Проектиране и Тестиране на Софтуер
ТУ, кат. КС, летен семестър 2012

Лекция 3

Тема:

ООП - Капсулиране на Данни

ООП - Наследяване

Част 1

ООП

Капсулиране на Данни

Съдържание:

- Въведение в концепцията за клас като колекция от данни компоненти (data components) и член функции – методи (member functions - methods).
- Класовете като потребителски дефинирани типове.
- Релацията **class – object.**

Концепцията ООП

- От структурно програмиране
  - Algorithms + Data Structures = Programs
  - Thinking in Functions
- Към обектно ориентирано програмиране
  - Classes + Objects = Programs
  - Thinking in Classes

Концепцията ООП

- Структурно Програмиране: гл. черта разделяне на програмата на функции
- Два проблема остават нерешени от Структурното Програмиране: F.Brooks, *The Mythical Man-Month*
  - Неограничен достъп до данни
  - Слабо моделиране (Real-World Modeling). Обектите в реалния свят включват:
    - Атрибути (Attributes)
    - Поведение (Behavior)

3-те главни ООП черти

- **Капсулиране/Скриване на Данни**
  - Данни и функциите за обработка на данните се капсулират в единно цяло (същност-entity).
  - Данни се скриват вътре в клас така, че да няма достъп по погрешка до тях чрез функции външни за класа.
- Наследяване
- Полиморфизъм

Кандидати за класове/обекти

- Физически обекти
  - Автомобили, държави, ел. компоненти
- Компоненти от визуалното програмиране
  - Прозорци, ресурси – елементи на потребителски интерфейс – бутон, етикет, текстово поле, ...
- Структури за съхранение на данни
  - Стек, опашка, списък, дърво, граф
- Х-ки на личността
  - Служител, студент, преподавател

Кандидати за класове/обекти

- Колекции от данни
  - Опис на продукт, файл с лични данни, речник
- Потребителски дефинирани типове данни
  - Time, Date, 2D обекти, 3D обекти
- Компоненти на компютърни игри
  - Предмети, играчи
- Други
  - Всичко родено от въображението на човека

Концепцията **Абстракция**

Abstraction е начин на представяне, при който се акцентира само върху основните, най-съществените характеристики/параметри.

Abstraction е лек срещу сложността в програмирането. Абстракцията има за цел да опрости софтуерния процес.

Два основни вида абстракция в ПЕ:

- Абстракция на ниво обработка (**process abstraction**);
- Абстракция на ниво данни (**data abstraction**).

Process abstraction – основава се на подхода с ПП

ПП капсулират процес (обработка), без да предоставят инфо как се решава проблемът.

Пример: Да се сортира масив от числени данни

Решение: *sortInt(list, listLen);*

Извикването е абстракция на ниво процес.

Важни атрибути: име на масива, тип данни, размер.

Неважни (скрити) атрибути: алгоритъм за сортиране

Process abstraction – основава се на подхода с ПП

"Properly designed functions permit to ignore **how** a job's done. Knowing **what** is done is sufficient."

B.Kernighan & D.Ritchie

"A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. "

B.Kernighan & D.Ritchie

Абстракция: от процес към данни

- Абстракцията на ниво данни следва (появява се след) процес абстракцията, т.к. централна част на всяка абстракция с данни са нейните обработки, които се олицетворяват от абстракцията на ниво процес (ПП).

Абстракция данни - увод

Един абстрактен тип данни (АТД) капсулира две неща:

- вътрешното представяне на съответния тип данни
- набор ПП, които предоставят операциите над този тип.

Отделен екземпляр (instance) от абстрактен тип данни се нарича обект (**object**).

Floating-Point като АТД

Всички вградени (основни) типове данни (цели, реални, символни, булеви) са по същество АТД.

- Могат да се дефинират променливи (обекти)
- Налице са операции
- Вътрешното им представяне е скрито

Дефинирани от потребителя АТД

Потребителски дефинираният АТД следва да има поведението на основните типове данни, а именно:

1/ възможност за дефиниране на променливи със скриване на формата за вътрешно представяне;

2/ набор от действия (операции или ПП) за манипулиране на обекти от указания тип.

Пример: типът **stack** като АТД

Дадено: АТД стек със следните операции:

create(stack), destroy(stack), empty(stack),
push(stack, element), pop(stack), top(stack)

Клиент на стека може да създаде сл. програмен текст

```
create(stk1);
push(stk1, color1); push(stk1,color2);
if (!empty(stk1))  temp = pop(stk1);
```

Не се налага промяна на горния клиентски код, в случай на промяна на вътрешното представяне на стека, ако интерфейсът (протоколът на всички ПП) се запази

Примери от ПЕ: ADT в C++

С цел работа с АТД, C++ предоставя две синтактични конструкции, които са подобни, но не еднакви:

`class` (съдържа данни и ПП за работа с данни)

`struct` (съдържа само данни ???)

Данните, дефинирни в C++ клас, са **data members**.

ПП, дефнирани в клас, са методи **member functions**.

C++ клас съдържа скрити и видими компоненти

Скриване на данни:

`private`

`public`

Примери от ПЕ: ADT в Java

Подобно на C++ със следните разлики:

Всички АТД са класове (няма структури)

Памет за обекти се заделя от купа (грамада), като достъпът е чрез reference променливи.

ПП (методи) се дефинират само вътре в класове.

Без заглавни хедър файлове, т.к няма изискване описанието да изпреварва позоваването.

Няма нужда от деструктор, т.к има вградена garbage collection системна програма.

Примери от ПЕ: ADT в C#

1/ C# атрибути за достъп: *private, public, protected.*

Два нови: *internal* и *protected internal.* C# борави с конструкция, по-голяма от клас: *assembly*. ***assembly*** е колекция от файлове в състава на application програми. An *internal* член на клас се вижда във всички класове на едно assembly, в което той се среща.

2/ Подобно на Java, C# класовете са heap dynamic. Default constructors са налични и те формират начални стойности като (*0* for integer, *false* for boolean).

3/ C# ползва garbage collection за своите heap обекти и рядко се прилагат деструктори.

Примерни класове: C++

- C++ физически обекти (classes *SmallObj, Part*);
- C++ типове данни (class *Distance*);
- C++ GP програмни елементи (classes *Counter, String*);
- Конструктори/деструктори. Предефинирани конструктори;
- Обекти аргументи на функции (*dist3.AddDist1(dist1,dist2);*);
- Обекти, върнати от функции

  dist4 = dist1.AddDist2(dist2);
- Методи (член функции) дефинирани извън класа;
- Класове, обекти и памет.

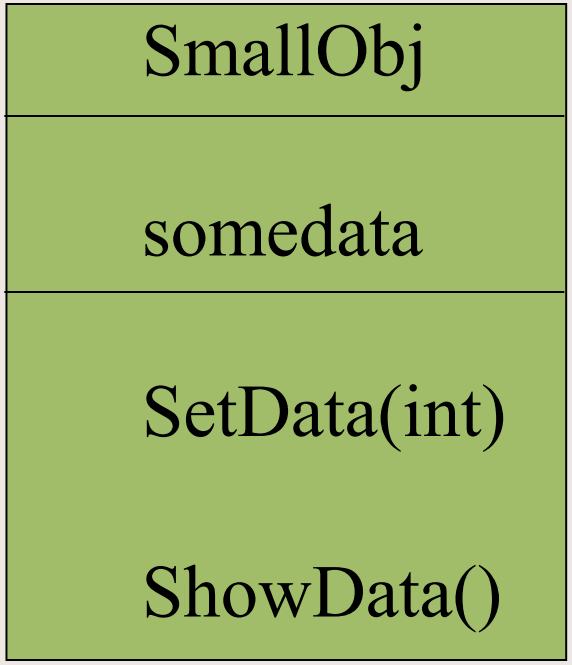
C++ обекти – физически обекти

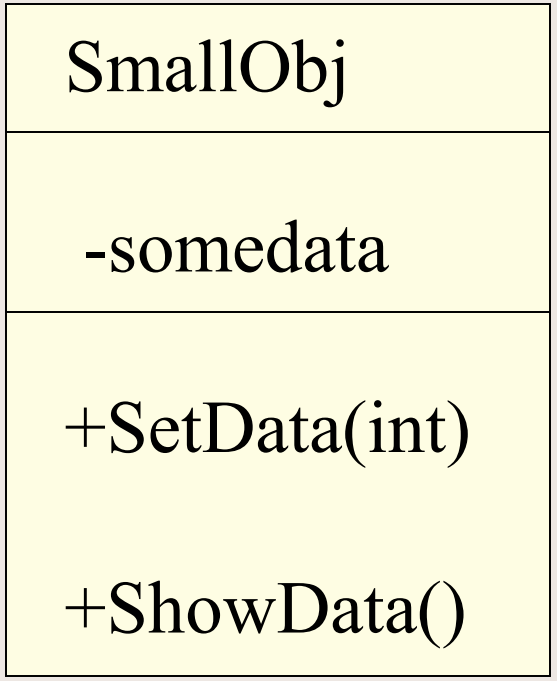
- Описание на клас SmallObj
  - OOP1a.cpp OOP1a.exe
- Общ синтаксис
- Квалификатори за достъп: *private, public*
- Член данни
  - Обикновено *private*
- Член функции
  - Обикновено *public*

Class SmallObj

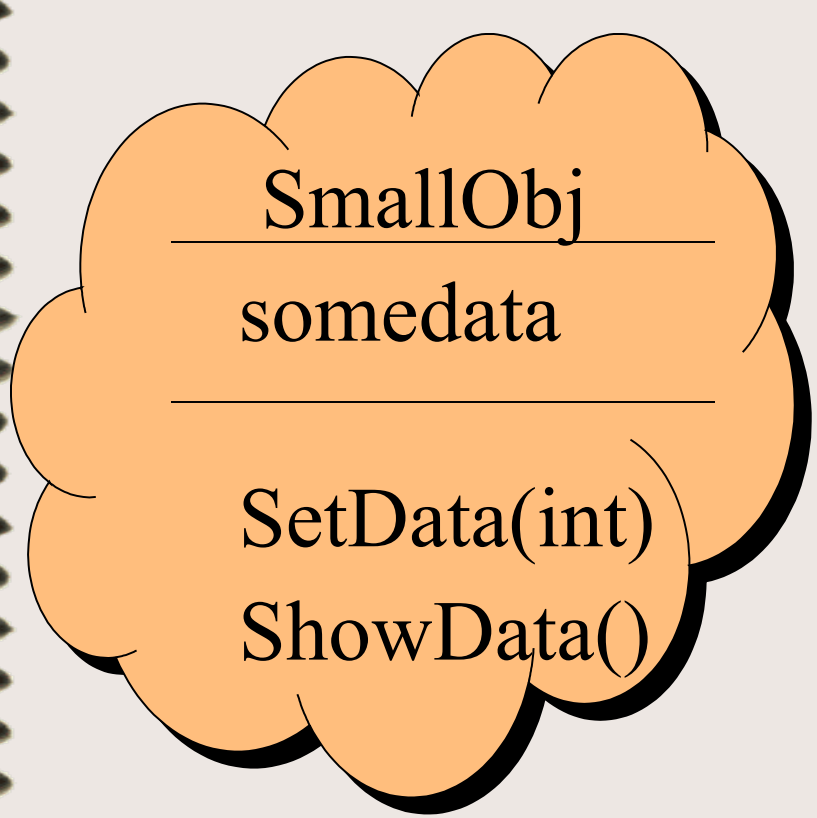
```
class SmallObj
{
 private: int somedata;

 public: void SetData(int d) { somedata = d; }

      void ShowData() { cout << "\nData is =" <<
                                    somedata; }
};
```

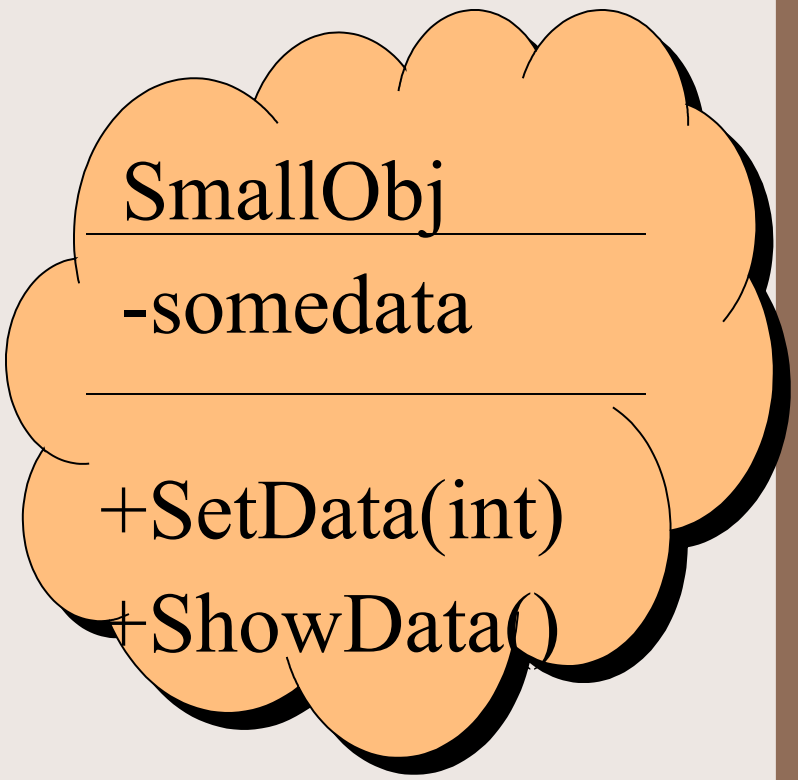
SmallObj – UML клас диаграма





SmallObj – UML клас диаграма според G.Booch





Приложение на клас

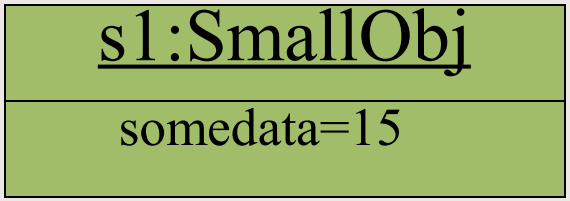
- Дефиниране на обекти
- Извикване на член функции (sending messages)

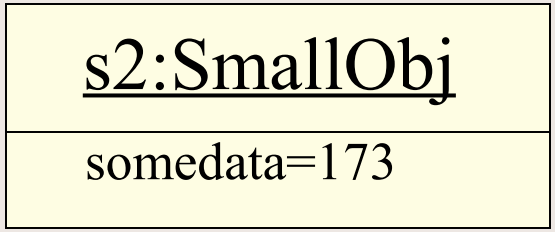
Клас SmallObj

```
void main()
{
  SmallObj s1, s2;

  s1.SetData(67);
  s2.SetData(123);

  s1.ShowData();
  s2.ShowData();
}
```

SmallObj – UML обект диаграми

Заразлика от класовете, обектите се подчертават. Името на класа и обекта се разделят с двуеточие.





