

Высокоуровневые методы информатики и программирования

Лекция 7

Создание классов

План работы

- Описание классов
- Состав классов – элементы класса
- Поля
- Методы
- Конструкторы и деструкторы

Классы

- **Классы это основные пользовательские типы данных.**
- Экземпляры класса – Объекты.
- Классы описывают все элементы объекта (данные) и его поведение (методы), также устанавливают начальные значения для данных объекта, если это необходимо.
- При создании экземпляра класса в памяти создается копия данных этого класса. Созданный таким образом экземпляр класса называется объектом.
- Экземпляр класса создается с помощью оператора *new*.
- Для получения данных объекта или вызова методов объекта, используется оператор `.` (точка).

`Student s;`

`s = new Student();`

`s.Name = "Иванов А.";`

- При создании экземпляра класса, копия данных, описываемых этим классом, записывается в память и присваивается переменной ссылочного типа

Составные элементы класса

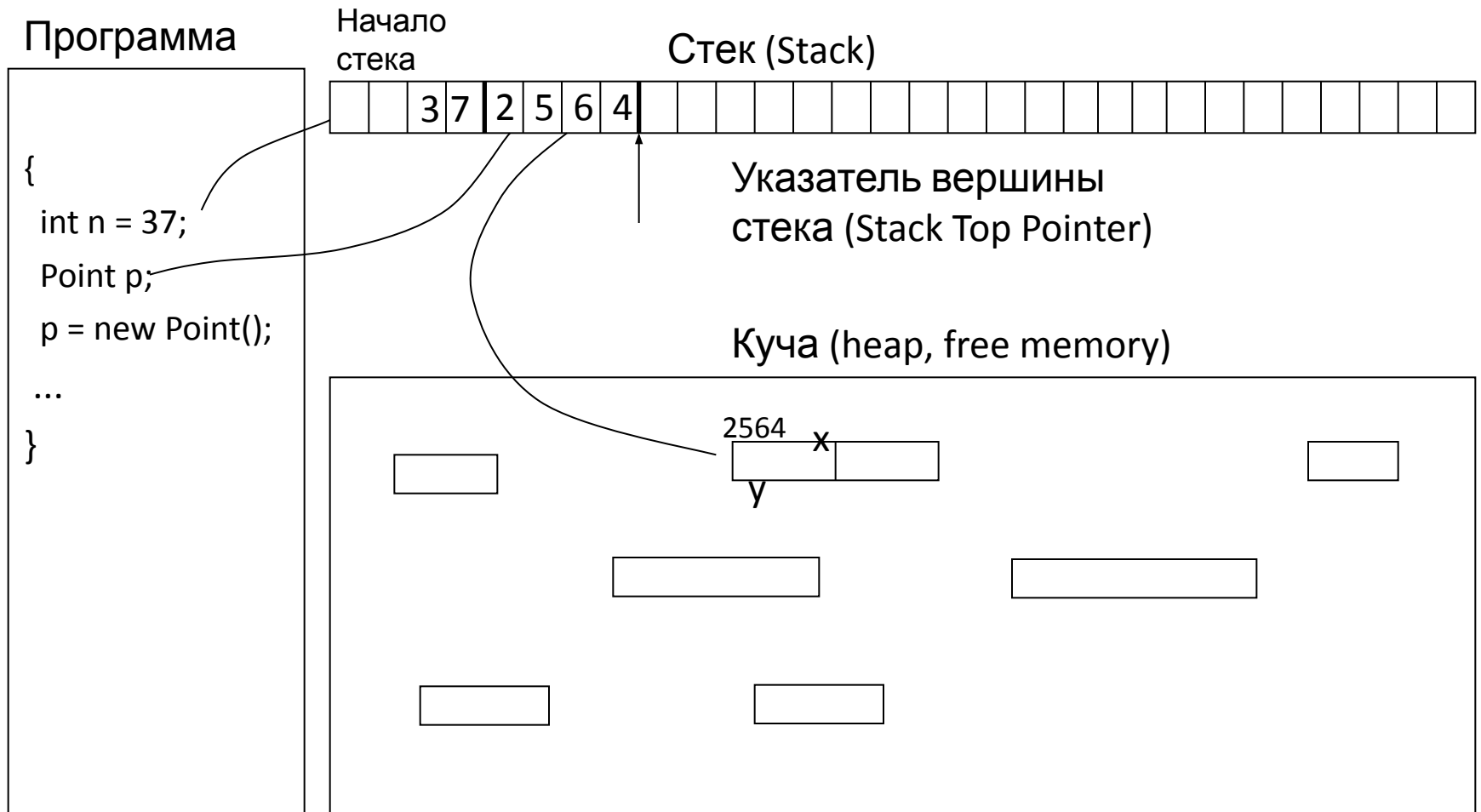
1. **Поля (field)** – обычно скрытые данные класса (внутренне состояние)
2. **Методы (methods)** – операции над данными класса (поведение) (можно называть функциями)
3. **Свойства (property)** – доступ к данным класса с помощью функций
 - **get** – получить
 - **set** – задать
4. **События (event)** – оповещение пользователей класса о том, что произошло что-то важное.

