Темы урока

Наследование

Наследование является одним из фундаментальных атрибутов объектно-ориентированного программирования.

Оно позволяет определить **дочерний класс**, который использует (наследует), расширяет или изменяет возможности **родительского класса**.

Класс, члены которого наследуются, называется **базовым или родительским классом**. Класс, который наследует члены базового класса, называется **производным или дочерним классом**.

C# и .NET поддерживают только **одиночное наследование**. Это означает, что каждый класс может наследовать члены только одного класса.

Зато наследование может быть выполнено “по цепочке”, т. е., класс D наследуется от класса C, C и B - от класса A.

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому все классы кроме своих собственных методов, также будут иметь и методы класса Object:

- ToString()
- Equals()
- GetHashCode()
- GetType()

Пример простого наследования

Давайте напишем класс Person:

```
// Person.cs

using System;

public class Person
{
  public string Name { get; set; }
  public DateTimeOffset DateOfBirth { get; set; }

  public string ShortDescription
  {
    get { return $"{GetType().Name} " +
                 $"name: {Name}, " +
                 $"date of birth: {DateOfBirth:dd-MM-yy}"; }
  }

  public void WriteShortDecription()
  {
    Console.WriteLine(ShortDescription);
  }
}
```

Теперь мы можем создать экземпляр этого класса и вызвать метод WriteShortDescription в основном коде программы:

```
// Program.cs

private static void Main(string[] args)
{
  Person p1 = new Person
  {
    Name = "Andrei",
    DateOfBirth = DateTimeOffset.Parse("1982-03-14")
  };

  p1.WriteShortDecription(); // Person name: Andrei, date of birth: 14-03-82
}
```

Кроме имени и даты рождения мы получим еще и имя класса в самом начале строки: Person.

Теперь совершим наследование.

Наш класс Person будет базовым или родительским. Создадим производный (или дочерний) класс Employee (сотрудник):

```
// Employee.cs

public class Employee : Person
{
}
```

Как видите, этот класс не содержит никаких собственных членов, но он базируется на классе Person, а значит имеет все его члены:

```
// Program.cs

private static void Main(string[] args)
{
  Person p1 = new Person
  {
    Name = "Andrei",
    DateOfBirth = DateTimeOffset.Parse("1982-03-14")
  };
  p1.WriteShortDecription(); // Person name: Andrei, date of birth: 14-03-82

  Employee e1 = new Employee
  {
    Name = "Andrei",
    DateOfBirth = DateTimeOffset.Parse("1982-03-14")
  };
  e1.WriteShortDecription(); // Employee name: Andrei, date of birth: 14-03-82
}
```

Теперь добавим несколько специфичных для работника членов в наш класс:

```
// Employee.cs

public class Employee : Person
{
  public string EmployeeCode { get; set; }
  public DateTimeOffset HireDate { get; set; }
}
```

Давайте установим значения новых членов при создании класса:

```
// Program.cs

private static void Main(string[] args)
{
  //...

  Employee e1 = new Employee
  {
    Name = "Andrei",
    DateOfBirth = DateTimeOffset.Parse("1982-03-14"),
    EmployeeCode = "000001",
    HireDate = DateTimeOffset.Parse("2016-10-01")
  };
  e1.WriteShortDecription(); // Employee name: Andrei, date of birth: 14-03-82
}
```

Запустим этот код. Мы видим, что поскольку метод WriteShortDecription() был унаследован от класса Person, то он и выводит только свойства класса Person, а наши новые – нет.

Сокрытие членов базового класса: new

Пример сокрытия

Мы можем захотеть изменить свойство, формирующее строку для вывода на экран для класса Employee, для этого, кажется, необходимо всего лишь переписать это свойство внутри дочернего класса:

```
// Employee.cs

public class Employee : Person
{
  public string EmployeeCode { get; set; }
  public DateTimeOffset HireDate { get; set; }

  public string ShortDescription
  {
    get
    {
      return $"{GetType().Name} " +
             $"code: {EmployeeCode}, " +
             $"name: {Name}, " +
             $"date of birth: {DateOfBirth:dd-MM-yy}, " +
             $"hire date: {HireDate:dd-MM-yy}";
    }
  }
}
```

Однако, если мы запустим такой код на выполнение, мы по-прежнему увидим старый формат вывода для сотрудника.

