Chapter 12

Inheritance and Aggregation

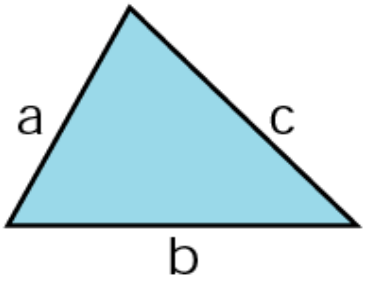
OBJECTIVES

After studying this chapter you will be able to:

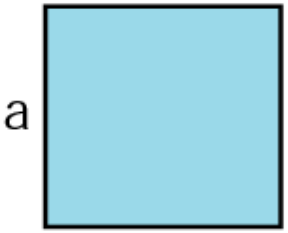
- Create a derived class from a base class.
- Use the *private*, *protected*, and *public* access types.
- Override a member function defined in the base class.
- Use dynamic binding to bind a function to an object during run time.
- Use polymorphism to create different functions with the same name.
- Understand how to use pure virtual functions in an abstract.
- Use upcasting and downcasting to cast base classes and derived classes.
- Use aggregation to create a class object that contains other object(s).
- Understand and describe the seven types of cohesion.

INHERITANCE

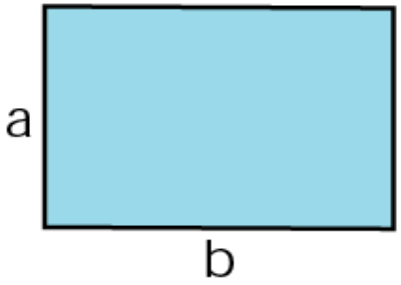
Figure 12-1 Simple polygons



perimeter = a + b + c
area = sqrt { (perimeter / 2)
* [(perimeter / 2) – a]
* [(perimeter / 2) – b]
* [(perimeter / 2) – c] }



perimeter = 4 * a
area = a^2



perimeter = 2 * (a + b)
area = a * b

Figure 12-2 Base and derived classes

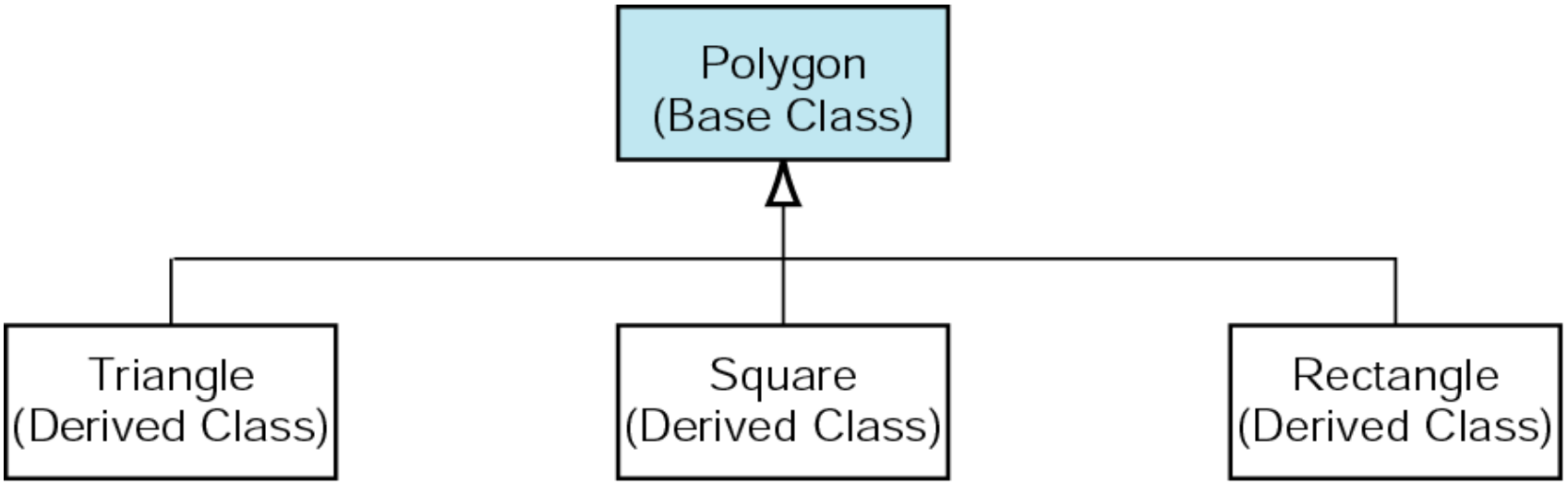
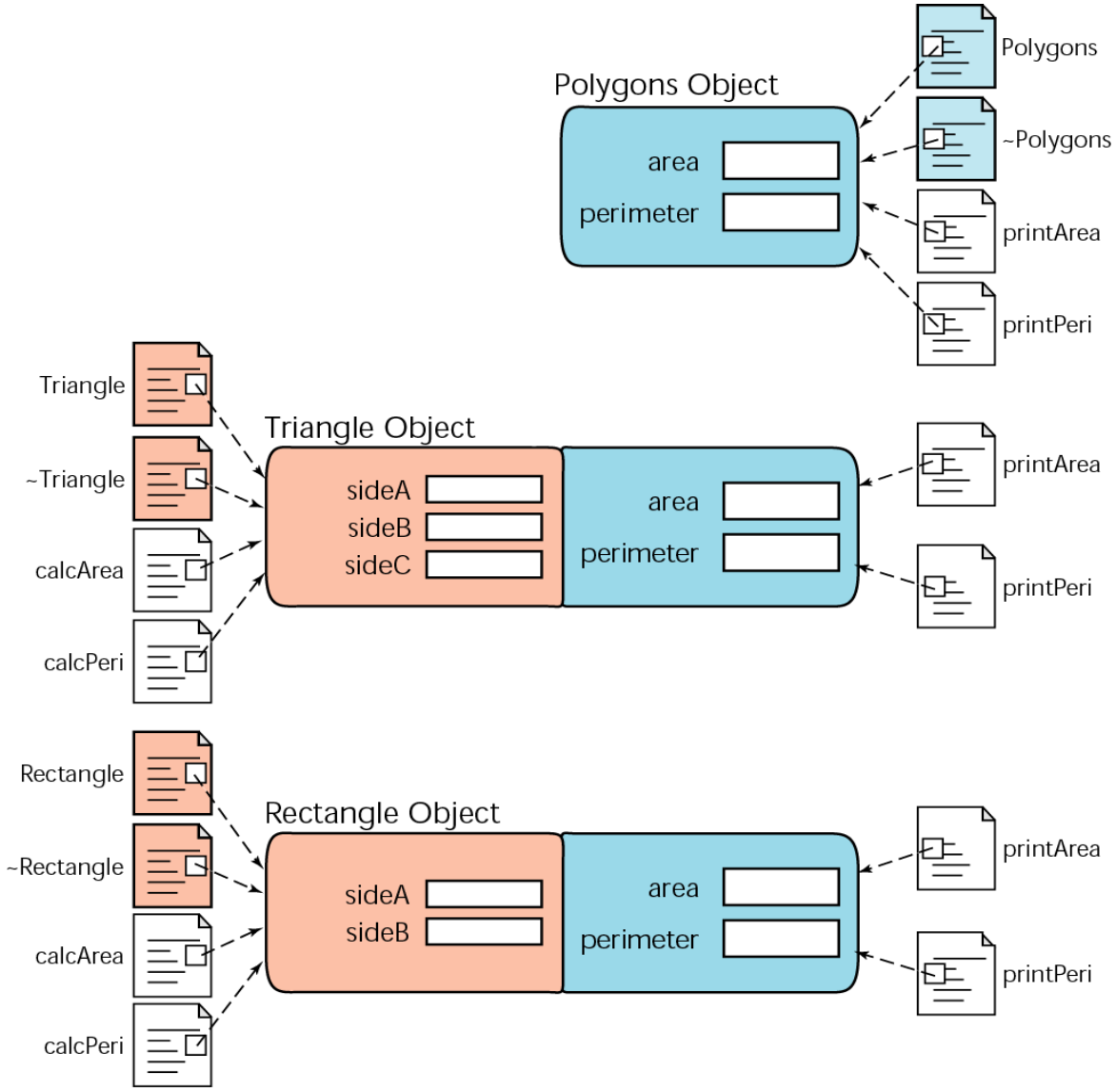


