CASE Study
– програмна система за плащане
(реализация с обекти)

The *case study* can be defined as a *research strategy*, an empirical inquiry that investigates a phenomenon within its real-life context

Wikipedia

Задачата:

- **Компания заплаща седмично**
- **Работещите са 4типа: salaried – с фиксирана седмична заплата;**
  - **Почасово заплатени с повишена 50% ставка над 40 часа;**
  - **Комисионери**
  - **Комисионери + седмична ставка.**

- **Компанията решава за текущия платежен период да увеличи 10% заплатите на тези, работещи на заплата +комисионна (четвъртия тип)**

- Проект:

създаваме abstract class Employee. Ще го разширяваме към Salariedemployee, CommisisonEmployee, , HourlyEmployee.
Имаме и клас BasePlusCommisionEmployee, наследник на Commissionemployee

Следва UML диаграма:

CASE – програмна система за плащане (реализация с обекти)

Задачата:

- Компания заплаща седмично
- Работещите са 4типа: salaried – с фиксирана седмична заплата;
  - Почасово заплатени с повишена 50% ставка над 40 часа;
  - Комисионери
  - Комисионери + седмична ставка.

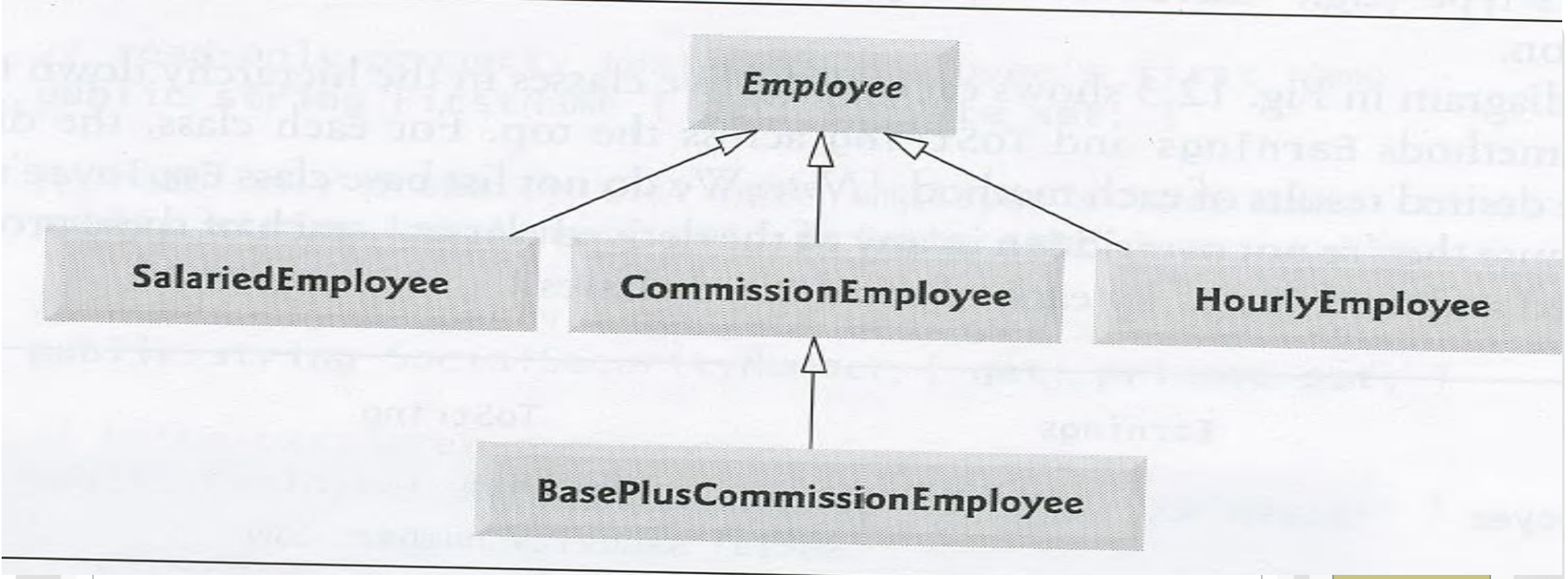
- Компанията решава за текущия платежен период да увеличи 10% заплатите на тези, работещи на заплата +комисионна (четвъртия тип)

- **Проект:**

създаваме abstract class Employee. Ще го разширяваме към Salariedemployee, CommisisonEmployee, , HourlyEmployee.