Это произошло потому, что мы вызываем метод WriteShortDecription() базового класса, а в базовом классе мы обращаемся к свойству базового класса. Давайте **скроем** и сам метод WriteShortDecription(), чтобы достичь желаемого результата:

```
// Employee.cs

public class Employee : Person
{
  public string EmployeeCode { get; set; }
  public DateTimeOffset HireDate { get; set; }

  public string ShortDescription
  {
    get
    {
      return $"{GetType().Name} " +
             $"code: {EmployeeCode}, " +
             $"name: {Name}, " +
             $"date of birth: {DateOfBirth:dd-MM-yy}, " +
             $"hire date: {HireDate:dd-MM-yy}";
    }
  }

  public void WriteShortDecription()
  {
    Console.WriteLine(ShortDescription);
  }
}
```

Теперь мы достигли желаемого результата: для класса Employee выводится полный набор его характеристик:

```
Employee code: 000001, name: Andrei, date of birth: 14-03-82, hire date: 14-10-01
```

То, что мы сделали, написав собственные реализации свойства и метода в дочернем классе называется **сокрытие** (**hiding**) членов базового класса. В нашем коде сейчас это происходит неявно, и Visual Studio показывает нам предупреждения к соответствующим строкам кода в дочернем классе:

```
Warning CS0108 'Employee.ShortDescription' hides inherited member
'Person.ShortDescription'. Use the new keyword if hiding was intended.
```

Мы можем избежать этих предупреждений, если сделаем **явное сокрытие** членов базового класса с **помощью ключевого слова new**:

```
// Employee.cs

public class Employee : Person
{
  public string EmployeeCode { get; set; }
  public DateTimeOffset HireDate { get; set; }

  new public string ShortDescription
  {
    get
    {
      return $"{GetType().Name} " +
             $"code: {EmployeeCode}, " +
             $"name: {Name}, " +
             $"date of birth: {DateOfBirth:dd-MM-yy}, " +
             $"hire date: {HireDate:dd-MM-yy}";
    }
  }

  new public void WriteShortDecription()
  {
    Console.WriteLine(ShortDescription);
  }
}
```

Самостоятельная работа

Напишите собственный базовый класс BaseDocument, который бы описывал произвольный документ и имел бы:

- Свойства:
  - DocName типа string: наименование документа
  - DocNumber типа string: номер документа
  - IssueDate типа DateTimeOffset: дата выдачи
  - PropertiesString типа string: **read-only** свойство, формирующее строку для вывода на экран свойств этого класса
- Метод
  - WriteToConsole()

Затем напишите производный класс Passport, унаследованный от BaseDocument, который бы имел дополнительно свойства Country (string) для хранения страны и PersonName (string) для хранения имени владельца.

Напишите новую реализацию свойств PropertiesString и метод WriteToConsole() чтобы произвести сокрытие членов базового класса.

Создайте по одному экземпляру каждого класса в основном потоке программы, инициализируйте их свойства и выведите их на экран используя метод WriteToConsole() соответствующих классов.

Решение

```
// BaseDocument.cs

public class BaseDocument
{
  public string DocName { get; set; }

  public string DocNumber { get; set; }

  public DateTimeOffset IssueDate { get; set; }

  public string PropertiesString
  {
    get
    {
      return $"{DocName} #{DocNumber} issued {IssueDate:dd-MM-yy}";
    }
  }

  public void WriteToConsole()
  {
    Console.WriteLine(PropertiesString);
  }
}
```

```
// Passport.cs

public class Passport : BaseDocument
{
  public string Country { get; set; }

  public string PersonName { get; set; }

  public string PropertiesString
  {
    get
    {
      return $"{DocName} #{DocNumber} issued "
             + $"{IssueDate:dd-MM-yy} in "
             + $"{Country} for {PersonName}";
    }
  }

  public void WriteToConsole()
  {
    Console.WriteLine(PropertiesString);
  }
}
```

```
// Program.cs

internal class Program
{
  private static void Main(string[] args)
  {
    BaseDocument bd1 = new BaseDocument
    {
      DocName = "Drive License",
      DocNumber = "AB32DS32",
      IssueDate = DateTimeOffset.Parse("2012-02-03")
    };

    bd1.WriteToConsole();

    Passport p1 = new Passport
    {
      DocName = "Passport",
      DocNumber = "32135432",
      IssueDate = DateTimeOffset.Parse("2012-02-03"),
      PersonName = "Andrei Golyakov",
      Country = "Russia"
    };

    p1.WriteToConsole();
  }
}
```

