New theme

Improving structure with inheritance

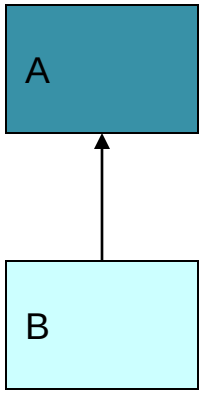
Main concepts to be covered

- Inheritance
- Subtyping
- Substitution
- Polymorphic variables

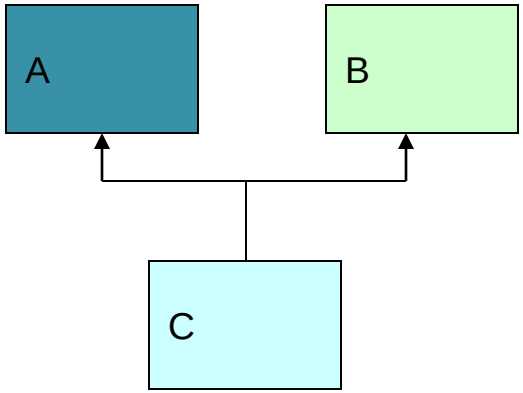
Inheritance: Introduction

- The inheritance allows subclasses to inherit all properties (variables and methods) of their parent classes. The different forms of inheritance are:
  - Single inheritance (only one super class)
  - Multiple inheritance (several super classes)
  - Hierarchical inheritance (one super class, many sub classes)
  - Multi-Level inheritance (derived from a derived class)
  - Hybrid inheritance (more than two types)
  - Multi-path inheritance (inheritance of some properties from two sources).

