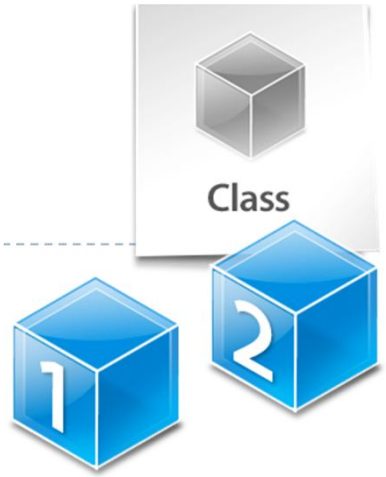
Программирование C#

5. Введение в классы, объекты и методы

Карбаев Д.С., 2016

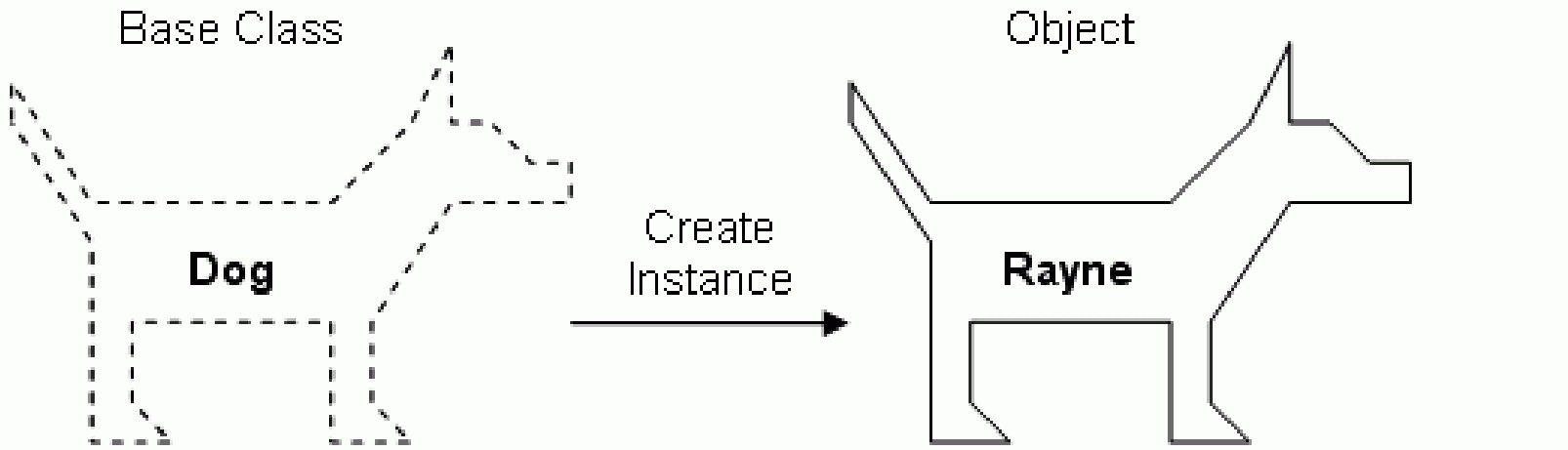
Классы



- **Класс** представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными.
- Класс, по существу, представляет собой ряд схематических описаний способа построения объекта.
- Класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан **объект** (экземпляр) этого класса.
- Общая форма:

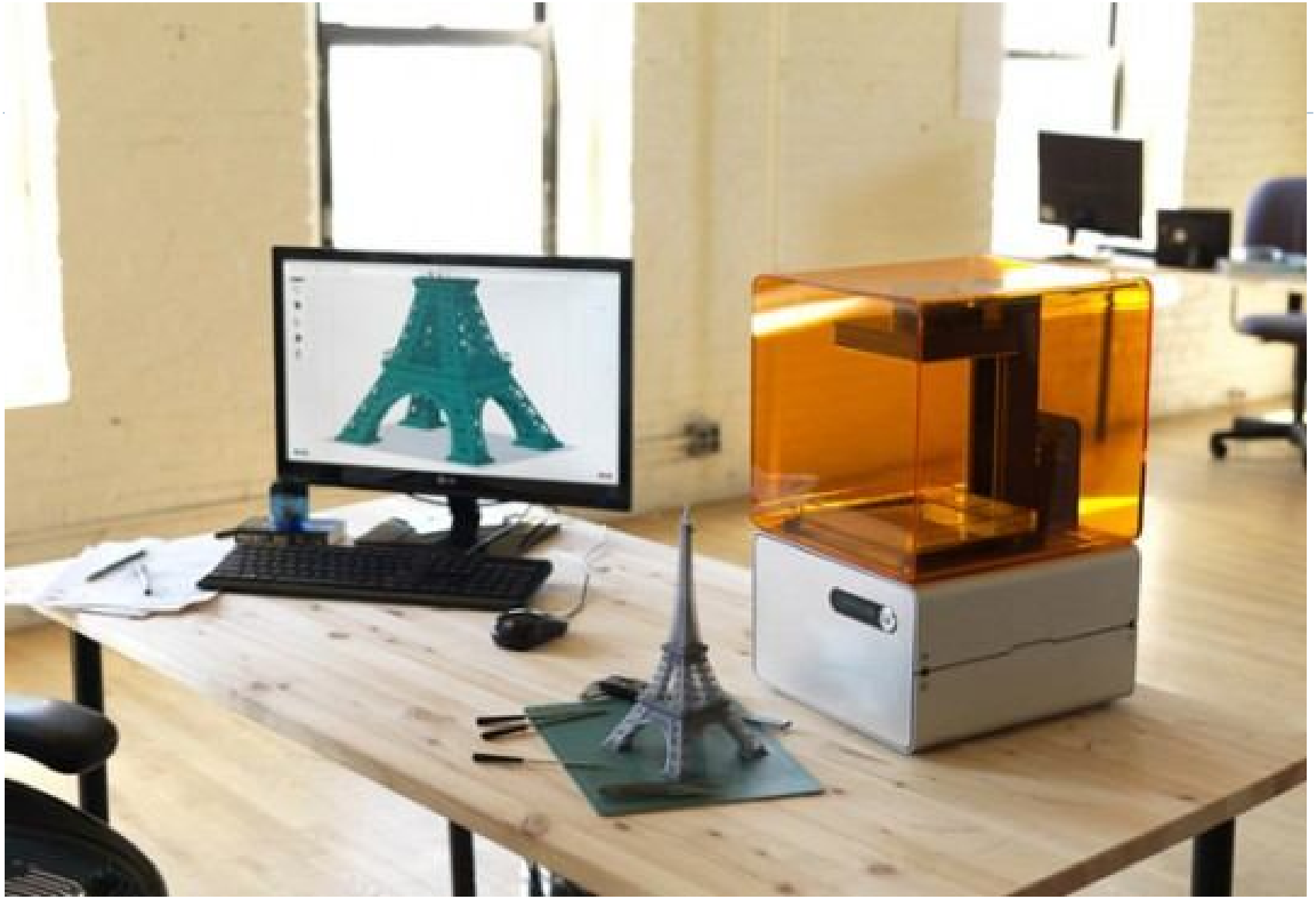
```
class имя_класса {
// Объявление переменных экземпляра.
доступ тип переменная1;
доступ тип переменная2;
//...
доступ тип переменнаяN;
// Объявление методов.
доступ возвращаемый_тип метод1 (параметры) {
// тело метода
}
//...
доступ возвращаемый_тип методM (параметры) {
// тело метода
} }
```

Классы



Properties	**Methods**
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property values	**Methods**
Color: Gray, White, and Black	Sit
Eye Color: Blue and Brown	Lay Down
Height: 18 Inches	Shake
Length: 36 Inches	Come
Weight: 30 Pounds	



Определение класса

```
class Building
{
    public int Floors; // количество этажей
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов
}
```

- Общая форма для объявления переменных экземпляра:

```
доступ тип имя_переменной;
```

где доступ обозначает вид доступа; тип — конкретный тип переменной, а имя_переменной — имя, присваиваемое переменной.