Поля класса

Поля класса

- Состояние объектов класса (а также структур, интерфейсов) задается с помощью переменных, которые называются *полями (fields)*.
- При создании объекта – экземпляра класса, в динамической памяти выделяется участок памяти, содержащий набор *полей*, определяемых классом, и в них записываются значения, характеризующие начальное состояние данного экземпляра.
- Объявление полей выполняется следующим образом:
`[<режим_доступа>] [модификаторы] <тип> <имя>;`
- Общим правилом является создание закрытых полей, имеющих режим доступа `private`. Данный режим задается полям по умолчанию.
- Любое воздействие на состояние объекта класса выполняется с использованием свойств или методов класса, которые контролируют последствия этих воздействий.
- Если полям класса не задается значение при объявлении, то они автоматически инициализируются значениями по умолчанию.
 - Для значащих переменных – это нулевое значение,
 - Для строк – это пустая строка,
 - Для ссылочных переменных – это стандартное значение `null`, как показано в комментариях описания класса `Person`.

Размещение полей в памяти программы



Поля класса (2)

- Поля класса создаются для каждого создаваемого объекта в выделенном ему участке памяти в "куче".
- Областью видимости полей являются все методы класса. При этом для использования поля требуется задавать только его имя.
- Например, метод вычисления возраста для объекта класса `Person` в днях может быть выполнено следующим образом:

```
public int CalcDays() { // вычисление возраста в днях
    int days = age * 365; // age - поле данного объекта
    return days;
}
```

- Если поле имеет режим `public`, то оно доступно там, где имеется ссылка на объект данного класса.
- Для обращения к этим полям из методов других классов (если поля открытые) нужно использовать ссылочную переменную, которая хранит ссылку на созданный объект.
- Например:

```
Person p; //объявление переменной типа Person
p = new Person(); //создание объекта и сохр. ссылки
p.Name = "Иванов П.И. "; //задание значения public поля
```


Поля класса (3)

- В качестве модификатора поля может использоваться ключевое слово `static`, обозначающее, что это статическое поле. Например, в классе `Person` может быть описано следующее статическое поле:

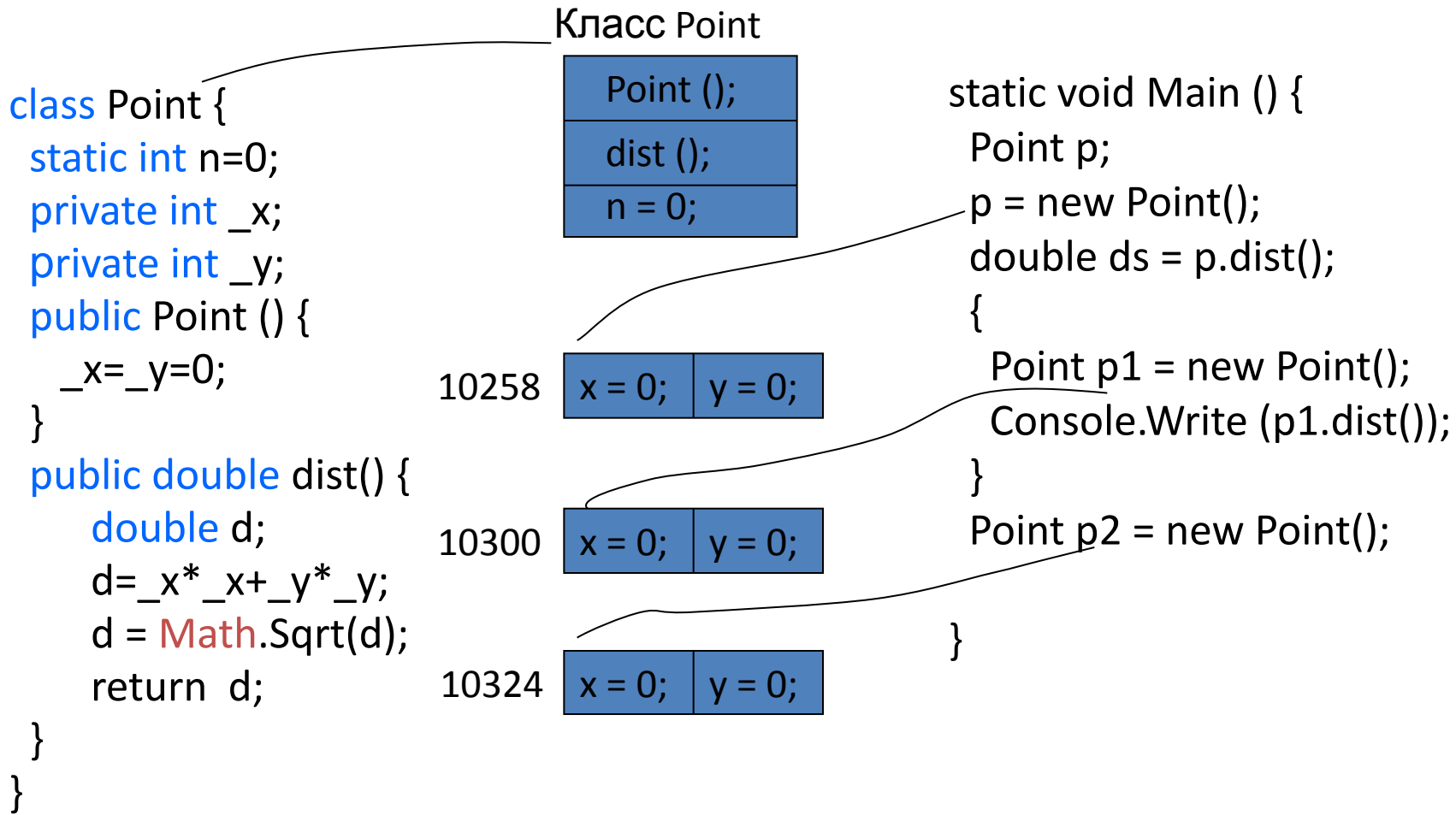
```
static int numPersons=0; // кол-во объектов класса
```

- Статическое поле класса создается только одно для всего класса.
- Для обращения к нему нужно указать имя класса и через точку имя статического поля. Например:

```
Person.numPersons++;
```

- Время существования полей определяется объектом, которому они принадлежат. Объекты в "куче", с которыми не связана ни одна ссылочная переменная, становятся недоступными и удаляются сборщиком мусора.