Figure 12-3 Inheritance



Note:

Constructors, destructors, nonmember functions, assignment operators, and virtual methods are not inherited in a derived class. If needed, they must be created.

PRIVATE, PROTECTED, PUBLIC

Note:

The combination of the base class access type and the inheritance type determines if and how the base class data and functions can be accessed by the derived class.

Figure 12-4 Inheritance example

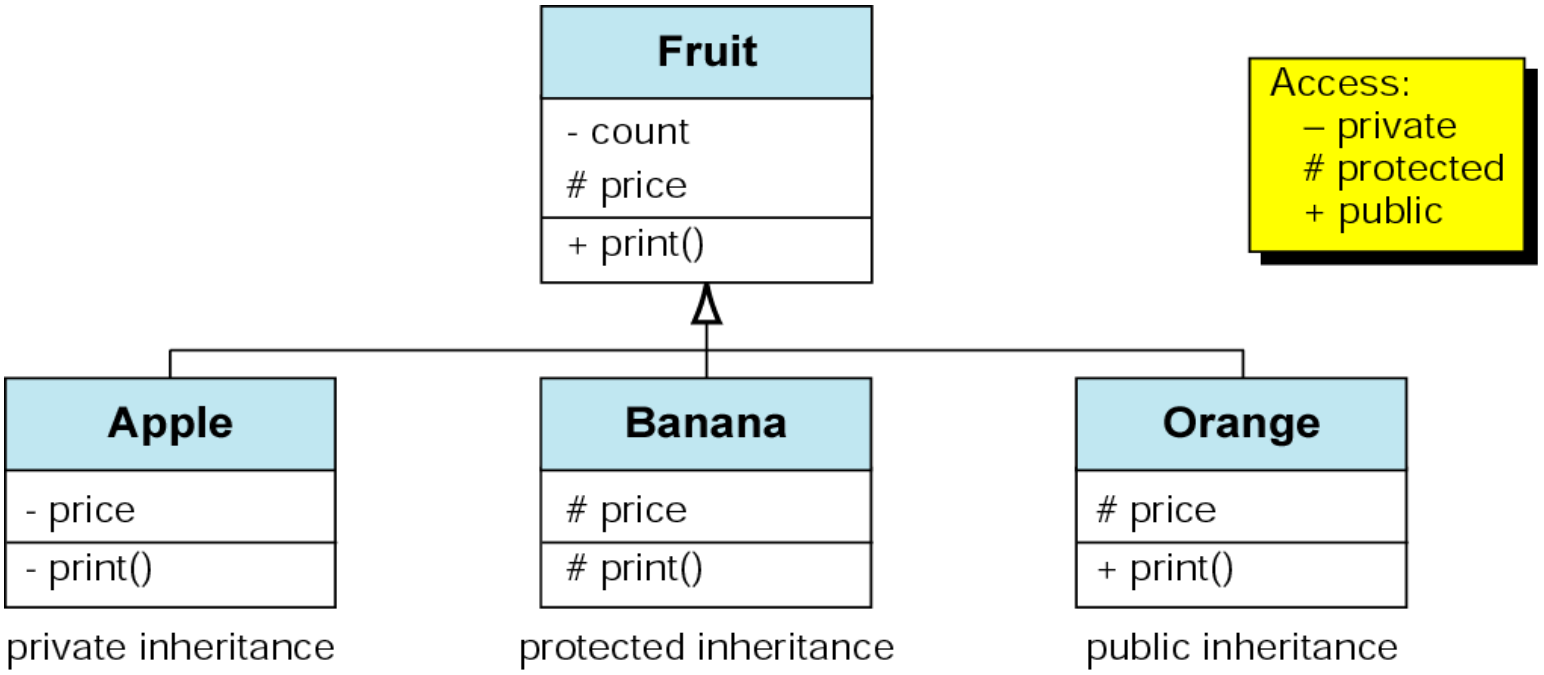
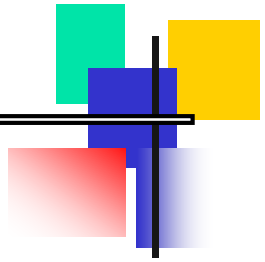


