

Лекция 3 . Основные принципы ООП

Парадигма разработки – это набор правил и критериев, соблюдаемых разработчиками, чтобы выдержать конкретную стилистику и модель написания кода.

Единая парадигма помогает избегать ошибок, упрощает работу в команде и ускоряет разработку. Ориентируясь на одну парадигму, можно корректно структурировать код приложения, зная четкие правила, выбранные командой, которая работает над конкретным проектом.

Существуют различные типы парадигм, например, процедурный, ориентированный на работу с функциями, или логический, подразумевающий решение базовых логических задач в духе «если $A = \text{true}$, то и $B = \text{true}$ ». Но есть и более интересный подход к решению задач разработки, и это ООП.

ООП – это одна из парадигм разработки, подразумевающая организацию программного кода, ориентируясь на данные и объекты, а не на функции и логические структуры.

Обычно объекты в подобном коде представляют собой полноценные блоки с данными, которые имеют определенный набор характеристик и возможностей.

Объект может олицетворять что угодно – от человека с именем, фамилией, номером телефона, паролем и другой информацией до мелкой утилиты с минимумом характеристик из реального мира, но увеличенным набором функций. Объекты могут взаимодействовать друг с другом, пользователем и любыми другими компонентами программы.

ООП заставляет разработчиков фокусироваться на объектах, которыми нужно манипулировать, а не на той логике, что позволяет изменять данные и как-то с ними взаимодействовать. Такой подход хорошо работает в случае с комплексными программными решениями, требующими постоянной поддержки со стороны большого числа программистов.

Объектно-ориентированное программирование исповедует ряд принципов, лежащих в основе правил создания и использования всех структурных элементов, включая классы, объекты, методы и прочие компоненты.

Ниже разберем основные принципы ООП.

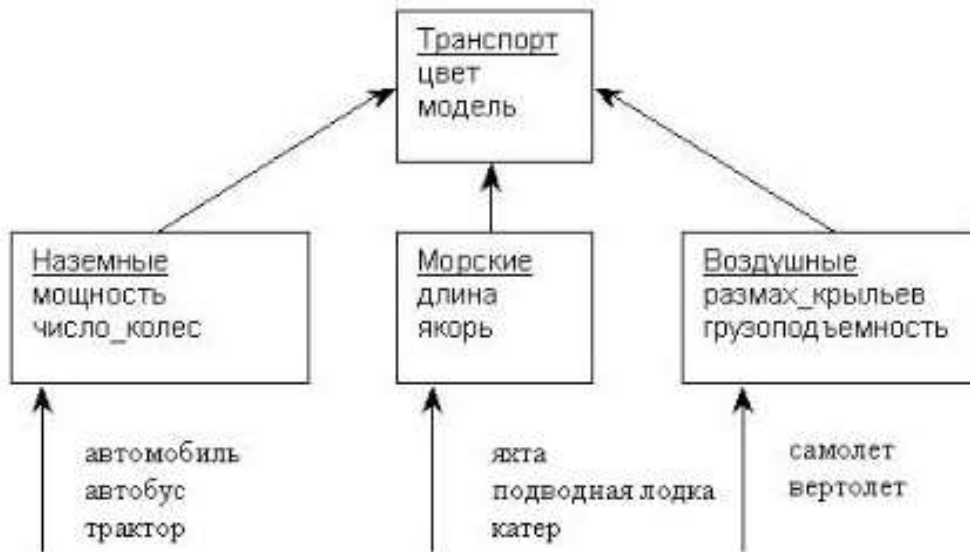
3.1 Наследование

Наследование - это отношение между классами, при котором класс использует структуру или поведение другого класса.

Наследование определяет взаимоотношений между классами и объектами.