C++ обекти – физически обекти
Клас Part OOP1b.cpp, OOP1b.exe

```
class Part
{
 private: int modelnumr, partnumr; float cost;

 public:  void SetPart(int mn, int pn, float cs)
          {
            modelnumr=mn; partnumr=pn; cost=cs;
          }
         void ShowPart()
         { cout << "modelnumber=" << modelnumr

         << "  " << "partnumber="  << partnumr

         << "  " << "cost="        << cost;
         }
};
```

Клас Part

```
void main()
{
  Part  p1,  p2;

  p1.SetPart(6244, 317, 21.68);
  p2.SetPart(8124, 516, 314.52);

 cout << "\n\nPart components:   ";
  p1.ShowPart();
 cout << "\n\nPart components:   ";
  p2.ShowPart();
}
```

Съвременната конвенция

Реализация, основана на:

interface file	OOP1am.h
implementation file	OOP1am.cpp
application – client program	OOp1amClient.cpp

interface file OOP1am.h

```
#ifndef OOP1AM_H  // used to avoid multiple
  definitions
#define OOP1AM_H

class SmallObj
{
 private: int somedata;

 public:  void SetData(int);
          void ShowData();
};
#endif
```

implementation file OOP1am.cpp

```
#include "oop1am.h"
#include <iostream>
using namespace std;

// implementation file oop1am.cpp
void SmallObj::SetData(int d)
      { somedata = d; }

void SmallObj::ShowData()
 { cout << "\nData is =" << somedata; }
```

application – client program OOp1amClient.cpp

```
// source text of client program using
   SmallObj class library
#include "oop1am.h"
#include <iostream>
using namespace std;
void main()
{
   SmallObj s1, s2;

   s1.SetData(67);    s2.SetData(123);

   s1.ShowData();     s2.ShowData();
}
```

C++ обекти – типове данни
Клас *Distance* OOP1ca.cpp, OOP1ca.exe

```
class Distance
 {
 private:   int feet;    float inches;
 public:
        void SetDist(int ft, float in)
             { feet = ft; inches = in; }
  void GetDist()
    { cout <<"\nEnter feet:" ;  cin >> feet;
      cout <<"\nEnter inches:"; cin >> inches;
        }
  void ShowDist()
    {
  cout <<"Feet=" << feet << " Inches="<<inches;
        }
};
```

Клас *Distance*

```
void main ()
{
 Distance d1, d2;

 d1.SetDist(3, 5.6);
 cout << "\nDistance components:  ";
  d1.ShowDist();

 d2.GetDist();
 cout << "\nDistance components:  ";
  d2.ShowDist();

 }
```

C++ обекти – елементи с общо предназначение
Клас *Counter* OOP1da.cpp, OOP1da.exe

```
class Counter
{
  private: unsigned count;
  public:
  void SetCount(int val) { count = val; }
  void GetData()
    { cout <<"\nEnter data:"; cin >> count; }
  void ShowData()
    { cout <<"\nData count is:" << count;}
  void IncCount() { count++; }
  unsigned GetCount() { return count; }
};
```

Клас *Counter*

```
void main()
{
    Counter c1;
    c1.GetData(); c1.ShowData();
    c1.IncCount(); c1.ShowData();
    cout << "\n" << c1.GetCount(); c1.ShowData();

    Counter c2;
    c2.SetCount(100);
    cout << "\n\nCounter c2 =" << c2.GetCount();
    c2.IncCount();
    c2.IncCount();
    cout << "\n\nCounter c2 =" << c2.GetCount();
 }
```

C++ обекти

Конструктори и Деструктори

клас Counter	клас Distance
OOP1db.cpp	OOP1cb.cpp
OOP1db.exe	OOP1cb.exe

Конструктори и деструктори
Клас *Counter*

```
class Counter
{
 private: unsigned count;
 public:
        Counter() { count = 0; }
        Counter(int val) { count = val; }
        void SetCount(int val) { count = val; }
        void GetData() { cout <<"\nEnter data:"; cin >> count;}
        void ShowData() { cout <<"\nData count is:" << count;}
        void IncCount() { count++; }
        unsigned GetCount() { return count; }
};
```

Конструктори и деструктори
Клас *Counter*

```
void main()
{
    Counter c1;
    c1.GetData(); c1.ShowData(); c1.IncCount();
   c1.ShowData();
    cout << "\n" << c1.GetCount();
    c1.ShowData();

    Counter c2(100);
    cout << "\n\nCounter c2 =" << c2.GetCount();
    c2.IncCount();
    c2.IncCount();
    cout << "\n\nCounter c2 =" << c2.GetCount();
 }
```

Конструктори и деструктори
Клас *Distance*

```
class Distance {
 private: int feet; float inches;
 public:
          Distance() { feet = 0; inches = 0.0; }
         Distance(int ft,float in) {feet=ft;inches=in;}
          void SetDist(int ft, float in)
                { feet = ft; inches = in; }
          void GetDist()
          {     cout <<"\nEnter feet:" ;  cin >>feet;
                cout <<"\nEnter inches:"; cin >> inches;
          }
          void ShowDist()
          {
        cout <<"Feet=" << feet <<"    Inches="<< inches;
          }
};
```

Конструктори и деструктори
Клас *Distance*

```
void main ()
{
 Distance d1, d2;
 d1.SetDist(3, 5.6);
 cout << "\nDistance components:  ";    d1.ShowDist();

 d2.GetDist();
 cout << "\nDistance components:  ";    d2.ShowDist();

 Distance d3, d4(7, 8.9);
 cout << "\nDistance components:  ";    d3.ShowDist();
 cout << "\nDistance components:  ";    d4.ShowDist();
}
```

Обекти – аргументи на функции

Клас Distance

```
Distance dist1(5,6.8),dist2(3, 4.5), dist3;
```

Задача: да се съберат (сума на) две дължини чрез следния метод:

```
dist3.AddDist1(dist1, dist2);
```

OOP1e.cpp OOP1e.exe

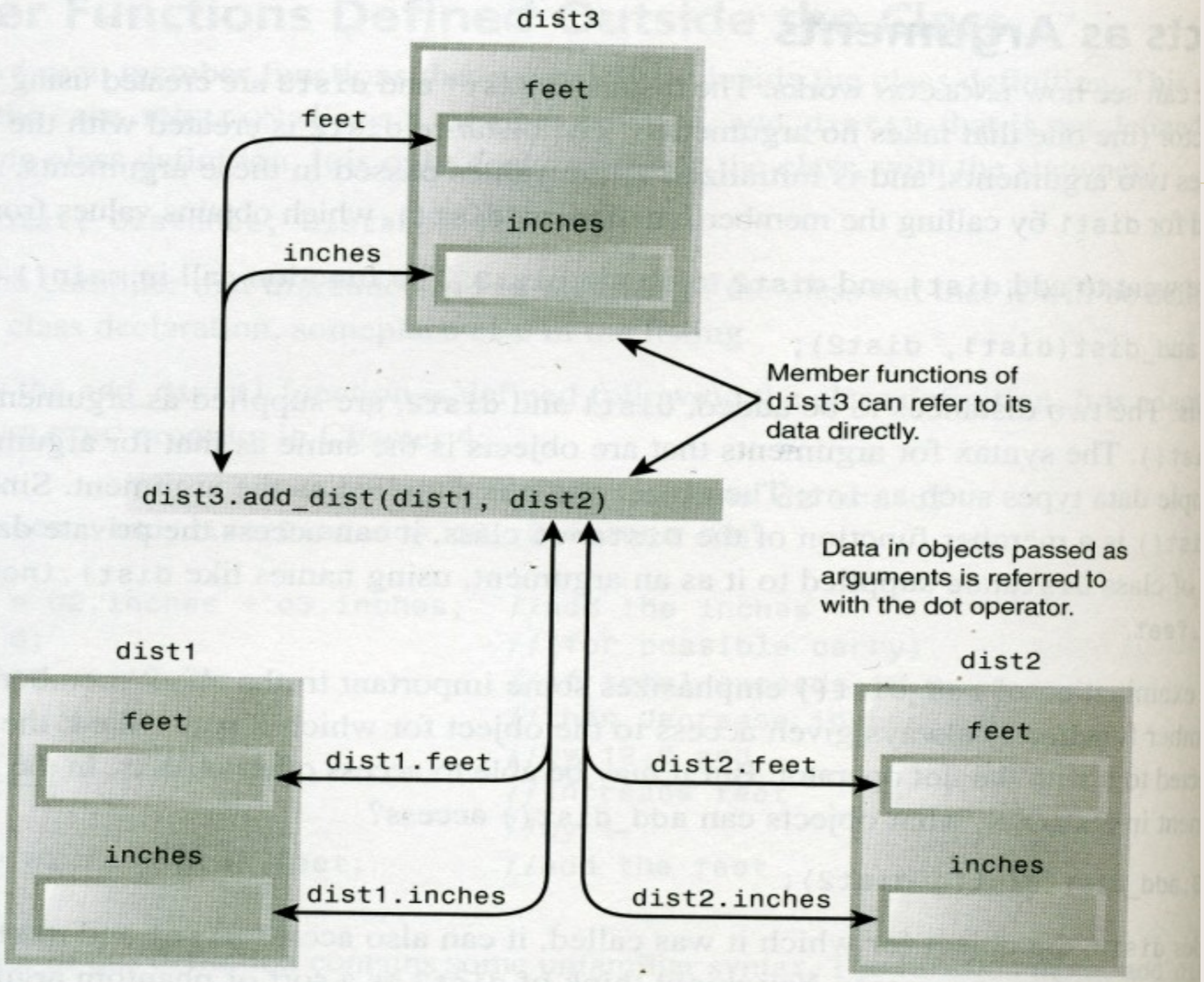


FIGURE 6.6

Result in this object.

Обекти – аргументи на функции

```
// void AddDist1(Distance d1, Distance d2);
//
        void AddDist1(Distance d1, Distance d2)
        {
              feet = d1.feet +d2.feet;
              inches = d1.inches + d2.inches;
              if (inches >= 12.)
               {
               inches -= 12.;   feet++;
               }
        }
```

Обекти – аргументи на функции

```
//      Member function defined outside the class
//      using scope resolution operator

 void Distance::AddDist1(Distance d1, Distance d2)
                {
                      feet = d1.feet +d2.feet;
                      inches = d1.inches + d2.inches;
                      if (inches >= 12.)
                       {
                       inches -= 12.;   feet++;
                       }
                }
```

Обекти, връщани от функции

Клас Distance

```
Distance dist1(5,6.8), dist2(3, 4.5), dist4;
```

Задача: да се съберат (сума на) две дължини чрез следния метод:

```
dist4 = dist1.AddDist2(dist2);
```