Figure 12-5 Overriding access specifier

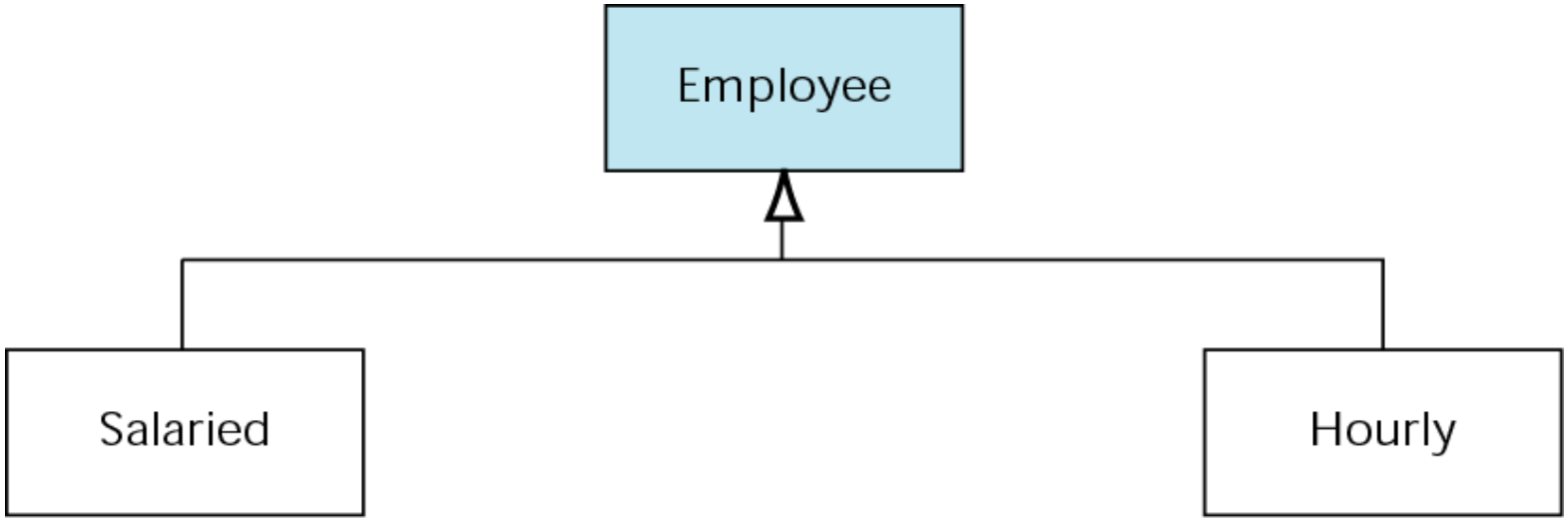


```
class B
{
        protected:
          int datum;
        public:
          void print();
          ...
};  // B
```

```
class D : private B
{
        protected:
          using B :: datum;


          ...
};  // D
```