Имаме и клас BasePlusCommisionEmployee, наследник на Commissionemployee

Следва UML диаграма:



- Employee декларира общ интерфейс (набор методи) за всеки „служител“
- Класът е абстрактен;
- Всеки наследил го следва да даде имплементация;
- Всеки наследил, може да се разглежда и като „Employee”
- Методите му се имплементират чрез полиморфизъм, т.е. повиквания се решават по време на изпълнение и се ползват имплементации за конкретния наследник.

Диаграмата от следващия слайд показва за всички класове от йерархията - имплементацията, която следва да имат методите Earning() и ToString():

	Earnings	ToString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage* hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate* base salary: *baseSalary*

Class Employee

Има

- **конструктор;**

- **Пропъртита с модификатор get;**

- **Метод ToString(), извеждащ низ с описание на служителя;**

- **Абстрактен метод Earnings(), който следва да се имплементира задължително във всички наследници.**

  Тогава приложението ще вика чрез полиморфизъм реализация на Earnings() за всеки конкретен тип служител.

```
public abstract class Employee
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor

   // return string representation of Employee object, using properties
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // abstract method overridden by derived classes
   public abstract decimal Earnings(); // no implementation here
} // end abstract class Employee
```

Клас SalriedEmployee

- Разширява Employee;
- Има конструктор за инициализация, препращащ към базовия конструктор;
- Property WeekSalary за достъп до даннов член с валидация при промяна;
- Реализация на метода Earnings();
- Реализация на ToString() използваща реализацията на базовия клас + собствена;

Ако нямахме реализация за Earning() – класът щеше отново да е абстрактен.
Ако нямахме реализация за ToString() – SalaryEmployee щеше да използва ToString() на Employee, без добавките за salaryemployee.

Методът използва и базовата реализация (code reusing), както и информация от собственото property

```
using System;

public class SalariedEmployee : Employee
{
   private decimal weeklySalary;

   // four-parameter constructor
   public SalariedEmployee( string first, string last, string ssn,
      decimal salary ) : base( first, last, ssn )
   {
      WeeklySalary = salary; // validate salary via property
   } // end four-parameter SalariedEmployee constructor

   // property that gets and sets salaried employee's salary
   public decimal WeeklySalary
   {
      get
      {
         return weeklySalary;
      } // end get
      set
      {
         if ( value >= 0 ) // validation
            weeklySalary = value;
         else
            throw new ArgumentOutOfRangeException( "WeeklySalary",
               value, "WeeklySalary must be >= 0" );
      } // end set
   } // end property WeeklySalary

   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return WeeklySalary;
   } // end method Earnings

   // return string representation of SalariedEmployee object
   public override string ToString()
   {
      return string.Format( "salaried employee: {0}\n{1}: {2:C}",
         base.ToString(), "weekly salary", WeeklySalary );
   } // end method ToString
} // end class SalariedEmployee
```

Клас HourEmployee

- Има конструктор, викащ отново и базовия;
- Има 2 пропъртита **Wage** и **Hours**, реализиращи и съответни верификации в set частта.
- Метод **Earnings()** правещ сметките;
- Метод **ToString()** допълващ (и ползващ) базовия.

```
// HourlyEmployee class that extends Employee.
using System;

public class HourlyEmployee : Employee
{
   private decimal wage; // wage per hour
   private decimal hours; // hours worked for the week

   // five-parameter constructor
   public HourlyEmployee( string first, string last, string ssn,
      decimal hourlyWage, decimal hoursWorked )
      : base( first, last, ssn )
   {
      Wage = hourlyWage; // validate hourly wage via property
      Hours = hoursWorked; // validate hours worked via property
   } // end five-parameter HourlyEmployee constructor

   // property that gets and sets hourly employee's wage
   public decimal Wage
   {
      get
      {
         return wage;
      } // end get
      set
      {
         if ( value >= 0 ) // validation
            wage = value;
         else
            throw new ArgumentOutOfRangeException( "Wage",
               value, "Wage must be >= 0" );
      } // end set
   } // end property Wage
```

```
   // property that gets and sets hourly employee's hours
   public decimal Hours
   {
      get
      {
         return hours;
      } // end get
      set
      {
         if ( value >= 0 && value <= 168 ) // validation
            hours = value;
         else
            throw new ArgumentOutOfRangeException( "Hours",
               value, "Hours must be >= 0 and <= 168" );
      } // end set
   } // end property Hours

   // calculate earnings; override Employee's abstract method Earnings
   public override decimal Earnings()
   {
      if ( Hours <= 40 ) // no overtime
         return Wage * Hours;
      else
         return ( 40 * Wage ) + ( ( Hours - 40 ) * Wage * 1.5M );
   } // end method Earnings

   // return string representation of HourlyEmployee object
   public override string ToString()
   {
      return string.Format(
         "hourly employee: {0}\n{1}: {2:C}; {3}: {4:F2}",
         base.ToString(), "hourly wage", Wage, "hours worked", Hours );
   } // end method ToString
} // end class HourlyEmployee
```