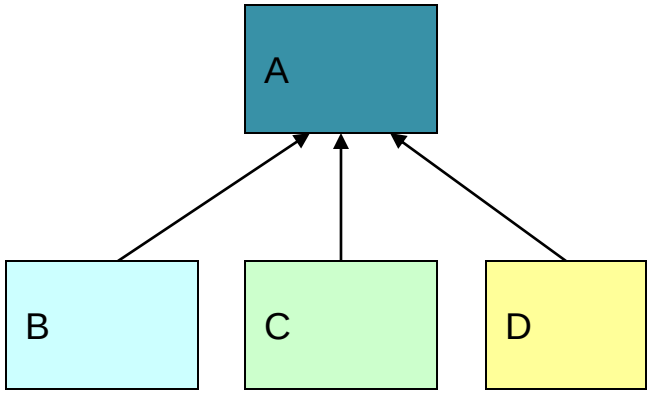
Forms of Inheritance



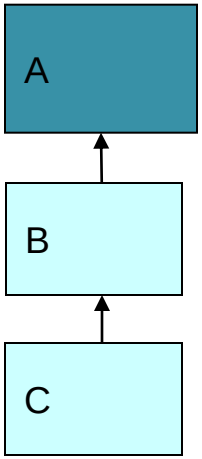
(a) Single Inheritance



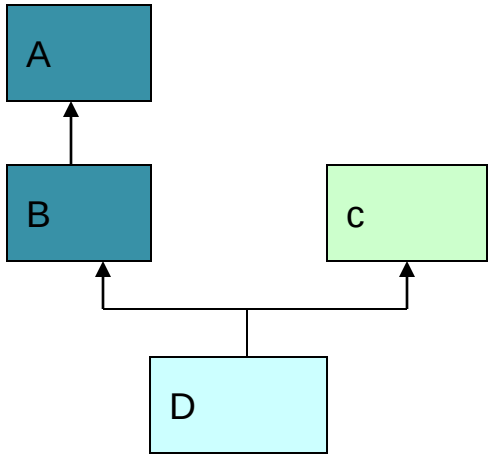
(b) Multiple Inheritance



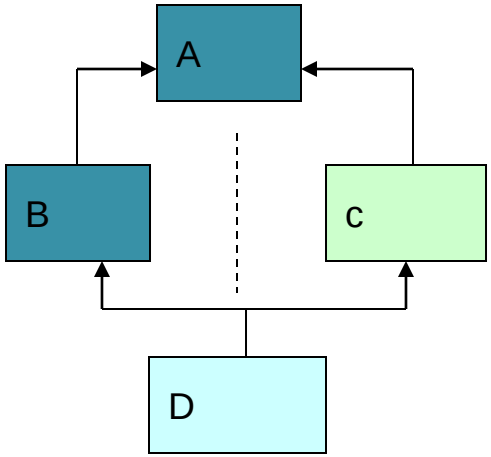
(c) Hierarchical Inheritance



(a) Multi-Level Inheritance



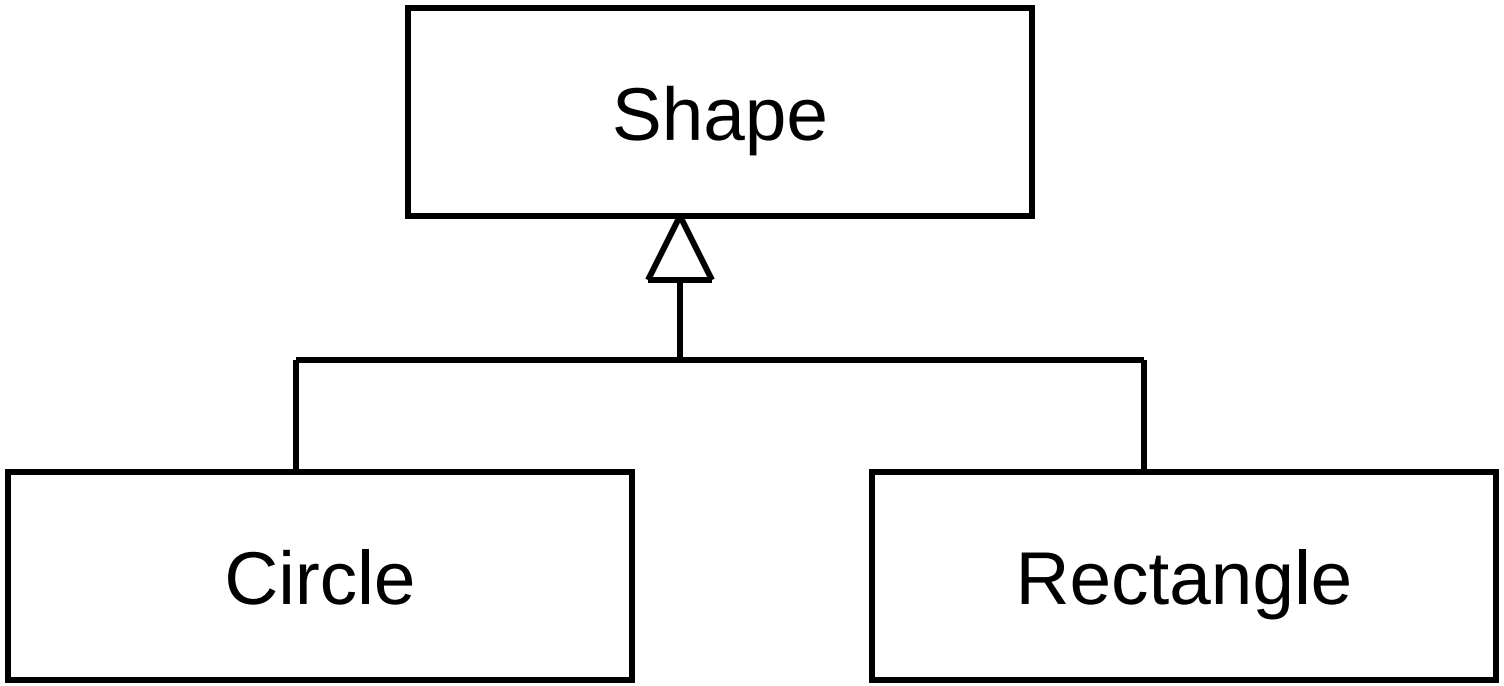
(b) Hybrid Inheritance



(b) Multipath Inheritance

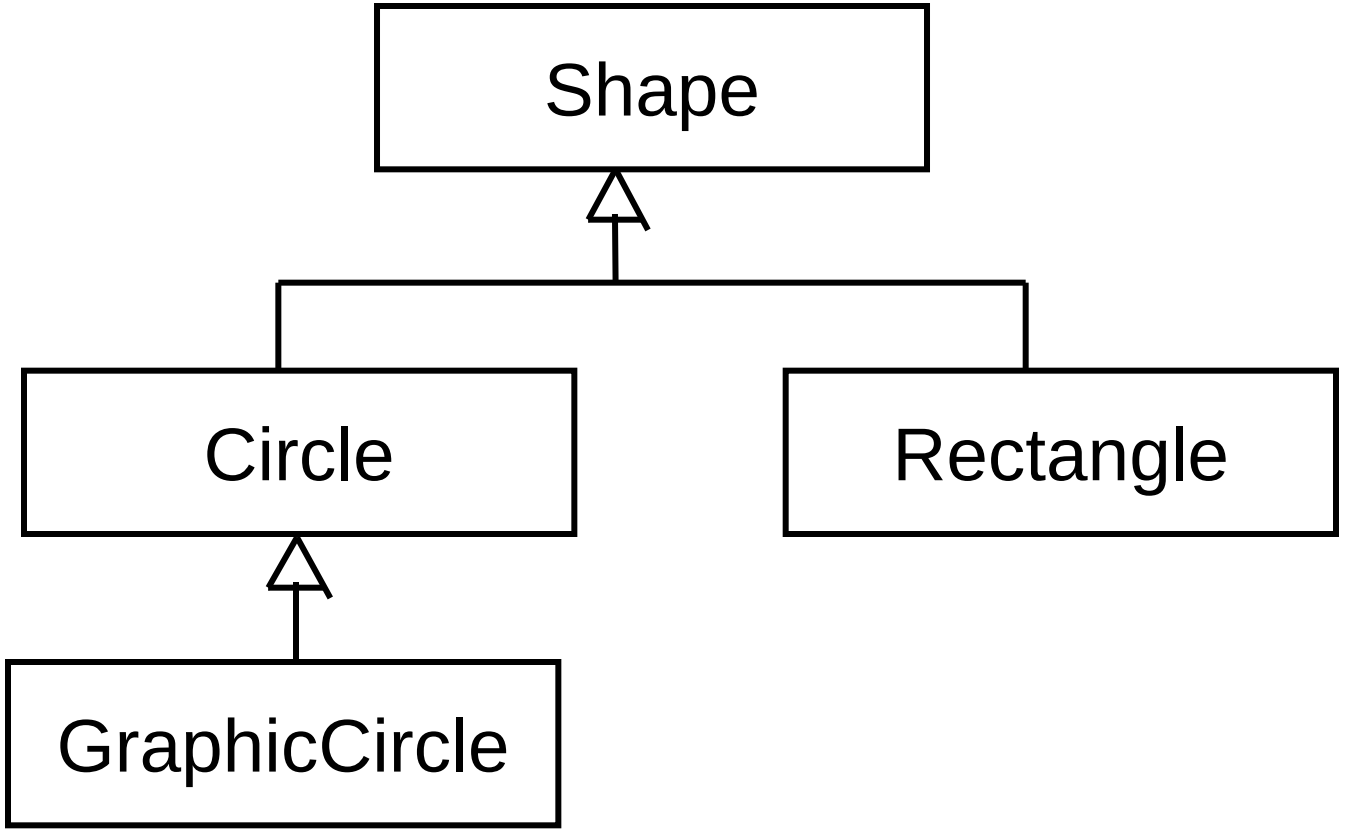
Inheritance - Example

- Circle Class can be a subclass (inherited from) of a parent class - Shape



Inheritance - Example

- Inheritance can also have multiple levels.



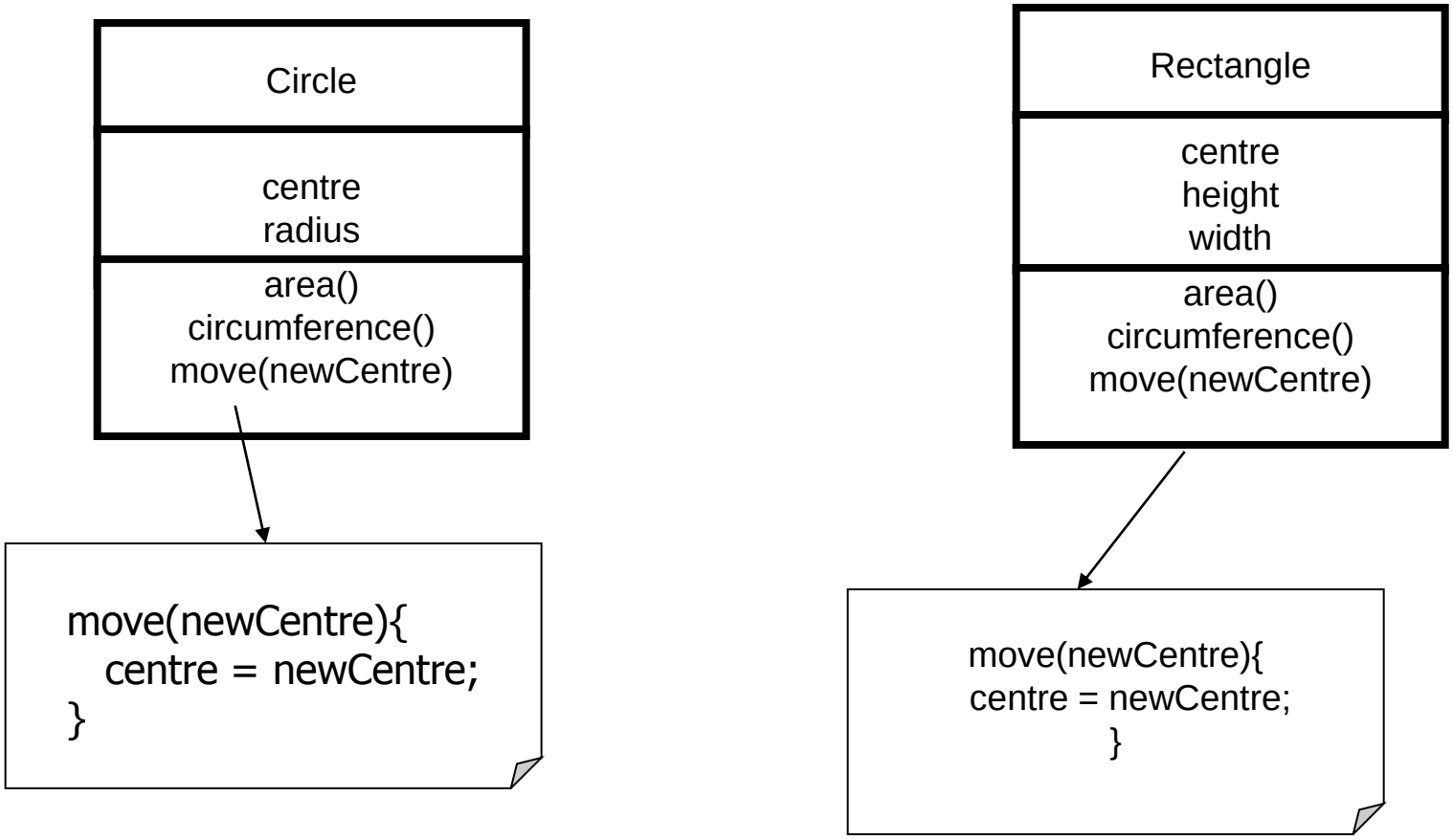
Uses of Inheritance - Reuse

- If multiple classes have common attributes/methods, these methods can be moved to a common class - parent class.

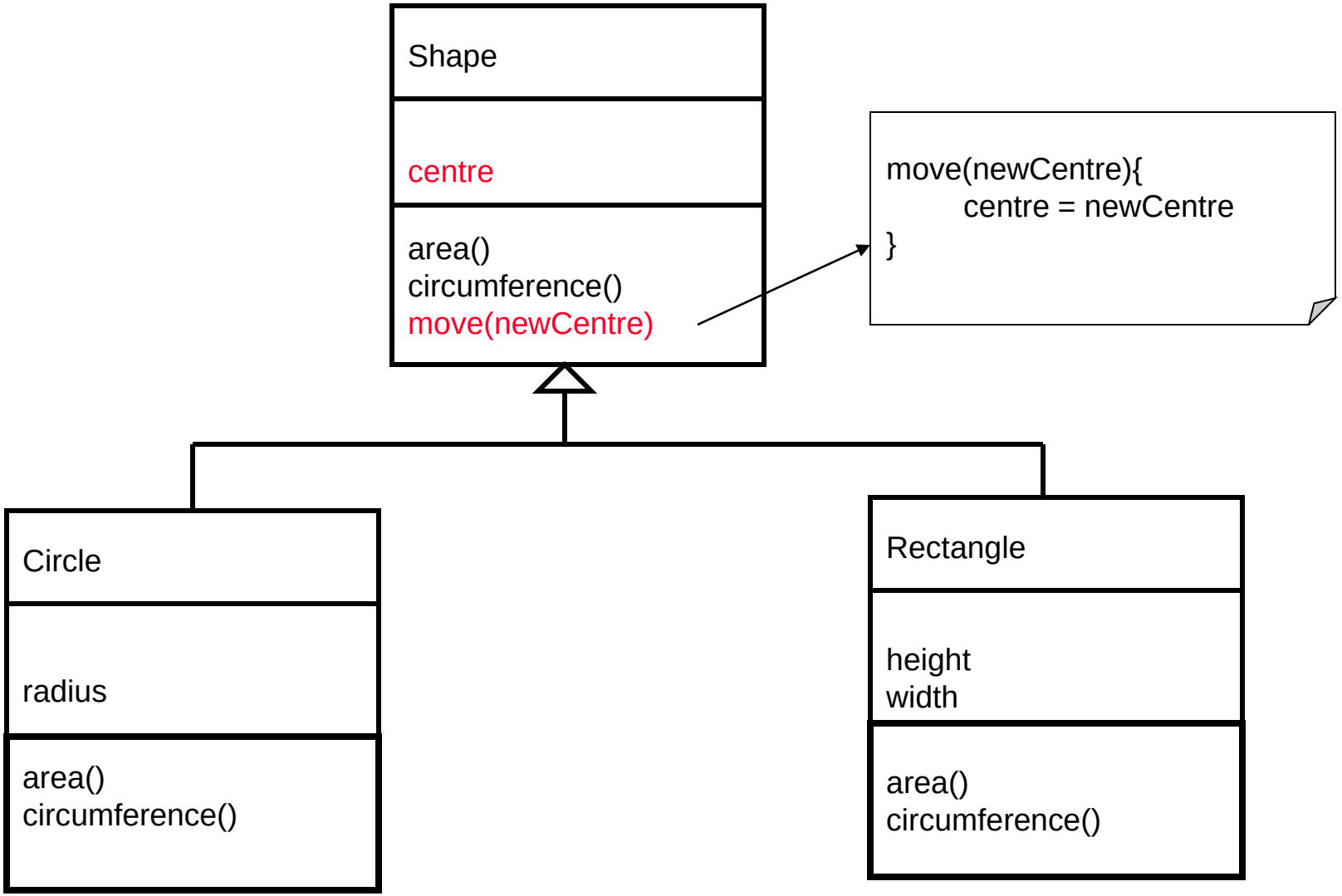
- This allows reuse since the implementation is not repeated.

