Объектно-ориентированное программирование

Лекции по курсу

Языки программирования

семестр 2

Объектно-ориентированное программирование

Ввиду увеличения сложности задач в программировании, акцент переместился от *алгоритмов* к ***объектам***, содержащим алгоритмы в качестве методов.

В больших системах нет главного алгоритма или он тривиален, но есть *большое число взаимосвязанных задач*.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов (Википедия)

Для программирования работы сложной системы мы:

- выявляем классы объектов, присутствующих в этой системе
- выявляем их свойства и методы
- выявляем ***то общее, что есть в различных классах***
- выявляем различные зависимости между классами и взаимодействие между объектами этих классов

Зависимости между классами. Наследование

Примеры зависимостей между классами

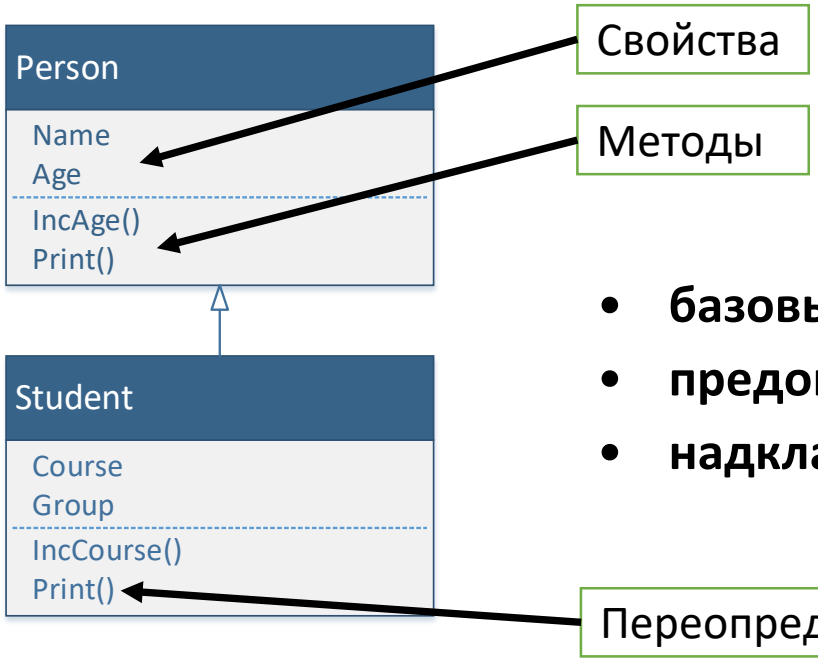
- класс содержит в качестве поля объект другого класса
- в методе класса параметром является объект другого класса
- метод класса вызывает статический метод другого класса
- один из классов является разновидностью другого

Определение. *Наследованием* называется такая зависимость между классами, при которой один из классов является разновидностью другого.

Наследование

При наследовании все члены базового класса (поля, методы, свойства) *наследуются* производным классом. Производный класс может добавлять новые члены (поля, методы, свойства) и переопределять (*замещать*) методы базового класса.

UML-диаграмма классов Person-Student



UML – Unified Modeling Language

- **базовый класс — производный класс**
- **предок — потомок**
- **надкласс — подкласс**

Наследование. Класс Person

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; private set; }
    public Person(string Name, int Age)
    {
        this.Name = Name;
        this.Age = Age;
    }
    public void IncAge()
    {
        Age += 1;
    }
    public void Print()
    {
        WriteLine(Name + " " + Age);
    }
}
```

Наследование. Класс Student

```
public class Student: Person
{
    public int Course { get; private set; }
    public int Group { get; set; }
    public Student(string Name, int Age, int Course, int Group):
      base(Name,Age)
    {
        this.Course = Course;
        this.Group = Group;
    }
    public void IncCourse()
    {
        Course += 1;
    }
    public new void Print()
    {
        base.Print();
        WriteLine(Course + " " + Group);
    }
}
```

Класс-предок

Вызов конструктора предка

new говорит о том, что новый метод Print() перекрывает действие старого

Вызов переопределённой функции предка

base – имя, использующееся для ссылки на предка

Наследование. Основная программа