- Для того чтобы создать конкретный объект типа Building, придется воспользоваться следующим оператором.

```
Building house = new Building(); // создать объект типа Building
```

- Оператор-точка связывает имя объекта с именем члена класса:

```
house.Floors = 2;
```

Пример программы, класс Building

```
using System;
class Building {
public int Floors; // количество этажей
public int Area; // общая площадь здания
public int Occupants; // количество жильцов
}
//В этом классе объявляется объект типа Building,
class BuildingDemo {
 static void Main() {
  Building house = new Building(); // создать объект
                                   //типа Building
  int areaPP; // площадь на одного человека
  // Присвоить значения полям в объекте house,
  house.Occupants = 4; house.Area = 1000;
  house.Floors = 2;
  // Вычислить площадь на одного человека.
  areaPP = house.Area / house.Occupants;
  Console.WriteLine("Дом имеет:\n " +
   house.Floors + " этажа\n " +
   house.Occupants + " жильцов\n " +
   house.Area +
    " кв. метров общей площади, из них\n " +
    areaPP + " приходится на одного человека");
}}
```

Допустим, что исходный текст приведенной программы сохранен в файле UseBuilding.cs

В результате ее компиляции создается файл UseBuilding.exe

При этом оба класса. Building и BuildingDemo, автоматически включаются в состав исполняемого файла.

- Результат:

```
Дом имеет:
2 этажа
4 жильца
1000 кв. метров общей площади, из них
250 приходится на одного человека
```

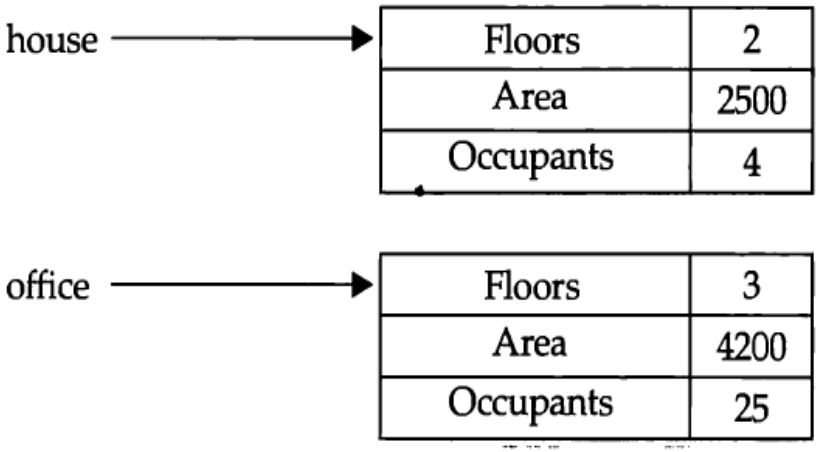
Создание нескольких объектов

```
static void Main() {
  Building house = new Building();
  Building office = new Building();
  int areaPP; // площадь на одного человека
  // Присвоить значения полям в объекте house
  house.Occupants = 4; house.Area = 1000;
  house.Floors = 2;
  // Присвоить значения полям в объекте office
  office.Occupants = 25; office.Area = 4200;
  office.Floors = 3;

  // Вычислить площадь на одного человека в жилом доме.
    areaPP = house.Area / house.Occupants;
  Console.WriteLine("Дом имеет:\n " + house.Floors +
  " этажа\n " + house.Occupants + " чел. (жильцы)\n " +
  house.Area +" кв. метров общей площади, из них\n " +
  areaPP + " приходится на одного человека");
  // Вычислить площадь на одного человека в учреждении.
    areaPP = office.Area / office.Occupants;
  Console.WriteLine("Учреждение имеет:\n " +
  office.Floors + " этажа\n" + office.Occupants +
  " работников\n " + office.Area +
  " кв. метров общей площади, из них\n " +
  areaPP + " приходится на одного человека");
}
```

- Результат:

```
Дом имеет:
2 этажа
4 чел. (жильцы)
1000 кв. метров общей площади, из них
250 приходится на одного человека
Учреждение имеет:
3 этажа
25 работников
4200 кв. метров общей площади, из них
168 приходится на одного человека
```



Создание объектов

- Для объявления объекта типа Building использовалась следующая строка кода.

```
Building house = new Building();
```

- Объявление переменной house можно отделить от создания объекта, на который она ссылается, следующим образом.

```
Building house; // объявить ссылку на объект
house = new Building(); // выделить память для объекта типа Building
```

- Во фрагменте кода

```
int x;
x = 10;
```

переменная x содержит значение 10, поскольку она относится к типу int, который является типом значения.