  Example : Rectangle and Circle method have a common method move(), which requires changing the centre coordinate.

Uses of Inheritance - Reuse



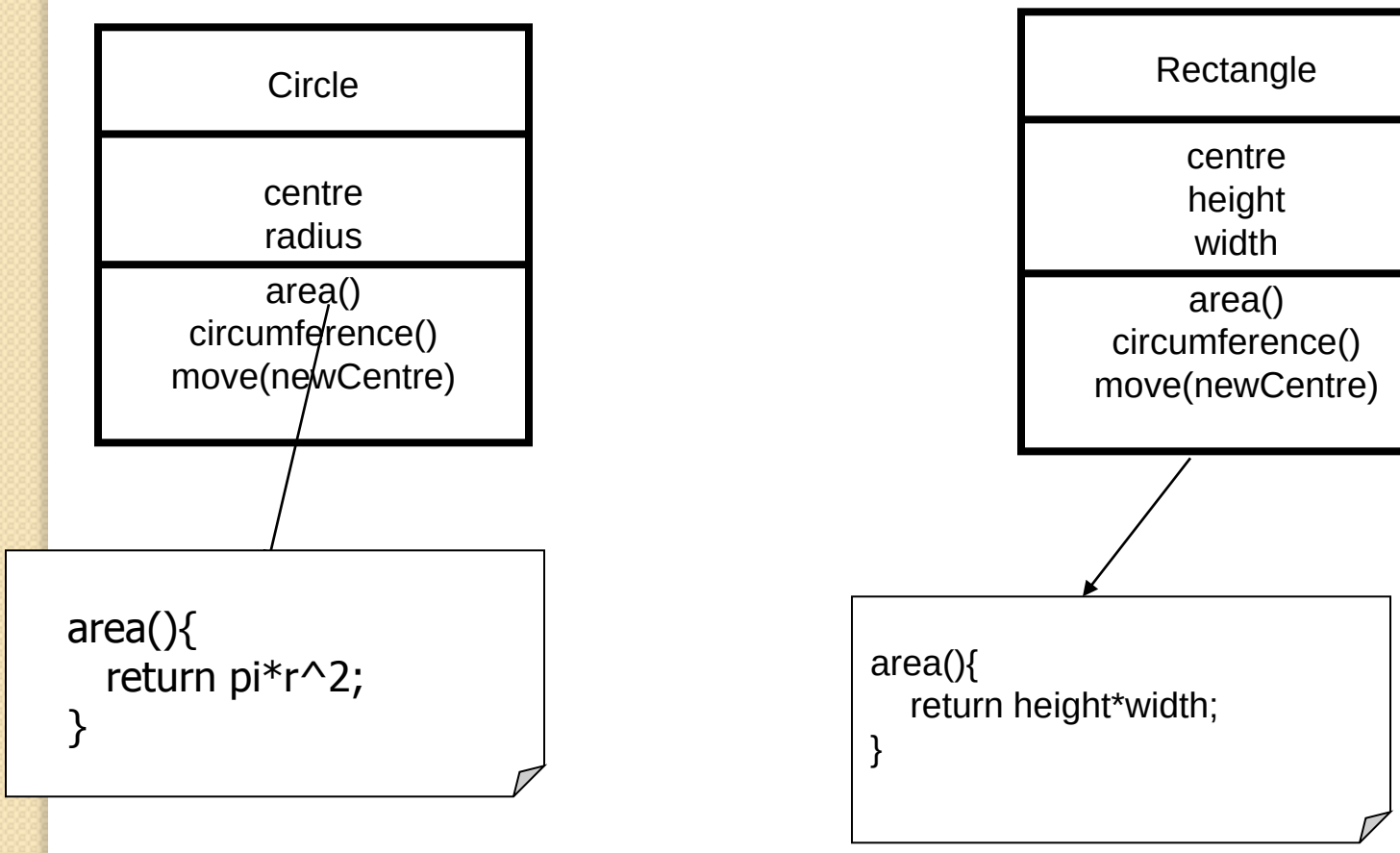
Uses of Inheritance - Reuse



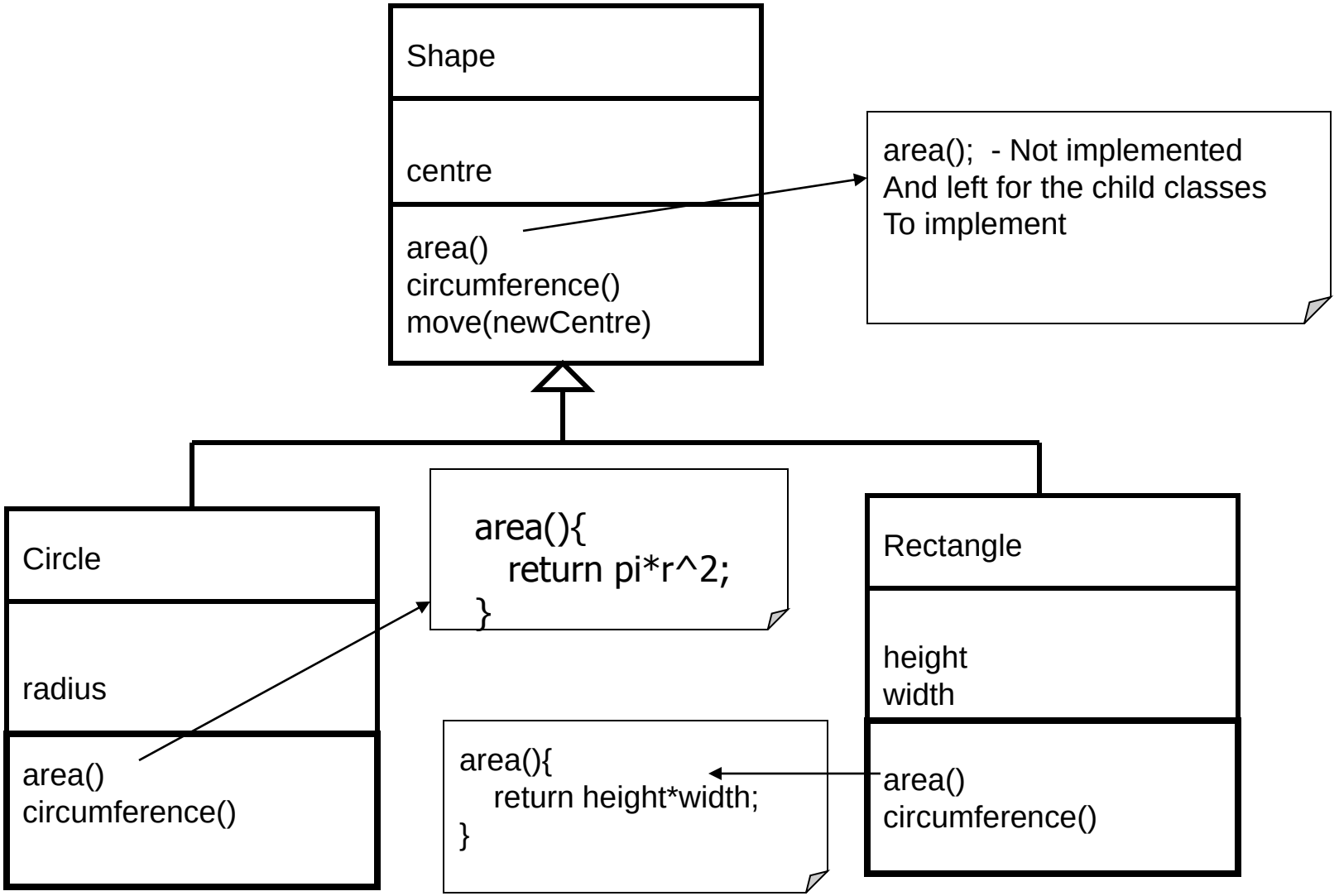
Uses of Inheritance - Specialization

- Specialized behavior can be added to the child class.
- In this case the behaviour will be implemented in the child class.
  - E.g. The implementation of area() method in the Circle class is different from the Rectangle class.
- area() method in the child classes override the method in parent classes().