MANAGER FUNCTIONS UNDER INHERITANCE

Figure 12-6 Employee class design



OVERRIDING MEMBER FUNCTIONS

POLYMORPHISM

Figure 12-7 Static function binding

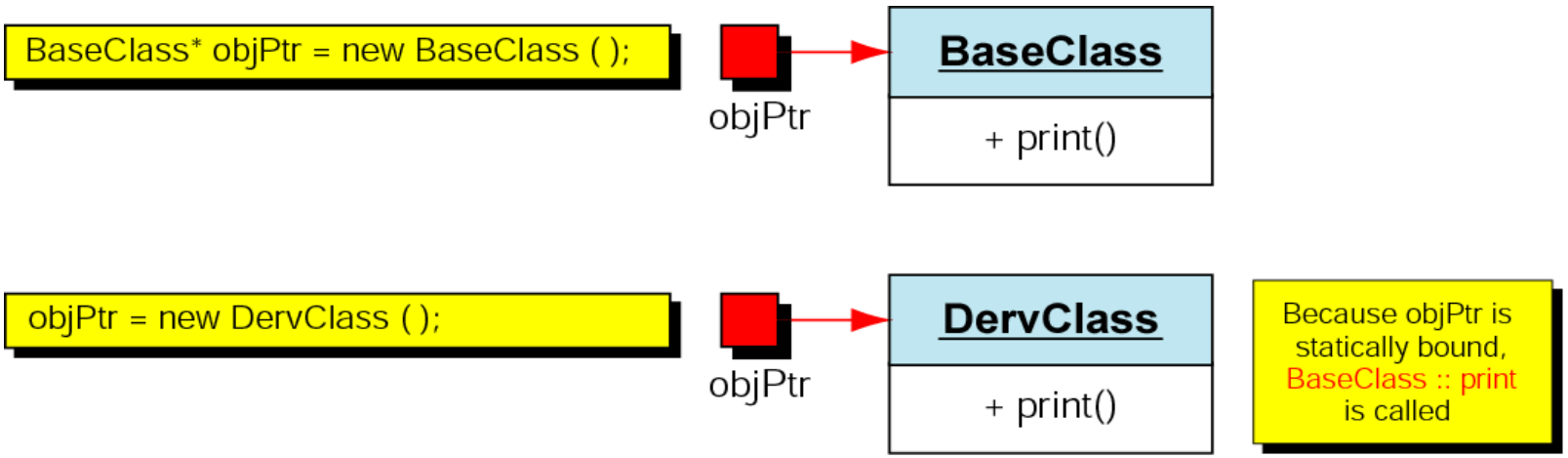


Figure 12-8 Dynamic binding through polymorphism

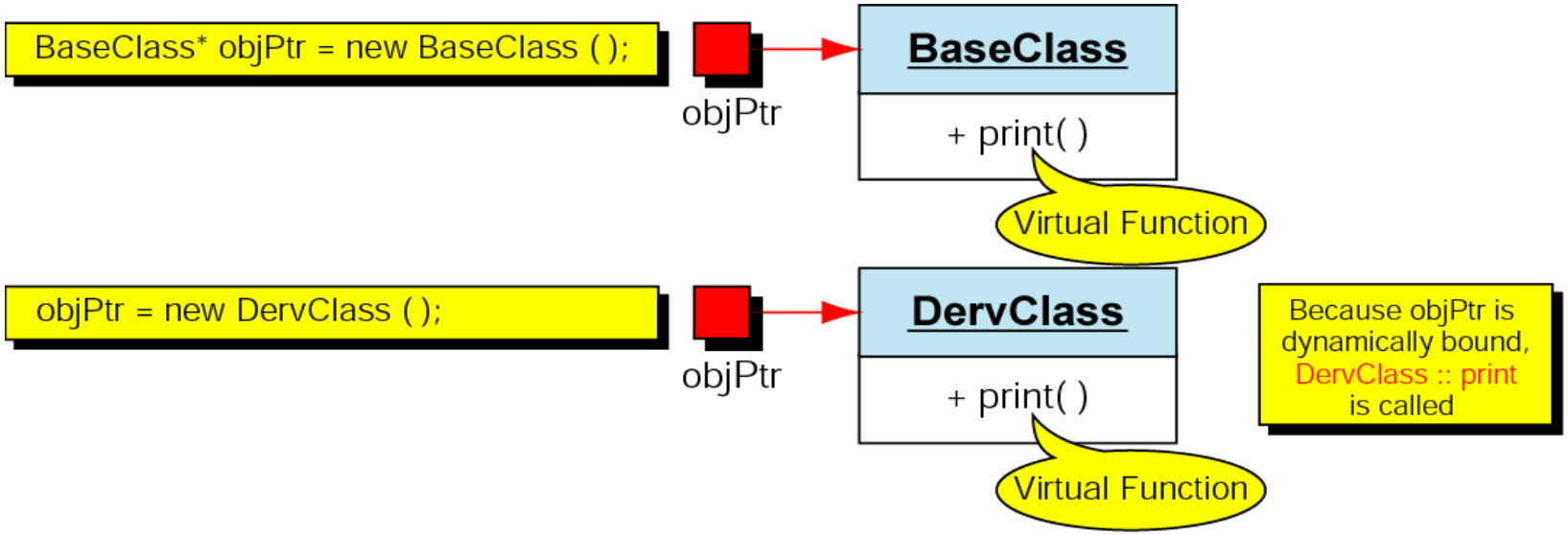
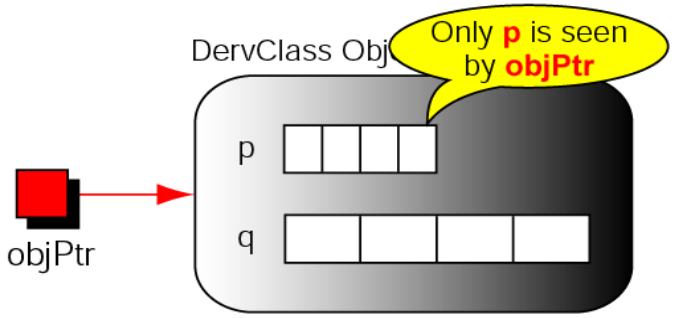
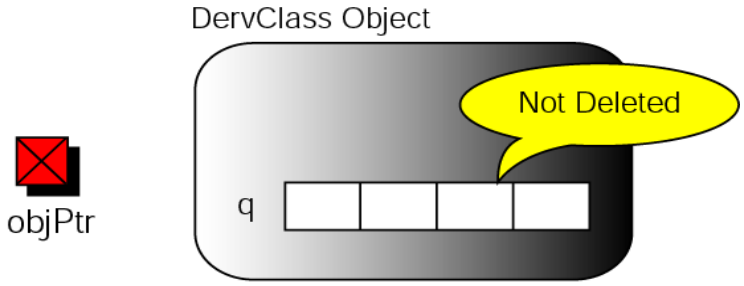


Figure 12-9 Virtual destructors

```
BaseClass
{
   private:
      int* p;
   public:
      BaseClass( ) {p = new int[4];}
     ~BaseClass( ) {delete[ ] p;}
}  // BaseClass

DervClass
{
   private:
      ifloat* q;
   public:
      DervClass( ) {q = new float[4];}
     ~DervClass( ) {delete[ ] q;}
}  // DervClass
...

int main ()
{
BaseClass* objPtr = new DervClass ( );
...

delete objPtr;
...
```



(a) After instantiation



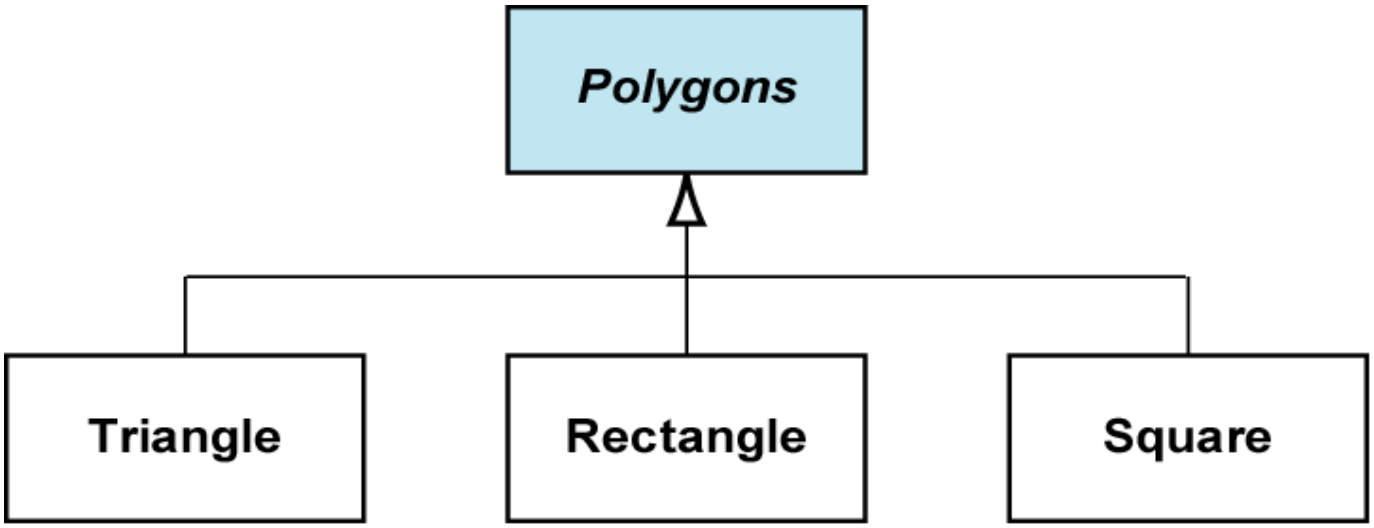
(b) After delete

Pure Virtual Functions: Abstract Classes

Note:

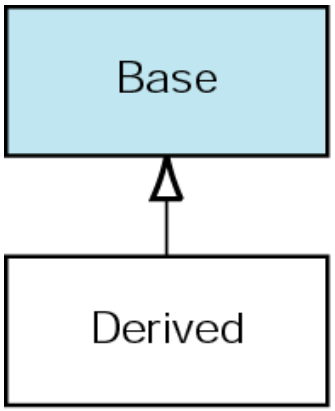
An abstract class combines the idea of inheritance, polymorphism, and modeling.