Клас CommissionEmployee

Има:

- **конструктор;**
- **Собствени даннови членове;**
- **Реализация на метода Earnings()**
- **Реализация на метод ToString(), ползваща и базовата.**

```
// CommissionEmployee class that extends Employee.
using System;

public class CommissionEmployee : Employee
{
   private decimal grossSales; // gross weekly sales
   private decimal commissionRate; // commission percentage

   // five-parameter constructor
   public CommissionEmployee( string first, string last, string ssn,
      decimal sales, decimal rate ) : base( first, last, ssn )
   {
      GrossSales = sales; // validate gross sales via property
      CommissionRate = rate; // validate commission rate via property
   } // end five-parameter CommissionEmployee constructor

   // property that gets and sets commission employee's gross sales
   public decimal GrossSales
   {
      get
      {
         return grossSales;
      } // end get
      set
      {
```

```
         if ( value >= 0 )
            grossSales = value;
         else
            throw new ArgumentOutOfRangeException(
               "GrossSales", value, "GrossSales must be >= 0" );
      } // end set
   } // end property GrossSales

   // property that gets and sets commission employee's commission rate
   public decimal CommissionRate
   {
      get
      {
         return commissionRate;
      } // end get
      set
      {
         if ( value > 0 && value < 1 )
            commissionRate = value;
         else
            throw new ArgumentOutOfRangeException( "CommissionRate",
               value, "CommissionRate must be > 0 and < 1" );
      } // end set
   } // end property CommissionRate

   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return CommissionRate * GrossSales;
   } // end method Earnings

   // return string representation of CommissionEmployee object
   public override string ToString()
   {
      return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
         "commission employee", base.ToString(),
         "gross sales", GrossSales, "commission rate", CommissionRate );
   } // end method ToString
} // end class CommissionEmployee
```

Създаване наследник на CommissionEmployee – класът

BasePlusCommissionemployee

- Създава втори ниво в йерархията;
- Има конструктор, викащ и базовия;
- Даннов член;
- Property, работещо със собствения даннов член;
- Метод Earnings() за определяне заплащането за този вид. Вика базовия от Commissionemployee
- Метод ToString(), викащ родителските реализации за едноименния метод.

Т.е. имаме chain of method calls включващ трите нива на йерархията ни.

```
public class BasePlusCommissionEmployee : CommissionEmployee
{
   private decimal baseSalary; // base salary per week

   // six-parameter constructor
   public BasePlusCommissionEmployee( string first, string last,
      string ssn, decimal sales, decimal rate, decimal salary )
      : base( first, last, ssn, sales, rate )
   {
      BaseSalary = salary; // validate base salary via property
   } // end six-parameter BasePlusCommissionEmployee constructor

   // property that gets and sets
   // base-salaried commission employee's base salary
   public decimal BaseSalary
   {
      get
      {
         return baseSalary;
      } // end get
      set
      {
         if ( value >= 0 )
            baseSalary = value;
         else
            throw new ArgumentOutOfRangeException( "BaseSalary",
               value, "BaseSalary must be >= 0" );
      } // end set
   } // end property BaseSalary
```

```
   // calculate earnings; override method Earnings in CommissionEmployee
   public override decimal Earnings()
   {
      return BaseSalary + base.Earnings();
   } // end method Earnings

   // return string representation of BasePlusCommissionEmployee object
   public override string ToString()
   {
      return string.Format( "base-salaried {0}; base salary: {1:C}",
         base.ToString(), BaseSalary );
   } // end method ToString
} // end class BasePlusCommissionEmployee
```