Uses of Inheritance - Specialization



Uses of Inheritance - Specialization



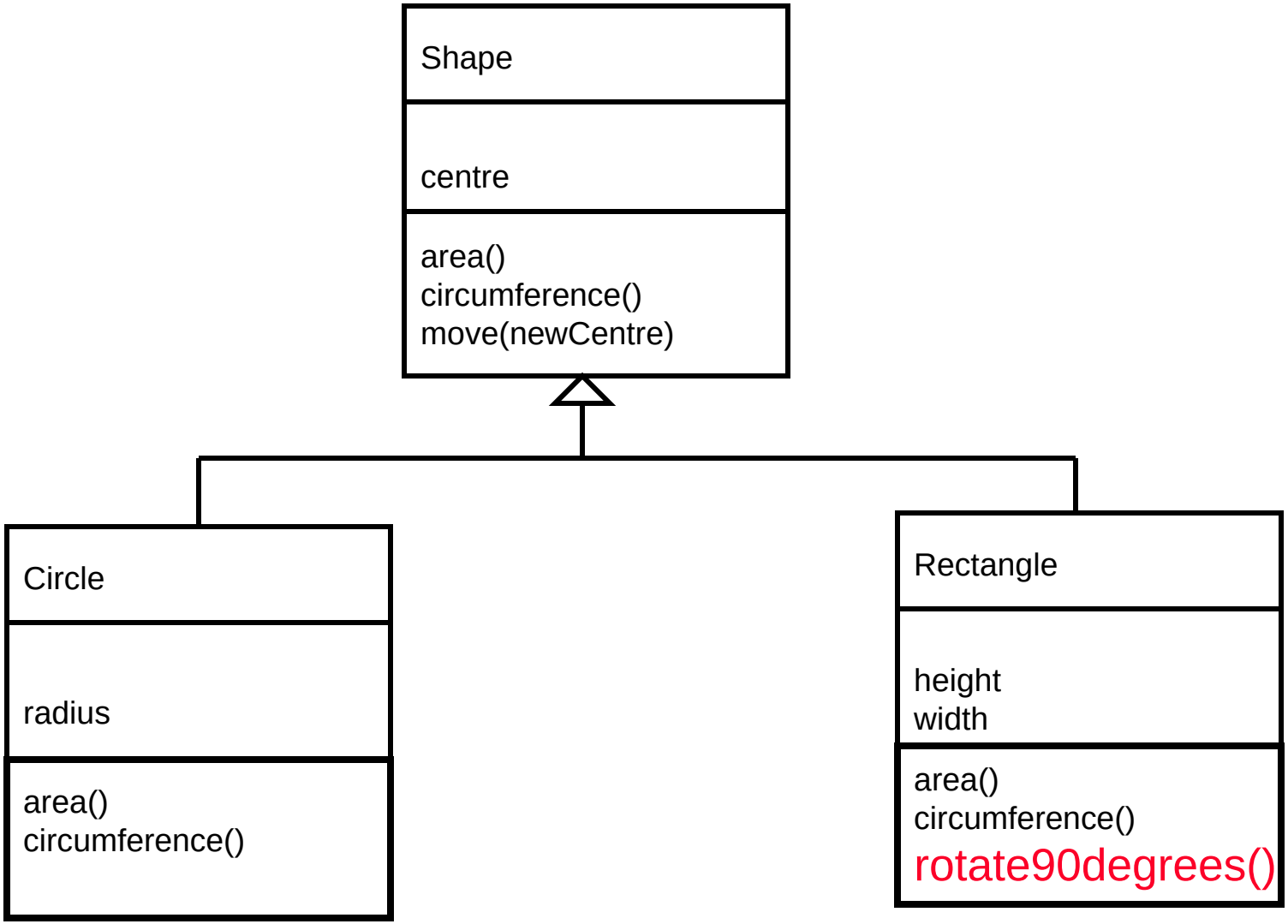
Uses of Inheritance – Common Interface

- All the operations that are supported for Rectangle and Circle are the same.
- Some methods have common implementation and others don't.
  - move() operation is common to classes and can be implemented in parent.
  - circumference(), area() operations are significantly different and have to be implemented in the respective classes.
- The Shape class provides a common interface where all 3 operations *move()*, *circumference()* and *area()*.

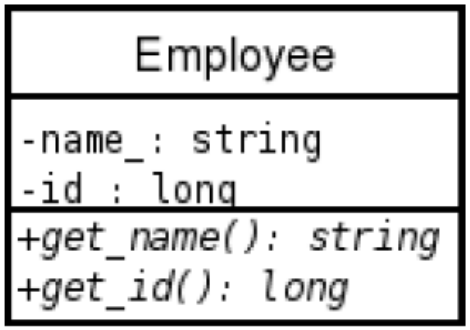
Uses of Inheritance - Extension

- Extend functionality of a class.
- Child class adds new operations to the parent class but does not change the inherited behavior.
  - E.g. Rectangle class might have a special operation that may not be meaningful to the Circle class – rotate 90 degrees()

Uses of Inheritance - Extension

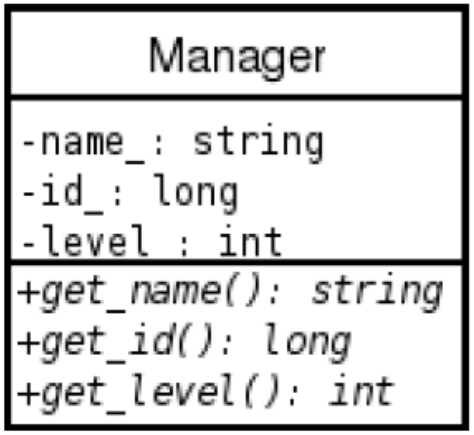


Клас Employee




```
1  class Employee {
2    string name_;
3    long id_;
4  public:
5    Employee(string name,long id)
6    : name_(name),id_(id)
7    {}
8    string get_name() const;
9    long get_id() const;
10 };
```

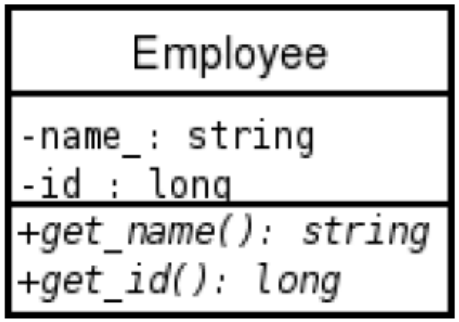
Клас Manager

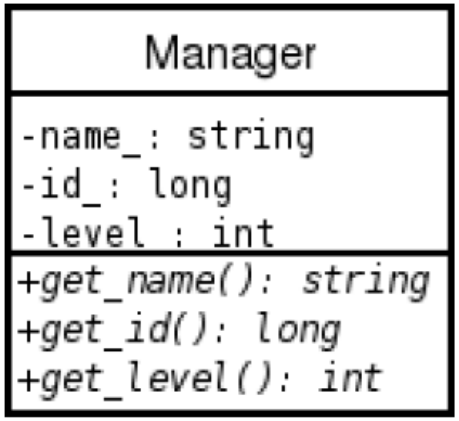



```
1  class Manager {
2    string name_;
3    long id_;
4    int level_;
5  public:
6    Manager(string name,
7         long id,
8         int level)
9    :name_(name), id_(id),
10    level_(level)
11   {}
12   string get_name() const;
13   long get_id() const;
14   int get_level() const;
15 };
```

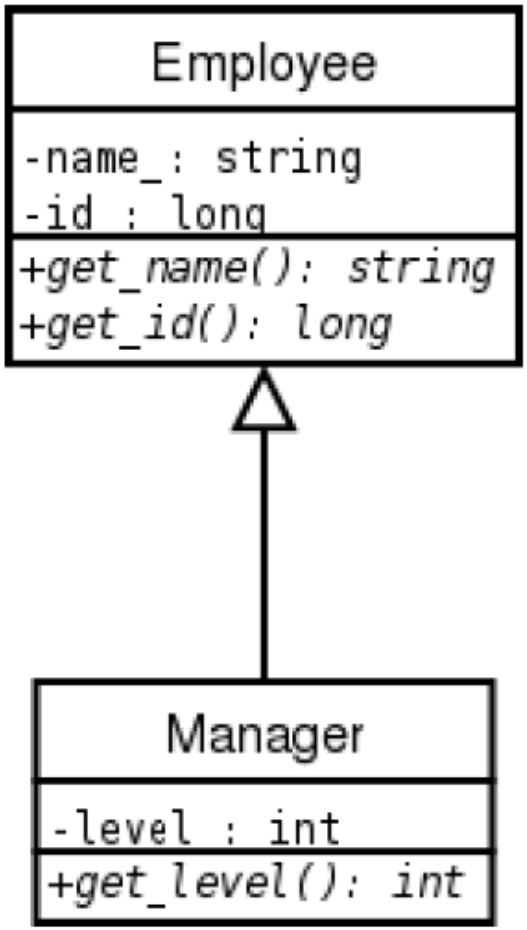
Класовете Employee и Manager

- Мениджърът (Manager) е вид работник (Employee).
- Класът Manager притежава всички атрибути и методи на класа Employee.
- Освен атрибутите и методите на класа Employee, класът Manager притежава някои допълнителни свойства:
  - ниво в йерархията (level_);
  - група от подчинени;