Чтобы не создавать кучу одинаковых объектов или классов, можно создать класс над классами с более общими характеристиками и функциями, а потом постепенно наследовать от него те или иные возможности. Используя специальную конструкцию, программист может забрать из класса ряд атрибутов или методов, оставить их в прежнем виде и дополнить новыми или же слегка переосмыслить на свое усмотрение, а потом создать из них уникальный объект или подкласс для дальнейшего наследования опций.

Наследование вводит иерархию классов.



Пусть у нас есть следующий класс Person, который описывает отдельного человека:

```
class Person
{
    private string _name = "";

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public void Print()
    {
        Console.WriteLine(Name);
    }
}
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
class Employee : Person
{
}

```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса с параметрами.

Таким образом, наследование реализует отношение is-a (является), объект класса Employee также является объектом класса Person:

```
Person person = new Person { Name = "Tom" };
person.Print();    // Tom
person = new Employee { Name = "Sam" };
person.Print();    // Sam

```

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

По умолчанию все классы наследуются от базового класса Object, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

3.1.1 Особенности наследования

- У одного родительского класса может быть сколько угодно классов-наследников.
- В дочернем классе свойства при наследовании повторно не описываются.
- Объекты производных классов обычно являются более узкоспециализированными.
- При наследовании потомок сохраняет свойства и методы родительского класса.
- При наследовании потомок добавляет новые свойства и методы.
- При наследовании потомок может менять старые свойства и методы.

3.1.2 Ограничения при наследовании

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа `internal`, то производный класс может иметь тип доступа `internal` или `private`, но не `public`.
- Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производный класс может наследовать только от класса, который имеет модификатор `public`.
- Нельзя унаследовать класс от статического класса.
- Если класс объявлен с модификатором **`sealed`**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

3.1.3 Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам `Person` и `Employee`. Хотя `Employee` наследует весь функционал от класса `Person`, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void PrintName()
    {
        Console.WriteLine(_name);
    }
}
```

Этот код не сработает и выдаст ошибку, так как переменная `_name` объявлена с модификатором `private` и поэтому к ней доступ имеет только класс `Person`. Но зато в классе `Person` определено общедоступное свойство `Name`, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
class Employee : Person
```

```

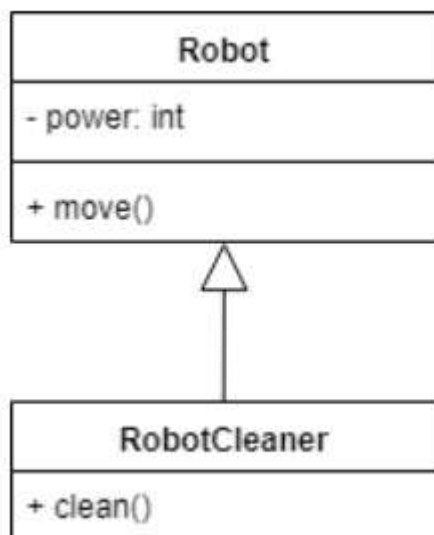
{
    public void PrintName()
    {
        Console.WriteLine(Name);
    }
}

```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private** **protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

3.1.4 Наследование в терминах UML

Для обозначения наследования в UML применяется взаимосвязь «обобщение». От дочернего класса идет сплошная стрелка к родительскому классу, что означает дочерний класс является наследником родительского класса.



3.1.5 Ключевое слово base

Добавим в классы **Person** и **Employee**, описанные выше, конструкторы:

```

class Person
{
    public string Name { get; set;}
    public Person(string name)
    {
        Name = name;
    }
    public void Print()
    {
        Console.WriteLine(Name);
    }
}

```

```

    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }
}

```

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения `base(name)`.

```

Person person = new Person("Bob");
person.Print();    // Bob
Employee employee = new Employee("Tom", "Microsoft");
employee.Print();  // Tom

```

3.1.6 Конструкторы при наследовании

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base**. То есть в классе Employee через ключевое слово `base` надо явным образом вызвать конструктор класса Person:

```

class Employee : Person
{
    public string Company { get; set; } = "";
    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }
}