```
static void Main(string[] args)
{
    var p = new Person("Ivanov", 19);
    p.Print();
    p.IncAge();
    p.Print();
    var s = new Student("Petrov", 18, 1, 9);
    s.Print();
    s.IncAge();
    s.IncCourse();
    s.Print();
}
```

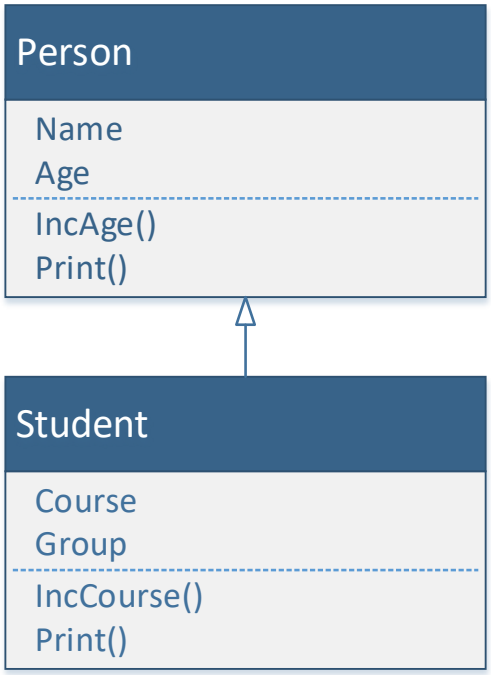
Вызов унаследованной функции

Вызов переопределяющей функции

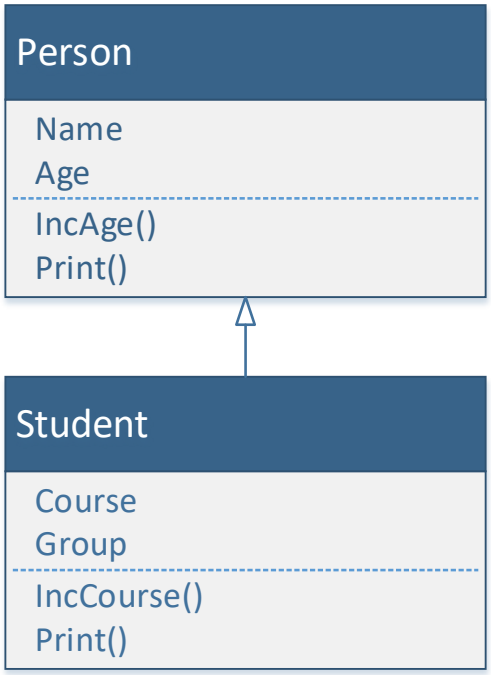
Цели наследования

- **Повторное использование кода**
  (Student может использовать все свойства и методы базового класса Person)
- **Создание единого интерфейса** для группы родственных классов (Person определяет единый интерфейс для всех наследников: IncAge(), Print())
- **Обеспечение вариабельности и изменчивости** кода: производные классы ***добавляют*** к базовому классу новую ***функциональность*** (Student.IncCourse()) и переопределяют функциональность базового класса (Student.Print()). Множество производных классов – каждый со своей добавленной функциональностью

Наследование – это расширение или сужение?

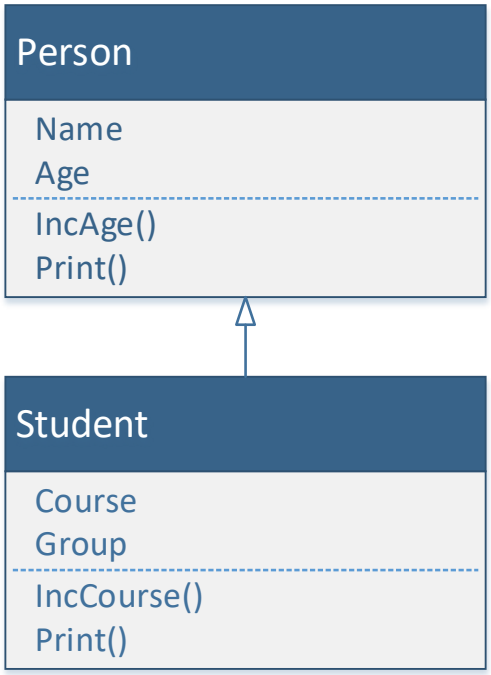


Наследование – это расширение или сужение?



- Наследование – это **расширение** интерфейса (добавляются новые методы и свойства)

Наследование – это расширение или сужение?



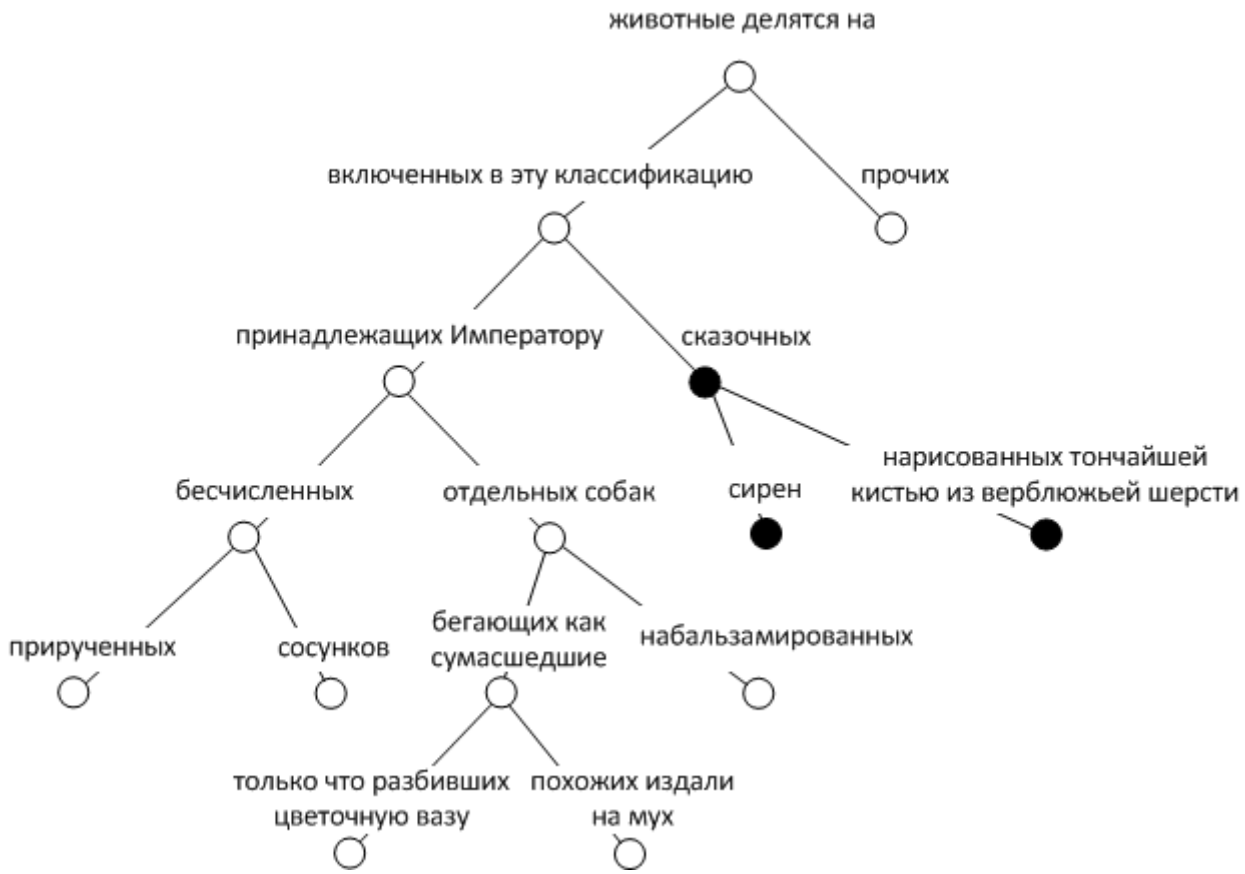
- Наследование – это **расширение** интерфейса (добавляются новые методы и свойства)
- Наследование – это **сужение** количества представителей (студентов меньше, чем персон)

Двойственность: увеличение одного показателя приводит к уменьшению другого

Иерархия наследования: примеры

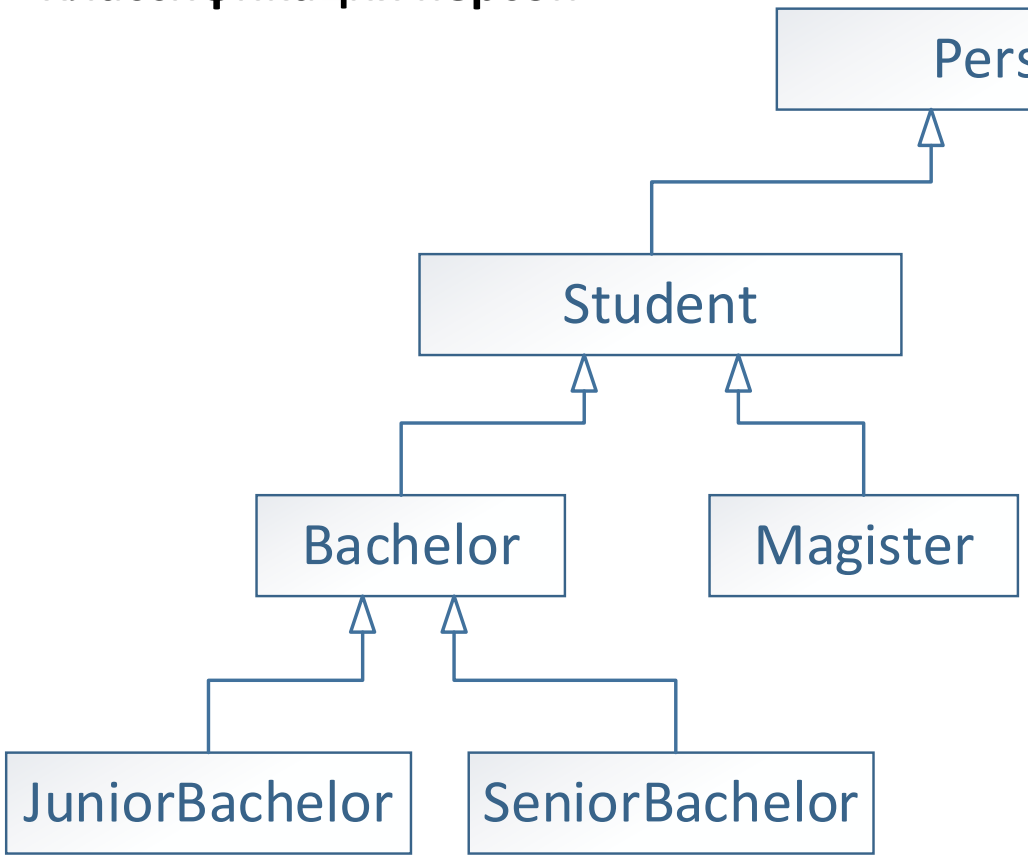
Иерархия наследования: примеры

Классификация животных (Умберто Эко "Поиски совершенного языка")



Иерархия наследования: примеры

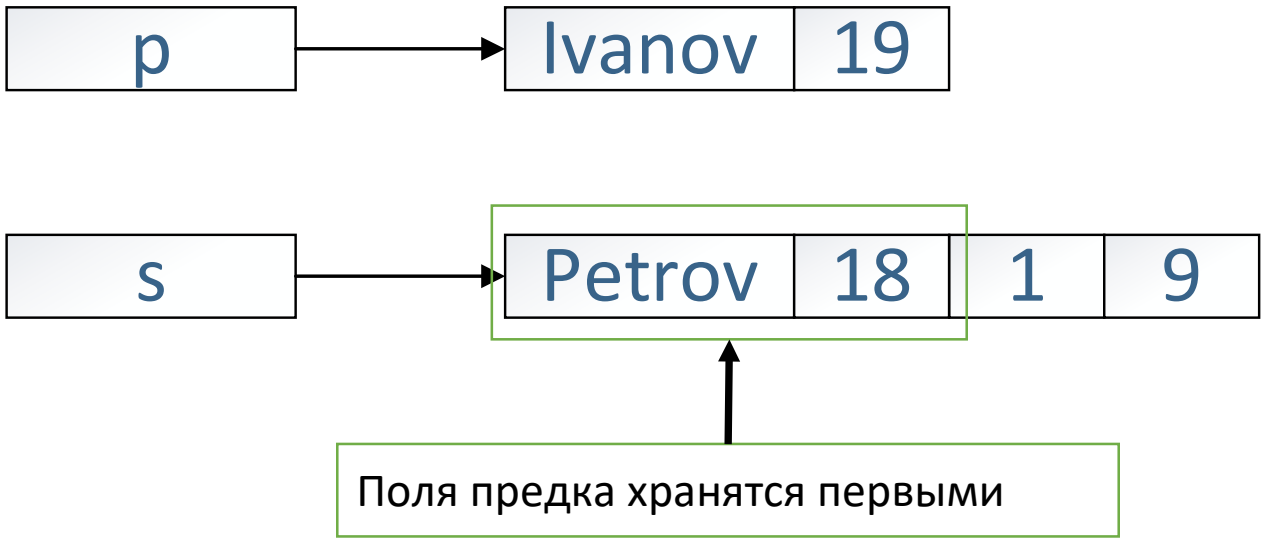
Классификация персон



Множество производных классов – каждый со своей добавленной функциональностью

Хранение в памяти