Размещение описания методов класса и объектов



Методы классов C#

Методы программы

- Описываются только в классах
- Имеют доступ
 - к закрытым и открытым переменным класса (полям)
 - Локальным переменным

Описание и вызов метода

- Описание метода содержит

```
<заголовок_метода>
{
    тело_метода
}
```
- Синтаксис заголовка метода
 - [модификаторы] {тип_результата} имя_метода
([список_формальных_параметров])
- Вызов метода
 - имя_метода([список_фактических_параметров])

Вызов метода

```
MyClass a = new MyClass();  
x = a.MyMethod(val1, val2, ... , valN);  
...  
d = (b + a.MyMethod(5))/3.14;
```

```
x = (5+2)/3.14;  
a = func(x);
```

```
float MyMethod (double a)  
{  
    double w;  
    x = a*25.7/5.6;  
    return x;  
}
```

Описание метода

- Заголовок метода

[режим доступа] <type> name (parameters) // method header

```
{  
    statement;  
    ...  
    statement;  
    return X;  
}
```

} Method body

- Например:

```
void A(int p) {...}
```

```
int B(){...}
```

```
public void C(){...}
```

- Описание параметра

[ref | out | params]тип_параметра имя_параметра

Формальные параметры методов

- По умолчанию параметры передаются по значению.
- Значением переменной ссылочного типа является ее адрес.
- Можно задать передачу параметров по ссылке
 - ref – параметр должен иметь начальное значение
 - out – параметр может не иметь начального значения
- Синтаксис объявления формального параметра
[ref | out] <тип> имя

Модификаторы параметров

Модификатор	Пояснение
(нет)	Если у параметра не задан модификатор, то предполагается что он передается по значению (т.е. вызываемый метод получает копию фактического параметра).
out	Данные передаются по ссылке. Данному параметру в вызываемом методе должно задаваться значение. Если вызываемый метод не задает значение данному параметру, то будет ошибка компиляции).
ref	Данные передаются по ссылке. Значение параметру задается вызывающим методом и может быть изменено в вызываемом методе.
params	Параметр с таким модификатором позволяет передавать переменное количество формальных параметров, как один логический параметр. В методе может быть только один формальный параметр с таким модификатором и он должен быть последним

Передача произвольного числа параметров

- Несмотря на фиксированное число формальных параметров, есть возможность при *вызове метода* передавать ему *произвольное число фактических параметров*.
- Для реализации этой возможности в списке формальных параметров необходимо задать ключевое слово `params`. Оно задается один раз и указывается только для последнего параметра списка, объявляемого как массив произвольного типа.
- При *вызове метода* этому формальному параметру соответствует *произвольное число фактических параметров*.
- Например:

```
void Sum(out long s, params int[] p)
{
    s = 0;
    for( int i = 0; i < p.Length; i++)
        p2 += (long)Math.Pow(p[i], 3);
    Console.WriteLine("Метод А-2");
}
```

Фактические параметры

- Фактические параметры должны соответствовать по количеству и типу формальным параметрам.
- Соответствие между типами параметров точно такое же, как и при присваивании:
 - Совпадение типов.
 - Тип формального параметра является производным от типа фактического параметра.
 - Задано неявное или явное преобразование между типами.
- Если метод перегружен, то компилятор будет искать метод с наиболее подходящей сигнатурой.

Выполнение вызова метода

- При *вызове метода* выполнение начинается с вычисления фактических параметров, которые являются выражениями.
- Упрощенно вызов метода можно рассматривать, как создание *блока*, соответствующий телу метода, в точке вызова. В этом *блоке* происходит замена имен формальных параметров фактическими параметрами.
- При передачи параметров *по значению* (нет модификаторов `ref` или `out`):
 - Создаются локальные переменные методов.
 - Тип локальных переменных определяется типом соответствующего формального параметра.
 - Вычисленные выражения фактических параметров присваиваются специально создаваемым локальным переменным метода.
 - ***Замена формальных параметров на фактические выполняется так же, как и оператор присваивания.***
- При передачи параметров *по ссылке* выполняется замена формальных параметров на реально существующий фактический параметр.
- Когда методу передается объект ссылочного типа, то все поля этого объекта могут меняться в методе любым образом, поэтому `ref` или `out` не часто появляются при описании параметров метода.