```

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
class Person
{
    public string Name { get; set; }
    // конструктор без параметров
    public Person()
    {
        Name = "Tom";
        Console.WriteLine("Вызов конструктора без параметров");
    }
    public Person(string name)
    {
        Name = name;
    }
    public void Print()
    {
        Console.WriteLine(Name);
    }
}
```

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор:

```
public Employee(string company)
{
    Company = company;
}
```

Фактически был бы эквивалентен следующему конструктору:

```
public Employee(string company)
    :base()
{
    Company = company;
}
```

3.1.7 Слово **sealed** для классов

Можно запретить наследоваться от класса, добавив к его объявлению ключевое слово **sealed**:

```
public sealed class C
{
    // код
}
```

```
public class D : C
{
// ошибка компиляции, нельзя наследоваться от sealed класса
}
```

Запрет наследования используют из соображений безопасности – ведь наследники могут переопределять методы как хотят и обращаться к protected членам, а хочется запретить менять реализацию класса.

Запрет наследования улучшает производительность, так как sealed методы не могут быть переопределены, реализованы как не виртуальные, поэтому их вызов быстрее.

3.1.8 Когда не нужно использовать наследование

- Если мы просто хотим использовать некоторые методы класса-родителя, чтобы выполнить свою работу.
- При этом наш класс логически не сильно связан с классом-родителем, либо большая часть методов класса-родителя ему вообще не нужна
- Допустим, есть класс «Окно операционной системы». Он очень сложно устроен и много чего умеет, например, отрисовываться на экране, получать события от пользователя о нажатиях мыши и клавиатуры и т.д. И еще у него есть ширина и высота и методы для работы с ними.
- Допустим, мы хотим создать свой класс для геометрических фигур, и нам тоже надо уметь работать с шириной и высотой. Отнаследовавшись от окна, мы бы получили реализацию этих методов, но тем самым мы:
 - Получили много лишнего кода
 - Наш класс может использоваться везде, где нужны окна, а это не нужно.

3.2 Инкапсуляция

Инкапсуляция - это сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей.

Вся важная информация, необходимая для работы объекта, в нем же и хранится. И только определенные данные доступны для внешних функций и объектов. Данные конкретного класса хранятся в пределах этого класса.

Вносить в них изменения, используя другие классы, нельзя. У окружения есть право только запрашивать «публичные» методы и атрибуты. Такой подход обеспечивает повышенный уровень безопасности, а также сокращает шансы на случайное повреждение данных внутри какого-то класса со стороны.

Также под инкапсуляцией понимают ограничение области видимости для переменных и функций классов.

3.2.1 Основные модификаторы доступа

Все поля, методы и остальные компоненты класса имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для компонентов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В языке C# применяются следующие модификаторы доступа:

- **private**: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
- **private protected**: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
- **file**: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с таким модификатором доступны только из текущего файла кода.
- **protected**: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
- **internal**: компоненты класса или структуры доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
- **protected internal**: совмещает функционал двух модификаторов **protected** и **internal**. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.
- **public**: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

3.3 Абстракция

3.3.1 Абстрактные классы

Абстрактный класс – это класс, экземпляры которого нельзя создать. Базовый класс объявляют абстрактным, если не имеет смысла создавать его экземпляры, а имеет смысл создавать только экземпляры его наследников.

Абстрактный класс помечается словом **abstract**.

Абстрактный класс может иметь абстрактные методы – методы без реализации. Их надо пометить словом `abstract`. Конструктор часто делают `protected` - все равно создать экземпляры нельзя. Абстрактный класс может иметь обычные поля и методы.

Пусть класс `Shape` – абстрактный:

```
public abstract class Shape
{
    private Color color;
    public abstract double GetWidth();
    public abstract double GetHeight();
    public abstract double GetArea();

    protected Shape(Color color)
    {
        this.color = color;
    }

    protected Color GetColor()
    {
        return color;
    }
}
```

Экземпляр такого абстрактного класса создавать нельзя:

```
Shape shape = new Shape(); // будет ошибка компиляции
                          //нельзя создавать экземпляр абстрактного класса
```

Если в классе есть хотя бы один абстрактный метод или от родителей достался нереализованный абстрактный метод, то класс обязан быть абстрактным.