```
var p = new Person("Ivanov", 19);
var s = new Student("Petrov", 18, 1, 9);
```



Принцип "открыт-закрыт": прелюдия

Как вводить модификации в код правильной программы?

Если программа доступна в исходниках и требуется изменить её функциональность, то допустимо ли менять её код?

Есть ли риск при этом превратить работающую программу в неработающую? Если мы меняем библиотеку, то есть ли риск, что весь зависимый код перестанет работать?

Принцип "открыт-закрыт": анекдот

Как вводить модификации в код правильной программы?

Если программа доступна в исходниках и требуется изменить её функциональность, то допустимо ли менять её код?

Есть ли риск при этом превратить работающую программу в неработающую? Если мы меняем библиотеку, то есть ли риск, что весь зависимый код перестанет работать?

Анекдот про программиста

Приходит сын программиста к папе

- Папа, почему Солнце всходит на Востоке, а заходит на Западе?
- Ты это проверял?
- Да.
- И давно так?
- Ну, я не знаю... Всегда...
- Не тормозит?
- Нет.
- Тогда сынок ничего лучше не трогай, пусть работает!

Принцип "открыт-закрыт": формулировка

Код программы должен быть:

закрыт для изменения текста и

открыт для модификации поведения и функциональности.

Как такое возможно?

Чтобы обеспечить этот принцип, используют ***наследование***:

от базового класса, в котором не хватает функциональности, порождают новый, добавляя и изменяя его функциональность.

Код базового класса при этом *не меняется*.

Принцип "открыт-закрыт": пример очереди с фильтрацией

```
public class QueueWithFilter<T>: Queue<T>
{
    public Predicate<T> Filter { get; }
    public QueueWithFilter(Predicate<T> Filter)
    {
        this.Filter = Filter;
    }
    public new void Enqueue(T x)
    {
        if (Filter(x))
            base.Enqueue(x);
    }
}
```

Вызывается конструктор предка по умолчанию – без параметров : base()

действие функции Enqueue() переопределяется. Все остальные методы наследуются без изменений

Весь код, в котором используется базовый класс Queue<T>, **будет продолжать работать без перекомпиляции**

Принцип "открыт-закрыт": использование очереди с фильтрацией

```
static void Main(string[] args)
{
    var q = new QueueWithFilter<int>(x => x % 2 == 0);
    for (int i=0; i<10; i++)
        q.Enqueue(i);
    while (q.Count > 0)
        Write(q.Dequeue() + " ");
}
```

Функция Dequeue() и свойство Count унаследованы из класса Queue<T> без изменений

Реализация очереди с фильтрацией с помощью включения