Figure 12-10 Abstract polygon class design



Type Conversion in Hierarchical Classes

Figure 12-11 Upcasting objects




```
class Base
{
   ...
} ; // Base
class Derived: public Base
{
   ...
} ; // Derived
   ...
int main (void)
{
// Local Definitions
Base    baseObj;
Derived derObj;
   ...
// Statements
   baseObj = derObj;
   ...
}  // main
```

Figure 12-12 Upcasting objects

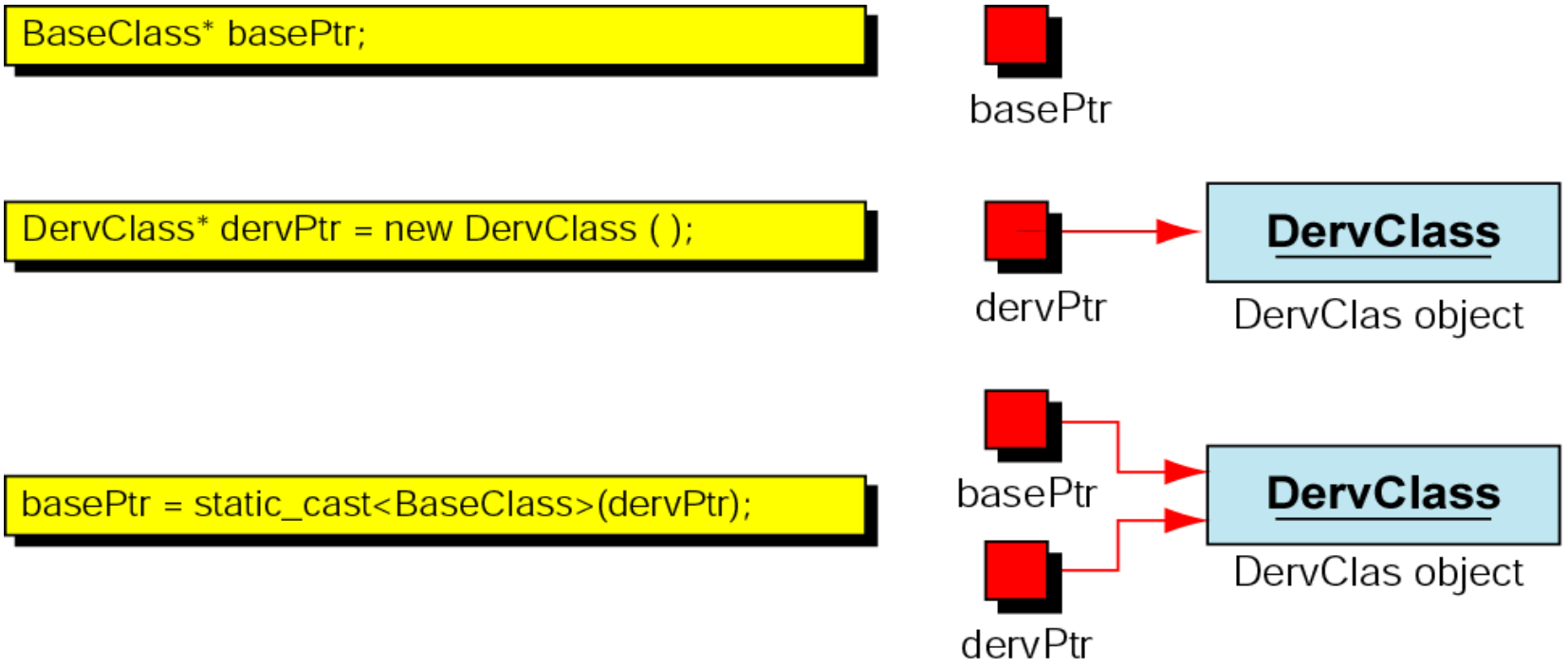


Figure 12-13 Valid downcasting

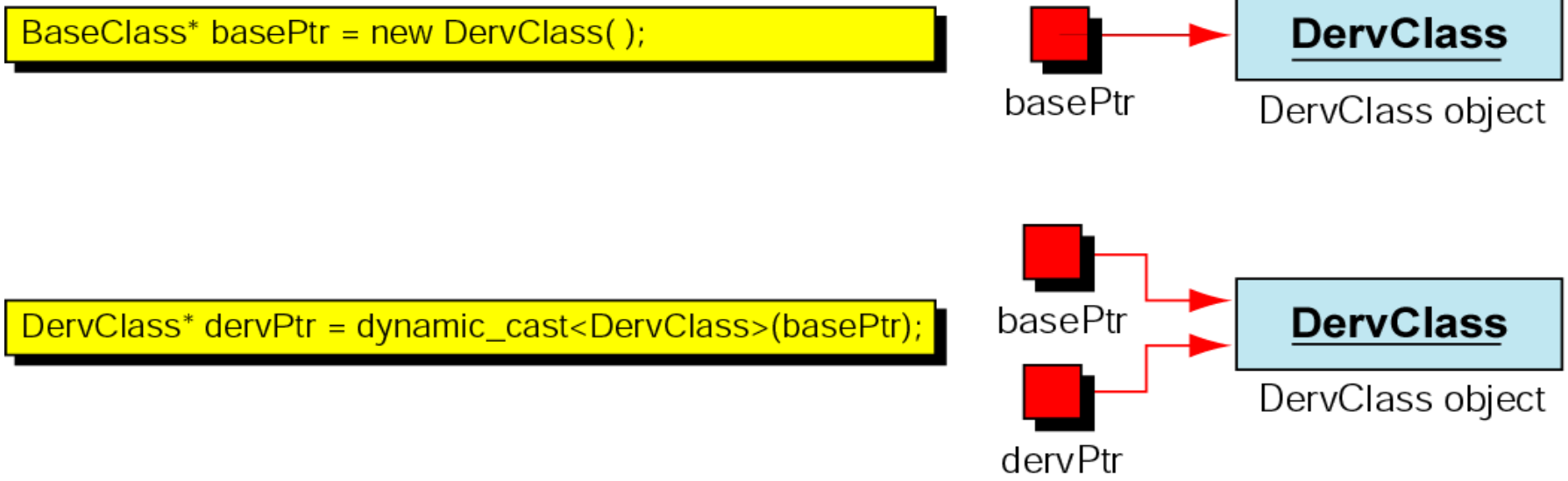
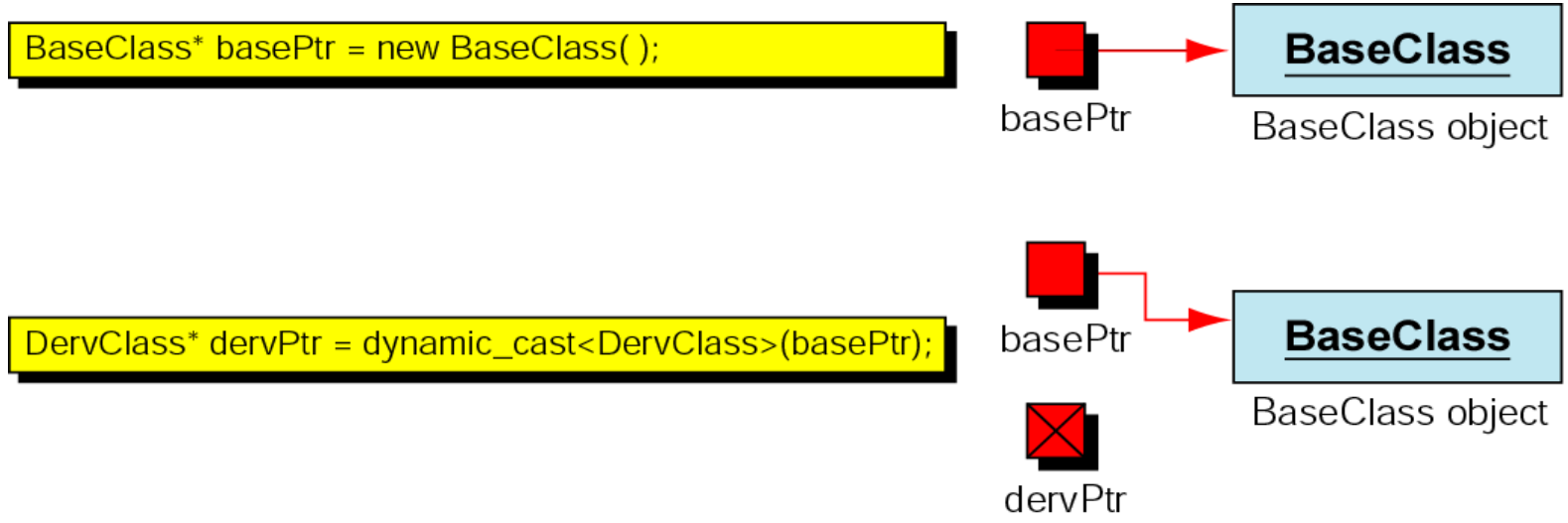
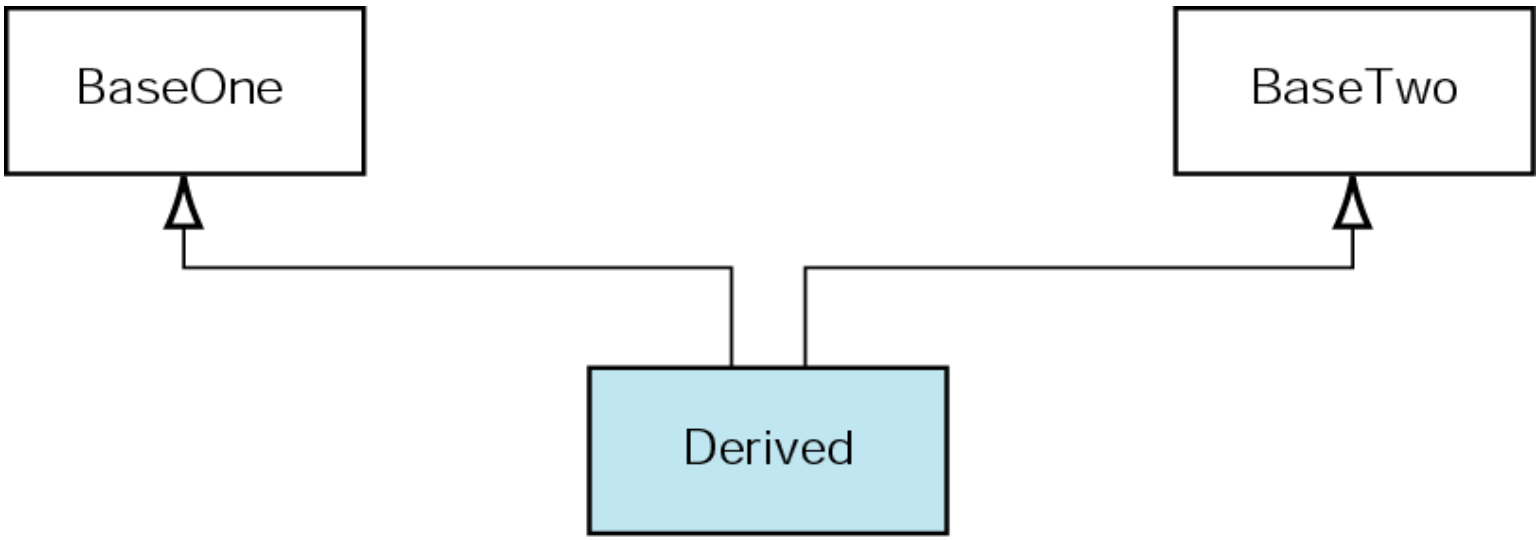