OOP1f.cpp OOP1f.exe

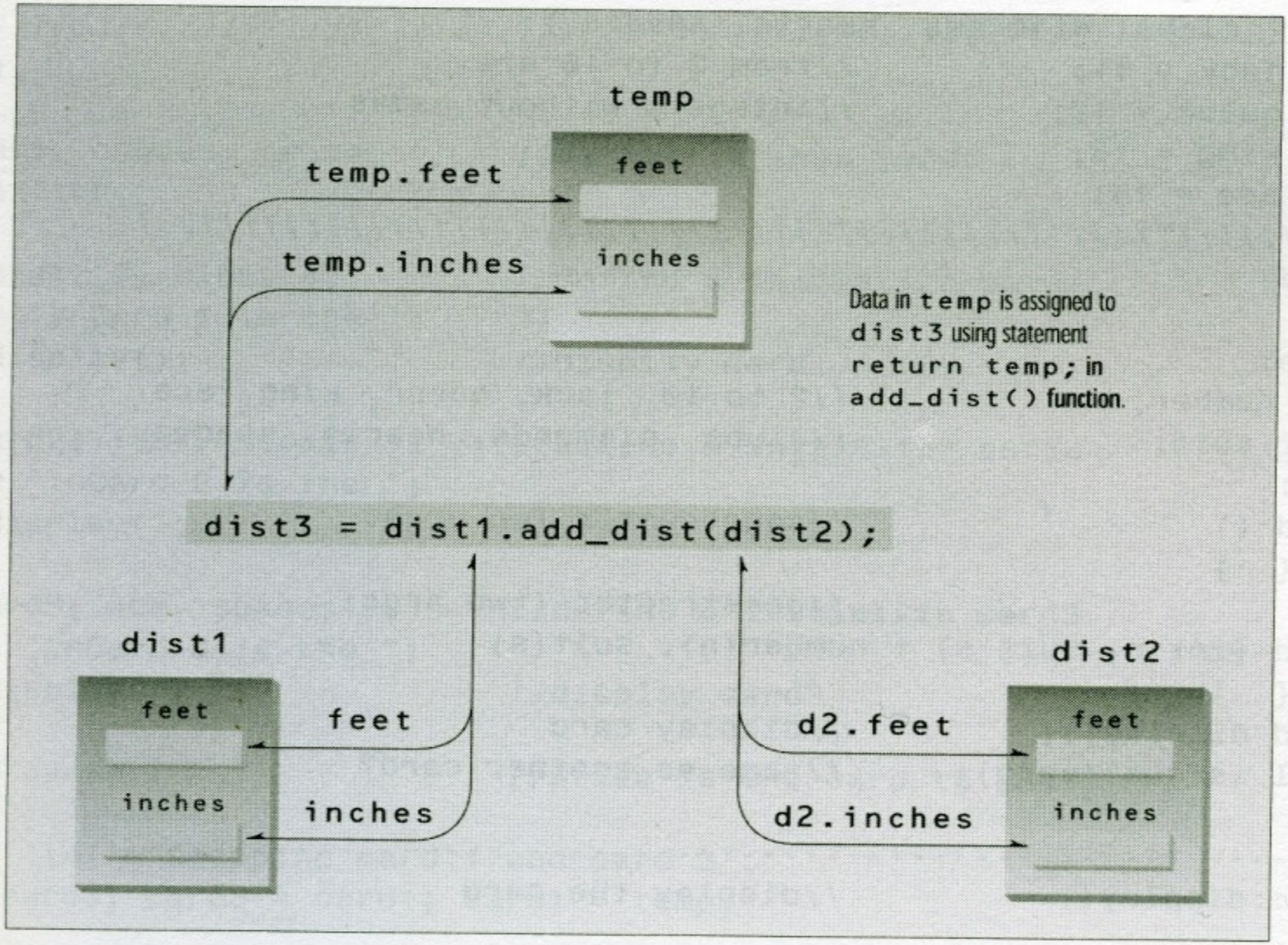


FIGURE 6.7

Result returned from the temporary object.

Обекти, връщани от функции

```
Distance AddDist2(Distance d1)
 {
      //  first version of source text
      int ft; float in;
      ft = feet + d1.feet;
      in = inches + d1.inches;
      if (in >= 12.)
        {
             in -= 12.;   ft++;
        }
      return Distance(ft, in);//anonymous object
 }
```

Обекти, връщани от функции

```
//  second  version of source text
Distance AddDist2(Distance d1)
 {
      Distance temp;
      temp.feet = feet + d1.feet;
      temp.inches = inches + d1.inches;
      if (temp.inches >= 12.)
        {
             temp.inches -= 12.;   temp.feet++;
        }
      return temp;
 }
```

Член функции, дефинирани вън от класа

Операция за принадлежност

:: scope resolution operator

Структури и Класове

- Ползват се по един и същ начин
- Единствена формална разлика:
  - При class: елементи са *private* by default
  - При struct: елементи са *public* by default

Класове, Обекти и Памет

Всички обекти от даден клас ползват едни и същи член функции.

Всички обекти от даден клас ползват свои уникални член данни.

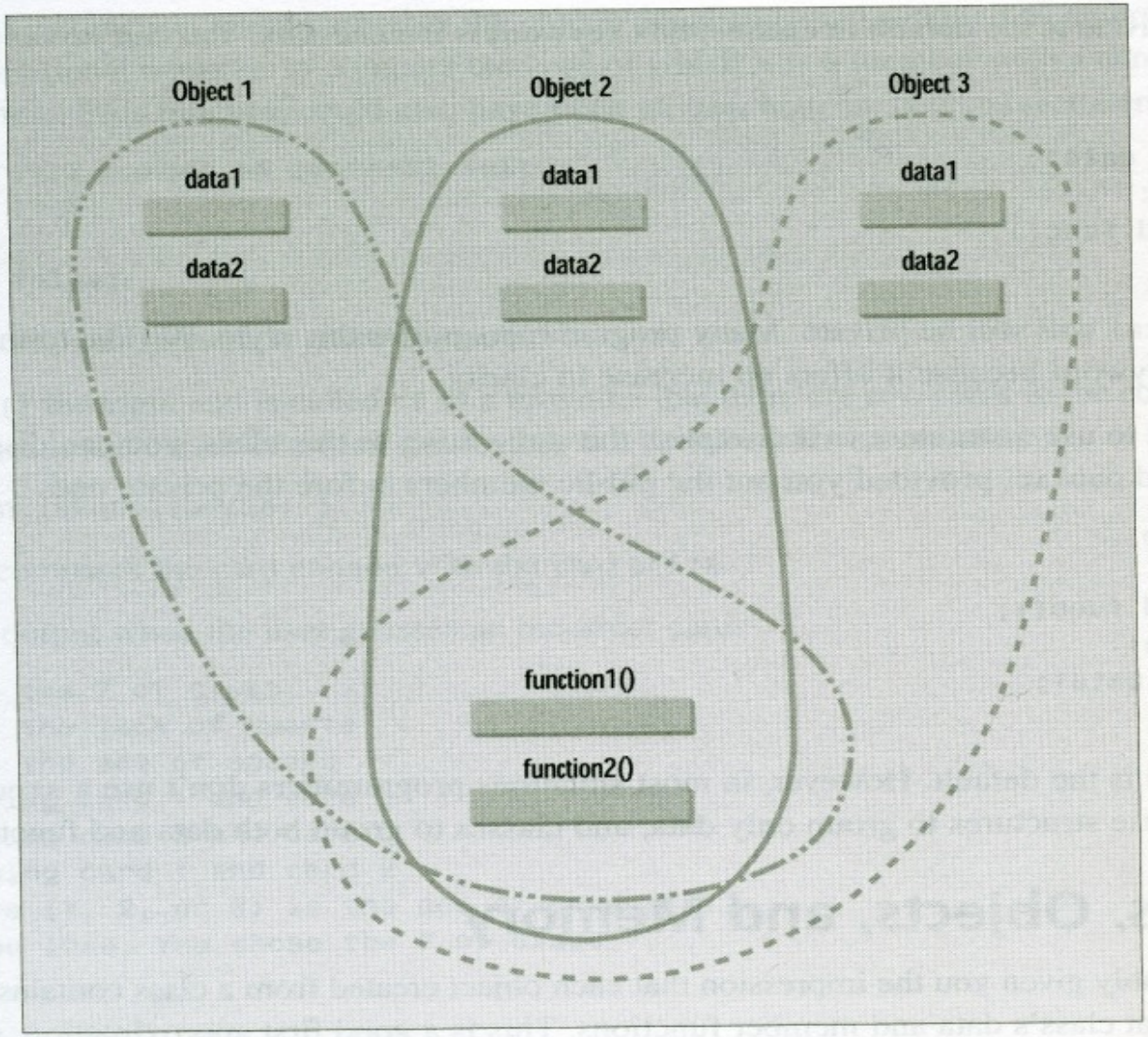
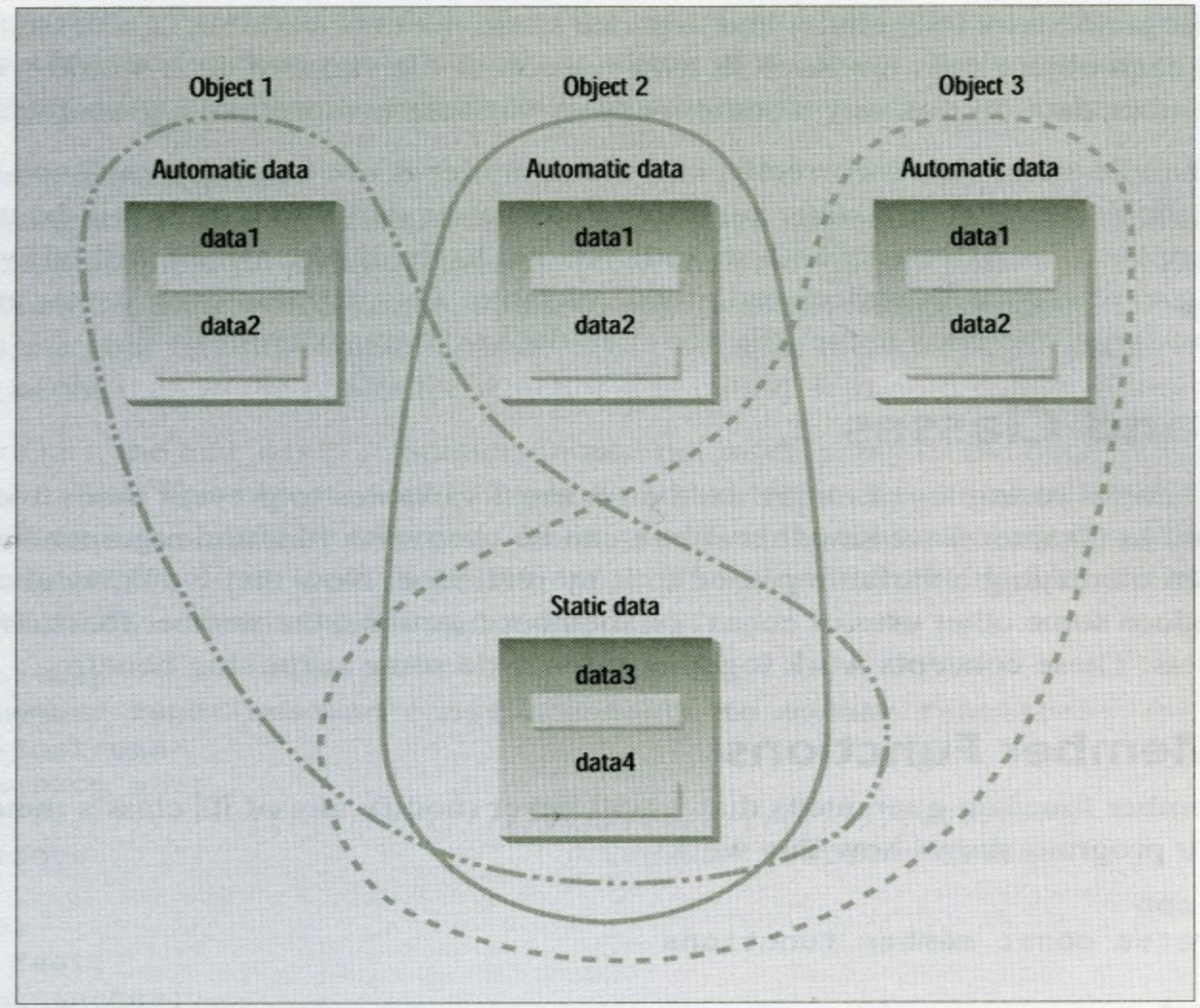


FIGURE 6.8

Objects, data, functions, and memory.

Класове, Обекти и Памет

Служебната дума static прави единствено копие на член данни, независимо колко обекта от даден клас са създадени.



URE 6.9
ic versus automatic member variables.

Статични данни на клас – отделна декларация и дефиниция

```
class foo
 {
   private: static int count;  //
  declaration only
   public: foo() { count++; }
   int getcount() { return count; }
 }:
 int foo::count = 0;      // definition
 void main()
 {
  foo f1, f2, f3;
  cout << '\n' << count;
 }
```

const и класове

const член функции на клас гарантират, че те няма да променят член данни на класа

```
class aClass
 { private: int alpha;
    public: void nonFunc() // non
  const function
          { alpha = 99; }  // OK
    void conFunc() const // const
  member fun
          { alpha = 999; } // ERROR
 };
```

const и класове

const Objects

Когато обект се декларира *const*, той не може да се модифицира. Следователно, само *const* член функции могат да се ползват с такъв обект, защото тяхната употреба гарантира, че обектът няма да се модифицира.