Полиморфизъм при изпълнението.
Оператор: is и Downcasting

- За да тестваме приложението си, създаваме инстанции за всеки от 4 класа;
- Приложението ще работи с тези инстанции както поотделно – чрез променливи от съответния тип, така и чрез масив от Employee (полиморфично);

- Тук ще трябва и реализция на повишението с 10% за BasePlusSalaryEmployee , което ще изисква определяне на конкретния тип в run-time.

- Обекти от всеки тип се създават динамично;

- При извеждане на екран, отново трябва да се опредли типа и заплащането за всеки обект от различен тип. При викане на WriteLine() и формиране на низ за извеждане, имплицитно се вика метод ToString() за конкретния тип

```
public static void Main( string[] args )
{
   // create derived-class objects
   SalariedEmployee salariedEmployee =
      new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00M );
   HourlyEmployee hourlyEmployee =
      new HourlyEmployee( "Karen", "Price",
      "222-22-2222", 16.75M, 40.0M );
   CommissionEmployee commissionEmployee =
      new CommissionEmployee( "Sue", "Jones",
      "333-33-3333", 10000.00M, .06M );
   BasePlusCommissionEmployee basePlusCommissionEmployee =
      new BasePlusCommissionEmployee( "Bob", "Lewis",
      "444-44-4444", 5000.00M, .04M, 300.00M );

   Console.WriteLine( "Employees processed individually:\n" );

   Console.WriteLine( "{0}\nearned: {1:C}\n",
      salariedEmployee, salariedEmployee.Earnings() );
   Console.WriteLine( "{0}\nearned: {1:C}\n",
      hourlyEmployee, hourlyEmployee.Earnings() );
   Console.WriteLine( "{0}\nearned: {1:C}\n",
      commissionEmployee, commissionEmployee.Earnings() );
   Console.WriteLine( "{0}\nearned: {1:C}\n",
      basePlusCommissionEmployee,
      basePlusCommissionEmployee.Earnings() );
```

1

Литерал, представящ реално число 800.00

Присвояване на обект от производен клас на указател към базов

- Кодът **А** присвоява на елемент на масива обекти от различен тип. Това е допустимо поради наследяемостта, т.е. може да присвоявате референции на производни класове на променливи от базовия тип, дори и когато базовия тип (Employee) е абстрактен клас.

- Кодът **Б** итерира през масива employees, като вика ToString() и Earnings() над променлива от тип Employee – т.е. на нея се присвояват различни по тип Employee.
Но винаги се вика подходящия метод!
Решението кой ToString() и кой Earnings() се ползва се взема в run-time на базата типа на текущо реферирания обект от currentEmployee!

Този процес е познат като dynamic binding или късно свързване

2

```
// create four-element Employee array
Employee[] employees = new Employee[ 4 ];

// initialize array with Employees of derived types
employees[ 0 ] = salariedEmployee;
employees[ 1 ] = hourlyEmployee;
employees[ 2 ] = commissionEmployee;
employees[ 3 ] = basePlusCommissionEmployee;

Console.WriteLine( "Employees processed polymorphically:\n" );

// generically process each element in array employees
foreach ( Employee currentEmployee in employees )
{
   Console.WriteLine( currentEmployee ); // invokes ToString

   // determine whether element is a BasePlusCommissionEmployee
   if ( currentEmployee is BasePlusCommissionEmployee )
   {
      // downcast Employee reference to
      // BasePlusCommissionEmployee reference
      BasePlusCommissionEmployee employee =
         ( BasePlusCommissionEmployee ) currentEmployee;

      employee.BaseSalary *= 1.10M;
```

А

Б

```
            Console.WriteLine(
               "new base salary with 10% increase is: {0:C}",
               employee.BaseSalary );
         } // end if

         Console.WriteLine(
            "earned {0:C}\n", currentEmployee.Earnings() );
      } // end foreach

      // get type name of each object in employees array
      for ( int j = 0; j < employees.Length; j++ )
         Console.WriteLine( "Employee {0} is a {1}", j,
            employees[ j ].GetType() );
   } // end Main
} // end class PayrollSystemTest
```