Figure 12-14 Invalid downcasting



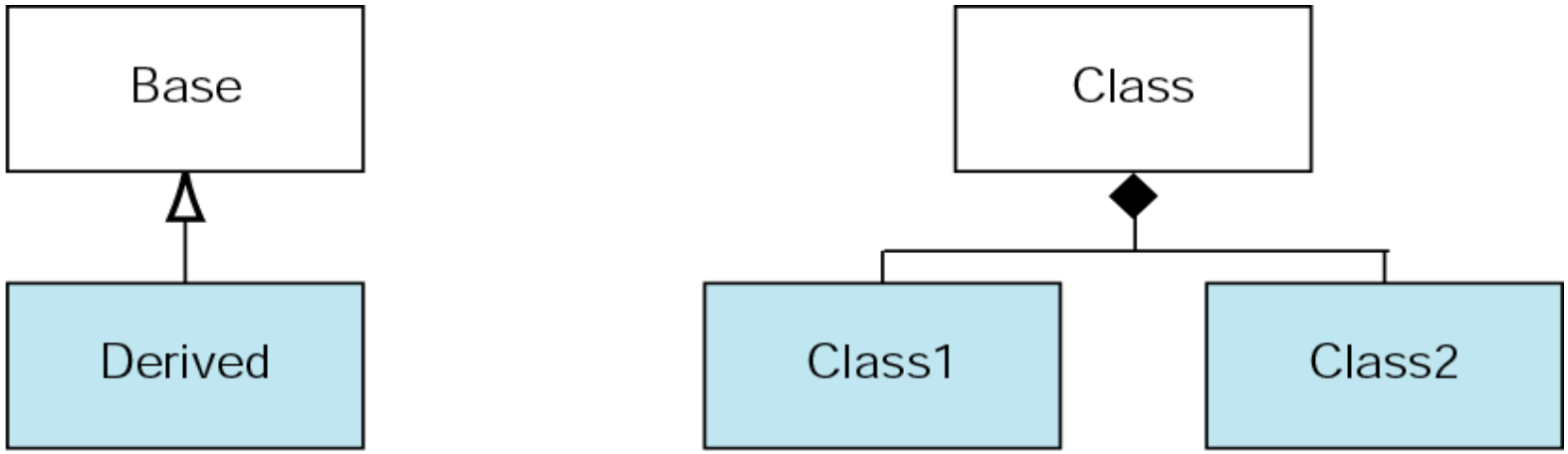
MULTIPLE INHERITANCE

Figure 12-15 Multiple inheritance



AGGREGATION

Figure 12-16 Aggregation



(a) Derived *is a* Base

(b) Class aggregates *two classes*

Figure 12-17 Student class

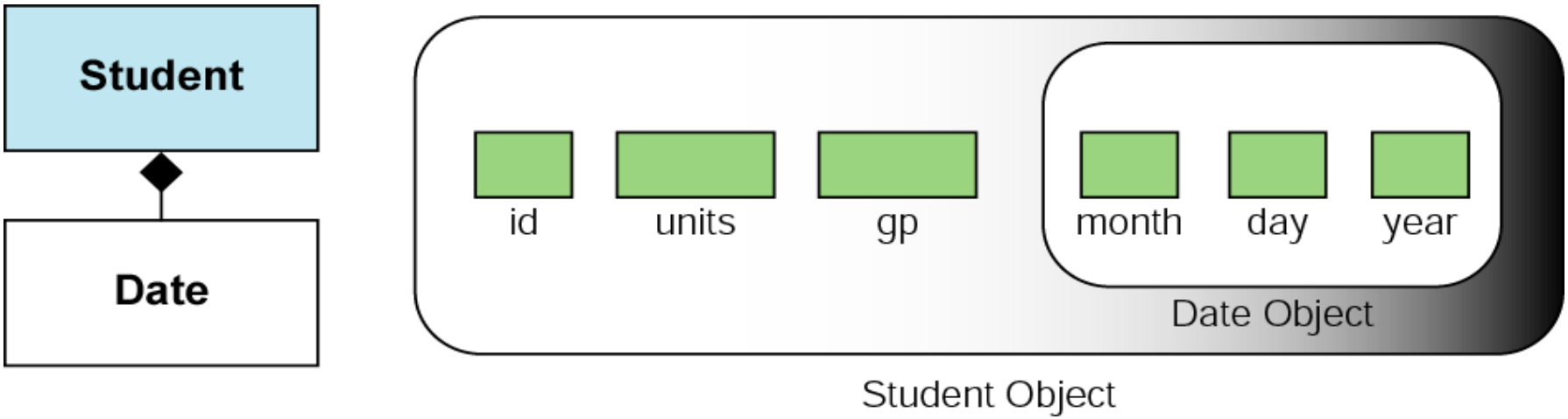
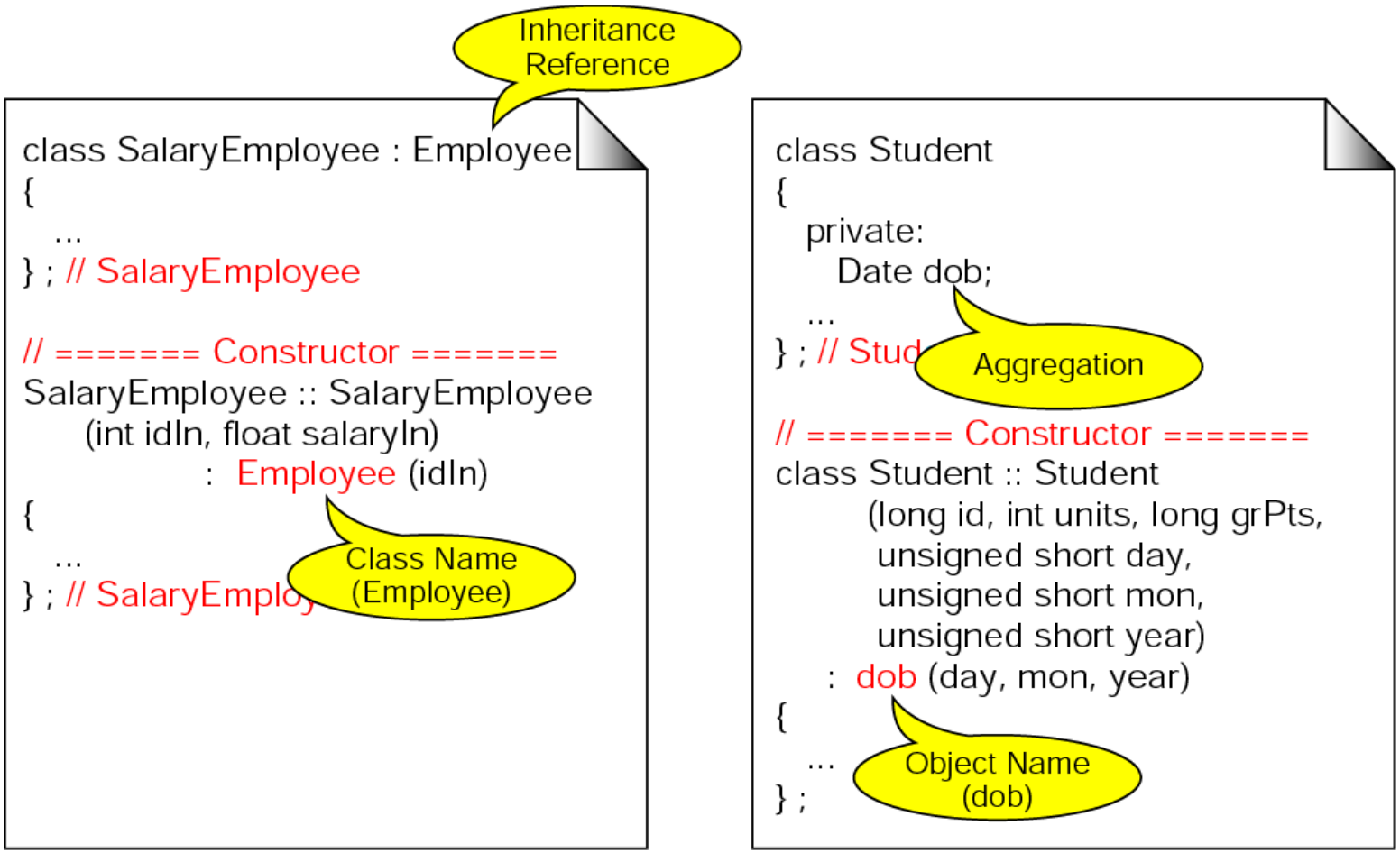