Lafore, pp 255

Еволюция на концепцията class

- Метод тип ShowData() не се прилага. Вмето него се въвеждат двойка методи за двупосочен достъп отделно до всички член данни:
- Метод Accessor (getX())
- Метод Mutator (setX())

- Microsoft развива надстройка - property

C++ Modern Object – native code

```
using namespace std;
class ModernObject    {
private: int x;

        // constructors
public: ModernObject() { x = 0; }
    public: ModernObject(int par) { x = par; }

        // obsolete method Show...type()
    public:  void showModernObject() { cout << endl << "Data = " << x ;  }

        // method accessor
    public:  int getModernObject() { return x; }

        // methods mutators
    public:  void setModernObject(int par) { x = par; }
    public:  void setModernObject(int par1, int par2) { x = par1 + par2; }
    public:  void setModernObject(int par1, int par2, int par3) { x = par1 + par2 + par3; }

        // property
//   properties not supported in unmanaged code
// end of property
};  // end of class ModernObject
```

C++ Modern Object – native code

```
void main()
{

 ModernObject a;
 ModernObject b(15);
 ModernObject c;

a.showModernObject();b.showModernObject();

// test accessor and mutators methods
c.setModernObject(20);       cout << endl << c.getModernObject();
c.setModernObject(20,50);    cout << endl << c.getModernObject();
c.setModernObject(20,30,40); cout << endl << c.getModernObject();

/*     // test property
// no properties, =>> ,no statements to test properties
*/
}
```

C++ Modern Object – managed code

```
using namespace System;
__gc class ModernObject    {
private: int x;

public: ModernObject() { x = 0; } // constructors
public: ModernObject(int par) { x = par; }

      // obsolete metod Show...type()
   public:  void showModernObject()  {
Console::Write("Data = "); Console::WriteLine(x) ;  }
      // method accessor
   public:  int getModernObject() { return x; }
      // methods mutators
   public:  void setModernObject(int par) { x = par; }
   public:  void setModernObject(int par1, int par2) { x = par1 + par2; }
   public:  void setModernObject(int par1, int par2, int par3) { x = par1 + par2 + par3; }

      // property  PropertyX
   public:  __property int get_PropertyX()      {            return x;          }

public:  __property void set_PropertyX(int par) {             x = par;          }

};  // end of class ModernObject
```

C++ Modern Object – managed code

```
void main( )
{
   ModernObject *a = new ModernObject();
ModernObject *b = new ModernObject(15);
ModernObject *c = new ModernObject();

a->showModernObject();b->showModernObject();

// test accessor and mutators methods
c->setModernObject(20);       Console::WriteLine( c->getModernObject() );
c->setModernObject(20,50);    Console::WriteLine( c->getModernObject() );
c->setModernObject(20,30,40); Console::WriteLine( c->getModernObject() );

    // test property
ModernObject *f = new ModernObject(24); Console::WriteLine(f->getModernObject());
   // test get property
Console::WriteLine(f->PropertyX);
   // test set property and get property
f->PropertyX = 135; Console::WriteLine(f->PropertyX);

Console::ReadLine();
 }
```

C# Modern Object – get(), set(), properties

```
class ModernObject      {
        private int x;
        public ModernObject() { x = 0; } // constructors
        public ModernObject(int par) { x = par; }
        // method accessor
        public int getModernObject() { return x; }
        // methods mutators
        public void setModernObject(int par) { x = par; }
        public void setModernObject(int par1, int par2) { x = par1 + par2; }
        public void setModernObject(int par1, int par2, int par3) { x = par1 + par2 + par3; }

        public int PropertyX            // property
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        } // end of property
    } // end of class ModernObject
```

C# Modern Object – get(), set(), properties

```
class Program
    {
        static void Main(string[] args)
        {
            ModernObject a = new ModernObject();
            ModernObject b = new ModernObject(15);
            ModernObject c = new ModernObject();
// test accessor and mutators methods
            c.setModernObject(20); Console.WriteLine(" {0}", c.getModernObject());
            c.setModernObject(20, 50); Console.WriteLine(" " + c.getModernObject());
            c.setModernObject(20, 30, 40); Console.WriteLine(" " + c.getModernObject());

            // test accessor and mutators methods
            ModernObject f = new ModernObject(115);
                 Console.WriteLine(" " + f.getModernObject());
            f.setModernObject(125); Console.WriteLine(" " + f.getModernObject());

            // test get property
            Console.WriteLine(" " + f.PropertyX);
            // test set property and get property
            f.PropertyX = 135; Console.WriteLine(" " + f.PropertyX);
        }
    }
```

Java Modern Object –get(), set()

```
class ModernObject     {
        private int x;
        public ModernObject() { x = 0; }
        public ModernObject(int par) { x = par; }

        // method accessor
        public int getModernObject() { return x; }

        // methods mutators
        public void setModernObject(int par) { x = par; }
        public void setModernObject(int par1, int par2) { x = par1 + par2; }
        public void setModernObject(int par1, int par2, int par3) { x = par1 + par2 + par3; }

    } // end of class ModernObject

public class ModernObjectProg1 {
    public static void main(String[] args) {
      ModernObject a = new ModernObject();    ModernObject b = new ModernObject(15);
 ModernObject c = new ModernObject();         ModernObject d = new ModernObject();
 ModernObject e = new ModernObject();

        // test accessor and mutators methods
c.setModernObject(20);       System.out.println(" " + c.getModernObject() );
c.setModernObject(20,50);    System.out.println(" " + c.getModernObject() );
c.setModernObject(20,30,40); System.out.println(" " + c.getModernObject() );
  } // end of method main()
} // end of class ModernObjectProg1
```

Част 2

ООП
Наследяване

Съдържание:

- Базов клас и Породен(и) клас(ове);
- Контрол на достъпа;
- Йерархия от класове;
- Множествено наследяване.

3-те главни ООП черти

- Капсулиране/Скриване на Данни
- **Наследяване**
  - Процес за създаване на нови класове, наречени породени класове, от съществуващи класове
- Полиморфизъм

Концепцията **Наследяване**

- **Inheritance** е процес на създаване на нови класове, наречени породени класове, от съществуващи класове, наречени базови. Породеният наследява базовия, като добавя и свои собствени елементи – данни и/или методи.
- Базовият клас не се влияе, не се променя.
- Породеният клас е функционално по-богат от базовия клас.

Термини

- АТД в ПЕ за ООП се наричат ***classes***.
- Инстанции на класовете се наричат ***objects***.
- Клас, създаден чрез наследяване, е ***derived class*** или ***sub-class***.
- Клас, от който се създава клас, е ***parent class*** или ***super-class***.
- ПП, които дефинират операциите над обекти в даден клас, са ***methods***.
- Обръщенията към методите са ***messages***.
- Колекцията от методи на обект се нарича ***message protocol*** или ***message interface***.

Наследяване: предимства

Основно предимство на наследяването е ***code reusability***.

"Once a base class is written and debugged, it needs not be touched but can be adapted to work in different situations."

R. Lafore

Връзки м/у класове(обекти)

http://www.zib.de/Visual/people/mueller/Course/Tutorial/tutorial.html

- **A-Kind-Of релация**
- **Is-A релация**
- **Part-Of релация**
- **Has-A релация**

Релация *A-Kind-Of*

Проблем: програма, която рисува различни обекти: точка, кръг, квадрат, триъгълник. За всеки обект се дефинира клас. Ето шаблон за клас точка в 2-Д координатна система:

```
class Point {
 attributes:      int x, y

 methods:  setX(int newX)
           getX()
           setY(int newY)
           getY()
       }
```

Релация *A-Kind-Of*

Първи подход: Независим клас за всеки примитив. Например класът за кръг се представя с координати на центъра и размер на радиуса:

```
class Circle {
 attributes:     int x, y, radius

 methods:  setX(int newX)
           getX()
           setY(int newY)
           getY()
           setRadius(int newRadius)
           getRadius()
          }
```

Релация *A-Kind-Of*

Сравнението на двата класа подсказва следното:

И двата класа имат член данни: x, y с еднакъв смисъл: координати на точка и координати на център. Член данните описват положението на асоцииран с тях обект като дефинират точка.

Двата класа имат един и същ набор методи, които обслужват член данните x, y.

Клас Circle “прибавя (adds)" нов член данна за радиус и методи за работа с него.

Знаейки свойствата на клас Point *we can describe a circle as a point plus a radius and methods to access it*. По този начин кръгът е вид точка “**a-kind-of**" point. Все пак, кръгът е разновидност на точка, нещо по-вече от точка.

Релация *A-Kind-Of*

На фигурата класовете са представени с правоъгълници. Имената им започват с главна буква. Стрелката показва посоката на релацията, която следва да се чете, интерпретира “Circle is ***a-kind-of*** Point."

Илюстрация на релация
"a-kind-of"

Релация *Is-A*

Разгледаната релация се използва на ниво класове и описва връзки между класове.

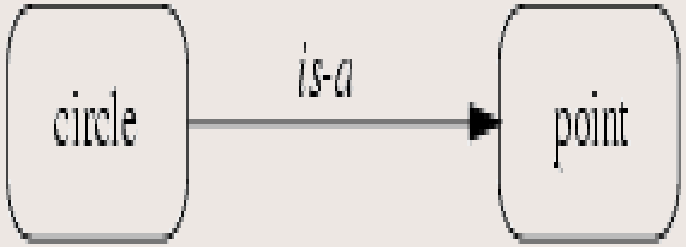
При работа с обекти от подобни свързани класове, релацията на ниво обекти се нарича “*is-a*"

Това е така, понеже бидейки клас *Circle* “a kind of “ клас *Point*, тогава обект от клас *Circle*, например *acircle*, се третира като точка “is a” *point*. На практика, всеки кръг има поведение на точка. Както точка, така и кръг могат да се движат в двете посоки чрез промяна на едната или другата координата.

За описание на тази релация, обектите се описват със заоблени правоъгълници и имената им започват с малка буква.

Релация *Is-A*

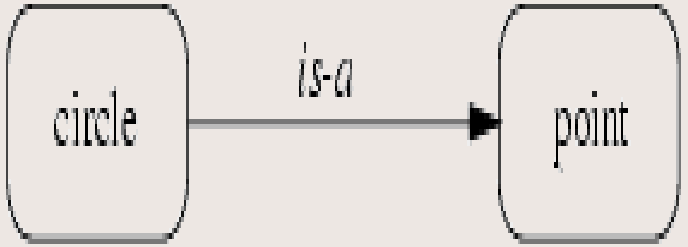
Илюстрация на релация "is-a"



Релации *a-Kind-of* и *Is-A*

Илюстрация на релация "a-kind-of"

Илюстрация на релация "is-a"



Релация ***Part-Of***

Тази релация е актуална, когато се създават нови обекти от други съществуващи обекти *build objects by combining them out of others*.

Пример: Програмата за рисуване на примитиви. Целим да създадем фигура, собствено лого, композирана от кръг и триъгълник.

Логото се състои от две части, или с други думи, кръгът и триъгълникът са части “part-of” от комплексната фигура лого:

Релация *Part-Of*

```
class Logo {
  attributes:
          Circle circle
          Triangle triangle

  methods:
          set(Point where)
  }
```

Илюстрацията на тази релация следва.

Релация ***Part-Of***

Илюстрация на релация
"part-of"

Релация *Has-A*

Тази релация е обратна на описаната релация part-of. Илюстрира се с насочена дъга в обратната посока спрямо посоката за релацията part-of.

Релация *Has-A*

Илюстрация на релация
"has-a"

Наследяване

Наследяване реализира релациите ***a-kind-of*** и ***is-a.*** На псевдо език пораждането на кръг от точка се представя така:

```
class Circle inherits from Point {
  attributes:  int radius
  methods: setRadius(int newRadius)
           getRadius()
  }
```

Клас Circle наследява член данните и методите на Point. Няма необходимост те да се дефинират повторно. Просто се ползват съществуващи данни и методи на базовия клас

Наследяване

Ето на ниво породени обекти пример как се активират методи на базовия клас и на породения клас

```
Circle acircle
acircle.setX(1)     /* Inherited from Point */
acircle.setY(2)
acircle.setRadius(3)  /* Added by Circle */
```

Наследяване

Релацията “***Is-a***" предполага, че обект от породения клас, т.е. кръг може да се ползва навсякъде, където се предполага работа с обект от базовия клас, т.е. Точка

Преместване на точка

```
move(Point apoint, int deltax)
  {
    apoint.setX(apoint.getX() + deltax)
  }
```

Преместване на кръг

```
Circle acircle
move(acircle,10)  /* Move circle by moving its center point */
```

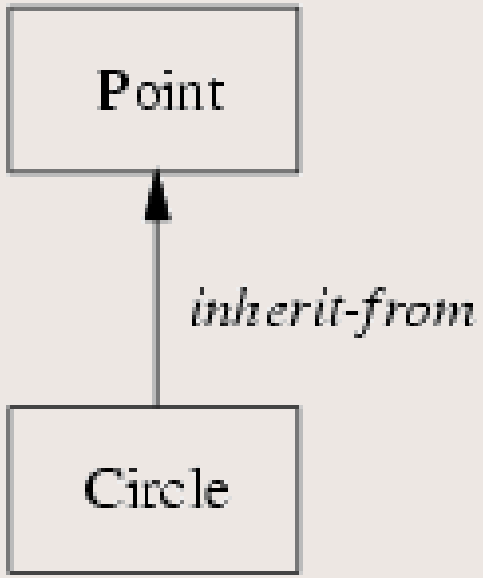
Наследяване

Definition (Inheritance) Inheritance *is the mechanism which allows a class A to inherit properties of a class B. We say "A inherits from B". Objects of class A thus have access to attributes and methods of class B without the need to redefine them.*

Definition (Superclass/Subclass) *If class A inherits from class B, then B is called* ***superclass*** *of A. A is called* ***subclass*** *of B.* Superclasses are also called *parent classes*. Subclasses may also be called *child classes*.