```
public class QueueWithFilter2<T>
{
    private Queue<T> q;
    public Predicate<T> Filter { get; }

    public QueueWithFilter2(Predicate<T> Filter)
    {
        this.Filter = Filter;
        q = new Queue<T>();
    }
    public void Enqueue(T x)
    {
        if (Filter(x))
            q.Enqueue(x);
    }
    public T Dequeue() => q.Dequeue();
    public T Peek() => q.Peek();
    public int Count { get => q.Count; }
}
```

Недостаток – приходится переопределять все методы и свойства, а не только те, которые изменились

Наследование или включение

Когда выбирать наследование, а когда — включение?

Есть два класса: A и B.
Зададим вопросы:

B **is** A?

B является разновидностью A?
Если ответ «да», то это — **наследование**.

Пример
Какаду **is** животное

A **has** B?

A состоит из B?
Если ответ «да», то это — **включение**.

Пример
Автомобиль **has** двигатель

A **use** B?

A использует объект B для реализации?
Если ответ «да», то это — **включение**.

Пример
Множество **use** БДП
(для реализации)

Спорные ситуации

Ellipse **?** Circle

Спорные ситуации

Ellipse ? Circle

Утверждение 1. Circle is Ellipse

(окружность – это эллипс, у которого радиусы равны)

Спорные ситуации

Ellipse ? Circle

Утверждение 1. Circle is Ellipse

(окружность – это эллипс, у которого радиусы равны)

Утверждение 2. Ellipse is Circle

(эллипс – это окружность, у которой добавлен второй радиус)

Спорные ситуации

Ellipse ? Circle

Утверждение 1. Circle is Ellipse

(окружность – это эллипс, у которого радиусы равны)

Утверждение 2. Ellipse is Circle

(эллипс – это окружность, у которой добавлен второй радиус)

Matrix ? Vector

Спорные ситуации

Ellipse ? Circle

Утверждение 1. Circle is Ellipse

(окружность – это эллипс, у которого радиусы равны)

Утверждение 2. Ellipse is Circle

(эллипс – это окружность, у которой добавлен второй радиус)

Matrix ? Vector

Утверждение 1. Matrix является разновидностью Vector

(матрица – это вектор, состоящий из векторов)

Спорные ситуации

Ellipse ? Circle

Утверждение 1. Circle is Ellipse
(окружность – это эллипс, у которого радиусы равны)
Утверждение 2. Ellipse is Circle
(эллипс – это окружность, у которой добавлен второй радиус)

Matrix ? Vector

Утверждение 1. Matrix является разновидностью Vector
(матрица – это вектор, состоящий из векторов)
Утверждение 2. Vector является разновидностью Matrix
(вектор-строка – это матрица с одной строкой)
(вектор-столбец – это матрица с одним столбцом)

Спорные ситуации

Ellipse ? Circle

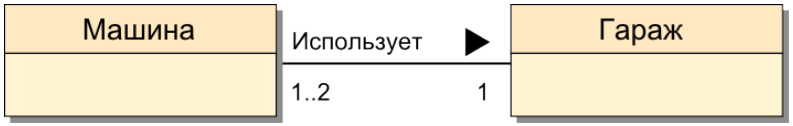
Утверждение 1. Circle is Ellipse
(окружность – это эллипс, у которого радиусы равны)
Утверждение 2. Ellipse is Circle
(эллипс – это окружность, у которой добавлен второй радиус)

Matrix ? Vector

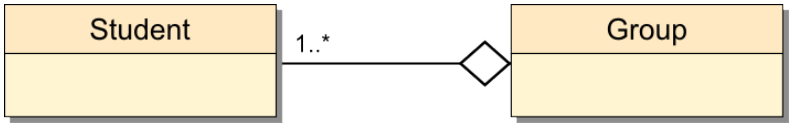
Утверждение 1. Matrix является разновидностью Vector
(матрица – это вектор, состоящий из векторов)
Утверждение 2. Vector является разновидностью Matrix
(вектор-строка – это матрица с одной строкой)
(вектор-столбец – это матрица с одним столбцом)
Утверждение 3. Matrix состоит из Vector
(матрица реализуется как массив векторов)

Виды отношений между классами

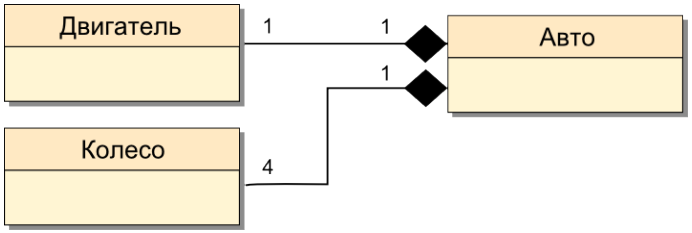
Ассоциация (связь)



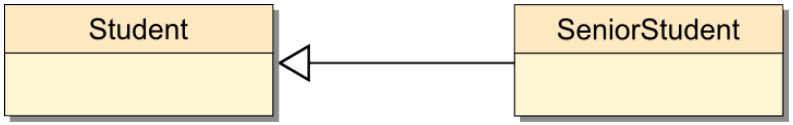
Агрегация (слабая форма **has** — состоит логически)



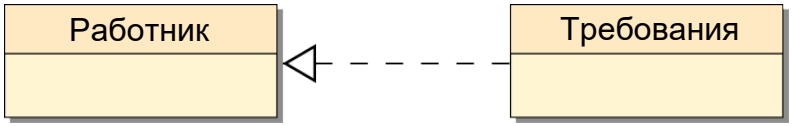
Композиция (сильная форма **has** — состоит физически)



Наследование (обобщение – **is**)

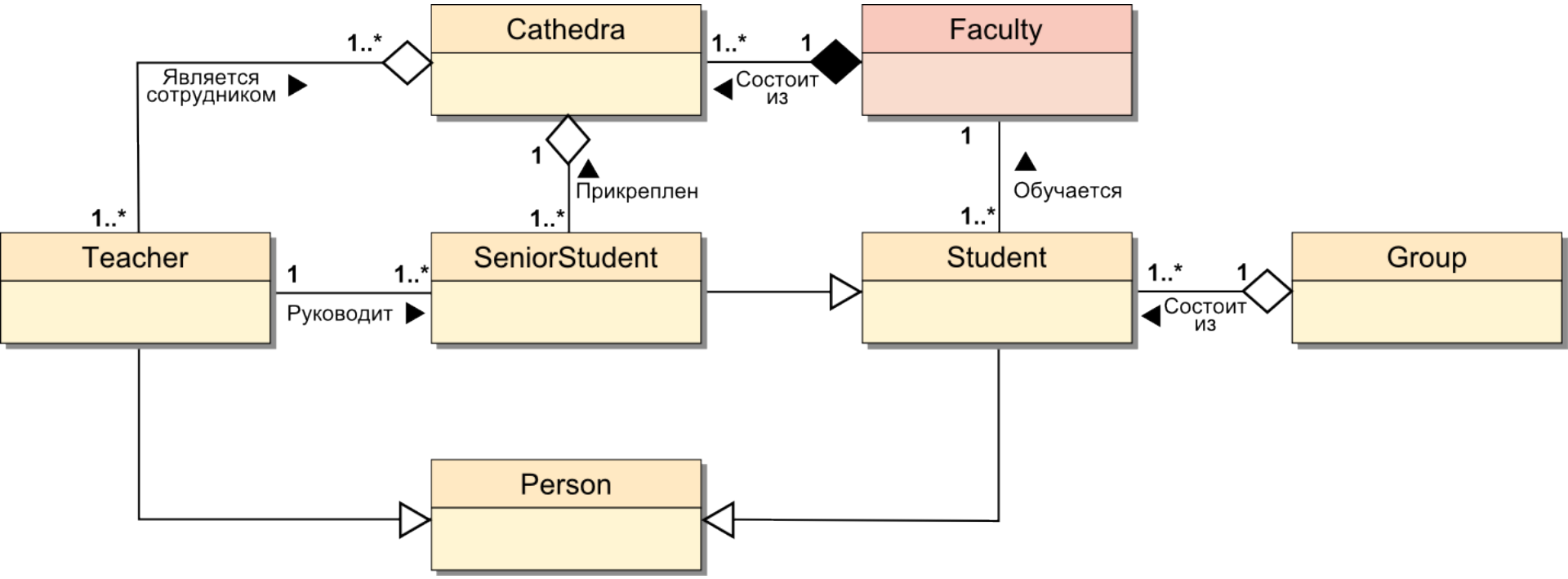


Реализация интерфейсов



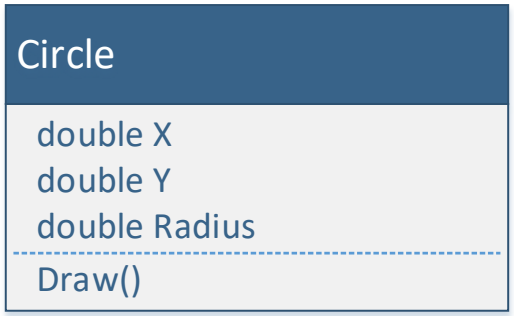
Графическая нотация UML диаграмм классов

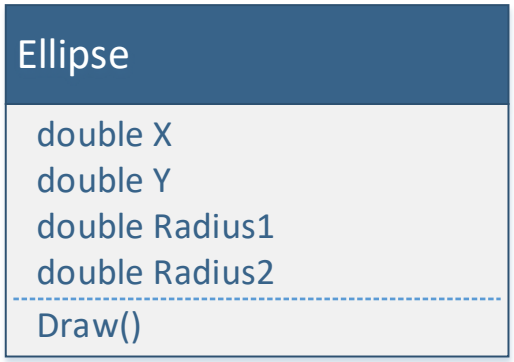
Пример UML-диаграммы классов



Наследование и выявление общего предка

Имеется два класса: Circle и Ellipse. Какой из них сделать наследником другого?



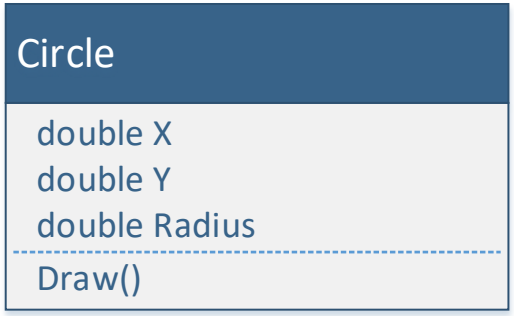


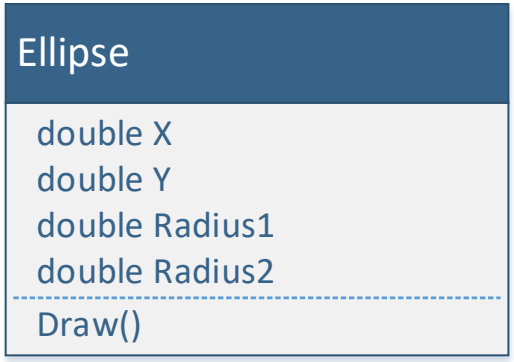
Наследование и выявление общего предка

Имеется два класса: Circle и Ellipse. Какой из них сделать наследником другого?

Вопрос – некорректный!

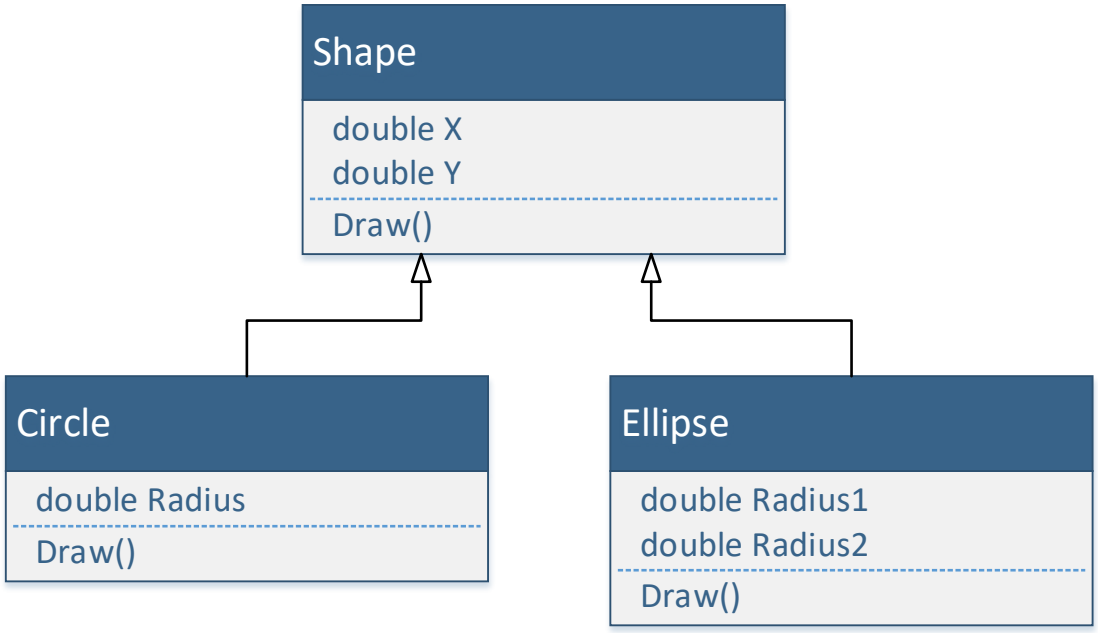
Необходимо выявить у них общее и унаследовать их от единого предка.





Наследование и выявление общего предка

Общий предок – Shape. Он содержит координаты и метод рисования Draw()



Модификатор доступа protected