```
public class TestClass
{
    // ошибка компиляции – класс должен быть помечен как abstract
    public abstract void TestMethod();
}
```

Класс можно делать абстрактным, даже если в нем нет собственных или унаследованных от родителей абстрактных методов.

```
public abstract class TestClass
{
    public void TestMethod()
    {
```

```
        Console.WriteLine("TestMethod");
    }
}
```

Если нет возможности или мы не хотим реализовывать абстрактный член, доставшийся от абстрактного базового класса, то данный класс тоже помечается как абстрактный.

```
abstract class Person
{
    public abstract string Name { get; set; }
}

abstract class Manager : Person {
    // абстрактный класс не обязан иметь реализацию абстрактных членов,
    // доставшихся от базового класса
}
```

Абстрактные классы нужны чтобы создавать базовые классы, которые реализуют некоторую общую логику, но при этом некоторые аспекты на данном уровне абстракции еще не известны.

3.3.2 Абстрактные свойства

Абстрактные свойства помечаются словом `abstract`.

Ниже приведен пример абстрактного свойства `Name` в классе `Person`:

```
abstract class Person
{
    public abstract string Name { get; set; }
}
```

В классе наследнике необходимо переопределить это свойство:

```
class Client : Person
{
    private string name;

    public override string Name
    {
        get
        {
            return "Mr/Ms. " + name;
        }
        set
        {
            name = value;
        }
    }
}
```

3.3.3 Когда следует делать класс абстрактным

- Когда нет смысла создавать экземпляры этого класса.
- На текущем уровне абстракции непонятно, как реализовать какие-то методы, и нет какой-то разумной реализации по умолчанию.

```
public abstract class Shape
{
    public abstract double GetWidth();

    public abstract double GetHeight();

    public abstract double GetArea();
}
```

Тут непонятно, каковы размеры и площадь фигуры, потому что мы не знаем её тип, положение.

3.3.4 Когда не следует делать класс абстрактным

- Объекты класса уже можно создавать и использовать

```
Shape s = new Square(Color.RED, 1);
Console.WriteLine(s.GetArea())
```

- Мы реализовали все абстрактные методы всех родителей, поэтому класс можно делать не абстрактным.
- Мы понимаем, как реализовать определенный метод.

```
public class Square : Shape
{
    private double sideLength;

    public Square(Color color, double sideLength) : base(color)
    {
        this.sideLength = sideLength;
    }

    public override double GetWidth()
    {
        return sideLength;
    }

    public override double GetHeight()
```

```

    {
        return sideLength;
    }

    public override double GetArea()
    {
        return sideLength * sideLength;
    }
}

```

Для квадрата мы уже понимаем, как посчитать размеры и площадь.

3.3.5 Интерфейсы в терминах ООП

Интерфейс обычно подобен абстрактному базовому классу, имеющему только абстрактные члены. В С# интерфейсы обозначаются при помощи ключевого слова **interface**. Для интерфейсов принято соглашение именования – начинать их с буквы I (от Interface)

```

public interface IShape
{
    double GetWidth();
    double GetHeight();
    double GetArea();
}

```

Интерфейсы в С# могут содержать:

- Методы
- Свойства
- События
- Индексаторы
- Константы
- Операторы
- Статический конструктор
- Вложенные типы
- Статические поля, методы, свойства, индексаторы и события

```

interface IMovable
{
    void Move();    // Метод
    string Name { get; set; }    // Свойство
    event MoveHandler MoveEvent;    // Событие
}

```