Наследяване базов/породен

Илюстрация на
граф наследяване



Point – базов / Circle - породен

- C++

```
class Point { …};
 class Circle :  public Point
 { … };
```

- Java

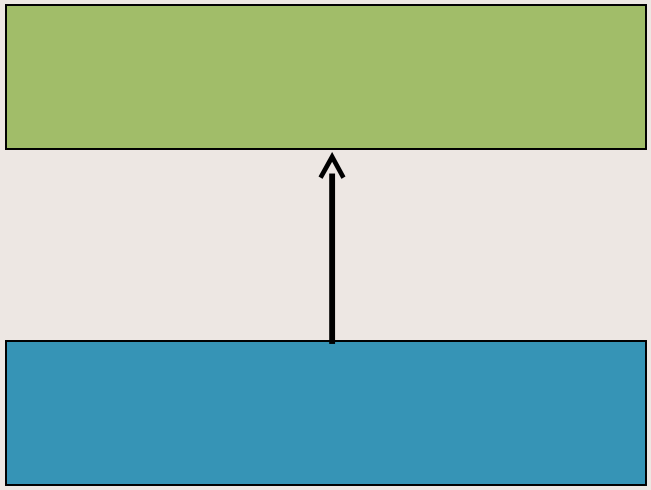
```
class Point { …}
 class Circle extends Point
 { … }
```

- C#

```
public class Point { … }
public class Circle  :  Point
{ … }
```

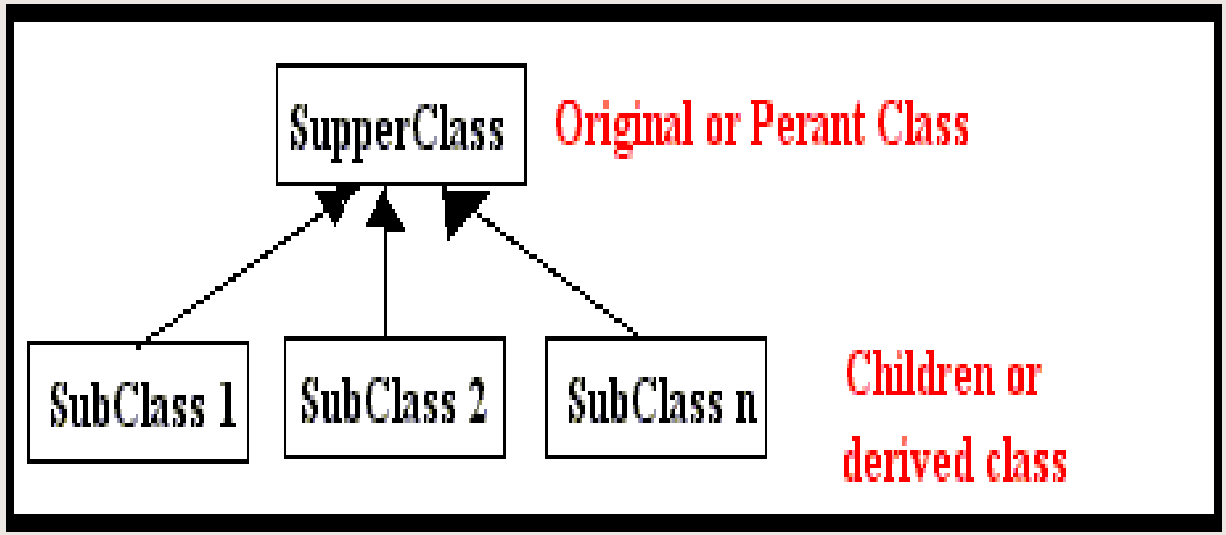
Релация Базов-Породен клас

Релацията базов породен клас илюстрирана на следната фигура:

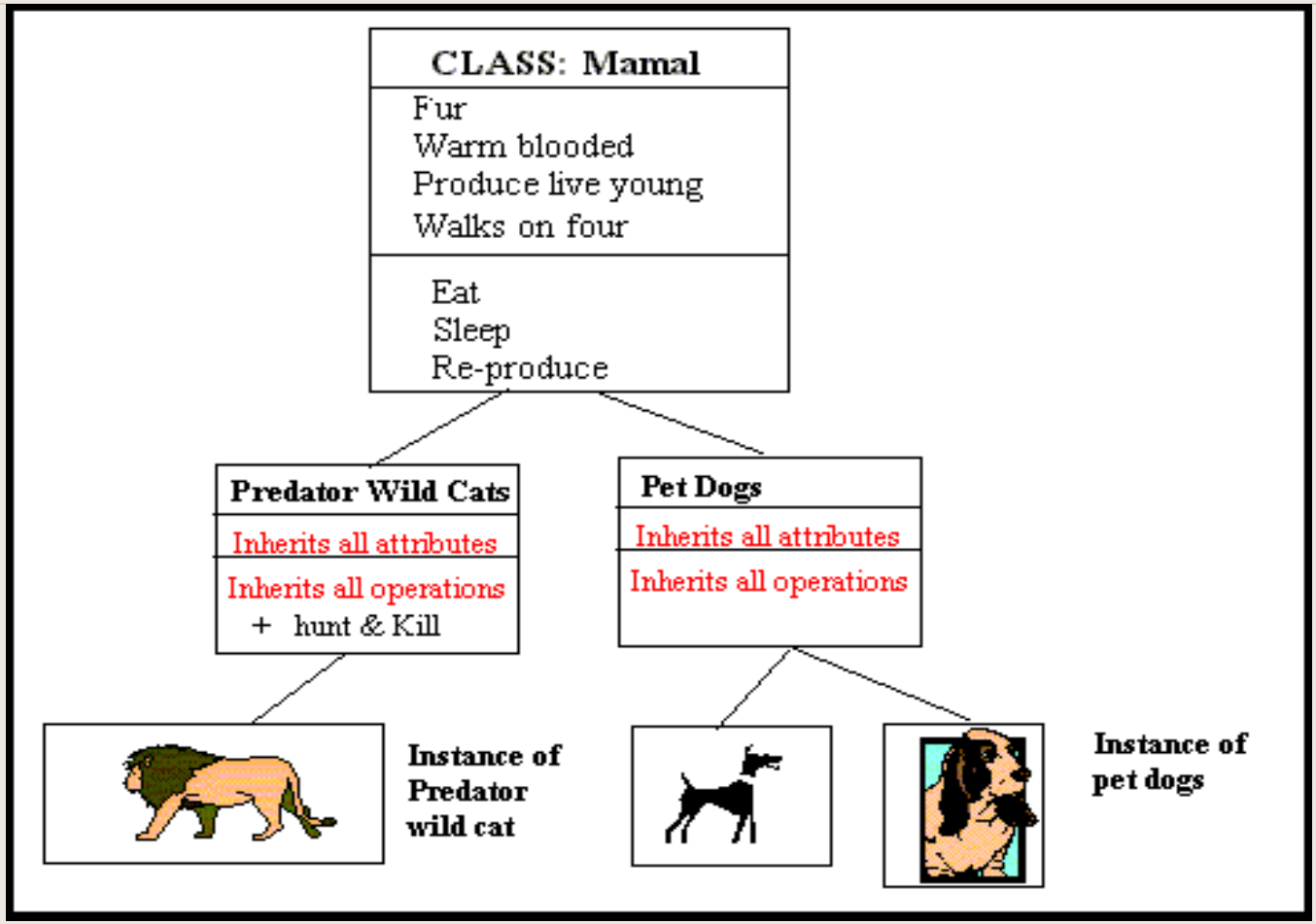


Пример

http://homepages.north.londonmet.ac.uk/~chalkp/proj/ootutor/inheritance.html

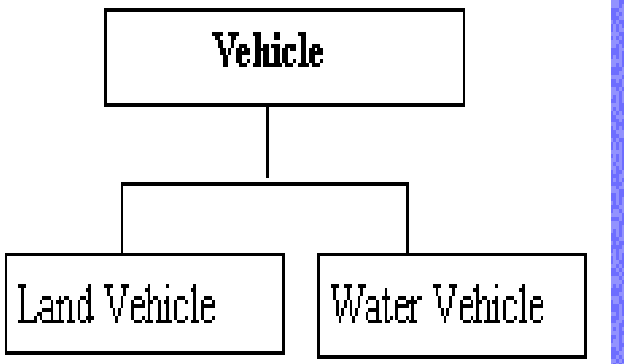


A **SubClass** inherits all the ***attributes*** and ***behaviors*** of the **SuperClass**, and may have additional attributes and behaviors.



Пример

- Пример за просто наследяване **Single Inheritance.**



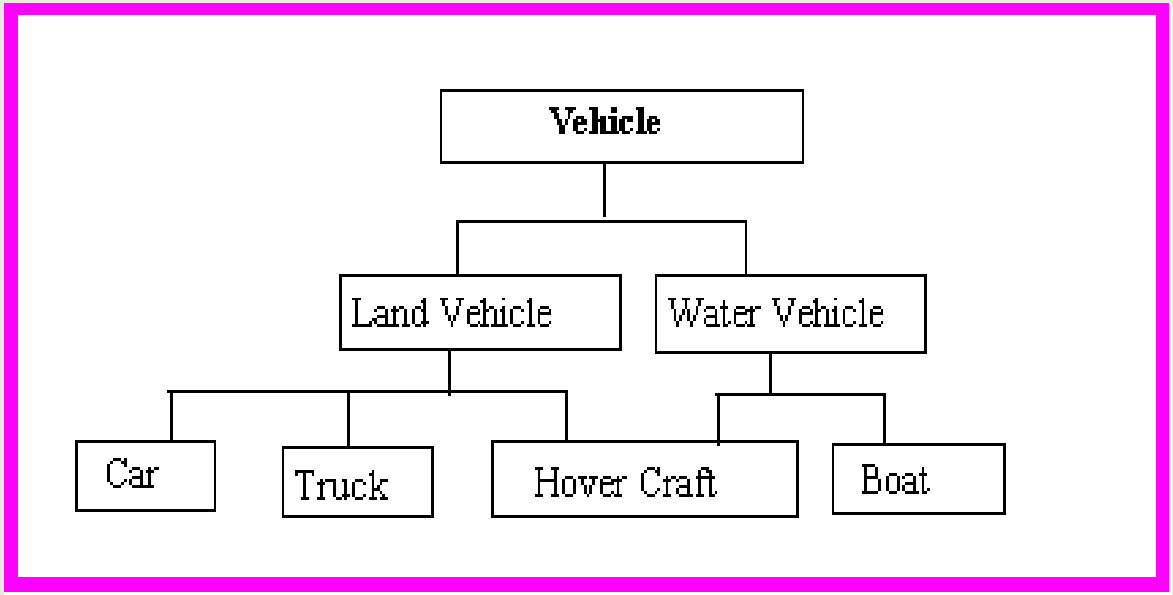
ATTRIBUTES

- registration Number
- seat Number
- engine Power
- capacity

OPERATIONS

- Started
- stoped
- parked
- reversed

Множествено наследяване



Задаване на Породен клас

```
class Base {    . . .    };
```

// Derived class **publicly** derived from Base class

```
class Derived : public Base {
  . . .
 };
```

// Derived class **privately** derived from Base class

```
class Derived : private Base {
  . . .
 };
```

Достъп до членове на базовия клас

- Атрибут за достъп ***protected***

```
class Base {
        private:   . . .

        protected:   . . .

        public: . . .
          };
```

Base class(es) and Derived class(es) - примери

Даден клас Counter ще служи като базов клас.

Нуждаем се от клас, чийто обекти броячи работят на изваждане /decrement/. Ще създадем нов, породен клас по две причини:

- Counter е добре настроен клас.
- Няма достъп до първичния код на Counter.

Създаване на породен клас

DecrementCounter (1st version)

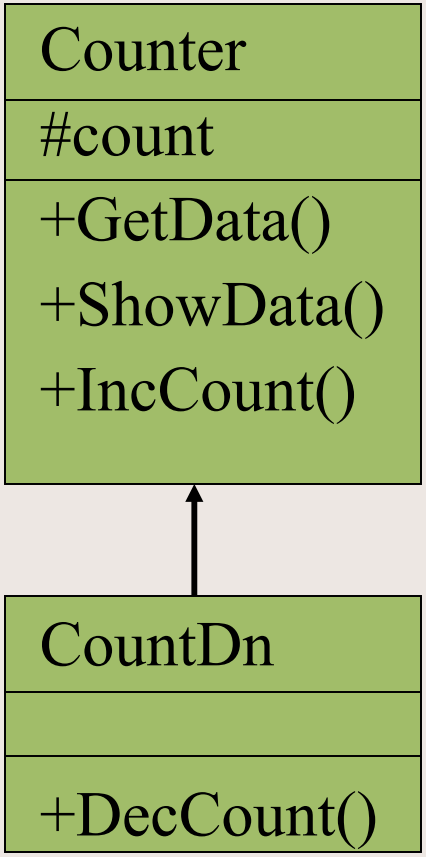
```
class CountDn:public Counter
  {
     . . .
  };
```

OOP2aa.cpp OOP2aa.exe

Class *CountDn* (1st idea)

```
class Counter
{
  protected: unsigned count;
  public:   void GetData() { cout <<"\nEnter data:"; cin >> count; }
            void ShowData() { cout <<"\nData count is:" << count;}
            void IncCount() { count++; }
};
class CountDn : public Counter
{
 public:   void DecCount() { count--; }
};
void main()
{
 Counter c1;   c1.GetData();   c1.ShowData();   c1.IncCount();   c1.ShowData();

 CountDn c3;  c3.GetData(); c3.incCount();  c3.IncCount();  c3.ShowData();
              c3.DecCount();  c3.ShowData():
}
```

Наследяване в UML клас диаграми



клас *CountDn* (работещо демо)

```
class Counter
{
  protected: unsigned count;
  public:    Counter() { count = 0; }
             Counter(int val ) { count = val; }
             void IncCount() { count++; }
             void GetData() { cout <<"\nEnter data:"; cin >> count;}
             void ShowData() { cout <<"\nData count is:" << count; }
             unsigned GetCount() { return count; }
             Counter  operator++(int ) { count++;  return Counter(count);  }
             Counter  operator++(    ) { count++;   return Counter(count);  }
};

class CountDn : public Counter
{
 public:  Counter operator--() { count--; return Counter(count);          }
};
```

клас *CountDn* (работещо демо)

```
void main()
{
 Counter c1;
 c1.GetData(); c1.ShowData(); c1.IncCount(); c1.ShowData();
 cout << "\n\n" << c1.GetCount();
 c1++;
 c1.ShowData();

 Counter c2;
 c2 = ++c1;   cout << "\nCounter c2 =" << c2.GetCount();
 c2 = ++c1;   cout << "\nCounter c2 =" << c2.GetCount();

 CountDn c3;
 ++c3; ++c3; ++c3;   --c3;
 cout << "\nCountDn c3 =" << c3.GetCount();

}
```

Конструктори на породен клас

DecrementCounter (2nd version)

```
class CountDn:public Counter
  {
     . . .
  };
```

OOP2ab.cpp OOP2ab.exe

Class *CountDn*

```
class Counter
{  protected: unsigned count;
   public:   Counter() { count = 0; }
             Counter(int val ) { count = val; }
             void IncCount() { count++; }
             void GetData() { cout <<"\nEnter data:"; cin >> count;}
             void ShowData() { cout <<"\nData count is:" << count; }
             unsigned GetCount() { return count; }
             Counter  operator++(int ) { count++;  return Counter(count);  }
             Counter  operator++(    ) { count++;   return Counter(count);  }
};

class CountDn : public Counter
{
 public:   CountDn() : Counter() { }
           CountDn(int val) : Counter(val)  { }
           Counter operator--() { count--; return CountDn(count);  }
};
```

Class *CountDn*

```
void main()
{
 Counter c1;
 c1.GetData();c1.ShowData();c1.IncCount();c1.ShowData();
 cout << "\n\n" << c1.GetCount();
 c1++;
 c1.ShowData();

 Counter c2;
 c2 = c1++;   cout << "\nCounter c2 =" << c2.GetCount();
 c2 = c1++;   cout << "\nCounter c2 =" << c2.GetCount();

 CountDn c3;
 c3++; c3++; c3++;   c3--;
 cout << "\nCountDn c3 =" << c3.GetCount();

 CountDn c4(100);
 c4--;
 cout << "\nCountDn c4 =" << c4.GetCount();
}
```

Препокриване - член функции

Overriding member functions

Функции със същото име в базов и породен клас

```
class Stack2 : public Stack
  {  . . . };
```

Правило: когато се извиква, отнася се за метода от породения клас. Ако се търси методът на базовия клас, то операция :: scope resolution.