```
class Base
{
    private int field1;
    protected int field2;
    private void Method1() { }
    protected void Method2() { }
    public void Method3() { }
}
class Derived: Base
{
    public void Method3()
    {
        field1 = 666; // field1 недоступен из-за его уровня защиты
        field2 = 777; // OK
        Method1(); // Method1 недоступен из-за его уровня защиты
        Method2(); // OK
    }
}
class Program
{
    static void Main(string[] args)
    {
        var b = new Base();
        b.Method1(); // Method1 недоступен из-за его уровня защиты
        b.Method2(); // Method2 недоступен из-за его уровня защиты
        b.Method3(); // OK
    }
}
```

protected – доступен в этом классе и его потомках, недоступен во внешнем коде

Принцип подстановки Барбары Лисков

Формулировка принципа подстановки. Во всех ситуациях, где используется объект базового класса, можно использовать объект производного класса без потери функциональности.

Правило. Переменной базового класса можно присваивать объект производного класса. Переменной производного класса ***нельзя*** присвоить объект базового класса.

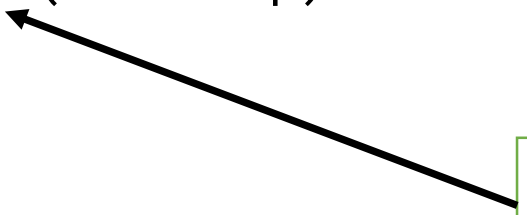
```
Person p = new Student("Петров", 18, 1, 9); // OK