Но в строке

```
Building house = new Building();
```

переменная house содержит не сам объект, а лишь **ссылку** на него.

Переменные ссылочного типа и присваивание

- Когда одна переменная ссылки на объект присваивается другой, то переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, на который ссылается переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. Пример:

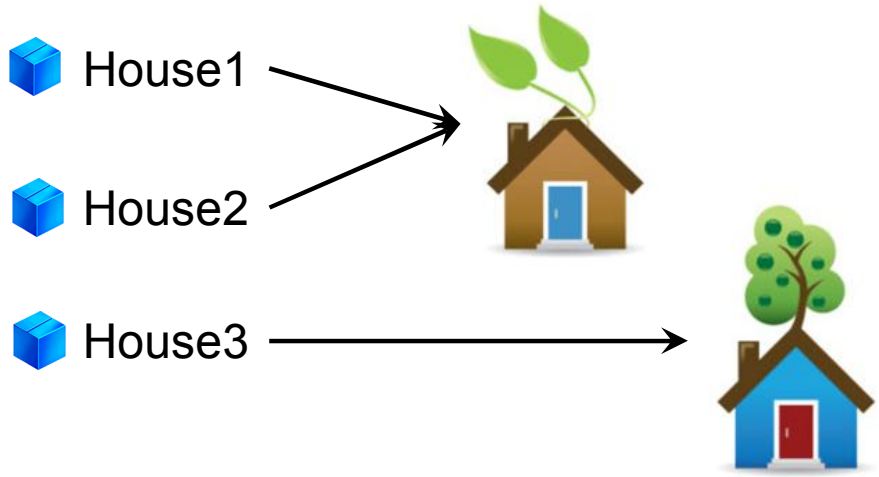
```
Building house1 = new Building();
Building house2 = house1;
```

После очередного присваивания

```
House1.Area = 2600;
```

оба метода WriteLine ()

```
Console.WriteLine(house1.Area);
Console.WriteLine(house2.Area);
```



выводят одно и то же значение: 2600.

- В результате следующей последовательности операций присваивания просто изменяется объект, на который ссылается переменная house2.

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();
house2 = house3; // теперь обе переменные, house2 и house3,
// ссылаются на один и тот же объект.
```

Методы

- Общая форма определения метода:

```
доступ возвращаемый_тип имя(список_параметров) {
// тело метода
}
```

где ***доступ*** — это *модификатор доступа*, определяющий те части программы, из которых может вызываться метод;
возвращаемый_тип обозначает тип данных, возвращаемых методом, имя обозначает конкретное имя, присваиваемое методу;
список_параметров — это последовательность пар, состоящих из типа и идентификатора и разделенных запятыми.

- Если модификатор доступа отсутствует, то метод оказывается закрытым (private) в пределах того класса, в котором он объявляется.
- Если метод не возвращает значение, то в качестве возвращаемого для него следует указать тип void.

Методы

```
//метод без параметров, не возвращает значения
  static void print() {
   Console.WriteLine("Напечатано из метода.");
  }

  //метод с 1 параметром, возвращает значения
  static void printWord(string s) {
   Console.WriteLine("Значение-параметр: "+s);
  }

 //метод с 2мя параметрами, возвращает целое
  static int sum(int x, int y) {
    int result=x+y;
    return result;
  }
```

```
static void Main(string[] args)
{
  print();

  string word = "тест";
  printWord(word);

  int a = 3, b = 5;
  int summa = sum(a, b);
  printWord(summa.ToString());
}
```

Результат:

```
Напечатано из метода.
Значение-параметр: тест
Значение-параметр: 8
```

Добавление метода в класс Building

```
class Building {
    public int Floors; // количество этажей
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов
     // Вывести площадь на одного человека,
     public void AreaPerPerson() {
        Console.WriteLine(" " + Area / Occupants +
        " приходится на одного человека");
     }
}
// Использовать метод AreaPerPerson()
class BuildingDemo {
    static void Main() {
        Building house = new Building();
        Building office = new Building();
     // Присвоить значения полям в объекте house…
    // Присвоить значения полям в объекте office…
        Console.WriteLine("Дом имеет:\n " +
        house.Floors + " этажа\n " +
        house.Occupants + " чел. (жильцы)\n " +
        house.Area + "кв. метров общей площади, из них");
         house.AreaPerPerson();
         …
         office.AreaPerPerson();
}   }
```

Возврат из метода

- В методе допускается наличие нескольких операторов **return**, особенно если имеются два или более вариантов возврата из него.

