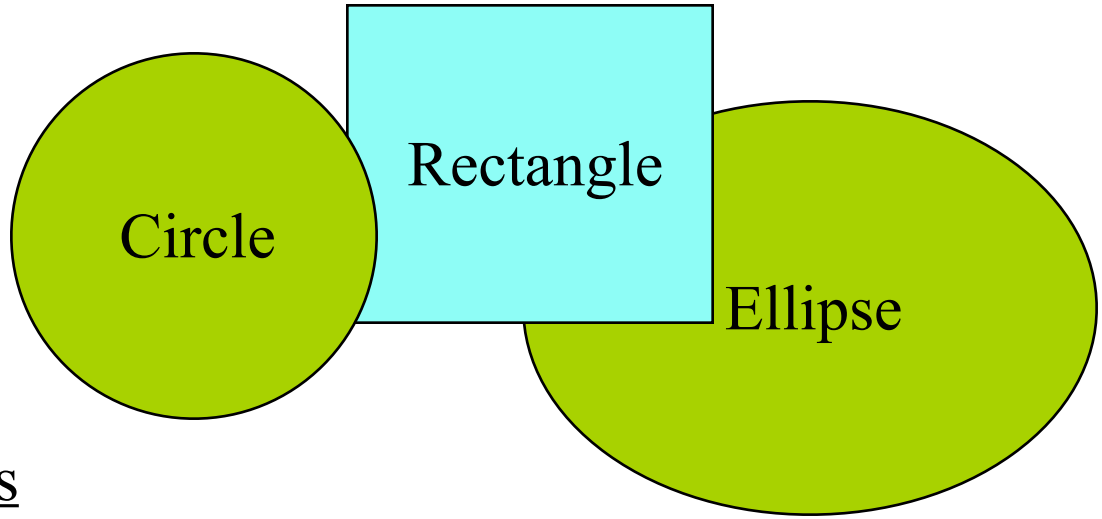
# Upcasting examples

```
Manager pete( "Pete", "444-55-6666", "Bakery");
Employee* ep = &pete;  // Upcast
Employee& er = pete;    // Upcast
```

- Lose type information about the object:

```
ep->print( cout );    //  prints base class version
```