Переопределение членов: virtual / override

Сокрытие — это весьма грубый способ работы с базовым классом. Фактически, вы просто создаете новые члены в наследнике, которые имеют такое же имя и никак не работают с членами базового класса!
Вместо того, чтобы скрывать члены базового класса, лучшим способом будет **переопределить** их используя ключевые слова virtual и override.

Пример

Чтобы сделать наследование более правильным, добавим ключевое слово **virtual** к свойству ShortDescription базового класса Person:

```
public class Person
{
  ...

  public virtual string ShortDescription
  {
    get
    {
      return $"{GetType().Name} " +
             $"name: {Name}, " +
             $"date of birth: {DateOfBirth:dd-MM-yy}";
    }
  }
}
```

Теперь у дочернего класса Employee поменяем ключевое слово new на **override**, а определение метода WriteShortDescription() закомментируем вовсе

```
public class Employee : Person
{
  ...

  public override string ShortDescription
  {
    get
    {
      return $"{GetType().Name} " +
             $"code: {EmployeeCode}, " +
             $"name: {Name}, " +
             $"date of birth: {DateOfBirth:dd-MM-yy}," +
             $"hire date: {HireDate:dd-MM-yy}";
    }
  }

  //public void WriteShortDecription()
  //{
  //  Console.WriteLine(ShortDescription);
  //}
}
```

Теперь вызовем наш код из основного потока:

```
// Program.cs

internal class Program
{
  private static void Main(string[] args)
  {
    BaseDocument bd1 = new BaseDocument
    {
      DocName = "Drive License",
      DocNumber = "AB32DS32",
      IssueDate = DateTimeOffset.Parse("2012-02-03")
    };

    bd1.WriteToConsole();

    Passport p1 = new Passport
    {
      DocName = "Passport",
      DocNumber = "32135432",
      IssueDate = DateTimeOffset.Parse("2012-02-03"),
      PersonName = "Andrei Golyakov",
      Country = "Russia"
    };

    p1.WriteToConsole();
  }
}
```

Мы видим, что метод базового класса WriteShortDecription() обратился к переопределенному в дочернем классе свойству ShortDescription:

```
Person name: Andrei, date of birth: 14-03-82
Employee code: 000001, name: Andrei, date of birth: 14-03-82, hire date: 01-10-16
```

Самостоятельная работа

Внесите соответствующие изменения в ваши классы BaseDocument и Passport, чтобы заменить новой имплементаций .

Удалите ставшую ненужной имплементацию метода WriteToConsole() у дочернего класса Passport.

Убедитесь, что код работает как и прежде:

```
// BaseDocument.cs

public class BaseDocument
{
  public string DocName { get; set; }

  public string DocNumber { get; set; }

  public DateTimeOffset IssueDate { get; set; }

  public virtual string PropertiesString
  {
    get
    {
      return $"{DocName} #{DocNumber} issued {IssueDate:dd-MM-yy}";
    }
  }

  public void WriteToConsole()
  {
    Console.WriteLine(PropertiesString);
  }
}
```

```
// Passport.cs

public class Passport : BaseDocument
{
  public string Country { get; set; }

  public string PersonName { get; set; }

  public override string PropertiesString
  {
    get
    {
      return $"{DocName} #{DocNumber} issued "
             + $"{IssueDate:dd-MM-yy} in "
             + $"{Country} for {PersonName}";
    }
  }
}
```

```
// Program.cs

internal class Program
{
  private static void Main(string[] args)
  {
    BaseDocument bd1 = new BaseDocument
    {
      DocName = "Drive License",
      DocNumber = "AB32DS32",
      IssueDate = DateTimeOffset.Parse("2012-02-03")
    };

    bd1.WriteToConsole();

    Passport p1 = new Passport
    {
      DocName = "Passport",
      DocNumber = "32135432",
      IssueDate = DateTimeOffset.Parse("2012-02-03"),
      PersonName = "Andrei Golyakov",
      Country = "Russia"
    };

    p1.WriteToConsole();
  }
}
```

Наследование конструкторов

Конструкторы не передаются производному классу при наследовании.