```
public void MyMeth() {
    int i;
    for (i = 0; i < 10; i++) {
        if (i == 5) return; // прервать на шаге 5
        Console.WriteLine();
    }
}
```

- Общая форма:

```
return значение;
```

- Используя возвращаемое значение, можно усовершенствовать рассматривавшийся ранее метод AreaPerPerson():

```
public int AreaPerPerson() {
    return Area / Occupants;
}
```

Возврат значения

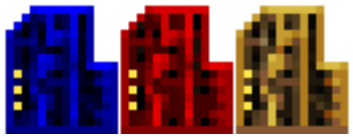
- Вызов метода AreaPerPerson() можно указать непосредственно в операторе, содержащем вызов метода WriteLine():

```
Console.WriteLine("Дом имеет:\n " +
house.Floors + " этажа\n " +
house.Occupants + " жильцов\n " +
house.Area +
" кв. метров общей площади, из них\n " +
house.AreaPerPerson() +
" приходится на одного человека");
```

- Можно сравнить величины площади на одного человека для двух зданий.

```
if (b1.AreaPerPerson() > b2.AreaPerPerson() )
Console.WriteLine("В здании b1 больше места для каждого человека");
```

Уровни доступа



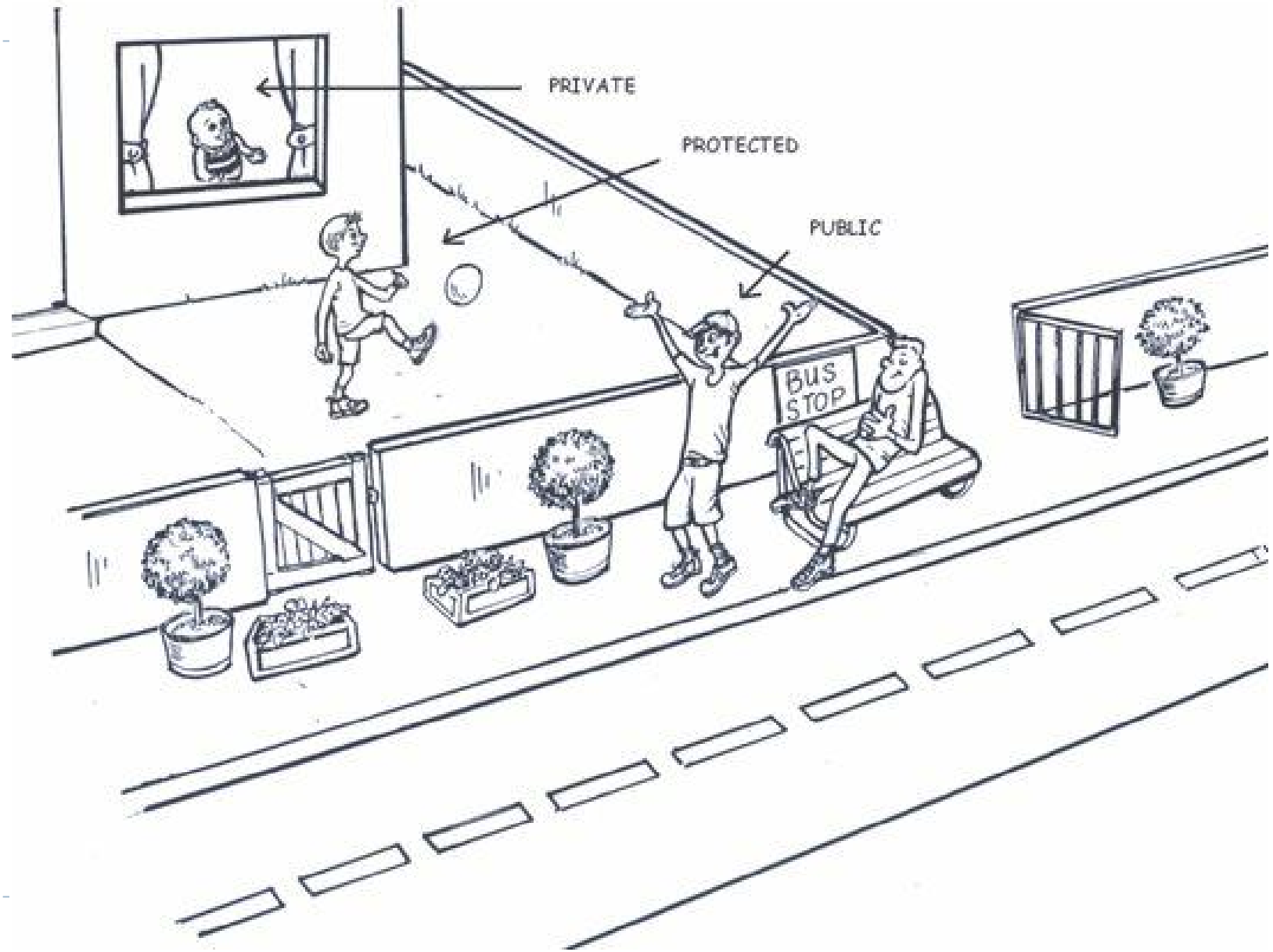
- Использование *модификаторов доступа* «private», «protected» и «public» означает насколько безопасным будет наш класс.

- **Private** (закрытое поле) означает, что объекты только этого класса могут обращаться к данному полю – используется по умолчанию;

  Public (открытое поле) означает, что объекты любого класса могут обращаться к этому полю;

- **Protected** (защищенное поле) означает, что только объекты тесно связанных классов могут обращаться к полю;

Уровни доступа