3

Б

Метод на Object

Вика се подходящия метод ToString() за типа върнат от GetType()

Увеличение с 10% на заплащането за BasePlusCommissionEmployee

Операторът is помага да се определи дали текущия Employee е BasePlusCommissionEmployee
Тази проверка би върнала true и за всеки наследник на BasePlusCommissionEmployee (ако имаше такъв).
Тогава извършваме downcast на currentEmployee от тип Employee към тип BasePlusCommissionEmployee (те са йерархично свързани).
Едва сега можем да достъпваме пропъртито BaseSalary. Опит да ползваме метод на производен клас директно от обект на базов клас е compilation error!
(downcasting генерира exception InvalidCastException при недопустимо преобразуване.
За да го избегнем в C# можем да използваме: as
(currentemployee as BasePlusCommissionEmployee) което връща null или BasePlusCommissionEmployee)

Разрешени присвоявания между
базов клас и производен клас

1. Присвояване на референция към базов клас към променлива от базовия клас е правилно;
2. Присвояване на референция на производен клас към променлива от същия производен клас е правилно;

3. референция на производен да се присвои към променлива от базов клас е „safe”, защото обектът от производния клас всъщност е обект и от базовия клас. Разбира се, имаме достъп само до членовете на базовия. Не можем да достъпваме членове само на производния клас през променлива от базовия му клас. Това е грешка при компилация;

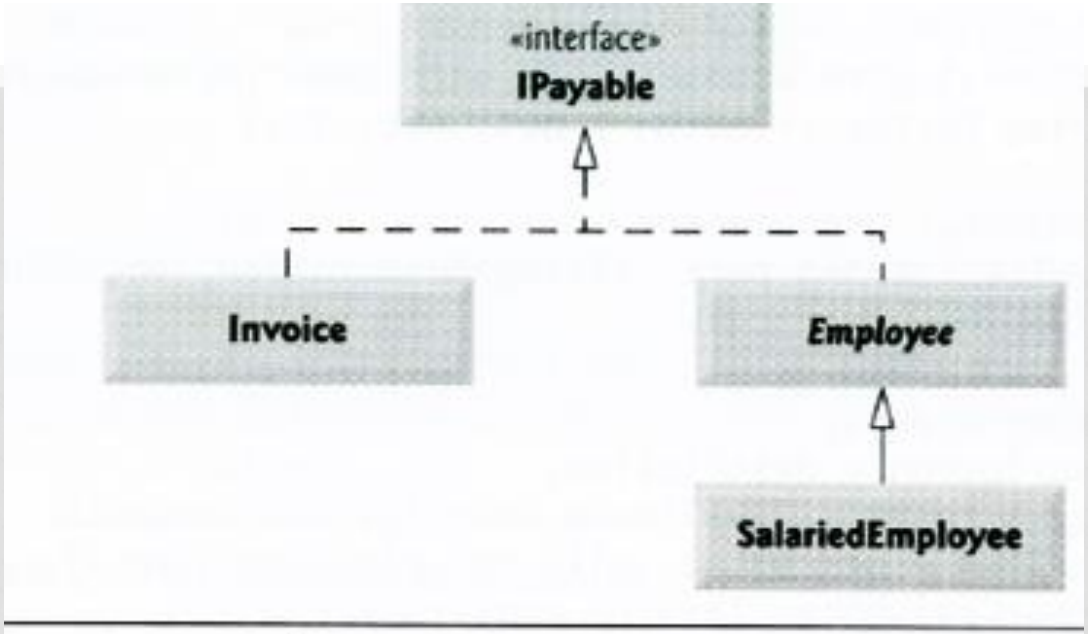
4. Опит за присвояване на референция от базов клас към променлива от производен клас е грешка при компилация. За да се избегне това, може референцията към базовия да се преобразува явно (cast) към такава на производен. Или да се преобразува чрез оператор „as“. Ако по време на изпълнение, сочения от такава променлива обект не е от производния тип, имаме exception (освен ако се е използвал оператор as).