OOP2b.cpp OOP2b.exe

Class *Stack2 : public Stack*

```
const int MAX = 100;
class Stack
  {  protected:     int st[MAX];  int sp;
     public:   Stack() {  sp = -1;  } // constructor
     void push(int var)     {  sp++;      st[sp] = var;}
   int pop(){int pom;  pom=st[sp]; sp--; return pom; }

   ~Stack() { cout << "\nGOOD Bye"; }  // Destructor
};
class Stack2 : public Stack
  {
  public: void push(int var)
       { if (sp<MAX-1)   Stack::push(var);
       else  {  cout << "\n Stack full\n"; exit(1);}
       }
       int pop()
       {  if (sp >=0 )   return Stack::pop();
       else  {  cout << "\n Stack empty \n"; exit(1);}
       }
};
```

Class *Stack2 : public Stack*

```
void main()
{

 Stack c1; cout << c1.pop();

 Stack2 M;    M.push(11);  M.push(22);  M.push(33);
              M.push(44);  M.push(55);  M.push(66);
 cout << "\n\nStack contents" << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;
 cout << M.pop() << endl;

}
```

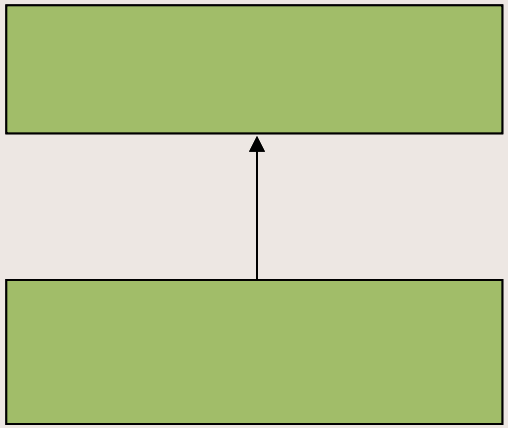
Йерархия от класове

- Base class – Derived class
- Base class – Derived classes
- Base cls – Derived/Base cls – Derived cls

Без ограничение в дълбочината на наследяване

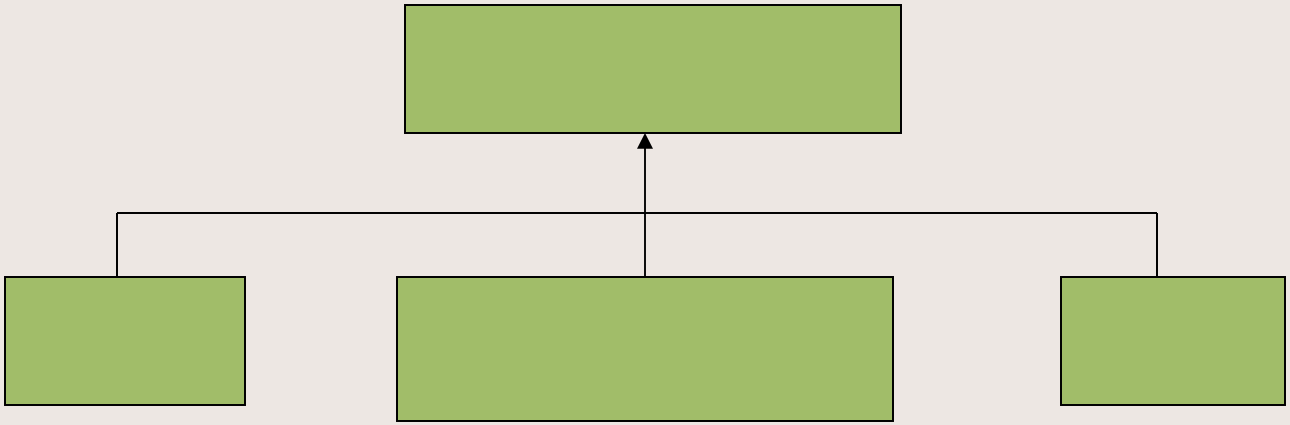
Йерархия от класове

- Base class – Derived class



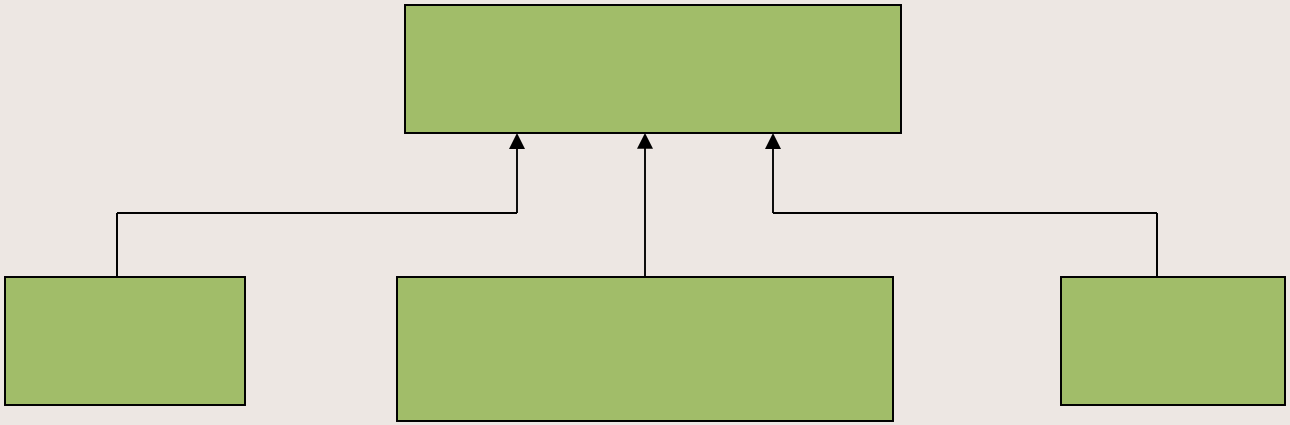
Йерархия от класове

- Base class – Derived classes



Йерархия от класове

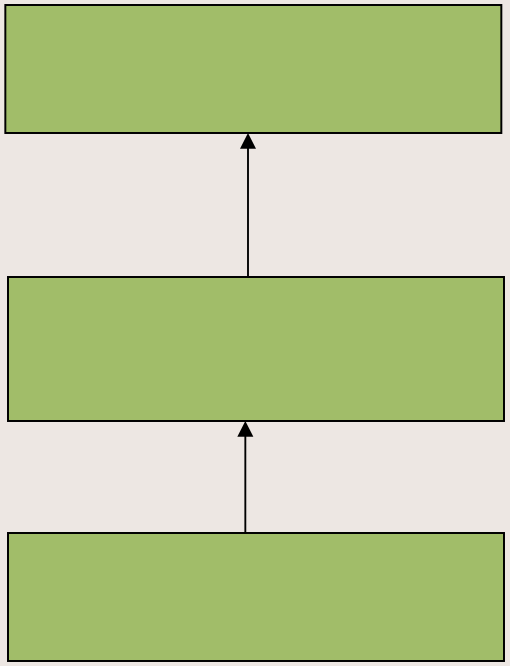
- Base class – Derived classes



Йерархия от класове

- Base cls – Derived/Base cls – Derived cls

Без ограничение в нивата на пораждане



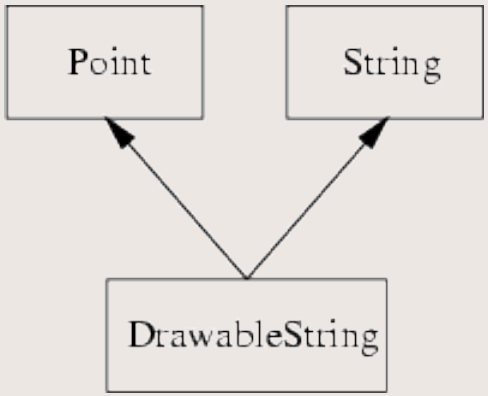
Множествено наследяване

Множественото наследяване – важен механизъм в ООП

Един подклас може да има два или повече супер-класове и да смесва техните свойства.

Множествено наследяване

Клас *DrawableString*, който наследява два базови класове *Point* и *String*.



Множествено наследяване

На псевдо език базовите класове се изброяват в списък, разделени със запетая:

```
class DrawableString inherits from Point, String {
attributes:
        /* All inherited from superclasses */

methods:
        /* All inherited from superclasses */
}
```

Multiple inheritance

Обекти от клас *DrawableString* имат поведение като point и като string.

Защото **drawablestring is-a point**, ние можем да го местим по екрана

```
DrawableString dstring
  ...
  move(dstring, 10)
  ...
```

Защото **drawablestring is-a string**, можем да добавяме текст:

```
dstring.append("The red brown fox ...")
```

Множествено наследяване

Възможни са конфликти на имена при схема на множествено наследяване като на фигурата долу

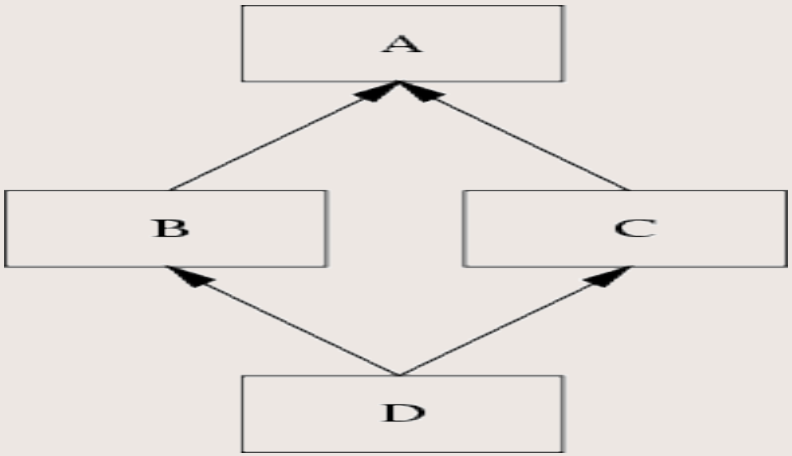
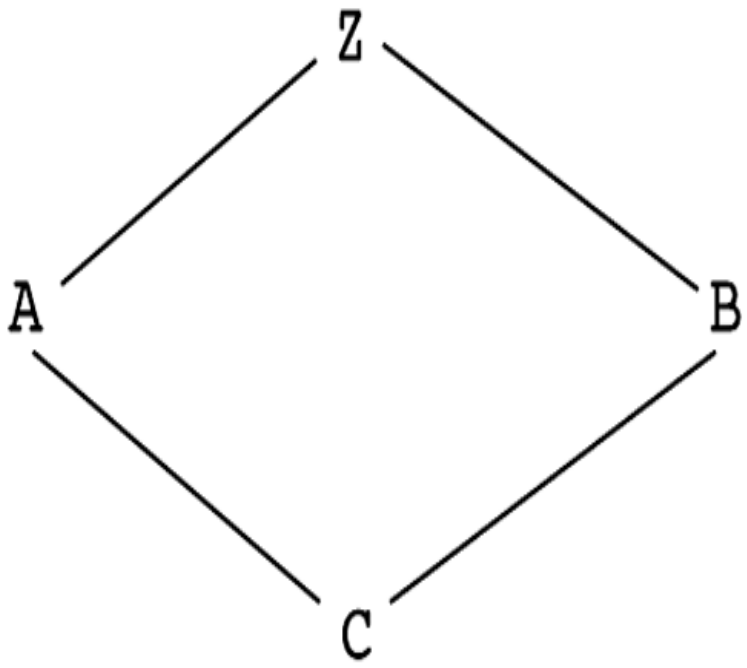