```
string this[int index] { get; set; } // Индексаторы
}
```

Интерфейс может иметь только **public** члены.

Модификаторы видимости указывать нельзя, они всегда подразумеваются **public**. Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

```
public interface IShape
{
    private double GetWidth();
    private double GetHeight();
    protected double GetArea();
}
```

3.3.6 Реализация интерфейса

Так как интерфейсы в терминах ООП – абстрактные классы, то нельзя создавать их экземпляры.

От интерфейса можно наследоваться, как от обычного класса, указав двоеточие:

```
public class Square : IShape
{
    // реализация методов GetHeight, GetArea, GetWidth
    public void GetWidth()
    {
        // реализация метода GetWidth
    }
}
```

Вместо слова «наследуют», про интерфейсы говорят что их «реализуют». То есть класс Square реализует интерфейс IShape. Implement с англ. – реализовывать. Если класс реализует интерфейс, то он должен реализовывать все его методы, либо быть абстрактным.

3.3.7 Реализация интерфейсов и наследование

Класс может реализовывать несколько интерфейсов, в этом отличие от классов.

```
public interface ITest1
{
    void TestMethod1();
}

public interface ITest2
{
    void TestMethod2();
}
```

```

}

public class TestClass : ITest1, ITest2
{
    public void TestMethod1()
    {
    }

    public void TestMethod2()
    {
    }
}

```

Можно одновременно наследоваться от некоторого класса и реализовывать сколько угодно интерфейсов. В этом случае класс должен идти первым после «:»

```

public abstract class Animal
{
    public abstract void Move();
}

public interface IEating
{
    void Eat();
}

public interface ISpeaking {
    void Speak();
}

public class Cat : Animal, IEating, ISpeaking
{
    public override void Move()
    {
    }

    public void Eat()
    {
    }

    public void Speak()
    {
    }
}

```

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию.

```

interface IMovable
{
    void Move()
    {
        Console.WriteLine("Walking");
    }

    // реализация свойства по умолчанию только для чтения
    int MaxSpeed
    {
        get
        {
            return 0;
        }
    }
}

```

Интерфейсы полезны в следующих случаях:

- Чтобы указать для класса признак, который не вписывается в иерархию классов



- Когда хочется выделить некоторую абстракцию, но непонятно, как она будет реализована. Или реализации могут быть абсолютно несхожими между собой

```

interface ILogger
{

```



```
// сообщает о предупреждении
void Warning(string text);

// сообщает об ошибке
void Error(string text);

// информационное сообщение
void Info(string text);
}
```

3.3.8 Применение интерфейсов в классах и структурах

```
interface IMovable
{
    void Move();
}

class Person : IMovable
{
    public void Move()
    {
        Console.WriteLine("Человек идет");
    }
}

struct Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

3.4 Полиморфизм

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

3.4.1 Виды полиморфизма

- **Ad-hoc - полиморфизм**

Позволяет определять поведение в зависимости от входных аргументов.

К Ad-hoc – полиморфизму относят:

- Перегрузка методов
- Методы с произвольным количеством аргументов

- Необязательные аргументы

- **Параметрический полиморфизм**

Позволяет создавать универсальные базовые типы и методы для них.

Повышает повторное использования кода.

К параметрическому полиморфизму относят:

- Обобщения (универсальные шаблоны)

- **Полиморфизм подтипов**

Упрощает разработку методов для разных типов данных.

К полиморфизму подтипов относят:

- Переопределение методов

3.4.2 Перегрузка методов

В одном классе можно создавать методы с одинаковыми именами, но разной сигнатурой:

```
public class Summator
{
    public double Sum(int a, int b)
    {
        return a + b;
    }

    public int Sum(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

3.4.3 Сигнатура метода для перегрузки

В сигнатуру входят: название метода, количество, тип, порядок и модификаторы аргументов.

В классе нельзя определить два метода с одинаковой сигнатурой.

В сигнатуру не входит возвращаемый тип!