Student s = new Person("Иванов", 19); // Ошибка!
```

Через переменную p можно вызывать только методы и свойства класса Person

Принцип подстановки Барбары Лисков

Следствие. В функцию, использующую в качестве параметра базовый тип, можно передавать объект производного типа

```
public void Some(Person p)
{
    p.Print();
    p.IncAge();
    p.Print();
}
...
Some(new Student("Петров", 18, 1, 9));
```



Функция Some ничего не знает о типе Student !

Иерархия наследования исключений .NET

Exception (базовый класс исключения)
  Свойства:
  e.Message - текстовое описание ошибки,
  e.StackTrace - последовательность вызовов функций, которая привела к исключению
    ApplicationException
        Все пользовательские исключения
    SystemException
        AccessViolationException (несанкционированный доступ к памяти)
        ArgumentException (один из передаваемых методу аргументов является недопустимым)
           ArgumentNullException
           ArgumentOutOfRangeException
        ArithmeticException
           DivideByZeroException (целочисленное деление на 0)
        IndexOutOfRangeException
        InvalidCastException (явное приведение к неправильному типу)
        FormatException (данные не соответствуют формату или форматная строка неправильная)
        NullReferenceException (попытка воспользоваться методом, полем или свойством через NULL-ссылку)
        OutOfMemoryException
        StackOverflowException
        NotImplementedException (метод не реализован)
        NotSupportedException (метод не поддерживается)
        System.Collections.Generic.KeyNotFoundException
        System.IO.IOException (пространство имен System.IO)
           System.IO.FileNotFoundException
           System.IO.EndOfStreamException (попытка чтения за концом потока)

Обработка исключений разных типов

```
static int MyMod(int a, int b)
{
    return a - a % b * b;
}
static void Main(string[] args)
{
    try {
        var a = int.Parse(Console.ReadLine());
        var b = int.Parse(Console.ReadLine());
        var r = MyMod(a, b) % (b - 1);
    }
    catch (FormatException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
    }
}
```

Обработка неверного ввода числа

Обработка ситуации b=0 или b=1

Правило. Обычно наследники SystemException не обрабатываются (за небольшим исключением: например, System.IO.FileNotFoundException)

Создание собственных типов исключений

```
class MyModException: ArithmeticException { }
static int MyMod(int a, int b)
{
    if (b == 0)
        throw new MyModException();
    return a - a % b * b;
}
static void Main(string[] args)
{
    try {
        var a = int.Parse(Console.ReadLine());
        var b = int.Parse(Console.ReadLine());
        var r = MyMod(a, b) % (b - 1);
    }
    catch (MyModException e)
    {
        Console.WriteLine("Арифметическое исключение");
    }
    catch (ArithmeticException e)
    {
        Console.WriteLine("Арифметическое исключение");
    }
}
```

Более специфические исключения обрабатываются первыми! Менять местами обработчики нельзя!

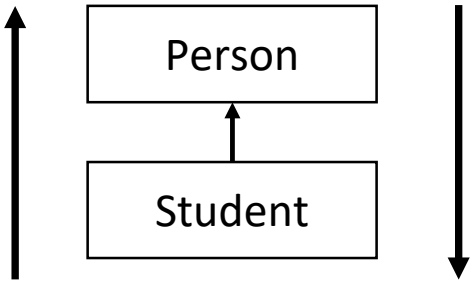
Обработка всех исключений

```
static void Main(string[] args)
{
    try
    {
        ...
    }
    catch (Exception e)
    {
        // Мне всё сойдёт с рук – гашу все исключения!
    }
}
```

Обычно обработка всех исключений Exception – признак слабости!
Лучше чтобы исключение неожиданного типа осталось необработанным, тогда мы будем знать о специфической проблеме

UpCast и DownCast

UpCast – приведение от производного типа к базовому.
Выполняется неявно (принцип подстановки)



DownCast – приведение от базового типа к производному.
Выполняется только явно.
Может приводить к исключению

```
Person p = new Student("Петров", 18, 1, 9); // UpCast
p.IncCourse; // Ошибка компиляции!
```

Но ведь в p – студент! Как его получить? Ответ: **Downcast**

```
((Student)p).IncCourse();
или
var s = (Student)p;
s.IncCourse();
```

Операции is и as

Операция приведения типа DownCast **небезопасна** и может приводить к исключению InvalidCastException:

```
Person p = new Person("Иванов", 19);
...
var s = (Student)p;
s.IncCourse();
```

Как это сделать **безопасно**?

Операция **as** возвращает null если приведение типа невозможно

Способ 1.

```
if (p is Student)
    ((Student)p).IncCourse();
```

Способ 2.

```
var s1 = p as Student;
if (s1 != null)
    s1.IncCourse();
```

Способ 3 (C# 7.1).

```
if (p is Student s1)
    s1.IncCourse();
```

Операции is и as

Операция приведения типа DownCast **небезопасна** и может приводить к исключению InvalidCastException:

```
Person p = new Student("Петров", 18, 1, 9);
...
var s = (Student)p;
s.IncCourse();
```

Как это сделать **безопасно**?

Способ 1.

```
if (p is Student)
    ((Student)p).IncCourse();
```

Способ 2.

```
var s1 = p as Student;
if (s1 != null)
    s1.IncCourse();
```

Переменная p имеет два типа:
статический (Person) – на этапе компиляции
динамический (Student) – на этапе выполнения

Полиморфизм

```
public class Person
{
    ...
    public void Print()
    {
        WriteLine(Name + " " + Age);
    }
}

public class Student : Person
{
    ...
    public new void Print()
    {
        base.Print();
        WriteLine(Course + " " + Group);
    }
}

Person p = new Student("Петров", 18, 1, 9);
p.Print();
```

Основной вопрос:
Какой Print вызовется?
Для класса Person или для класса Student?

Полиморфизм

```
public class Person
{
    ...
    public void Print()
    {
        WriteLine(Name + " " + Age);
    }
}

public class Student : Person
{
    ...
    public new void Print()
    {
        base.Print();
        WriteLine(Course + " " + Group);
    }
}

Person p = new Student("Петров", 18, 1, 9);
p.Print();
```

Правильный ответ:
Для класса Person.
Решение принимается на этапе компиляции (**рано**)

Полиморфизм

```
public class Person
{
    ...
    public void Print()
    {
        WriteLine(Name + " " + Age);
    }
}

public class Student : Person
{
    ...
    public new void Print()
    {
        base.Print();
        WriteLine(Course + " " + Group);
    }
}