При наследовании конструкторов работают следующие правила

1. Производный и базовый класс могут иметь свои наборы конструкторов
2. В момент создания дочернего объекта вызываются все конструкторы его предков в порядке от самого базового до конструктора текущего класса
3. Если в базовом классе отсутствует конструктор без параметров, необходимо явно вызывать конструктор базового класса, используя ключевое слово **base**.

Пример

Давайте определим конструктор для класса Person с параметрами для инициализации его свойств:

```
public class Person
{
  ...
  public Person(string name, DateTimeOffset dateOfBirth)
  {
    Name = name;
    DateOfBirth = dateOfBirth;
    Debug.WriteLine("Constructor Person(name, dateOfBirth) called");
  }
  ...
}
```

При попытке скомпилировать код после такого изменения мы получаем ошибку в классе Employee, потому что мы нарушили правило №3.

- Класс Person больше не имеет конструктора без параметров, поскольку, при наличии хотя бы одного конструктора с параметрами, конструктор без параметров уже надо указывать явно.
- А вот класс Employee по-прежнему имеет только “дефолтный” конструктор без параметров.
- Когда мы попытаемся вызвать конструктор класса Employee без параметров ему придется (согласно правилу №2) вызвать конструктор базового класса, а он требует указания 2 аргументов, значения для которых неоткуда взять.

У нас есть 2 пути:

1. Реализовать в базовом классе Person явно ещё и конструктор без параметров
2. Реализовать в производном классе Employee конструктор требующий наличия как минимум двух аргументов для базового класса.

Второй вариант более последователен, так как, по смыслу, раз мы хотим при создании экземпляра класса Person сразу иметь определенные значения имени и даты рождения, при создании класса Employee мы тоже должны хотеть иметь как минимум эти параметры определенными.

Реализуем второй вариант. В минимальнов виде он выглядит так:

```
// Employee.cs

public class Employee : Person
{
  ...

  public Employee(string name, DateTimeOffset dateOfBirth)
    : base(name, dateOfBirth)
  {
    Debug.WriteLine("Constructor Employee(name, dateOfBirth) called");
  }

  ...
}
```

Имеет смысл также добавить и еще один конструктор, определяющий и ненаследуемые свойство класса Employee: EmployeeCode и HireDate:

```
// Employee.cs

public class Employee : Person
{
  ...

  public Employee(
    string name,
    DateTimeOffset dateOfBirth,
    string employeeCode,
    DateTimeOffset hireDate)
     : base(name, dateOfBirth)
  {
    EmployeeCode = employeeCode;
    HireDate = hireDate;
    Debug.WriteLine(
      "Constructor Employee(name, dateOfBirth, employeeCode, hireDate) called");
  }

  ...
}
```

Теперь объекты будут создаваться с использованием своих конструкторов:

```
// Program.cs

internal class Program
{
  private static void Main()
  {
    Person p1 = new Person("Andrei", DateTimeOffset.Parse("1982-03-14"));
    p1.WriteShortDecription(); // Person name: Andrei, date of birth: 14-03-82

    Employee e1 = new Employee(
      "Andrei",
      DateTimeOffset.Parse("1982-03-14"),
      "000001",
      DateTimeOffset.Parse("2016-10-01"));
    e1.WriteShortDecription();
    // Employee code: 000001, name: Andrei,
    // date of birth: 14-03-82, hire date: 01-10-16
  }
}
```

Самостоятельная работа

Создайте конструкторы для классов BaseDocument и Passport, заполняющие их свойства.

Для конструктора класса Passport не нужно запрашивать параметр docName, нам и так известно, что все документы типа Passport имеют имя “Passport”. Передайте строку “Passport” в качестве строкового литерала при вызове наследуемого конструктора посредством base.