```
abstract public double Sum(int a, int b);

abstract public double Sum(double a, double b);

abstract public int Sum(int a, int b, int c);

abstract public double Sum(int a, int b); // Конфликт именования методов

abstract public int Sum(int a, int b); // Конфликт именования методов
```

3.4.4 Методы с произвольным количеством аргументов

```
public static double Average(params double[] numbers)
{
    double total = 0.0;
    foreach (double d in numbers)
    {
        total += d;
    }
    return total / numbers.Length;
}
```

```
Average(5); // выведется 5
Average(2, 4); // выведется 3
```

Использовать **params** в списке параметров метода можно только один раз. При этом такой параметр обязательно должен быть **последним** в списке аргументов

```
public static double Example1(int a, params double[] numbers)
{
    // ОК
}

public static double Example2(params double[] numbers, int a)
{
    // Ошибка компиляции, params double[] numbers
    // должен быть последним параметром
}
```

3.4.5 Необязательные аргументы

Сигнатура метода может указывать, являются ли его параметры обязательными или нет

```
// Метод создания квадрата с заданным цветом
public static Square CreateSquare(int size, int color)
{
    return new Square(size, color);
}

// Метод создания красного квадрата
public static Square CreateRedSquare(int size)
{
    return new Square(size, Color.Red);
}

// Универсальный метод
public static Square CreateSquare(int size, int color = Color.Red)
{
}
```

```
    return new Square(size, color);
}

// Пример вызова метода с необязательными аргументами
var redSquare = CreateSquare(5);
var square = CreateSquare(5, Color.BLUE);
```

Определение каждого необязательного параметра содержит его значение по умолчанию.

Необязательные параметры (их может быть несколько) определяются в конце списка параметров после всех обязательных параметров.

3.4.6 Обобщения (универсальные шаблоны)

Иногда нужно производить одинаковые операции над данными разного типа.

Пример: идентификатор банковского счета может быть любого типа (число, строка и др.)

```
class Account
{
    public int Id { get; set; }
    public int Sum { get; set; }
}
```

Решение 1:

Объявить идентификатор как object:

```
class Account
{
    public object Id { get; set; }
    public int Sum { get; set; }
}

Account acc1 = new Account { Id = 1 };
Account acc2 = new Account { Id = "705b58a" };
int id1 = (int)acc1.Id;
string id2 = (string)acc2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

В данном случае необходимо конвертировать тип идентификатора при каждом обращении, возникает путаница и есть вероятность ошибки.

Решение 2:

Использовать обобщения:

```

class Account<T>
{
    public T Id { get; set; }
    public int Sum { get; set; }
}

Account<int> acc1 = new Account<int> { Id = 1 };
Account<string> acc2 = new Account<string> { Id = "705b58a" };
int id1 = acc1.Id;
string id2 = acc2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);

```

Тип идентификатора задается переменной (в данном случае T).
 При инициализации объекта переменная T инициализируется типом (в данном случае int и string), конвертировать типы не нужно.

3.4.7 Использование нескольких универсальных параметров

```

class Transaction<U, V>
{
    public U FromAccount { get; set; } // с какого счета перевод
    public U ToAccount { get; set; } // на какой счет перевод
    public V Code { get; set; } // код операции
    public int Sum { get; set; } // сумма перевода
}

```

3.4.8 Наследование обобщенных типов

```

// Обобщенный базовый класс
abstract class Account<T>
{
    public T Id { get; private set; }
    protected Account(T id)
    {
        Id = id;
    }
}

// Необобщенный класс-наследник
class StringAccount : Account<string>
{
    public StringAccount(string id) : base(id) { }
}

// Класс-наследник, который типизирован тем же типом, что и базовый
class UniversalAccount<T> : Account<T>
{