```
class Animal {
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height; public int length;
    public string color;
    bool hasTail; protected bool isMammal;
    private bool spellingCorrect;

    public string GetFoodInfo() {
 // Открытый метод, получающий информацию о том,
 // чем питается животное
 //…
        return "";
    }

    private void SpellingCorrect() {
   // Закрытый метод для проверки правильности
   // написания вида животного
   //…
    }

    protected bool IsValidAnimalType(){
   // Защищенный метод, принимающий решение о
   // существовании указанного вида животного
   //…
        return true;
    }
}
```

```
class Zoo {
  static void Main(string[] args) {
     Animal a = new Animal();
     a.name = "Kangaroo";
     string food;
     bool animalExists;

     // Следующий код будет выполнен успешно,
     //поскольку классу «Zoo» разрешено
     // обращаться к открытым методам в
     //классе «Animal»

     // Вызов открытого метода
     food = a.GetFoodInfo();

     // Обе следующие строки НЕ будут
     //выполнены, поскольку классу «Zoo»
     // не разрешено обращаться к закрытым
     //или защищенным методам

     // Попытка вызова закрытого метода
     a.SpellingCorrect();
     // Попытка вызова защищенного метода
 animalExists = a.IsValidAnimalType();
   }
 }
```

Использование параметров

- При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется **аргументом**.
- Переменная, получающая аргумент, называется формальным параметром, или просто **параметром**.
- Параметры объявляются в скобках после имени метода.
- Областью действия параметров является тело метода.

Пример применения параметра

```
// Простой пример применения параметра.
using System;
class ChkNum {
    // Возвратить значение true, если значение
    // параметра x окажется простым числом
    public bool IsPrime(int x) {
        if(x <= 1) return false;
        for (int i=2; i <= x/i; i++)
            if((x%i) == 0) return false;
        return true;
}   }
class ParamDemo {
  static void Main() {
    ChkNum ob = new ChkNum();
     for (int i = 2; i < 10; i++)
      if (ob.IsPrime(i)) Console.WriteLine(i
                  + " простое число");
      else Console.WriteLine(i + " непростое число)");
}   }
```

Результат:

```
2 простое число
3 простое число
4 непростое число)
5 простое число
6 непростое число)
7 простое число
8 непростое число)
9 непростое число)
```

Пример метода, принимающего 2 аргумента