Наследяване

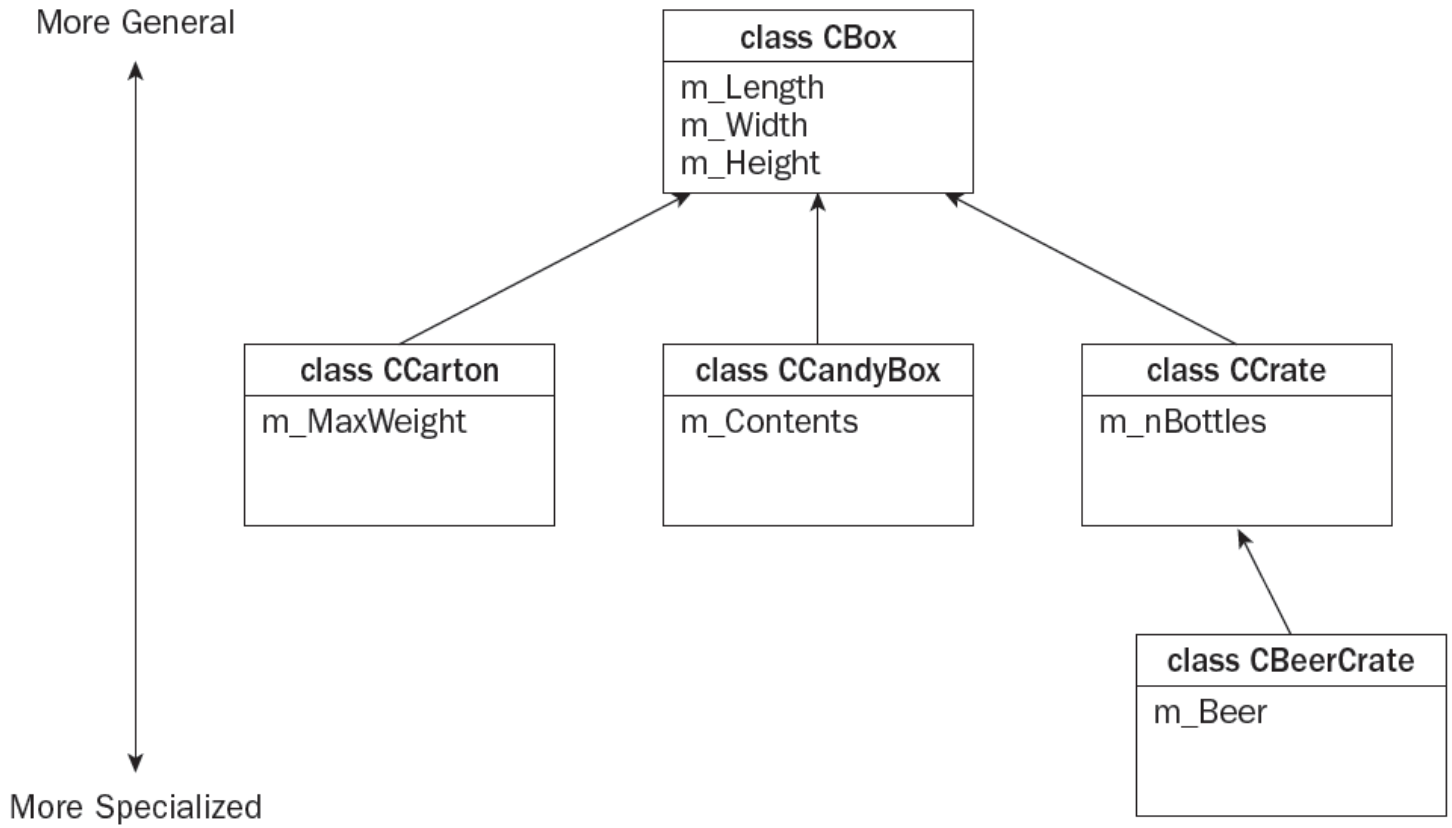


- За да се моделира подобен вид отношение между класовете в обектно-ориентираното програмиране се използва наследяване.
- Класът Employee се нарича базов клас или супер клас.
- Класът Manager се нарича производен клас или наследник.

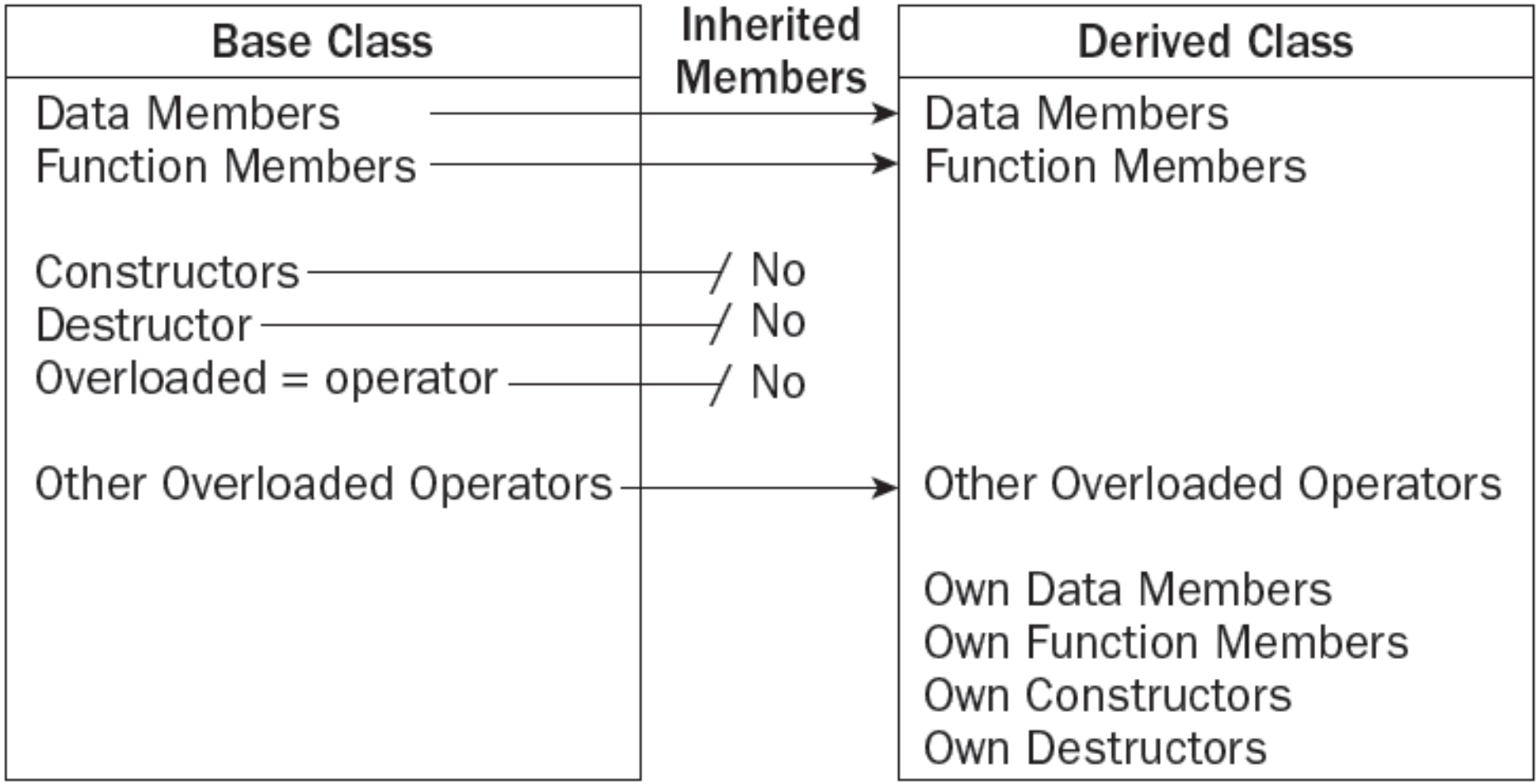
Концепцията Наследяване

- **Наследяване** 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.

Йерархия на класовете



Базов клас и наследник



Изключения

Не се наследяват:

- Конструкторите и деструктора
- Операторът =
- Приятелските функции и класове

Синтаксис (C++)

class derived-class**:access** base-class

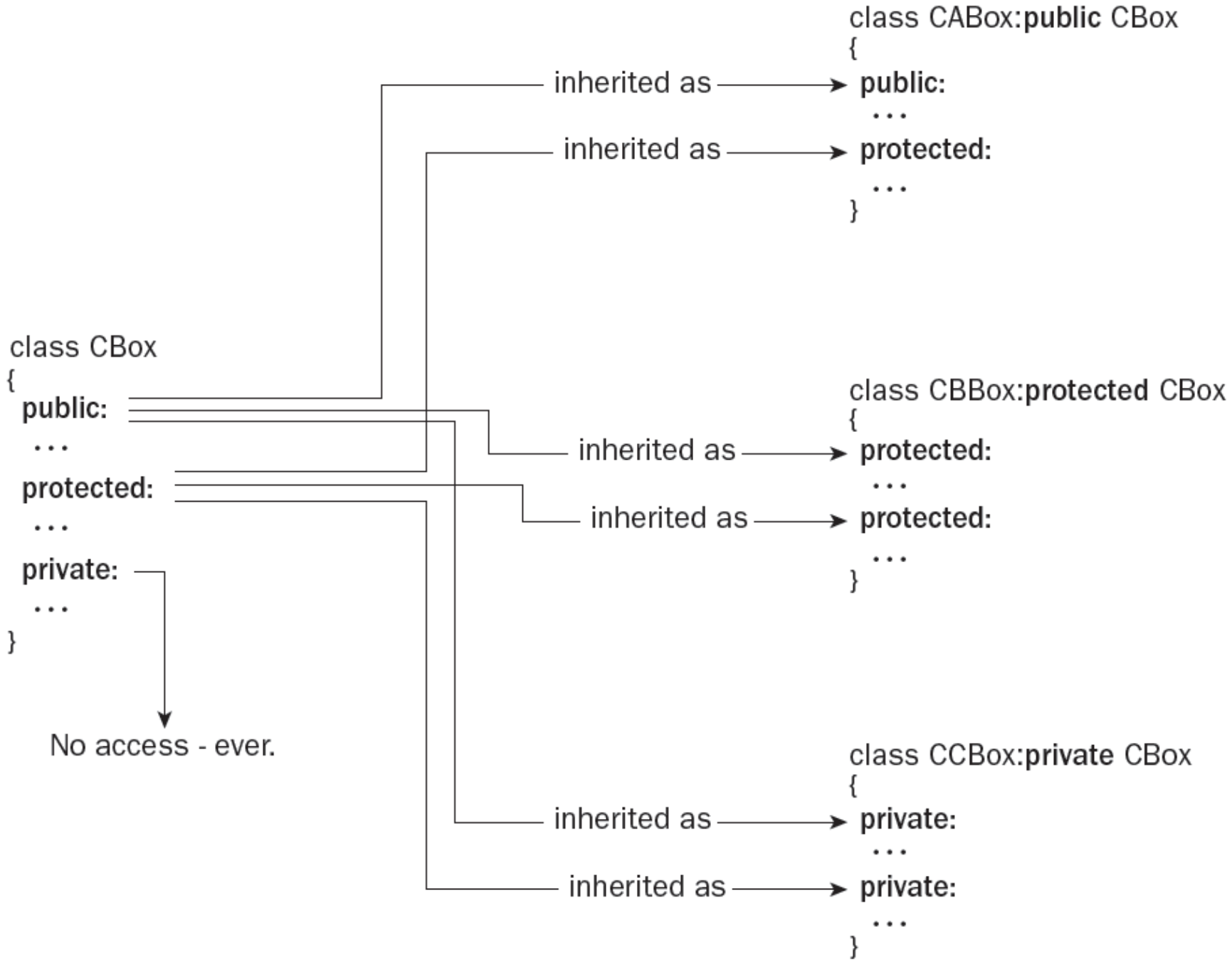
{

body of new class

}

Модификатор за достъп

- private
- public
- protected


class CABox:public CBox
{
inherited as
public:
...
inherited as
protected:
...
}
class CBox
{
public:
...
protected:
...
private:
...
}
class CBBox:protected CBox
{
inherited as
protected:
...
inherited as
protected:
...
}
No access - ever.
class CCBox:private CBox
{
inherited as
private:
...
inherited as
private:
...
}