Обновите создание объектов, используя новые конструкторы.

Убедитесь, что код работает как и прежде.

Решение

```
// BaseDocument.cs

using System;

public class BaseDocument
{
  public string DocName { get; set; }

  public string DocNumber { get; set; }

  public DateTimeOffset IssueDate { get; set; }


  public BaseDocument(string docName, string docNumber, DateTimeOffset issueDate)
  {
    DocName = docName;
    DocNumber = docNumber;
    IssueDate = issueDate;
  }

  ...
}
```

```
// Passport.cs

using System;

public class Passport : BaseDocument
{
  public string Country { get; set; }

  public string PersonName { get; set; }

  public Passport(
    string docNumber,
    DateTimeOffset issueDate,
    string country,
    string personName): base("Passport", docNumber, issueDate)
  {
    Country = country;
    PersonName = personName;
  }

  ...
}
```

```
// Program.cs

using System;

internal class Program
{
  private static void Main(string[] args)
  {
    BaseDocument bd1 = new BaseDocument(
      "Drive License",
      "AB32DS32",
      DateTimeOffset.Parse("2012-02-03"));
    bd1.WriteToConsole();

    Passport p1 = new Passport(
      "32135432",
      DateTimeOffset.Parse("2012-02-03"),
      "Andrei Golyakov",
      "Russia");
    p1.WriteToConsole();
  }
}
```

Порядок вызова конструкторов

Пример

Если мы посмотрим на конструкторы классов Person и Employee внимательно, мы увидим, что там добавлен вывод информации о вызове конструктора с уникальной строкой.

Ещё раз смотрим на последовательность создания объектов в основном потоке программы:

1. BaseDocument bd1
2. Passport p1

Давайте запустим этот код ещё раз и посмотрим в окошко Output (from Debug):

```
…
Constructor Person(name, dateOfBirth) called
…
Constructor Person(name, dateOfBirth) called
Constructor Employee(name, dateOfBirth, employeeCode, hireDate) called
```

Когда мы вызываем конструктор класса Passport, сначала вызывается конструктор наиболее базового класса из цепочки (включая конструктор класса Object). Поэтому мы видим цепочку вызова двух конструкторов для производного класса Passport.

Однако, в нашем примере мы видим, что у нас уже есть и свой собственный конструктор класса Employee, который имеет ту же сигнатуру, что и конструктор базового класса.

Если мы хотим использовать в качестве основы для конструктора более общий конструктор данного класса, мы можем использовать ключевое слово **this** по аналогии со словом base:

```
// Passport.cs

using System;

public class Passport : BaseDocument
{
  public string Country { get; set; }

  public string PersonName { get; set; }

  public Passport(
    string docNumber,
    DateTimeOffset issueDate,
    string country,
    string personName): this("Passport", docNumber, issueDate)
  {
    Country = country;
    PersonName = personName;
  }

  ...
}
```

Если мы запустим этот код, то увидим, что теперь в Output Debug при создании объекта класса Passport через конструктор с четырьмя параметрами логируется последовательный вызов в следующем порядке:

```
…
Constructor Person(name, dateOfBirth) called
…
Constructor Person(name, dateOfBirth) called
Constructor Employee(name, dateOfBirth) called
Constructor Employee(name, dateOfBirth, employeeCode, hireDate) called
```

Итак, надо запомнить, конструкторы вызываются в порядке от наиболее базовых до непосредственно вызываемого.

Ссылки на базовый класс и объекты производных классов

C# является строго типизированным языком программирования. В этом языке строго соблюдается принцип совместимости типов. Это означает, что переменная ссылки на объект класса одного типа, как правило, не может ссылаться на объект класса другого типа.

Вообще говоря, переменная ссылки на объект может ссылаться только на объект своего типа.

Но из этого принципа строгого соблюдения типов в C# имеется одно важное исключение: **переменной ссылки на объект базового класса может быть присвоена ссылка на объект любого производного от него класса**.