```

```
public UniversalAccount(T id) : base(id) { }  
}
```

3.4.9 Ограничения обобщений

```
class NumbersCollection<T> where T : ICollection<int>, new()  
{  
    public T Numbers { get; set; } = new T();  
    // добавление чисел в коллекцию  
    public void Add(int i)  
    {  
        Numbers.Add(i);  
    }  
    // вывод чисел  
    public void Print()  
    {  
        foreach (int v in Numbers)  
        {  
            Console.WriteLine(v);  
        }  
    }  
}
```

С помощью выражения `where T : ICollection` мы указываем, что используемый тип `T` обязательно должен реализовывать интерфейс `ICollection` и соответственно обращаться к `Numbers` как к коллекции.

Стандартное ограничение `new()` позволяет создавать новые экземпляры универсального типа `T` с помощью общедоступного (`public`) конструктора без параметров.

```
// создание коллекции на основе списка  
var numbersListCollection = new NumbersCollection<List<int>>();  
  
numbersListCollection.Add(5);  
numbersListCollection.Add(1);  
numbersListCollection.Add(2);  
numbersListCollection.Print(); // выведется 5 1 2  
  
// создание коллекции на основе отсортированного множества  
var numbersSortedSetCollection = new  
NumbersCollection<SortedSet<int>>();  
  
numbersSortedSetCollection.Add(5);  
numbersSortedSetCollection.Add(1);  
numbersSortedSetCollection.Add(2);  
numbersSortedSetCollection.Print(); // выведется 1 2 3
```

3.4.10 Обобщенные методы

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}

int a = 1, b = 2;
Swap<int>(ref a, ref b); // или Swap(ref a, ref b);
string s1 = "hello", s2 = "world";
Swap<string>(ref s1, ref s2); // или Swap(ref s1, ref s2);
```

3.4.11 Обобщения в стандартных библиотеках C#

Обобщенные коллекции (System.Collections.Generic):

- List
- Dictionary
- LinkedList
- Queue
- и др.

Пример:

```
class Product
{
    public Product(string title, double price, string description)
    {
        Title = title;
        Price = price;
        Description = description;
    }
    public string Title { get; private set; }
    public double Price { get; private set; }
    public string Description { get; private set; }
}

Dictionary<long, Product> products = new Dictionary<long, Product>();
// добавление элементов
products.Add(0, new Product("Хлеб", 30, "Бородинский"));
products.Add(1, new Product("Молоко", 50, "Простоквашино"));
products.Add(2, new Product("Стиральный порошок", 30, "Для цветной стирки"));
// проверка на наличие элемента по заданному ключу
if (products.ContainsKey(0))
```

```
{  
    // обращение к элементу по ключу  
    Console.WriteLine(products[0].Title);  
    // удаление элемента по ключу  
    products.Remove(0);  
}
```

3.4.12 Переопределение методов

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечаются модификатором **virtual**. Такие методы и свойства называют **виртуальными**.

А чтобы переопределить метод в классенаследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

```
public class A  
{  
    public virtual void F()  
    {  
        Console.WriteLine(1);  
    }  
}  
  
public class B : A  
{  
    public override void F()  
    {  
        Console.WriteLine(2);  
    }  
}  
  
A a = new A();  
a.F(); // 1  
B b = new B();  
b.F(); // 2  
A c = new B();  
c.F(); // 2
```

Для виртуальной функции вызывается та реализация, которая определена для фактического типа объекта, а не для типа ссылки.

Пример – геометрические фигуры


```
public class Shape
{
    public virtual double GetArea()
    {
        return 0; // нет разумной реализации
    }
}

public class Rectangle : Shape
{
    // опущен код полей и конструктора
    public override double GetArea()
    {
        return width * height;
    }
}

// И тогда эти примеры будут работать правильно
Shape s1 = new Rectangle(10, 2);
Console.WriteLine(s1.GetArea()); // 20 (вызывается реализация для
                                   // Прямоугольника)
```