Person p = new Student("Петров", 18, 1, 9);
p.Print();
```

А нельзя ли чтобы
Решение принималось на этапе выполнения (**поздно**) и зависело от динамического типа p?

Полиморфизм. Виртуальные функции

```
public class Person
{
    ...
    public virtual void Print()
    {
        WriteLine(Name + " " + Age);
    }
}

public class Student : Person
{
    ...
    public override void Print()
    {
        base.Print();
        WriteLine(Course + " " + Group);
    }
}

Person p = new Student("Петров", 18, 1, 9);
p.Print();
```

Решение: Использовать виртуальные функции
Важно: Виртуальные функции в потомках должны иметь ровно такие же параметры, как и у предка
Говорят, что виртуальные функции предка и потомков образуют **цепочку виртуальности**

Виртуальный метод вызывается немного дольше обычного, но обеспечивает бОльшую гибкость

Полиморфизм. Определения

Полиморфизм = много форм

Полиморфизм – способность родственных объектов, связанных иерархией наследования, выполнять одно и то же действие немного по-разному

Раннее связывание – связывание имени метода с конкретным телом на этапе компиляции (рано)

Позднее связывание – связыванием имени метода с конкретным телом на этапе выполнения (поздно). Реализуется с помощью виртуальных функций. Обеспечивает полиморфизм родственных объектов

Полиморфная переменная – переменная, у которой статический тип (при описании) и динамический тип (при выполнении) могут не совпадать. **Полиморфное действие** – метод, выполняемый по-разному потомками разных типов (как правило, виртуальный метод)

Полиморфизм использует ссылочную объектную модель, поэтому работает только для классов и не работает для труктур struct

Виртуальные методы как блоки для замены кода

```
public class BaseStud
{
    public virtual void Sleep() => WriteLine("Sleep");
    public virtual void Eat() => WriteLine("Eat");
    public virtual void Think() => WriteLine("Think");
}
public class BadStud: BaseStud
{
    public override void Eat() => WriteLine("Sleep");
    public override void Think() => WriteLine("Sleep");
}
...
    public static void Сессия(BaseStud s)
    {
        WriteLine("Сессия:");
        s.Sleep();
        s.Eat();
        s.Think();
    }
    static void Main(string[] args)
    {
        Сессия(new BaseStud());
        Сессия(new BadStud());
    }
```

Функция-каркас, содержащая сменные модули

Блоки кода, которые можно менять, переопределяя функции в потомках

Полиморфные контейнеры

Контейнер называется **полиморфным**, если он состоит из полиморфных переменных базового типа

Примеры: List<Person>, Person[], HashSet<Person> и т.д.

Полиморфные контейнеры служат для вызова полиморфных действий:

```
var l = new List<Работник>();
l.Add(new Программист());
l.Add(new Программист());
l.Add(new Бухгалтер());
l.Add(new СистемныйАдминистратор());
l.Add(new Уборщица());

foreach (var worker in l)
    worker.Work();
```

Виртуальный метод

Важное преимущество данного кода – **разделение обязанностей**: начальник не должен знать метод Work() своих подчинённых, но должен вовремя вызывать код работы всего отдела

Полиморфные контейнеры и is - as

В полиморфных контейнерах можно вызывать невиртуальные методы для определённых классов:

```
var l = new List<Person>();
l.Add(new Person(...));
l.Add(new Student(...));
l.Add(new Student(...));

foreach (var p in l)
    if (p is Student)
        ((Student)p).IncCourse();
```

Класс Object – предок всех классов

Предком всех типов в .NET является класс **Object**.

Все методы, которые есть в Object, наследуются всеми классами.

Класс Object имеет следующий интерфейс (сокращено):

```
public class Object
{
    public virtual bool Equals(Object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}
```

Переопределение ToString()

Метод ToString() – строковое представление объекта

По умолчанию ToString() возвращает имя типа . Если мы хотим это изменить, то переопределяем его в потомках

Console.WriteLine(p) эквивалентно Console.WriteLine(p.ToString())

```
public class Person
{
    ...
    public override string ToString() => $"{Name} {Age}";
    public void Print() => WriteLine(ToString());
}
public class Student : Person
{
    ...
    public override string ToString() =>
        base.ToString()+$" {Course} {Group}";
}

Console.WriteLine(p);
```

Переопределение ToString()

Метод ToString() – строковое представление объекта

По умолчанию ToString() возвращает имя типа . Если мы хотим это изменить, то переопределяем его в потомках

Console.WriteLine(p) эквивалентно Console.WriteLine(p.ToString())

Невиртуальный метод Print(), ведущий себя полиморфно (т.к.вызывает виртуальный метод!). Переопределять в потомке его не надо!

```
public class Person
{
    ...
    public override string ToString() => $"{Name} {Age}";
    public void Print() => WriteLine(ToString());
}
public class Student : Person
{
    ...
    public override string ToString() =>
        base.ToString()+$" {Course} {Group}";
}

Console.WriteLine(p);
```

GetType() и его использование

Метод GetType() возвращает объект типа Type, однозначно характеризующий тип.

У двух разных объектов одного типа их GetType() равны

Операция **typeof**(тип) возвращает объект Type для данного типа

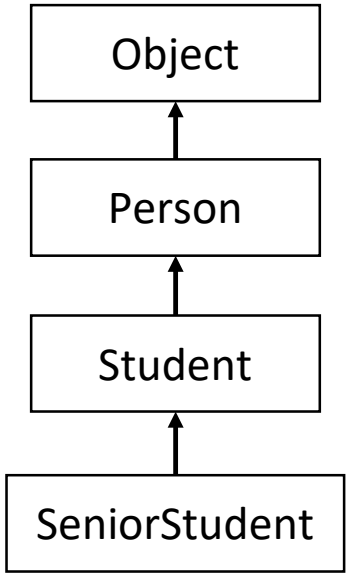
Пример 1.

```
Person p1 = new Student("Petrov", 18, 1, 9);
WriteLine(p1.GetType().Name); // Student
WriteLine(p1.GetType().BaseType.Name); // Person
```

Пример 2.