Да разширим примера:
въвеждаме и плащания през фактури

- Т.е. ще имаме някои общи с преди действия: например, изчисляване на плащането (преди беше на заплащането само);
- Добре е да въведем interface за обекти, поддържащи плащане. Напр. IPayable;
- Интерфейсът започва с ключова дума interface и съдържа само abstract методи, абстрактни пропъртита и абстрактни events. Т.е. интерфейсните елементи неявно са public и abstract. Всеки интерфейс може да е наследник на други интерфейси;
- Интерфейсът служи да накара несвързани класове да съдържат общи методи с различна реализация. За нашия пример, и Employee и Invoice класовете могат да имплементират IPayable;

- Интерфейсът се използва вместо абстрактен клас, когато липсва реализация по default, която ще се наследява (полета или методи)

IPayable ще съдържа например GetPaymentAmount();
Йерархията класове ще се видоизмени:



IPayable interface and class hierarchy UML class diagram.

```
public interface IPayable
{
        decimal GetPaymentAmount();
}
```

Създаване на клас Invoice

```
// Invoice class implements IPayable.
using System;

public class Invoice : IPayable
{
   private int quantity;
   private decimal pricePerItem;

   // property that gets and sets the part number on the invoice
   public string PartNumber { get; set; }

   // property that gets and sets the part description on the invoice
   public string PartDescription { get; set; }

   // four-parameter constructor
   public Invoice( string part, string description, int count,
      decimal price )
   {
      PartNumber = part;
      PartDescription = description;
      Quantity = count; // validate quantity via property
      PricePerItem = price; // validate price per item via property
   } // end four-parameter Invoice constructor
```

C# не позволява множествено наследяване на класове, но позволява множествено наследяване на интерфейси

```
// property that gets and sets the quantity on the invoice
public int Quantity
{
   get
   {
      return quantity;
   } // end get
   set
   {
      if ( value >= 0 ) // validate quantity
         quantity = value;
      else
         throw new ArgumentOutOfRangeException( "Quantity",
            value, "Quantity must be >= 0" );
   } // end set
} // end property Quantity

// property that gets and sets the price per item
public decimal PricePerItem
{
   get
   {
      return pricePerItem;
   } // end get
   set
   {
      if ( value >= 0 ) // validate price
         quantity = value;
```

```
            else
               throw new ArgumentOutOfRangeException( "PricePerItem",
                  value, "PricePerItem must be >= 0" );
         } // end set
      } // end property PricePerItem

      // return string representation of Invoice object
      public override string ToString()
      {
         return string.Format(
            "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
            "invoice", "part number", PartNumber, PartDescription,
            "quantity", Quantity, "price per item", PricePerItem );
      } // end method ToString

      // method required to carry out contract with interface IPayable
      public decimal GetPaymentAmount()
      {
         return Quantity * PricePerItem; // calculate total cost
      } // end method GetPaymentAmount
   } // end class Invoice
```

Промяна в класа Employee за да ползва IPayable

1. Класът е почти идентичен с предходната реализация с 2 изключения;
2. Наследява IPayable и го имплементира. За целта метода Earnings() е преименован на GetPaymentAmount(). Както и преди, GetPaymentAmount()се имплементира в отделните подтипове на Employee. Затова той е abstract.
   Всъщност, можем в тялото на GetPaymentAmount() да викаме просто Earnings() и да не променяме другите класове от йерархията;

3. Employee става abstract защото има абстрактен метод;

```
// Employee abstract base class.
public abstract class Employee : IPayable
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor

   // return string representation of Employee object
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // Note: We do not implement IPayable method GetPaymentAmount here, so
   // this class must be declared abstract to avoid a compilation error.
   public abstract decimal GetPaymentAmount();
} // end abstract class Employee
```

Модифициране на SalariedEmployee с оглед IPayable

- Разбира се, следва да се имплементира GetPaymentAmount();
- Същото следва да се направи и с останалите класове, наследили Employee;