Перегрузка методов

- Перегруженные (overloaded) методы – это методы с одинаковым именем, но с разной **сигатурой**.
- **Сигатура** это возвращаемый тип результата и типы передаваемых параметров.
- Например:
`int <имя метода> (int, float, double)`
- Перегрузка методов выполняется для следующих целей:
 - чтобы метод принимал разное количество параметров.
 - чтобы метод принимал параметры разного типа, между которыми нет неявного преобразования.

Специальная переменная класса

this

- В методах класса можно использовать переменную this.
- this это ссылка на тот объект для которого данный метод используется.
- this нигде не объявляется
- Чаще всего используется для обращения к полям класса, если имя параметров совпадает с именем поля.

- Например:

```
public Box (int Width, int Hight)
{
    this.Width = Width;
    this.Hight = Hight;
}
```

Формальные параметры методов

- Описание формального параметра
[ref|out|params] тип_параметра имя_параметра
- **По умолчанию параметры передаются по значению.**
- Значением переменной ссылочного типа является ее адрес.
- Можно задать передачу параметров по ссылке
 - ref – параметр должен иметь начальное значение
 - out – параметр может не иметь начального значения
- Синтаксис объявления формального параметра
[ref | out] <тип> имя

Пример

```
class Point {  
    public void swap(ref int a, ref int b, out int c){  
        //int c;  
        c = a;  
        a = b;  
        b = c;  
    }  
    ...  
}  
....  
int x = 5, y = 7, z;  
Point p;  
p = new Point();  
p.swap(ref x, ref y, out z);
```


Пример передачи объектов по ссылке и значению

```
class Program
{
    static void Main(string[] args)
    {
        MyClass mc;
        mc = new MyClass();
        MyMetod(ref mc);

        MyClass mc2;
        mc2 = new MyClass();

        swapMetod(ref mc, ref mc2);

        swapMetod2(mc, mc2);

        Console.WriteLine("Привет Мир!");
    }

    static void MyMetod(ref MyClass x)
    {
        x.a = 10;
    }
}
```

```
static void swapMetod(ref MyClass x, ref MyClass y)
{
    MyClass z;
    z = x;
    x = y;
    y = z;
}

static void swapMetod2(MyClass x, MyClass y)
{
    MyClass z;
    z = x;
    x = y;
    y = z;
}

class MyClass
{
    public int a;
    public MyClass()
    {
        a = 0;
    }
}
```

Перегрузка методов

- В C# не требуется уникальности имени метода в *классе*. Существование в *классе* методов с одним и тем же именем называется *перегрузкой*, а сами методы называются **перегруженными**.
- Уникальной характеристикой перегруженных методов является их **сигнатура**.
- Перегруженные методы, имея одинаковое имя, должны отличаться
 - либо числом параметров,
 - либо их типами,
 - либо модификаторами (заметьте: с точки зрения сигнатуры, ключевые слова `ref` или `out` не отличаются).
- Уникальность сигнатуры позволяет вызвать требуемый перегруженный метод.

Пример перегрузки методов

Перегрузка метода A():

```
void A(out long p2, int p1){
    p2 =(long) Math.Pow(p1,3);
}
void A(out long p2, params int[] p){
    p2=0;
    for(int i=0; i <p.Length; i++)
        p2 += (long)Math.Pow(p[i],3);
    Console.WriteLine("Метод A-2");
}
void A(out double p2, double p1){
    p2 = Math.Pow(p1,3);
}
void A(out double p2, params double[] p){
    p2=0;
    for(int i=0; i <p.Length; i++)
        p2 += Math.Pow(p[i],3);
}
```