# A drawing program




Operations

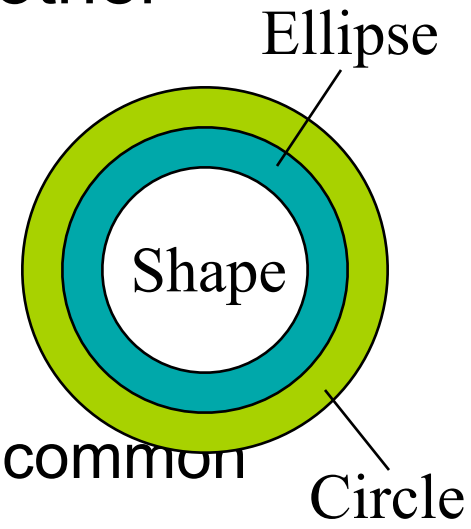
- render
- move
- resize

Data

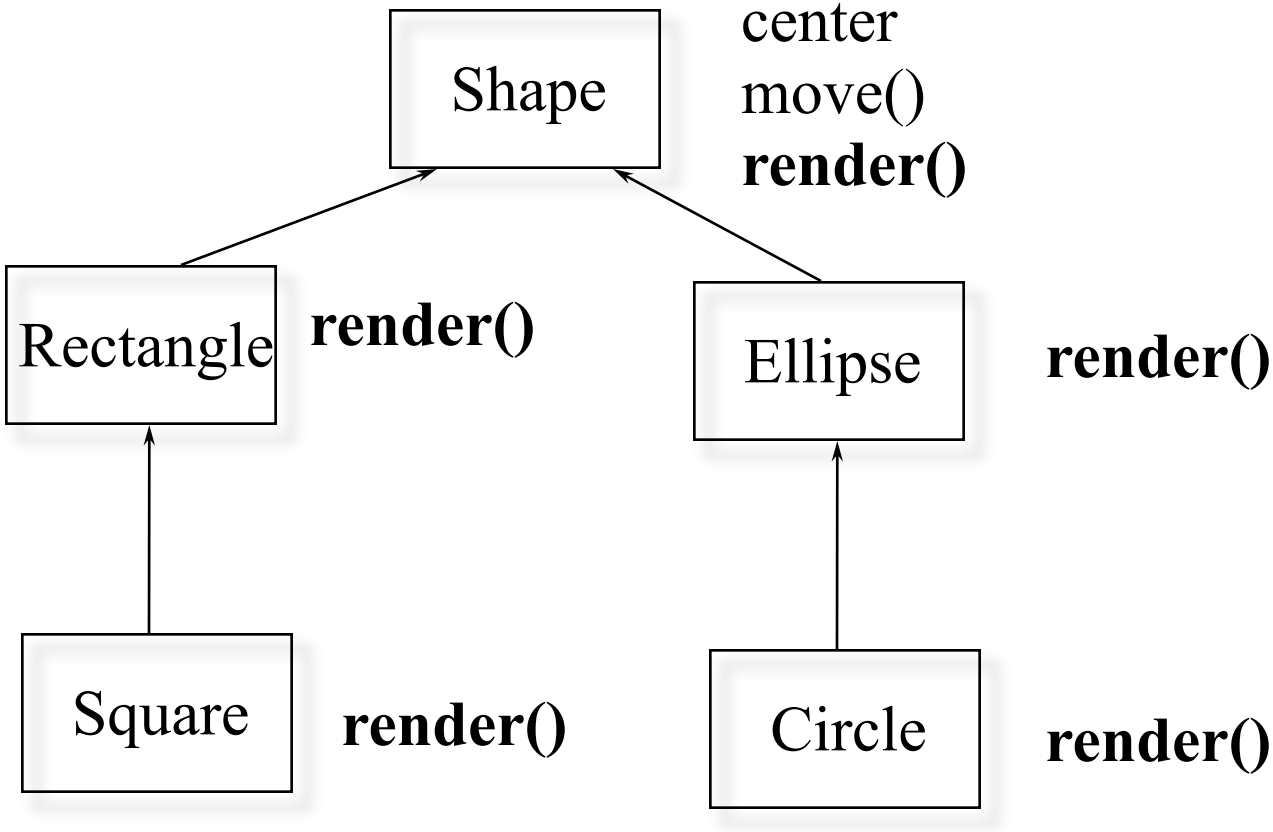
+ center

# Inheritance in C++

- Can define one class in terms of another
- Can capture the notion that
  - An ellipse is a shape
  - A circle is a special kind of ellipse
  - A rectangle is a different shape
  - Circles, ellipses, and rectangles share common
    - attributes
    - services
  - Circles, ellipses, and rectangles are not identical



# Conceptual model




*Note: Deriving Circle from Ellipse is a poor design choice!*

# In C++

- Define the general properties of a Shape

```
class XYPos{ ... };   // x,y point
class Shape {
public:
  Shape();
  virtual ~Shape();
  virtual void render();
  void move(const XYPos&);
  virtual void resize();
protected:
  XYPos center;
};
```

# Polymorphism

- Upcast: take an object of the derived class as an object of the base one.
  - Ellipse can be treated as a Shape
- Dynamic binding:
  - Binding: which function to be called
    - Static binding: call the function as the code
    - Dynamic binding: call the function of the object

# Virtual functions

- Non-virtual functions
  - Compiler generates ***static***, or direct call to stated type
  - Faster to execute
- Virtual functions
  - Can be ***transparently*** overridden in a derived class
  - Objects carry a pack of their virtual functions
  - Compiler checks pack and ***dynamically*** calls the right function
  - If compiler knows the function at compile-time, it can generate a static call

# Add new shapes

```
class Ellipse : public Shape {
public:
  Ellipse(float maj, float minr);
  virtual void render(); // will define own
protected:
   float major_axis, minor_axis;
};

class Circle : public Ellipse {
public:
  Circle(float radius) : Ellipse(radius, radius){}
  virtual void render();
};
```

# Example

```
void render(Shape* p) {
  p->render();    // calls correct render function
}                 // for given Shape!
void func() {
  Ellipse ell(10, 20);
  ell.render();   // static -- Ellipse::render();

  Circle circ(40);
  circ.render();  // static -- Circle::render();

  render(&ell);   // dynamic -- Ellipse::render();
  render(&circ);  // dynamic -- Circle::render()
}
```

# Abstract base classes

- An *abstract base class* has pure virtual functions
  - Only interface defined
  - No function body given
- *Abstract base classes cannot be instantiated*
  - Must derive a new class (or classes)
  - Must supply definitions for all pure virtuals before class can be instantiated

# In C++

- Define the general properties of a Shape

```
class XYPos{ ... };   // x,y point
class Shape {
public:
  Shape();
  virtual void render() = 0; // mark
  render() pure
  void move(const XYPos&);
  virtual void resize();
protected:
  XYPos center;
};
```

# Abstract classes

- Why use them?
  - Modeling
  - Force correct behavior
  - Define interface without defining an implementation
- When to use them?
  - Not enough information is available
  - When designing for interface inheritance

# Protocol/Interface classes

- Abstract base class with
  - All non-static member functions are *pure* virtual except destructor
  - Virtual destructor with empty body
  - No non-static member variables, inherited or otherwise
    - May contain static members

# Example interface

- Unix character device

```
class CDevice {
public:
        virtual ~CDevice();

        virtual int read(...)   = 0;
        virtual int write(...)  = 0;
        virtual int open(...)   = 0;
        virtual int close(...)  = 0;
        virtual int ioctl(...)  = 0;
};
```