- Що се отнася до резултата, върнат от “is” при евентуална употреба:
  - Employee is an IPayable;
  - SalariedEmployee is an IPayable;
  - SalariedEmployee обект може да бъде присвояван на променлива от тип Employee;
  - Обект от тип SalariedEmployee може да бъде присвоен на променлива от тип IPayable;
  - Обект от тип Invoice също може да бъде присвояван на променлива от тип IPayable.

```
// SalariedEmployee class that extends Employee.
using System;

public class SalariedEmployee : Employee
{
   private decimal weeklySalary;

   // four-parameter constructor
   public SalariedEmployee( string first, string last, string ssn,
      decimal salary ) : base( first, last, ssn )
   {
      WeeklySalary = salary; // validate salary via property
   } // end four-parameter SalariedEmployee constructor

   // property that gets and sets salaried employee's salary
   public decimal WeeklySalary
   {
      get
      {
         return weeklySalary;
      } // end get
      set
      {
         if ( value >= 0 ) // validation
            weeklySalary = value;
         else
            throw new ArgumentOutOfRangeException( "WeeklySalary",
               value, "WeeklySalary must be >= 0" );
      } // end set
   } // end property WeeklySalary
```

```
   // calculate earnings; implement interface IPayable method
   // that was abstract in base class Employee
   public override decimal GetPaymentAmount()
   {
      return WeeklySalary;
   } // end method GetPaymentAmount

   // return string representation of SalariedEmployee object
   public override string ToString()
   {
      return string.Format( "salaried employee: {0}\n{1}: {2:C}",
         base.ToString(), "weekly salary", WeeklySalary );
   } // end method ToString
} // end class SalariedEmployee
```

Използване на IPayable за полиморфична употреба на Employees и Invoices

Тестовият клас PayableInterfaceTest показва как в едно приложение могат да се използват (полиморфично) методи на Invoices и Employees при наличието на IPayable.

- Създаваме масив от 4 payableObjects (от тип интерфейс IPayable);
- После ги инициализираме : и с Invoice, и сус SalariedEmployee обекти;
- После в цикъл "foreach" обработваме последователно всеки един от тях (полиморфично), извеждайки типа обект и плащанията по него.
  Така WriteLine() в цикъл вика ToString() за всеки IPayable обект, макар че ToString() не е деклариран в интерфейса. Това е възможно, защото всички (включително и интерфейсния тип) са наследници на Object и като такива поддържат ToString().
  Вика се GetPaymentAmount() за всеки обект от масива payableObjects, независимо от типа му.
  Т.е. вика се различна имплементация за ToString() и за GetPaymentAmount().
  Така например, когато currentPayable реферира Invoice, изпълняват се ToString() и GetPaymentAmount() на Invoice!

```
// Tests interface IPayable with disparate classes.
using System;

public class PayableInterfaceTest
{
   public static void Main( string[] args )
   {
      // create four-element IPayable array
      IPayable[] payableObjects = new IPayable[ 4 ];

      // populate array with objects that implement IPayable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00M );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95M );
      payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith",
         "111-11-1111", 800.00M );
      payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes",
         "888-88-8888", 1200.00M );

      Console.WriteLine(
         "Invoices and Employees processed polymorphically:\n" );

      // generically process each element in array payableObjects
      foreach ( var currentPayable in payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         Console.WriteLine( "{0}\npayment due: {1:C}\n",
            currentPayable, currentPayable.GetPaymentAmount() );
      } // end foreach
   } // end Main
} // end class PayableInterfaceTest
```

примерни изходни данни биха изглеждали така:

Invoices and Employees processed polymorphically:

Invoice:
part number: 01234 (seat)
quantity: 2
price per item: $123.00
payment due: $ 246.00

Invoice:
part number: 02564 (tire)
quantity: 4
price per item: $79.95
payment due: $ 319.80

salaried employee: Ivan Petrov
social security number: 123-11-222
weekly salary: $100.00
payment due: $100.00

…….

Често ползвани интерфейси на .NET FCL

Използването им е подобно на използването на IPayable от предните слайдове;

Interface	Description
`IComparable`	As you learned in Chapter 3, C# contains several comparison operators (e.g., `<`, `<=`, `>`, `>=`, `==`, `!=`) that allow you to compare simple-type values. In Section 12.8 you'll see that these operators can be defined to compare two objects. Interface `IComparable` can also be used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `CompareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `CompareTo` to return a value indicating whether the object on which it's invoked is less than (negative integer return value), equal to (`0` return value) or greater than (positive integer return value) the object passed as an argument, using any criteria you specify. For example, if class `Employee` implements `IComparable`, its `CompareTo` method could compare `Employee` objects by their earnings amounts. Interface `IComparable` is commonly used for ordering objects in a collection such as an array.
`IComponent`	Implemented by any class that represents a component, including Graphical User Interface (GUI) controls (such as buttons or labels). Interface `IComponent` defines the behaviors that components must implement.