Вызовы перегруженного метода A():

```
public void TestLoadMethods(){
    long u=0; double v =0;
    A(out u, 7); A(out v, 7.5);
    Console.WriteLine ("u= {0}, v= {1}", u,v);
    A(out v,7);
    Console.WriteLine("v= {0}",v);
    A(out u, 7,11,13);
    A(out v, 7.5, Math.Sin(11.5)+Math.Cos(13.5),
15.5);
    Console.WriteLine ("u= {0}, v= {1}", u,v);
}
```

Основные методы класса

- **Конструктор**

- Метод с именем класса (например, `Point()`),
- вызывается автоматически при создании экземпляра класса

- **Деструктор**

- Метод с тильдой и названием класса (`~Point()`)
- Активно не используется, так как при автоматической сборке мусора неизвестно когда будет вызываться.

Конструкторы

- Метод с именем класса.
- Нет типа результата (даже `void`).
- Может перегружаться.
- Вызывается с помощью операции `new`.

Конструктор по умолчанию

- Конструктор класса без параметров называется конструктором по умолчанию (default constructor)
- Конструктор по умолчанию без операторов автоматически создается компилятором в классе (если нет никакого другого конструктора).
- Например

```
class Student
{
    String Name;
    int course;
}
```

...

```
// используем автоматически созданный конструктор
```

```
Student st = new Student();
```

- Если в классе описан хотя бы один конструктор, то конструктор по умолчанию автоматически не создается.

```
class Student
```

```
{
    String Name;
    int course;
    public Student(string nm, int crs) {Name=nm; Course=crs;}
}
```

...

```
// используем автоматически созданный конструктор, а его уже нет!!!
```

```
Student st = new Student(); // ошибка!
```

Деструктор

Деструктор используется для освобождения неуправляемых ресурсов:

```
// Override System.Object.Finalize() via finalizer syntax.
class MyResourceWrapper
{
    ~MyResourceWrapper ()
    {
        // Clean up unmanaged resources here.
        // Beep when destroyed (testing purposes only!)
        Console.Beep ();
    }
}
```

Использование `this` для цепочки конструкторов

- Другим способом применения ключевого слова `this` является использование шаблона проектирования называемого цепочкой конструкторов (constructor chaining).
- Такой шаблон полезен, когда программируется класс с большим количеством конструкторов.
- Учитывая тот факт, что конструкторы часто проверяют переданные параметры в перегруженных конструкторах может быть много повторяющихся проверок.
- Для того, чтобы это избежать разрабатывается конструктор с максимальным количеством параметром – главный конструктор (master constructor), в котором выполняется полная проверка всех передаваемых значений. Другие конструкторы выполняют вызов главного конструктора с помощью ключевого слова `this`.

```
public <имя класса>(параметры) :  
    this(параметры) {}
```

Например:

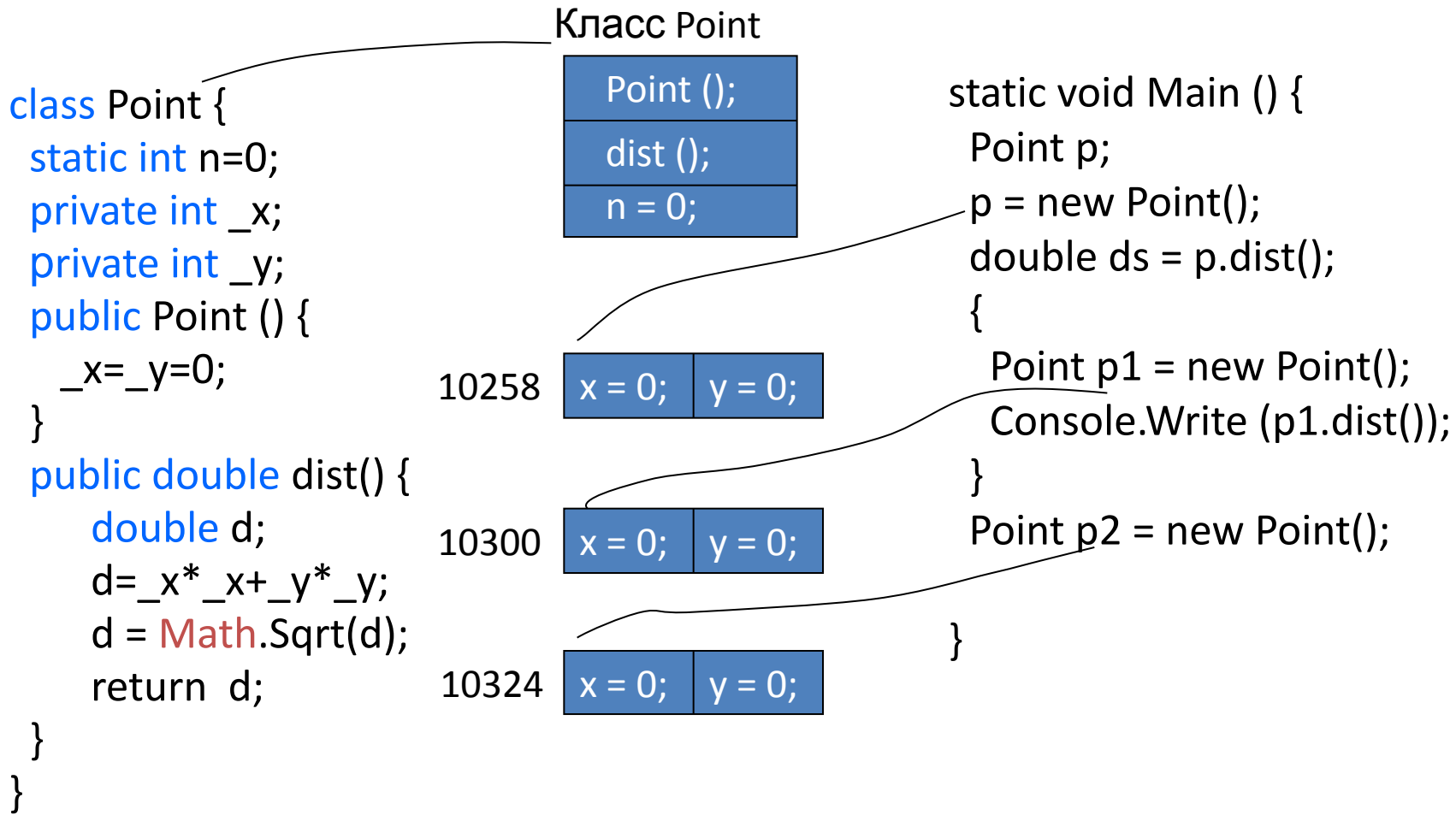
```
public Motorcycle(int intensity) :  
    this(intensity, "") {}
```


Пример с главным конструктором

```
class Motorcycle
{
public int driverIntensity;
public string driverName;
public Motorcycle() { }
// Избыточная логика конструкторов!
public Motorcycle(int intensity)
{
    if (intensity > 10)
    {
        intensity = 10;
    }
    driverIntensity = intensity;
}
public Motorcycle(int intensity, string name)
{
    if (intensity > 10)
    {
        intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
}
...
}
```

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;
    // Constructor chaining.
    public Motorcycle() {}
    public Motorcycle(int intensity) :
        this(intensity, "") {}
    public Motorcycle(string name) :
        this(0, name) {}
    // Это главный конструктор, который
    // делает всю реальную работу.
    public Motorcycle(int intensity,
        string name)
    {
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}
```

Размещение описания методов класса и объектов



Сборка мусора

- В C/C++ - приложениях очень часто возникают утечки памяти (memory leaks):