Figure 12-18 Instantiating an object



Software Engineering and Programming Style

Note:

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. . . . Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

Figure 12-19 Types of cohesion

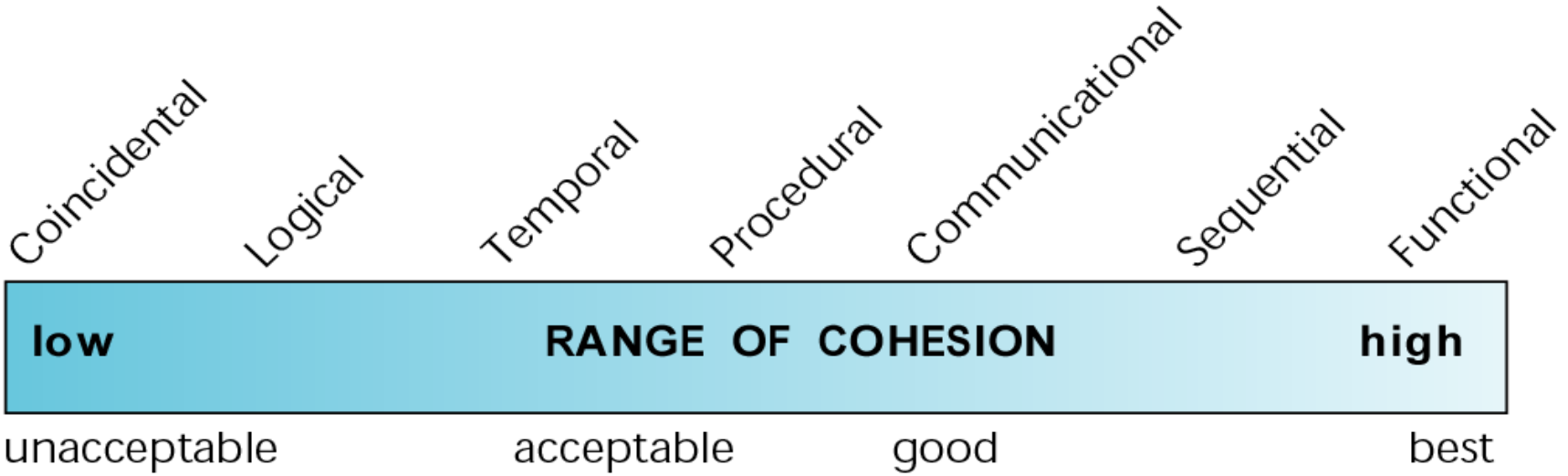
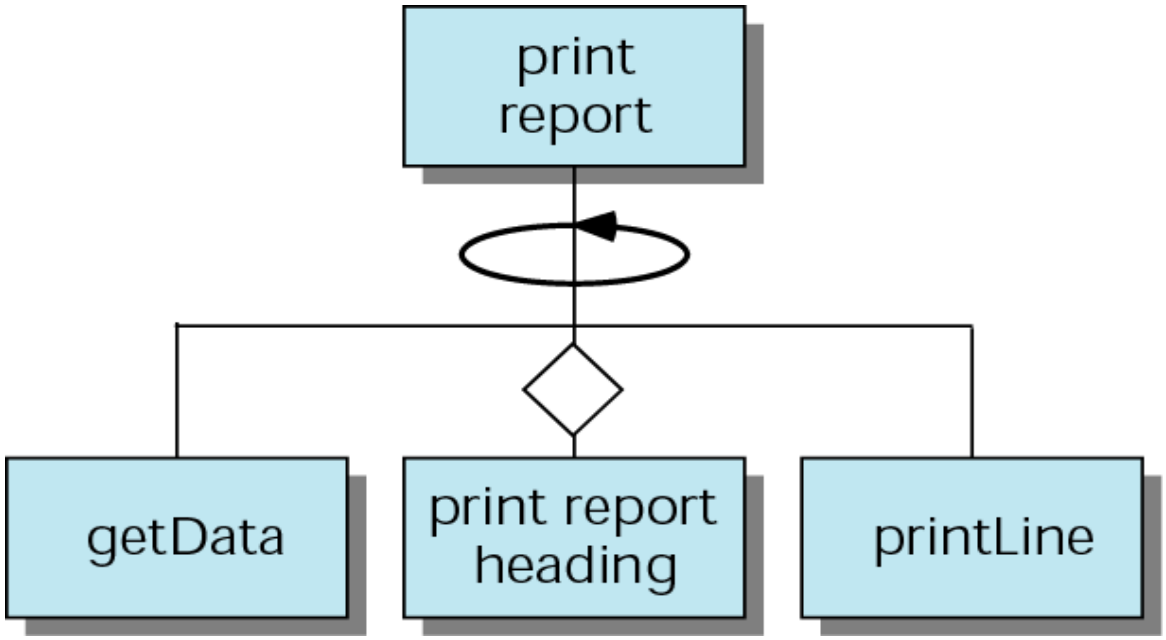


Figure 12-20 Example of functional cohesion



Note:

Well-structured programs are highly cohesive and loosely coupled.