Пример за базов клас

```
class Employee {
      public:
         Employee(string theName, float
thePayRate);
         string getName() const;
         float getPayRate() const;
         float pay(float hoursWorked) const;

      protected:
            string name;
            float payRate;
};
```

```
Employee::Employee(string theName, float thePayRate)
{
  name = theName;
  payRate = thePayRate;
}

string Employee::getName() const
{
  return name;
}

float Employee::getPayRate() const
{
  return payRate;
}

float Employee::pay(float hoursWorked) const
{
  return hoursWorked * payRate;
}
```

Пример за наследник

```
class Manager : public Employee {
public:
  Manager(string theName,
          float thePayRate,
          bool isSalaried);

  bool getSalaried() const;

  float pay(float hoursWorked) const;

protected:
  bool salaried;
};
```

```
Manager::Manager(string theName,
                 float thePayRate,
                 bool isSalaried)
  : Employee(theName, thePayRate)
{
  salaried = isSalaried;
}

bool Manager::getSalaried() const
{
  return salaried;
}

float Manager::pay(float hoursWorked) const
{
  if (salaried)
    return payRate;
  /* else */
  return Employee::pay(hoursWorked);
}
```

Пример за използване на класовете

```
Employee empl("John Burke", 25.0);
// Print out name and pay (based on 40 hours work).
cout << "Name: " << empl.getName() << endl;
cout << "Pay: " << empl.pay(40.0) << endl;

Manager mgr("Jan Kovacs", 1200.0, true);
// Print out name and pay (based on 40 hours work).
cout << "Name: " << mgr.getName() << endl;
cout << "Pay: " << mgr.pay(40.0) << endl;
```

Методи

- При реализацията на производен клас се използват само публичните методи и член-променливи на базовия клас.
- Като компромис могат да се използват защитени (protected) член-променливи и методи.
- Защитените членове на базовия клас се държат като публични (public) за членовете на производните класове и като скрити (private) за всички останали функции.

- В базовия и производния класове може да се дефинират методи с еднакви имена.
- В класовете Employee и Manager е дефиниран метод с едно и също име — print().
- Когато в класа наследник искаме да използваме метода print(), дефиниран в базовият клас, трябва да използваме пълното му име.

```
void Manager :: print ( void ) const {
        Employee :: print ();
        //...
}
```

- Следната дефиниция е некоректна (безкрайна рекурсия):

```
void Manager :: print ( void ) const {
        print ();
        //...
}
```

Конструктори

- При създаване обект от производен клас: първо се създава базовият клас, след това се създават член-променливите на производния клас и след това самият производен клас.

```
1 class Manager: public Employee {
2   int level_;
3 public:
4   Manager(string name, long id, int level)
5   : Employee(name,id),
6     level_(level)
7   {}
8   ...
9 };
```

Конструктори

- В конструктора на производния клас могат да се инициализират член-променливите, които са декларирани в производния клас.
- Не е възможно да се инициализират директно член-променливите на базовия клас.
- Ако базовият клас няма конструктор по подразбиране, то в производния клас задължително трябва да се извика конкретен конструктор на базовият клас.
- Ако в производния клас не е изрично указано кой конструктор на базовия клас трябва да се извика, то ще се извика конструкторът по подразбиране на базовия клас.

Inheritance and Polymorphism in a program

(second example & point of view)

- must have base class with may be some (virtual) functions with different meaning in future
- let see an idea of base class (will be used for planets and spaceships as well):

```
class Orbiter
{
    private:                                          //data members
            double m_mass;                            //m_ is a prefix convention for data members
            XY m_prior, m_current, m_thrust;          //thrust is needed for fly() – initial force.For
                                                      // planets thrust is always (0,0)

    public:                                           // member functions
            Orbiter( XY current, XY prior, double mass);          // constructor
            XY GetPosition() const;                   // only reads data members
            void Fly();
};
```

Inheritance and polymorphism

- Now -an enriched class version with a display() method. This method differs for planets and for flying objects:

```
class Orbiter
{
    protected:
        double m_mass;
        XY m_current, m_prior, m_thrust;
    public:

        Orbiter( XY current, XY prior, double mass)
        { m_current = current;  m_prior = prior; m_mass = mass;                    }
        XY GetPosition() const;
        void Fly();
        virtual void Display() const;                          // virtual !!
                                          //we must think about object future polymorphism
};
```

Inheritance and polymorphism

- the following example assumes that orbiterArray[ ] contains pointers to a mixture of objects of classes derived from Orbiter:

```
extern Orbiter* orbiterArray[];
for( int i = 0; i < MAX; i++)
{
    orbiterArray[i] ->Fly();
    orbiterArray[i] -> Display();
}
```

- idea of pure virtual functions

(if one exist in a class declaration, it's forbidden to construct an object of that class)

Example:

```
virtual void Display() const =0;
```

Derived classes

```
class Planet : public Orbiter
{     public:
            Planet( XY current, XY prior, double mass) :Orbiter( current, prior, mass) {}
            void Display() const;
};

class SpaceShip : public Orbiter
{     private: double m_fuel; XY m_orientation;
      public:
            SpaceShip( XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
                                                                  : Orbiter(current, prior, mass)
            {m_orientation = orientation; m_fuel = fuel;
             m_thrust = thrust;                                   // m_thrust was  Orbiter's data member
            }
      void Display() const;
      …                                                          // maybe another members of the class follows
};
```

If omitted – mistake!. The compiler would have tried to use default, but this is missed in Orbiter.

Полиморфизъм

- **Полиморфизмът** е свойство на член-функциите и се реализира с **виртуални функции**.
  Той се състои в осигуряване възможността генерирания код да се държи по различен начин в зависимост от условия, определяни по време на изпълнението на програмата.
  Идеята на тези функции се основава на вида на свързването им (**статично** или **ранно** и съответно **динамично** или **късно**)

Статичното свързване

- **Статичното свързване** на типа на обекта с извикваната член-функция става по време на компилацията и не се променя при изпълнение на програмата. Свързващият редактор замества имената на извикваните функции с конкретните им адреси.

Динамичното свързване

- **Динамичното** свързване на типа на обекта с извикваната член-функция е по време на изпълнение на програмата. При извикването на функция се задава само името, но не и конкретния адрес на прехода. Или по време на изпълнение се дава възможността за промяна на връзките между имена и адреси. Късното свързване се определя чрез думата **virtual**.

Характеристики на виртуалните функции

- Членове на класа;
- Декларират се с ключовата дума **virtual;**
- В породения клас има различна функционалност.

Пример

```
class Animal {
    public:
        virtual void eat()
        {
            cout << "I eat
like a      generic Animal."
<< endl;
        }
};
```

virtual functions

- virtual functions can be called in a base class as well as from outside the class;

- if Orbiter class contains

  Orbiter::Fly()

  and that function uses angular momentum for his computations, which is specific to every derived class. So it's convenient to declare it in Orbiter like this:

protected:

virtual XY GetSpecificCalculation() const = 0;

and every derived class is obliged to provide his overriding function implementation.

Неопределен виртуален метод

- Т.нар. чисти виртуални функции
- Метод, който има декларация, но няма дефиниция.
- Използва се присвояване на стойност 0.