Interface	Description
`IDisposable`	Implemented by classes that must provide an explicit mechanism for *releasing* resources. Some resources can be used by only one program at a time. In addition, some resources, such as files on disk, are unmanaged resources that, unlike memory, cannot be released by the garbage collector. Classes that implement interface `IDisposable` provide a `Dispose` method that can be called to explicitly release resources.
`IEnumerator`	Used for iterating through the elements *of a collection* (such as an array) one element at a time. Interface `IEnumerator` contains method `MoveNext` to move to the next element in a collection, method `Reset` to move to the position before the first element and property `Current` to return the object at the current location.

Обобщение : ООП, полиморфизъм, използване на интерфейси

- With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the app.

- With polymorphism, the same method name and signature can be used to cause different actions to occur, depending on the type of object on which the method is invoked.
- Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated in a system without requiring modification of the base system.

Demonstrating Polymorphic Behavior

- Invoking a method on a derived-class object via a base-class reference invokes the derived-class functionality—the type of the referenced object determines which method is called.
- A base-class reference can be used to invoke only the methods declared in the base class. If an app needs to perform a derived-class-specific operation on a derived-class object referenced by a base-class variable, the app must first downcast the base-class reference to a derived-class reference.

Abstract Classes and Methods

- Abstract base classes are incomplete classes for which you never intend to instantiate objects.
- The purpose of an abstract class is primarily to provide an appropriate base class from which other classes can inherit, and thus share a common design.
- Classes that can be used to instantiate objects are called concrete classes.
- You make a class abstract by declaring it with keyword `abstract`.
- Each concrete derived class of an abstract base class must provide concrete implementations of the base class's `abstract` methods and properties.
- Failure to implement a base class's `abstract` methods and properties in a derived class is a compilation error unless the derived class is also declared `abstract`.
- Although we cannot instantiate objects of `abstract` base classes, we can use them to declare variables that can hold references to objects of any concrete class derived from those `abstract` classes.

- By declaring a method `abstract`, we indicate that each concrete derived class must provide an appropriate implementation.
- All `virtual` method calls are resolved at execution time, based on the type of the object to which the reference-type variable refers. This process is known as dynamic binding or late binding.
- The `is` operator determines whether the type of the object in the left operand matches the type specified by the right operand and returns true if the two have an *is-a* relationship.
- The `as` operator performs a downcast that returns a reference to the appropriate object if the downcast is successful and returns `null` if the downcast fails.
- Every object knows its own type and can access this information through method `GetType`, which all classes inherit from class `object`.
- Assigning a base-class reference to a derived-class variable is not allowed without an explicit cast or without using the `as` operator. The `is` operator can be used to ensure that such a cast is performed only if the object is a derived-class object.

`sealed` *Methods and Classes*

- A method that's declared `sealed` in a base class cannot be overridden in a derived class.
- A class that's declared `sealed` cannot be a base class (i.e., a class cannot extend a `sealed` class). All methods in a `sealed` class are implicitly `sealed`.

- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
- An interface declaration begins with keyword `interface` and can contain only `abstract` methods, properties, indexers and events.
- All interface members are implicitly declared both `public` and `abstract`. They do not specify any implementation details, such as concrete method declarations.
- Each interface can extend one or more other interfaces to create a more elaborate interface that other classes can implement.
- To use an interface, a class must specify that it implements the interface by listing it after the colon (:) in the class declaration.
- A class that implements an interface but doesn't implement all the interface's members must be declared `abstract` and contain an abstract declaration of each unimplemented interface member.
- The UML expresses the relationship between a class and an interface through a realization. A class is said to "realize," or implement, an interface.
- To implement more than one interface, use a comma-separated list of interface names after the colon (:) in the class declaration.
- Inheritance and interfaces are similar in their implementation of the *is-a* relationship. An object of a class that implements an interface may be thought of as an object of that interface type.
- All methods of class `object` can be called by using a reference of an interface type—the reference refers to an object, and all objects inherit the methods of class `object`.