```
*C p = new C();  
//...  
p = new C();
```

 - Распределение памяти “вручную”
 - Нет явных правил принадлежности объектов
- В серверных приложениях вообще не должно быть утечек памяти. Должны работать месяцы и годы без единой утечки памяти.
- Решение: автоматическое управление памятью (деструкторов нет)
- Стратегия сборки мусора .NET аналогична сборке мусора в языке Java.
- GC (Garbage Collector – сборщик мусора) автоматически удаляет все неиспользуемые объекты
- Более эффективные методы управления памятью
- Простые в использовании, не допускающие утечек памяти

C#: Недетерминированный процесс уничтожения объектов

- Недостатки схемы, основанной на сборке мусора:
 - Сборка выполняется в какой-либо, заранее не известный момент в будущем
- Вместо этого предлагается реализовывать интерфейс `IDisposable`, используя оператор `using`:

Внимание: Код не должен зависеть от деструкторов, освобождающих внешние ресурсы

```
using( File f = new File("c:\\xyz.xml") )  
{  
    . . .  
}
```

- `IDisposable.Dispose()` вызывается при выходе из блока

Свойства класса

- Весьма полезный гибрид поля и методов: поле, доступ к которому и присваивание которому – программируемые операции
- Используются для:
 - Реализации элементов данных, которые не могут быть изменены (для этого достаточно просто не включать метод *set* в описание свойства)
 - Проверок перед присваиванием (например, проверок полномочий)
 - Реализации вычисляемых (*активных*) значений
- Пример:

```
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Описание свойств класса

- Свойства являются частными случаями *методов класса* и описываются следующим образом:

```
режим_доступа <тип> <название>
{
    // методы для задания значения свойства
    set
    {
        // задание значения value некоторому закрытому полю
    }
    // методы для получения значения свойства
    get
    {
        return(<закрытое поле>);
    }
}
```

- Например:

```
public int Age
{
    set {if (value > 0) _age = value;}
    get {return(_age);}
}
```

Пользователь объектов класса работает со *свойствами* так, как если бы они были обычными *полями объектов*:

```
Person pr = new Person();
pr.Age = 25;
```

Пример простого класса с методами

```
class Point // наш класс   МММ.Point
{
    private int _x, _y; // поля класса
    public int X // свойства класса
    {
        get {return _x;}
        set {_x = value;}
    }
    // методы - конструктор
    public Point () {x=0; y=0;}
    // вычисление длины
    public double dist ()
    {
        return Math.Sqrt(_x*_x + _y*_y);
    }
}
```

Использование класса Point

```
namespace TestProg    // наше пространство имен
{
    class Progr
    {
        static void Main( )
        {
            Point a;
            a = new Point(3,4);
            a.x = 4;    // correct
            double r;
            r = a.dist();
        }
    }
}
```


Автоматически реализуемые свойства

- Вместо обычного определения свойства, например:

```
private int myItem;  
public int MyItem  
{  
    get {return myItem;}  
    set {myItem = value;}  
}
```

- Можно сделать следующее описание:

```
public int MyProperty { get; set; }
```

Инициализация объектов класса

- Значения свойствам разрешается назначать объектам во время создания.
- Например, если имеется описание класса, имеющего два свойства A и B:

```
public class MyClass  
{  
    public int A { get; set; }  
    public int B { get; set; }  
}
```

- то можно создать и инициализировать объект данного класса следующим образом:

```
MyClass myObject = new MyClass() { A = 5, B = 10 };
```

Анонимные типы

- Анонимные типы позволяют описать тип данных без формального описания класса

- Например:

```
var book1 = new
{
    ISBN = "978-0-470-17661-0",
    Title="Professional Windows Vista Gadgets Programming",
    Author = "Wei-Meng Lee",
    Publisher="Wrox"
};
```

- Теперь book1 является объектом с 4 свойствами: ISBN , Title , Author , and Publisher.

- Можно использовать имена переменных при назначении значений свойствам, например:

```
var Title = "Professional Windows Vista Gadgets Programming";
var Author = "Wei-Meng Lee";
var Publisher = "Wrox";
var book1 = new { ISBN = "978-0-470-17661-0", Title, Author, Publisher };
```

static

- Статическими могут быть:
 - поля;
 - методы;
 - конструкторы;
 - классы.
- ...