Абстрактен клас

- Клас, в който е деклариран поне един неопределен виртуален метод.

```
class Shape {
        public:
        virtual void draw ( void ) const =0;
};
```

Пример

```
class Abstract
{
  public:
     virtual void pure_virtual() =0;
  };
```

Особености 1/2

- Обекти от абстрактни класове не могат да бъдат създавани:

```
Abstract x;  //Error
```

- Абстрактните класове могат да се използват като базови за други класове.
- Абстрактните класове не могат да се използват, като тип на параметри и тип на върнатата стойност на функциите:

```
Abstract func(); //Error
Float f(Abstract); //Error
```

Особености 2/2

- Методите на абстрактните класове могат да се извикват от техните конструктори, но извикване на чиста виртуална функция не е възможно и води до грешка по време на изпълнение на програмите;
- Допустимо е дефиниране и използване на указатели към абстрактни класове и псевдоними на абстрактни класове:

```
Abstract *ptr;
Abstract& f(Abstract &);
```

Пример 1/3

```
#include <iostream>
using namespace std;
class CPolygon
{
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; }
        virtual int area (void) =0;
};
```

Пример 2/3

```
class CRectangle: public CPolygon
{
    public:
        int area (void)
        {return (width * height); }
};
class CTriangle: public CPolygon
{
    public:
        int area (void)
        { return (width * height / 2);
  }
};
```

Пример 3/3

```
int main ()
{
      CRectangle rect;
      CTriangle trgl;
      CPolygon * ppoly1 = &rect;
      CPolygon * ppoly2 = &trgl;
      ppoly1->set_values (4,5);
      ppoly2->set_values (4,5);
      cout << ppoly1->area() << endl;
      cout << ppoly2->area() << endl;
      return 0;
}
```

Abstract base class and pure virtual functions. Example

```
// Specification file for the Student class (Student.h)
#include <Cstring>   // For strcpy

const int NAME_SIZE = 51;
const int ID_SIZE = 21;

class Student
{
protected:
   char name[NAME_SIZE];                    // Student name
   char idNumber[ID_SIZE];                  // Student ID
   int yearAdmitted;                        // Year student was admitted
public:
   // Default constructor
   Student()                  { name[0] = '\0';       idNumber[0] = '\0';       yearAdmitted = 0; }

   // Constructor
   Student(const char *n, const char *id, int year)                  { set(n, id, year); }

   // The set function sets the attribute data.
   void set(const char *n, const char *id, int year)
     { strncpy(name, n, NAME_SIZE);    // Copy the name
       name[NAME_SIZE - 1] = '\0';       // Place a null character
       ….}
   // Accessor functions
   const char *getName() const                        { return name; }

   const char *getIdNum() const                       { return idNumber; }

   int getYearAdmitted() const                        { return yearAdmitted; }

   // Pure virtual function
   virtual int getRemainingHours() const = 0;
};
```

Abstract base class and pure virtual functions. Example

```
// Specification file for the CsStudent class (Cstudent.h)

#include "Student.h"

// Constants for required hours
const int MATH_HOURS = 20;   // Math hours
const int CS_HOURS = 40;     // Computer science hours
const int GEN_ED_HOURS = 60; // General Ed hours

class CsStudent : public Student
{ private:
    int mathHours;    // Hours of math taken
  int csHours;      // Hours of Computer Science taken
  int genEdHours;   // Hours of general education taken
public:
  // Default Constructor
  CsStudent() : Student()      { mathHours = 0;              csHours = 0;                genEdHours = 0; }

  // Constructor
  CsStudent(const char *n, const char *id, int year) :      Student(n, id, year)
    { mathHours = 0;       csHours = 0;       genEdHours = 0; }

  // Mutator functions
  void setMathHours(int mh)                 { mathHours = mh; }

  void setCsHours(int csh)                  { csHours = csh; }

  void setGenEdHours(int geh)               { genEdHours = geh; }

  // Overridden getRemainingHours function, defined in CsStudent.cpp
  virtual int getRemainingHours() const;
};
```

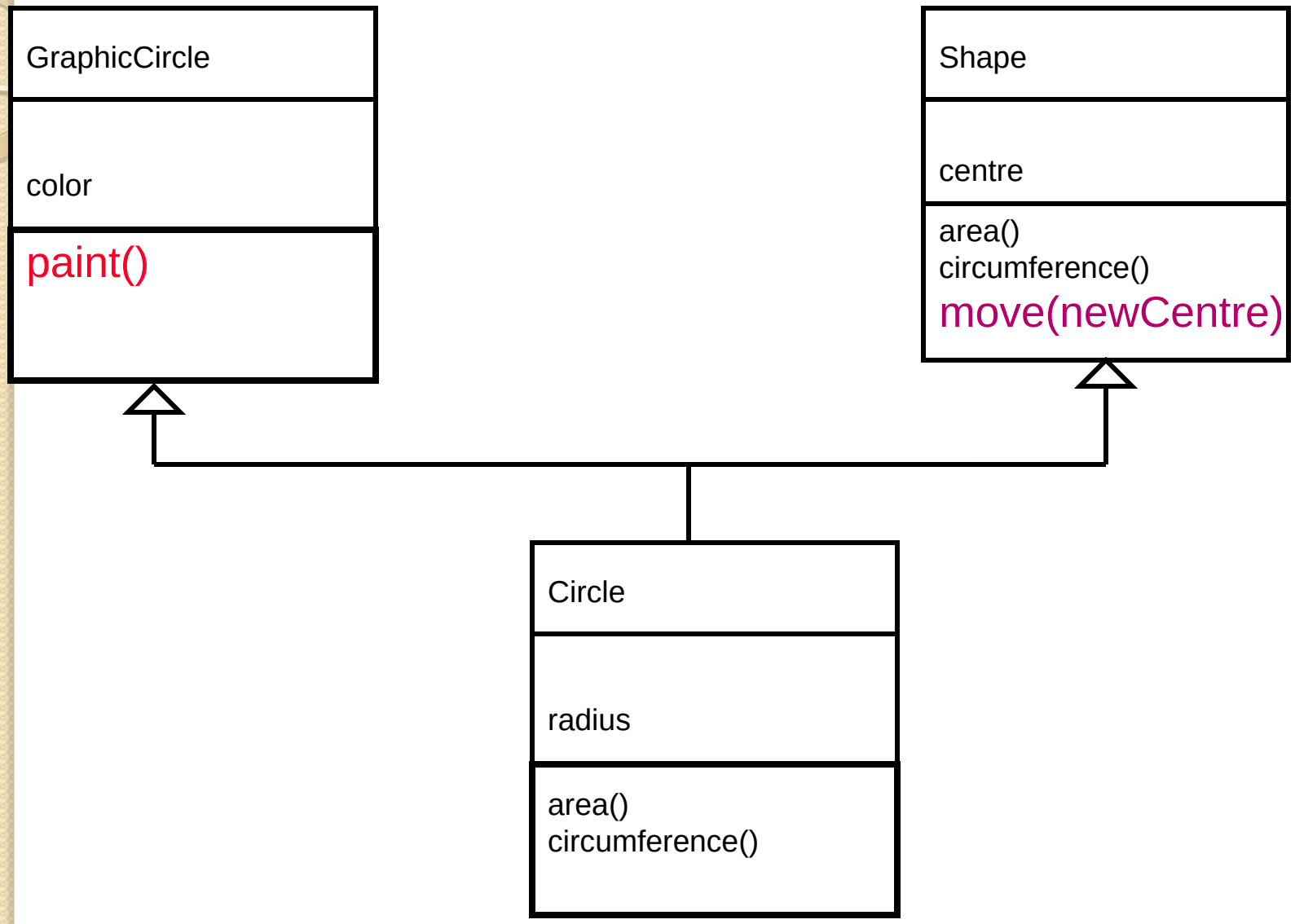
Abstract base class and pure virtual functions. Example

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

//*************************************************
// The CsStudent::getRemainingHours function returns *
// the number of hours remaining to be taken.       *
//*************************************************

int CsStudent::getRemainingHours() const
{
   int reqHours,    // Total required hours
   remainingHours;  // Remaining hours

   // Calculate the required hours.
   reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;

   // Calculate the remaining hours.
   remainingHours = reqHours - (mathHours + csHours +
                    genEdHours);

   // Return the remaining hours.
   return remainingHours;
}
```

Abstract base class and pure virtual functions. Example

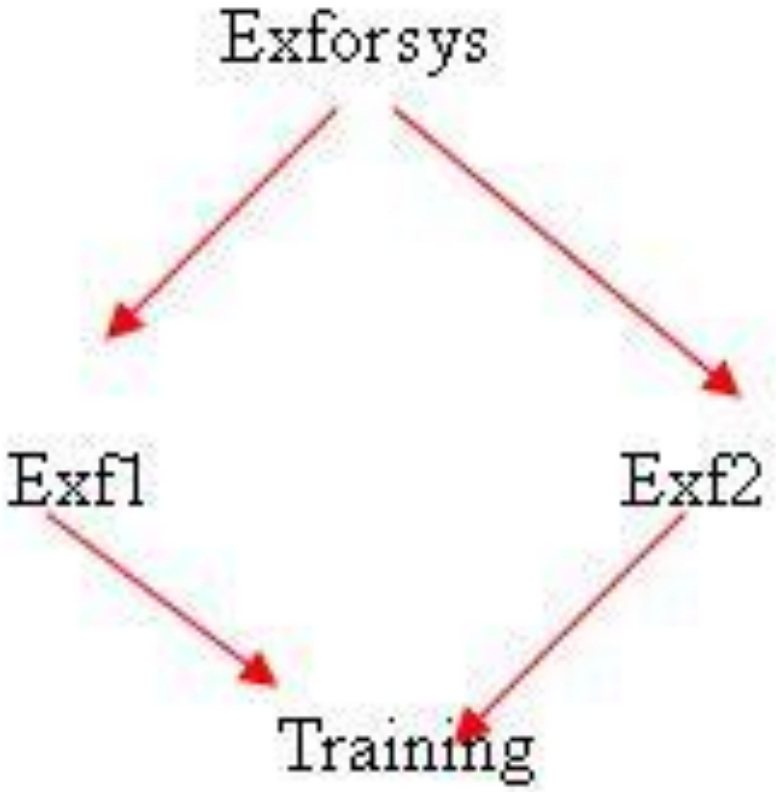
```
// This program demonstrates the CsStudent class, which is
// derived from the abstract base class, Student.
#include <iostream>
#include "CsStudent.h"
using namespace std;

int main()
{
   // Create a CsStudent object for a student.
   CsStudent student("Jennifer Haynes", "167W98337", 2006);

   // Store values for Math, Computer Science, and General  Ed hours.
   student.setMathHours(12);  // Student has taken 12 Math hours
   student.setCsHours(20);    // Studeht has taken 20 CS hours
   student.setGenEdHours(40); // Student has taken 40 Gen Ed hours

   // Display the number of remaining hours.
   cout << "The student " << student.getName()
        << " needs to take " << student.getRemainingHours()
        << " more hours to graduate.\n";

   return 0;
}
```

Uses of Inheritance – Multiple Inheritance



Множество наследяване

- Множествено наследяване се нарича наследяване-то, при което един клас е пряк наследник на повече от един предшестве-ници.