Такое присваивание считается вполне допустимым, поскольку экземпляр объекта производного типа инкапсулирует экземпляр объекта базового типа. Следовательно, по ссылке на объект базового класса можно обращаться к объекту производного класса.

Пример

```
// Program.cs

using System;

internal class Program
{
  private static void Main()
  {
    Person p1 = new Person("Andrei", DateTimeOffset.Parse("1982-03-14"));
    p1.WriteShortDecription(); // Person name: Andrei, date of birth: 14-03-82

    Person e1 = new Employee(
      "Andrei",
      DateTimeOffset.Parse("1982-03-14"),
      "000001",
      DateTimeOffset.Parse("2016-10-01"));

    e1.WriteShortDecription();
    // Employee code: 000001, name: Andrei,
    // date of birth: 14-03-82, hire date: 01-10-16
  }
}
```

Следует особо подчеркнуть, что доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается. Это означает, что если ссылка на объект производного класса присваивается переменной ссылки на объект базового класса, то доступ разрешается только к тем частям этого объекта, которые определяются базовым классом. И в этом есть своя логика, поскольку базовому классу ничего не известно о тех членах, которые добавлены в производный от него класс.

Например, мы не можем обратиться к полям объекта `e1.EmployeeCode` или `e1.HireDate`, так как они не определены в базовом классе.

Однако можно воспользоваться проверкой типа внутреннего объекта используя операторы is и приведением через as или просто явным приведением:

```
// Program.cs

using System;

internal class Program
{
  private static void Main()
  {
    var persons = new Person[3];

    persons[0] = new Person(
      "Maria", DateTimeOffset.Parse("1987-03-01"));

    persons[1] = new Person(
      "Sergey", DateTimeOffset.Parse("1981-12-27"));

    persons[2] = new Employee(
      "Andrei",
      DateTimeOffset.Parse("1982-03-14"),
      "000001",
      DateTimeOffset.Parse("2016-10-01"));

    foreach (var person in persons)
    {
      if (person is Employee)
      {
        var employee = (Employee) person;
        Console.WriteLine(
          $"{employee.GetType().Name} " +
          $"code: {employee.EmployeeCode}, " +
          $"name: {employee.Name}, " +
          $"date of birth: {employee.DateOfBirth:dd-MM-yy}, " +
          $"hire date: {employee.HireDate:dd-MM-yy}");
      }
      else
      {
        Console.WriteLine(
          $"{person.GetType().Name} " +
          $"name: {person.Name}, " +
          $"date of birth: {person.DateOfBirth:dd-MM-yy}");
      }
    }
  }
}
```

Один из самых важных моментов для присваивания ссылок на объекты производного класса переменным базового класса наступает тогда, когда конструкторы вызываются в иерархии классов. Как вам должно быть уже известно, в классе нередко определяется конструктор, принимающий объект своего класса в качестве параметра. Благодаря этому в классе может быть сконструирована копия его объекта.

Самостоятельная работа

Добавьте классу Passport метод ChangeIssueDate(DateTimeOffset newIssueDate), создайте массив объектов базового класса и проверяя, и перебирая их в цикле поменяйте всем паспортам IssueDate на сегодняшнее число, и выведите на экран информацию о документе используя метод базового класса WriteToConsole().

Домашнее задание

Унаследовать от класса ReminderItem, написанного в прошлой домашней работе 2 класса:

1. PhoneReminderItem
    a. с дополнительным свойством PhoneNumber (string) номер телефона, куда нужно послать сообщение.
    b. с конструктором, который кроме параметров базового класса имеет также дополнительный параметр (для заполнения нового поля)
    c. с переопределенным методом WriteProperties()
2. ChatReminderItem
    a. с дополнительными свойствами
        i. ChatName (string) Название чата
        ii. AccountName (string) Имя аккаунта в чате, которому нужно послать сообщение
    b. с конструктором, который кроме параметров базового класса имеет также два дополнительных (для заполнения новых полей)
    c. с переопределенным методом WriteProperties(), выводящим все свойства

Обновить WriteProperties всем классам таким образом, чтобы в самом начале выводился тип объекта.

Создать лист объектов базового типа инициализированный как минимум 3-мя объектами разных типов. Вывести на экран их свойства WriteProperties().