```
Person p1 = new SeniorStudent("Petrov", 18, 1, 9);
WriteLine(p1.GetType() == typeof(SeniorStudent)); // True
WriteLine(p1.GetType() == typeof(Student)); // False
WriteLine(p1 is Student)); // True
```

GetType() и родословная объекта




```
var t = p.GetType();
while (t != null)
{
    WriteLine(t.Name);
    t = t.BaseType;
}
```

Вывод:

```
Student
Person
Object
```

Переопределение GetHashCode() и Equals()

Метод GetHashCode() возвращает целое, представляющее **хеш-код** объекта. Хеш-код должен вычисляться быстро по всем полям

Для классов T, у которых переопределён GetHashCode(), можно определять множества HashSet<T> (сложность всех операций Θ(1))

Если объекты равны, то их хеш-коды совпадают. Однако два объекта с одинаковыми хеш-кодами не обязательно равны

Метод GetHashCode() всегда переопределяется вместе с методом **Equals**(), т.к. иначе два объекта с равным хеш-кодом будут не равны в смысле Equals(), что неправильно

Наша задача: добиться чтобы можно было определять

```
var st = new HashSet<Person>();
```

Переопределение GetHashCode() и Equals()

```
public class Person
{
    ...
    public override int GetHashCode() =>
        Name.GetHashCode() ^ Age.GetHashCode();

    public override bool Equals(Object o)
    {
        if (o == null)
            return false;
        if (o.GetType() != GetType())
            return false;
        var p = o as Person;
        return p.Name == Name && p.Age == Age;
    }
}
// Ура! теперь можно
var st = new HashSet<Person>();
```

Побитовое "или" перемешивает значения полей

Интерфейсы

Интерфейс – набор требований, которым должен удовлетворять класс. В интерфейсе содержатся заголовки методов и свойств. В интерфейсе не могут содержаться поля и реализация методов и свойств.

Все методы интерфейса неявно **виртуальные** (!!!)

Интерфейс по умолчанию internal, его члены – public. public для членов интерфейса ставить нельзя.

```
public interface IPrintable
{
    void Print();
}
public interface ICloneable
{
    ICloneable Clone();
}
public interface IPoint
{
    int X { get; set; }
    int Y { get; set; }
}
```

Реализация интерфейсов классами

Класс, реализующий интерфейс, должен реализовывать все свойства и методы интерфейса **public**-образом. Все реализуемые методы – **неявно override** (!!!)

Класс может реализовывать несколько интерфейсов (но должен наследоваться от единственного класса)

```
public class Person: IPrintable,ICloneable
{
    public void Print() => WriteLine(this);
    public ICloneable Clone() => new Person(Name, Age);
}
public class Student : Person, IPrintable, ICloneable
{
    // Print унаследовано – значит реализовано
    public new ICloneable Clone() => new Student(Name, Age,Course,Group);
}
...
ICloneable ic = new Student("Иванов",18,1,9);
Object o = ic.Clone(); // создаётся клон объекта, тип которого – в ic
...
public static void GlobalPrint(IPrintable ip) => ip.Print();
...
GlobalPrint(new Person("Петров",19))
```

Реализация интерфейсов классами (2)

Абсолютно разнородные классы могут реализовывать одни и те же интерфейсы.

Благодаря интерфейсам объекты этих классов можно помещать в один полиморфный список

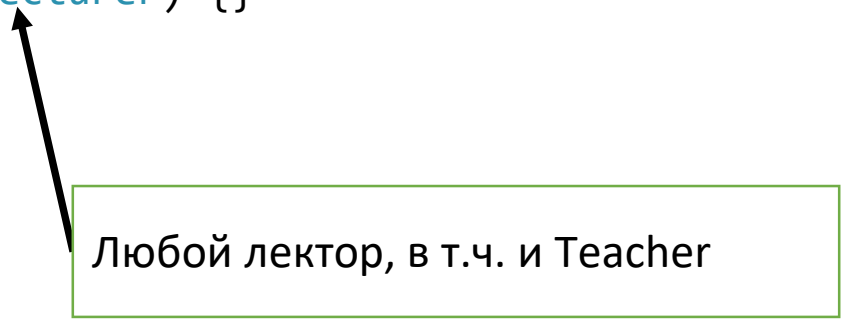
```
public class Point: ICloneable, IPoint
{
    public int X { get; set; }
    public int Y { get; set; }
    public Point(int x, int y) => (X, Y) = (x, y);
    public ICloneable Clone() => new Point(X, Y);
}
...
// Клонирование полиморфного списка
var l = new List<ICloneable>();
l.Add(new Person("Иванов", 19));
l.Add(new Student("Петров", 18, 1, 9));
l.Add(new Point(2, 3));

var l1 = new List<ICloneable>();
foreach (var ic in l)
    l1.Add(ic.Clone());
```

Интерфейсы как роли

Класс может поддерживать множество интерфейсов.

Интерфейсы можно рассматривать как роли объекта в различных ситуациях. С каждой ролью связана группа действий, которые объект может в этой ситуации выполнять.

```
public class Teacher: IPassenger, ICitizen, ILecturer, ICustomer ...

public struct Int32 : IComparable, IFormattable, IConvertible,
  IComparable<Int32>, IEquatable<Int32>

public static void Lecture(l: ILecturer) {}
```



Интерфейс сравнения IComparable<T>

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

Все стандартные типы T, которые можно сравнивать на "меньше", поддерживают интерфейс `IComparable<T>`

`i.CompareTo(j) < 0` означает, что `i < j`

Реализация IComparable<T> для Person

```
public class Person: IComparable<Person>
{
    ...
    public int CompareTo(Person p)
    {
        var nb = Name.CompareTo(p.Name);
        if (nb == 0)
            return Age.CompareTo(p.Age);
        else return nb;
    }
}
```

Благодаря поддержке этого интерфейса можно определять

```
SortedSet<Person> sp = new SortedSet<Person>();
```

Интерфейс сравнения IComparer<T>

```
public interface IComparer<T>
{
    int Compare(T t1, T t2);
}
```

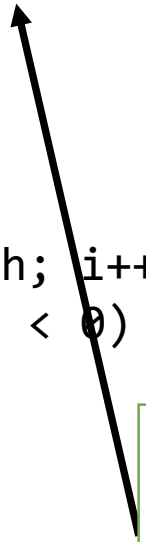
Преимущества: является внешним по отношению к типу. Это позволяет для одного типа определять несколько критериев сравнения

Пример использования:

```
public class ComparePersonByAge: IComparer<Person>
{
    public int Compare(Person p1, Person p2) =>
      p1.Age.CompareTo(p2.Age);
}
...
var l = new List<Person>();
l.Sort(new ComparePersonByAge());
```

Ограничения на параметры обобщений в методах

```
public static T Min<T>(T[] arr) where T: IComparable<T>
{
    if (arr.Length == 0)
        return default(T);
    var min = arr[0];
    for (var i = 1; i < arr.Length; i++)
        if (arr[i].CompareTo(min) < 0)
            min = arr[i];
    return min;
}
```

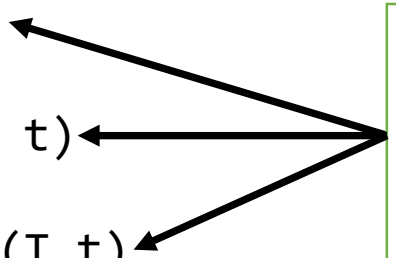


Ограничения в where имеют вид:

T: интерфейс

Ограничения на параметры обобщений в классах

```
class MySortedSet<T> where T: IComparable<T>
{
    bool Add(T t)
    { }
    bool Remove(T t)
    { }
    bool Contains(T t)
    { }
}
```



Ранее мы не могли реализовать эти методы, т.к. для переменных типа T была необходима операция "меньше"

Q & A