```
class Exforsys
{
protected:
int x;
};

class Exf1:public Exforsys
{ };

class Exf2:public Exforsys
{ };

class Training:public Exf1,public Exf2
{
public:
int example()
{
return x;
}
};
```

```
class Exforsys
{
protected:
int x;
;


class Exf1:virtual public Exforsys
{ };

class Exf2:virtual public Exforsys
{ };

class Training:public Exf1,public Exf2
{
public:
int example()
{
return x;
}
};
```

- Променя се **типът на наследяване** на базовия клас от преките му наследници, като се описва, че той се наследява **виртуално**. В този случай в класа ще се наследят само по едно копие на член-данните на базовия клас.

Multiple inheritance. Example:

```
// Specification file for the Date.h class
class Date
{
protected:
   int day;
   int month;
   int year;
public:
   // Default constructor
   Date()
      { day = 1; month = 1; year = 1900; }

   // Constructor
   Date(int d, int m, int y)
      { day = d; month = m; year = y; }

   // Accessors
   int getDay() const
      { return day; }

   int getMonth() const
      { return month; }

   int getYear() const
      { return year; }
};
```

Multiple inheritance. Example:

```
// Specification file for the Time.h class
class Time
{
protected:
   int hour;
   int min;
   int sec;
public:
   // Default constructor
   Time()
      { hour = 0; min = 0; sec = 0; }

   // Constructor
   Time(int h, int m, int s)
      { hour = h; min = m; sec = s; }

   // Accessor functions
   int getHour() const
      { return hour; }

   int getMin() const
      { return min; }

   int getSec() const
      { return sec; }
};
```

Multiple inheritance. Example:

```
// Specification file for the DateTime.h class

#include "Date.h"
#include "Time.h"

// Constant for string size
const int DT_SIZE = 20;

class DateTime : public Date, public Time
{
protected:
   char dateTimeString[DT_SIZE];
public:
   // Default constructor
   DateTime();

   // Constructor
   DateTime(int, int, int, int, int, int);

   // Accessor function
   const char *getDateTime() const
      { return dateTimeString; }
};
```

Multiple inheritance. Example:

```
// Implementation file for the DateTime.cpp class
#include <cstring>      // For strcpy and strcat
#include <cstdlib>      // For itoa
#include "DateTime.h"

// Constant for temp array size
const int TEMP_SIZE = 10;

// Default constructor                          *
DateTime::DateTime() : Date(), Time()
{               strcpy(dateTimeString, "1/1/1900 0:0:0");  }

// Constructor                                  *
DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :   Date(dy, mon, yr), Time(hr, mt, sc)
{   char temp[TEMP_SIZE]; // Temporary work area for itoa()

   // Store the date in dateTimeString, in the form MM/DD/YY
   strcpy(dateTimeString, itoa(getMonth(), temp, TEMP_SIZE));
   strcat(dateTimeString, "/");
   strcat(dateTimeString, itoa(getDay(), temp, TEMP_SIZE));
   strcat(dateTimeString, "/");
   strcat(dateTimeString, itoa(getYear(), temp, TEMP_SIZE));
   strcat(dateTimeString, " ");

   // Store the time in dateTimeString, in the form HH:MM:SS
   strcat(dateTimeString, itoa(getHour(), temp, TEMP_SIZE));
   strcat(dateTimeString, ":");
   strcat(dateTimeString, itoa(getMin(), temp, TEMP_SIZE));
   strcat(dateTimeString, ":");
   strcat(dateTimeString, itoa(getSec(), temp, TEMP_SIZE));       }
```

Multiple inheritance. Example:

```
// This program demonstrates a class with multiple inheritance.
#include <iostream>
#include "DateTime.h"
using namespace std;

int main()
{
   // Define a DateTime object and use the default  constructor to initialize it.
   DateTime emptyDay;

   // Display the object's date and time.
   cout << emptyDay.getDateTime() << endl;

   // Define a DateTime object and initialize it  with the date 2/4/60 and the time 5:32:27.
   DateTime pastDay(2, 4, 60, 5, 32, 27);

   // Display the object's date and time.
   cout << pastDay.getDateTime() << endl;
   return 0;
}
```

Program output:

1/1/1900 0:0:0
4/2/60 5:32:27

•To call function or data from the super class (as BankAcount), we can write:

__super::function(..);

•In managed code you can define a class as 'sealed'. This means the class cannot be inherited from. It's useful security measure

•Inheritance can be made through interfaces as well through parent classes.
An interface is similar to a class, but all the member functions in it are pure virtual.
So, an istantiation of interface cannot be made!!
Interfaces are useful for defining common capabilities for diverse classes with Different realizations.
•Interface example in managed C++ :

```
__gc __interface IStorableAsXml
{
        void ReadFromXmlFile(String *XmlFileName);
        void WriteToXmlFile(String *XmlFileName);
};

…
__gc __sealed class SavingAccount : public BankAccount, public IStorableAsXml
{
public:
// override functions from the interface
…
// other members, as before
…
}
```

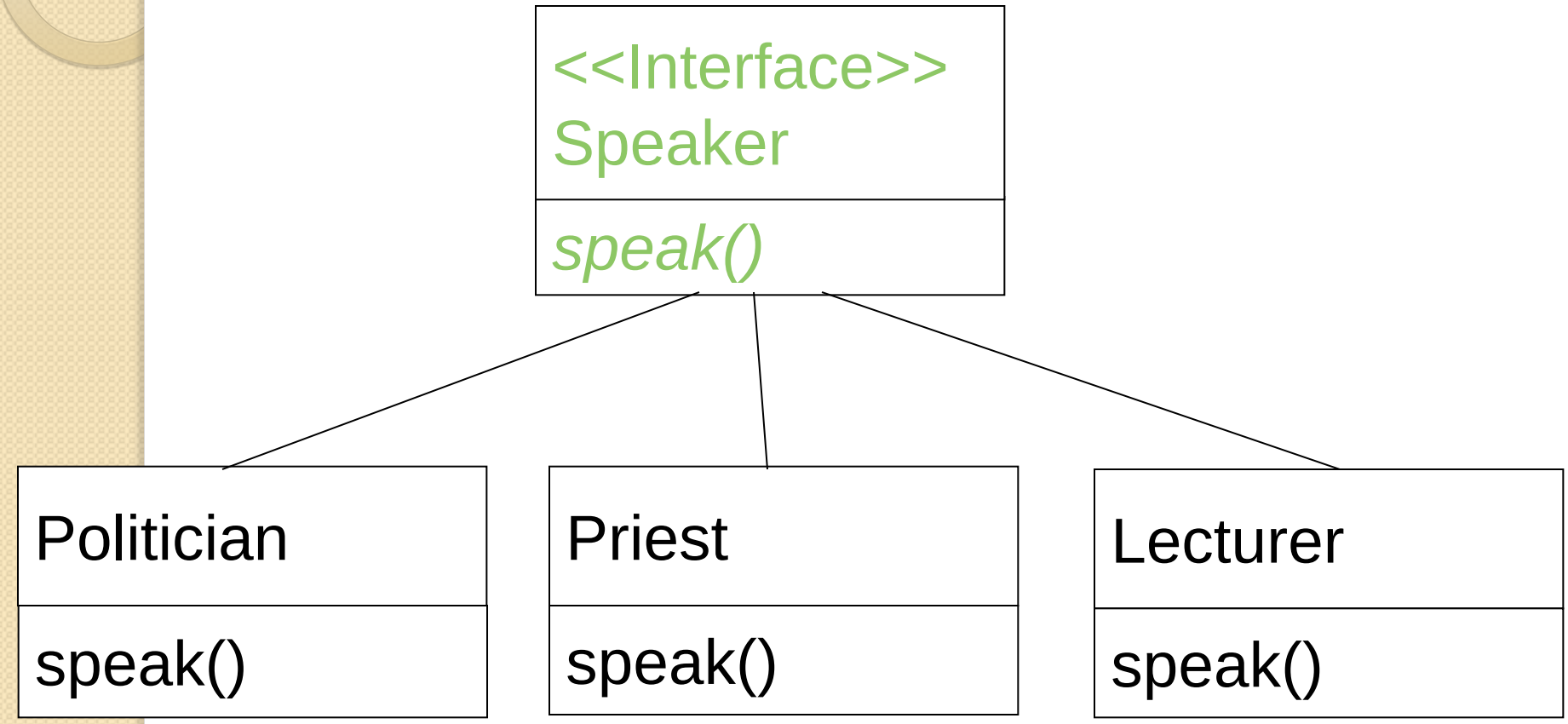
Interfaces

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

Interfaces: An informal way of realising multiple inheritance

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realising multiple inheritance in Java.

Interface - Example



Review

- Inheritance allows the definition of classes as extensions of other classes.
- Inheritance
  - avoids code duplication
  - allows code reuse
  - simplifies the code
  - simplifies maintenance and extending
- Variables can hold subtype objects.
- Subtypes can be used wherever supertype objects are expected (substitution).