Figure 12.1

An example of diamond inheritance



Множествено наследяване

Пример: 2 Base classes, 3+1 Derived classes

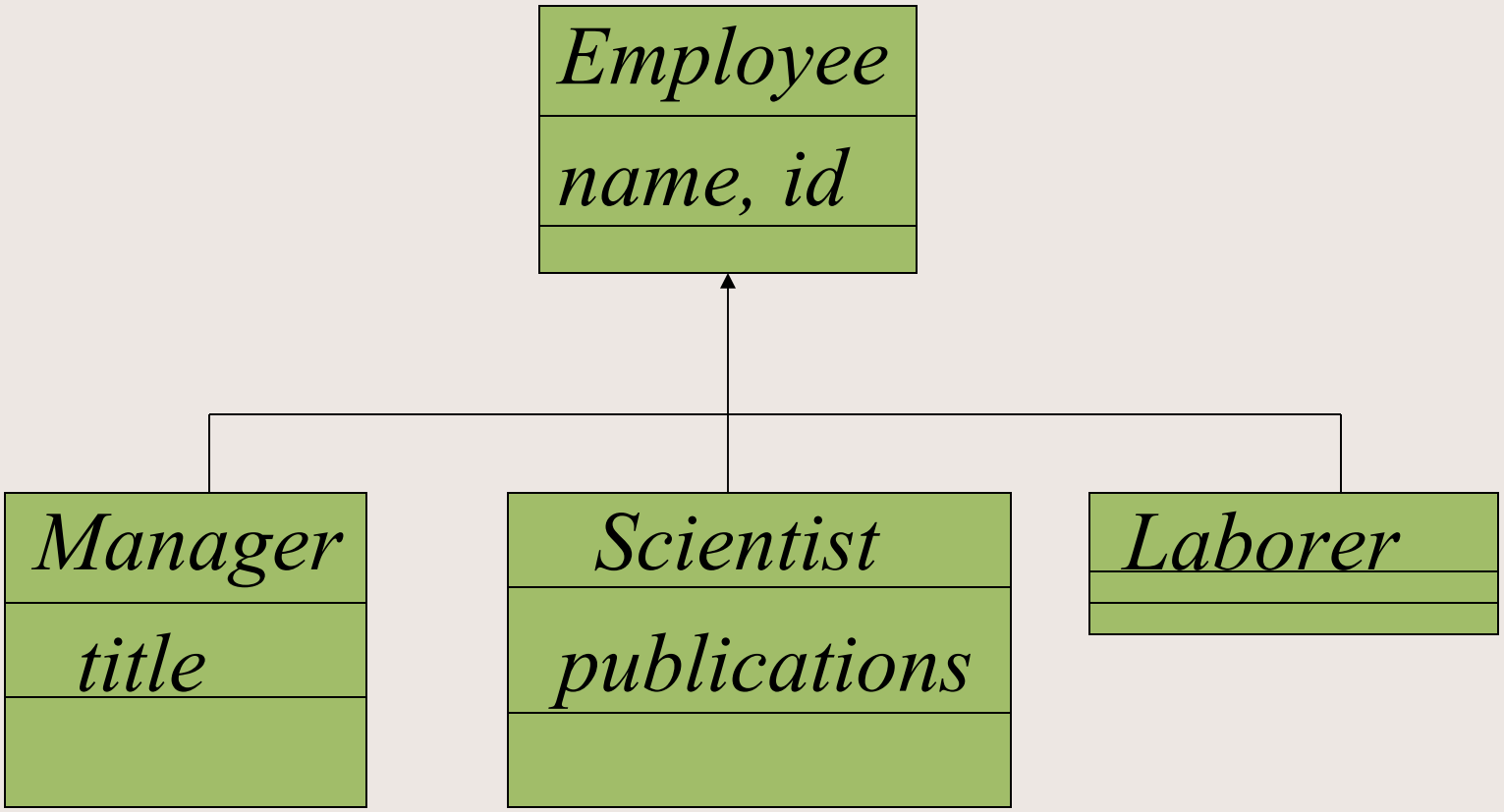
Базови класове:

```
class Employee
class Student
```

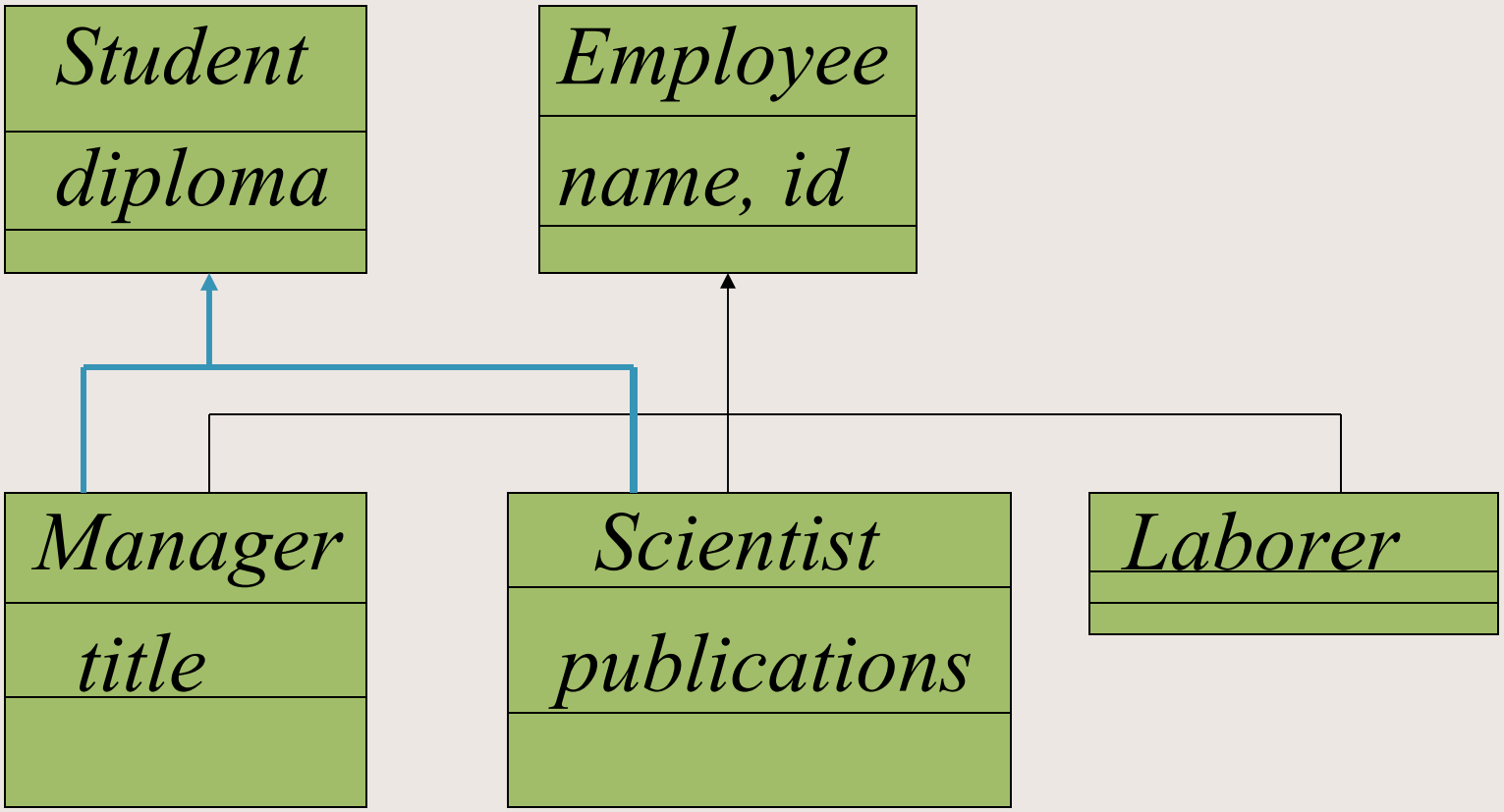
Породени класове

```
class Manager
class Scientist
class Laborer

class Foreman
```

Множествено наследяване- Да/Не?



Реално Множествено наследяване



Неоднозначности при множествено наследяване

```
class A { public: void Show()
 { cout<<"A"; } };
class B { public: void Show()
 { cout<<"B"; } };
class C : public A, public B {...};
void main() {
 C objc;
 objc.Show();    // ambiguous call
 objc.A::Show();    //   OK
 objc.B::Show();    //   OK
}
```

Пример с наследяване

```
class DistSign:public Distance
  {  . . . };
```

Конструктори в клас *DistSign*:

Двата конструктора извикват конструкторите на базовия клас и вършат още работа.

Член функции в клас *DistSign*:

Методите *GetDist(), ShowDist()* вкл. Извикване на overriden методи на базовия клас с ::

OOP2c.cpp OOP2c.exe

Class *DistSign : public Distance*

```
class Distance {   protected: int feet; float inches; public:    .... };
enum posneg {pos, neg};
class DistSign : public Distance
{
 private:  posneg sign;
 public:   DistSign() : Distance()  {  sign = pos;  }
           DistSign(int ft, float in, posneg sg=pos) : Distance(ft, in) { sign = sg; }
           void GetDist()
            {
                    Distance::GetDist(); // overriden function
                    char ch;   cout << "\nEnter sign + or -"; cin >> ch;
                    sign = (ch=='+')?pos:neg;
            }
           void ShowDist()
            {
                    cout << "\n" << (  (sign==pos)?"(+)":"(-)"    );
                    Distance::ShowDist(); // overridden function
            }
```

Class *DistSign : public Distance*

```
void main()
{
 cout << "\n Inheritance demo \n\n";
 DistSign alpha, beta(11, 5.65), gamma(8, 3.4, neg);

 alpha.ShowDist();  cout << "\t   DistSign alpha";   getch();

 beta.ShowDist();   cout << "\t   DistSign beta";    getch();

 gamma.ShowDist();  cout << "\t   DistSign gamma";   getch();

 DistSign delta;
 delta.GetDist();
 delta.ShowDist();  cout << "\t   DistSign delta";   getch();
}
```

Public и Private Наследяване

Пример:

Public и Private Наследяване

Пояснение относно квалификатори *private/public*

Метод на клас винаги има достъп до членовете на клас без значение дали са *private* или *public*.

НО! Обект на класа, деклариран от вън има достъп само (чрез dot операция за достъп) до *public* членове на класа. Няма достъп до *private* членове. Виж фиг>>

Public и Private Наследяване

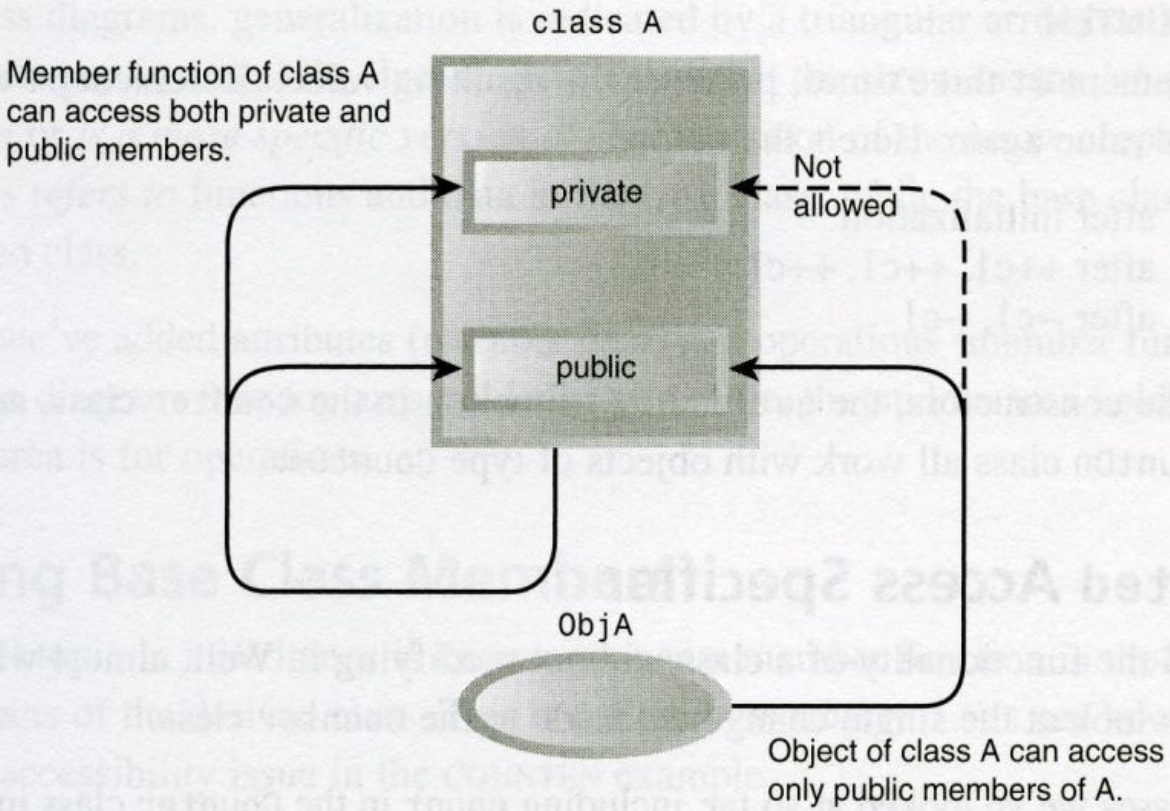


FIGURE 9.3

Access specifiers without inheritance.

Public и Private Наследяване

Пояснения при наследяване:

```
class Base { ... };
class Derv : public Base { ... };
```

В.: Член функции на породения клас имат ли достъп до членовете на базовия клас?

О.: Да, Член функции на породения клас имат достъп до членовете на базовия клас само, ако те са *public* или *protected*.

Public and Private Наследяване

Един *protected* член е достъпен от член функците на собствения си клас или от всеки клас, който е породен от собствения му клас.

Такъв *protected* член не е достъпен от ф-ии извън изброените по-горе класове, например ф-ята *main()*. Вж сл. фигура >>

Public and Private Наследяване

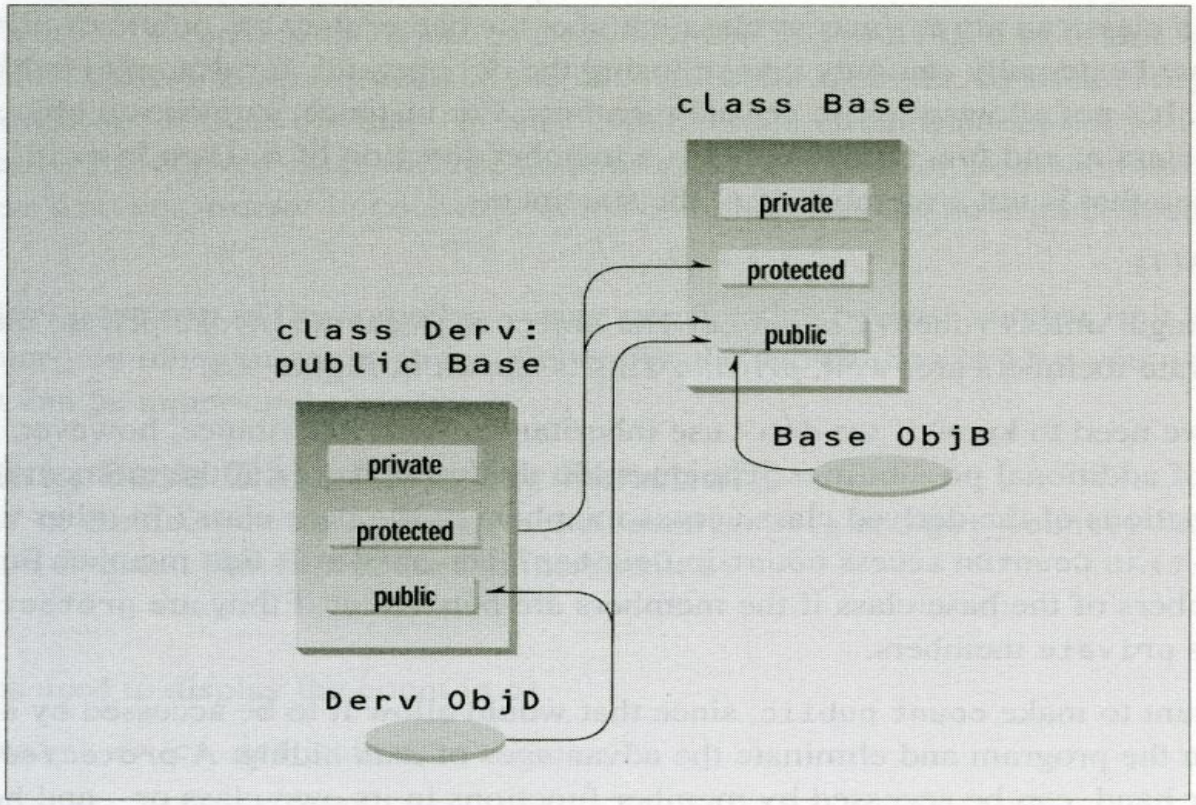


FIGURE 9.4
Access specifiers with inheritance.

Public and Private Наследяване

```
class A { . . . };
class B : public A { . . . };
class C : private A { . . . };
```

Дума *public* определя, че обекти от породения клас имат достъп до *public* методи на базовия клас.

Дума *private* определя, че обекти от породения клас нямат достъп до *public* методи на базовия клас. Т.к. Обектите нямат достъп до *private или protected* членове на класа, резултатът е, че никой член на базовия клас не е достъпен от обекти на породения клас. Виж сл. фигура >>

Public and Private Наследяване

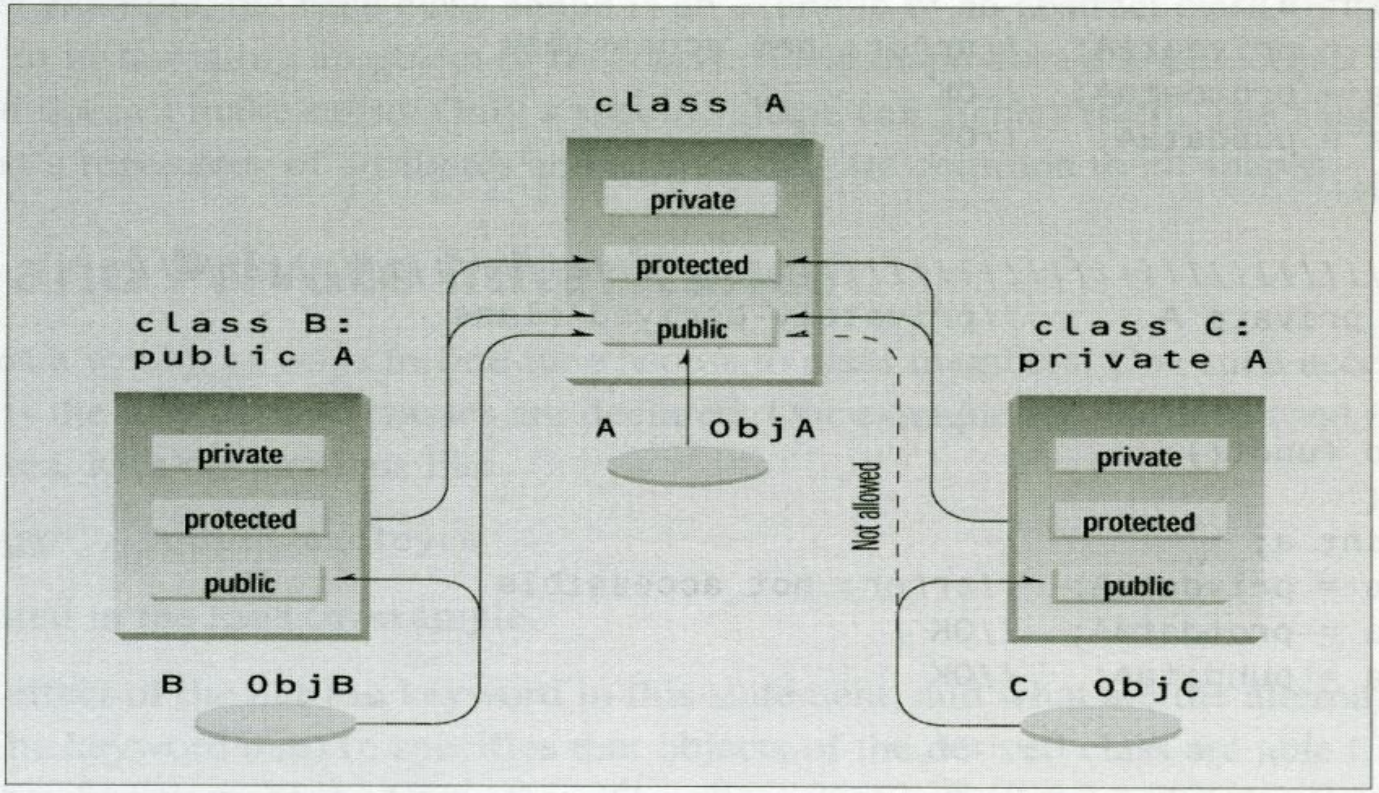


FIGURE 9.7
Public and private derivation.

Containership

- Класове в класове. Възможни релации:

B is **a kind of** A.

Релация основана на наследяване

```
class A { ... };
class B : public  A
{ . . . };
```

B **has an** object of class A.

Релация с независими класове

```
class A { ...};
     class B
{ . . .
 A obja;
};
```

Containership

- Classes within classes.

  Has-a relation се нарича още агрегация: Казва се:

  Library has a book.

  Invoice has a line item.

Aggregation е известна и като“part-whole” “част-цяло” релация.

The book is part of the library.

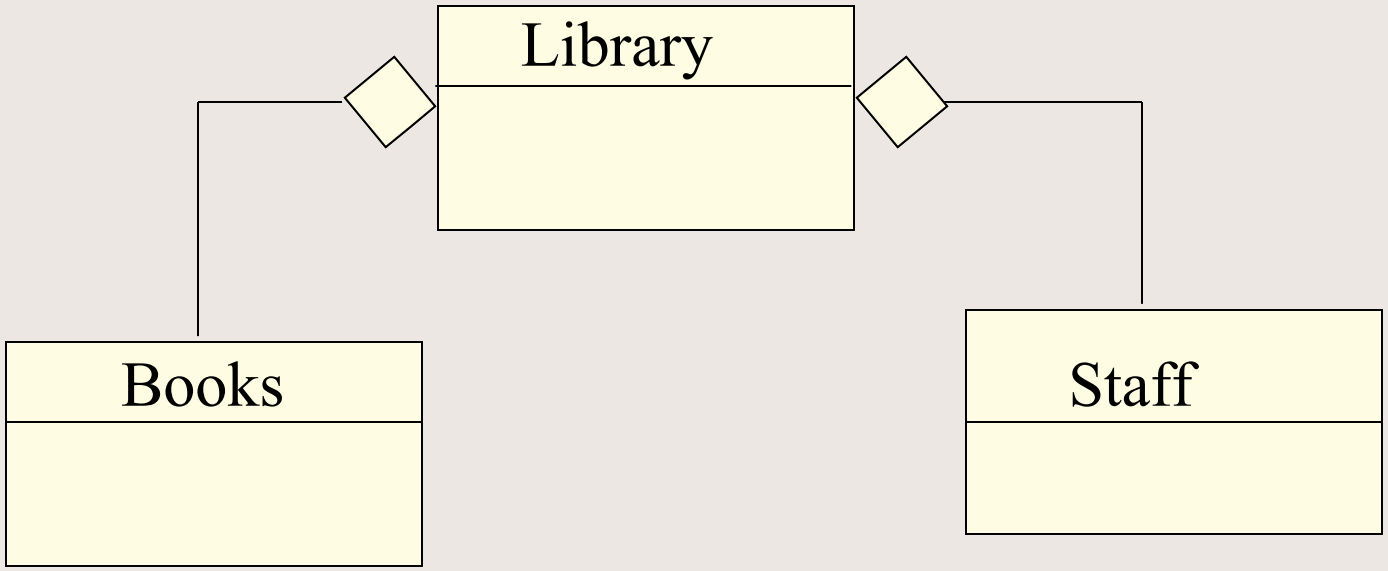
‘Has-a’ релация илюстрирана

- Програма като **Top-Down** низходяща йерархия от синтактични обекти /категории/:
  - Program // Program has function(s)
  - Function(s) // Function has statement(s)
  - Statement(s) // Statement has expression(s)
  - Expression(s) // Expression has operator(s) and operand(s)
  - Operator(s)
  - Operand(s)
  - Data

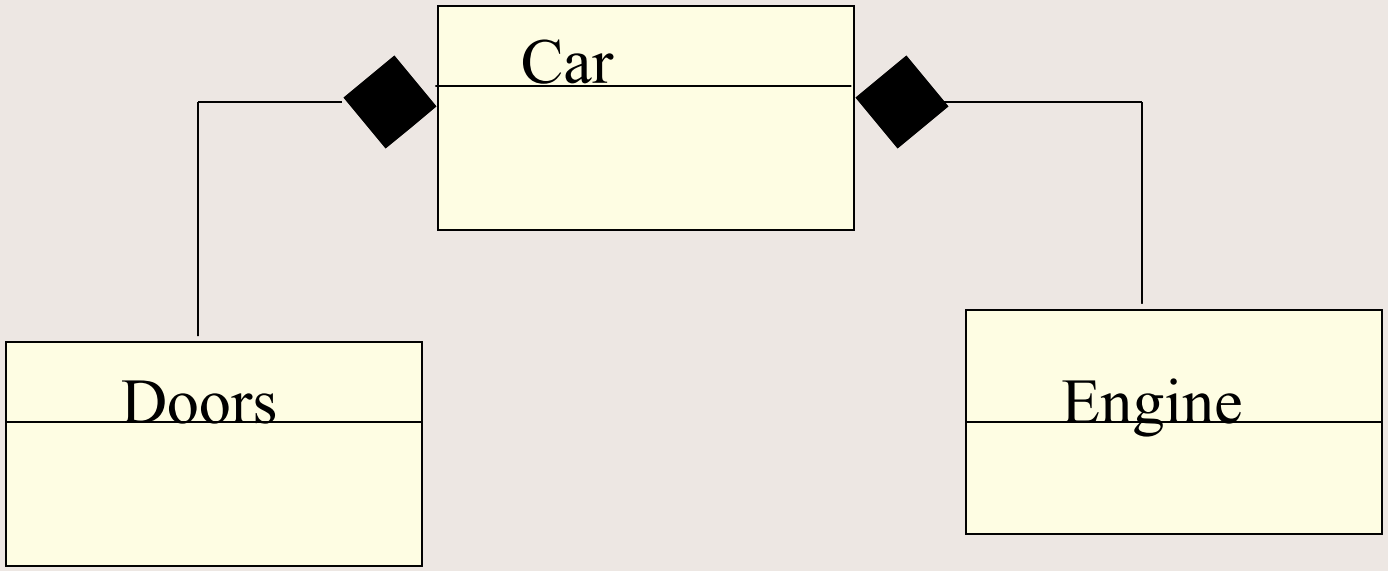
‘Part-of’ релация илюстрирана

- Програма като **Bottom-Up** възходяща йерархия от синтактични обекти /категории/:
  - Program
  - Function(s) // Function(s) are part of a program
  - Statement(s) // Statement(s) are part of a function
  - Expression(s) // Expression(s) are part of a statement
  - Operator(s) // Operator(s) are part of an expression
  - Operand(s) // Operand(s) are part of an expression
  - Data

Aggregation



Composition: по-силна Aggregation



Благодаря
За
Вниманието