```
class ChkNum {
    public bool IsPrime(int x) {…}
    // Возвратить наименьший общий множитель
    public int LeastComFactor(int a, int b) {
        int max;
        if (IsPrime(a) || IsPrime(b)) return 1;
        max = a < b ? a : b;
        for (int i = 2; i <= max / 2; i++)
            if (((a % i) == 0) && ((b % i) == 0)) return i;
        return 1;
   } }
class ParmDemo {
  static void Main() {
    ChkNum ob = new ChkNum();
     int a, b;  a = 7; b = 8;
     Console.WriteLine("Наименьший общий множитель чисел " +
    a + " и " + b + " равен " + ob.LeastComFactor(a, b));
     a = 100; b = 8;
     Console.WriteLine("Наименьший общий множитель чисел " +
    a + " и " + b + " равен " + ob.LeastComFactor(a, b));
     a = 100; b = 75;
     Console.WriteLine("Наименьший общий множитель чисел " +
    a + " и " + b + " равен " + ob.LeastComFactor(a, b));
} }
```

Результат:

```
Наименьший общий множитель чисел 7 и 8 равен 1

Наименьший общий множитель чисел 100 и 8 равен 2

Наименьший общий множитель чисел 100 и 75 равен 5
```

Добавление параметризированного метода в класс Building

- Если в методе используется несколько параметров, то для каждого из них указывается свой тип, отличающийся от других:

```
int MyMeth(int a, double b, float c) { //…
```

- С помощью параметризированного метода можно дополнить класс Building новым средством, позволяющим вычислять максимальное количество жильцов в здании, исходя из определенной величины минимальной площади на одного человека. Этим новым средством является приведенный ниже метод MaxOccupant ():

```
public int MaxOccupant(int minArea) {
  return Area / minArea;
}
```

Исключение недоступного кода

- Если создать метод, содержащий недоступный код, компилятор выдаст предупреждающее сообщение соответствующего содержания.
- Рассмотрим следующий пример кода.

```
public void MyMeth() {
  char a, b;
  // ...
  if(a==b) {
    Console.WriteLine("равно");
    return;
  } else {
    Console.WriteLine("не равно") ;
    return;
  }
  Console.WriteLine("это недоступный код");
}
```

Конструкторы

- В приведенных выше примерах программ переменные экземпляра каждого объекта типа Building приходилось инициализировать вручную, используя, в частности, следующую последовательность операторов.

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

- **Конструктор** инициализирует объект при его создании.
- У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу.

Конструкторы

- В приведенных выше примерах программ переменные экземпляра каждого объекта типа Building приходилось инициализировать вручную, используя, в частности, следующую последовательность операторов.

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

- **Конструктор** инициализирует объект при его создании.
- У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу.
- У конструкторов нет возвращаемого типа, указываемого явно.
- Общая форма конструктора:

```
доступ имя_класса (список_параметров) {
// тело конструктора
}
```

- У всех классов имеются конструкторы. Для большинства типов данных значением по умолчанию является нулевое, для типа bool — значение false, а для ссылочных типов — пустое значение.

Пример применения конструктора

```
// Простой конструктор.
using System;
class MyClass {
    public int x;
    public MyClass() {
        x = 10;
    }
}
class ConsDemo {
    static void Main() {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        Console.WriteLine(t1.x + " " + t2.x);
    }
}
```

Результат:

```
10 10
```

Параметризованные конструкторы

```
// Параметризированный конструктор.
using System;
class MyClass {
    public int x;
    public MyClass(int i) {
        x = i;
    }
}
class ParmConsDemo {
    static void Main() {
        MyClass t1 = new MyClass(10);
        MyClass t2 = new MyClass(88);
        Console.WriteLine(t1.x + " " + t2.x);
    }
}
```

Результат:

```
10 88
```

Добавление конструктора в класс Building

```
class Building {
    public int Floors; // количество этажей
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов
    // Параметризированный конструктор для класса Building,
    public Building(int f, int a, int o) {
        Floors = f;
        Area = a;
        Occupants = o;
    }
    public int AreaPerPerson() {/*…*/}
    public int MaxOccupant(int minArea) {/*…*/}
}
class BuildingDemo {
        static void Main() {
            Building house = new Building(2, 2500, 4);
            Building office = new Building(3, 4200, 25);
            //...
}       }
```

Оператор new

- Общая форма:

```
new имя_класса (список_аргументов)
```

где имя_класса обозначает имя класса, реализуемого в виде экземпляра его объекта.

- Можно создавать объекты для числовых типов (int, double и т.д.)
  Т.к. основные типы данных (например int или char) не преобразуются в ссылочные типы, существенно повышается производительность программы.
- Оператор new разрешается использовать вместе с типами значений:

```
using System;
class newValue {
  static void Main() {
    int i = new int(); // инициализировать переменную i нулевым значением
    Console.WriteLine("Значение переменной i равно: " + i);
} }
```

- Результат:

```
Значение переменной i равно: 0
```

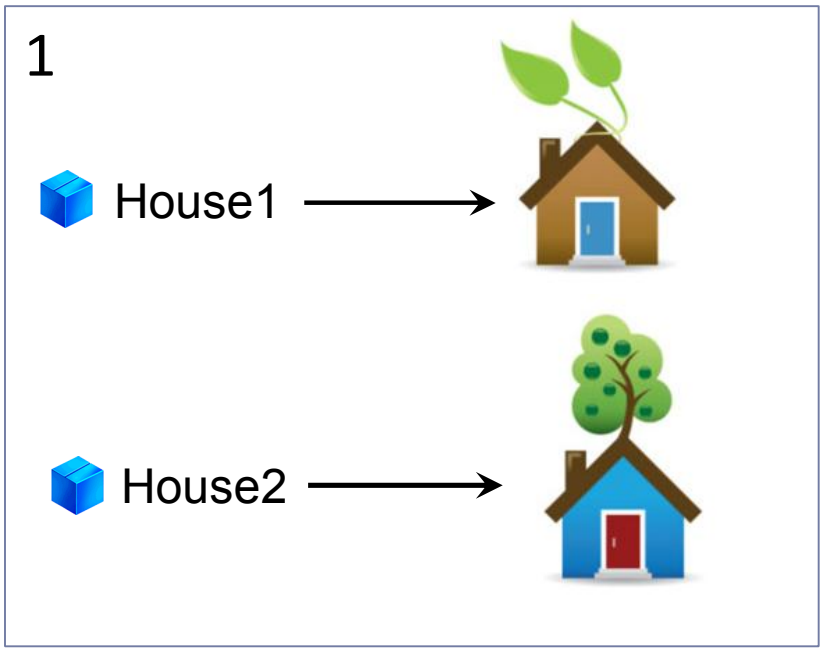
Сборка мусора и деструкторы

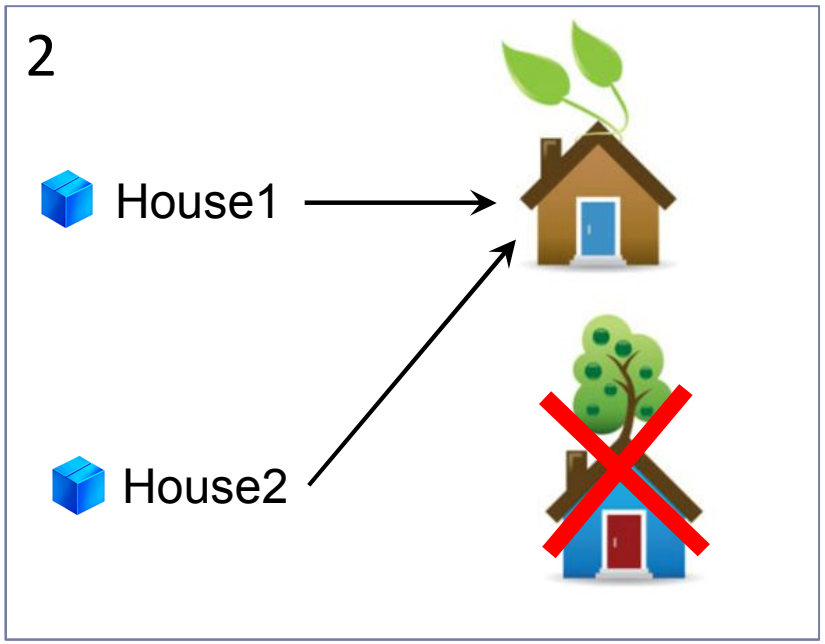
- **"Сборка мусора"** происходит следующим образом: если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается и накапливается. Эта утилизированная память может быть затем распределена для других объектов.
- "Сборка мусора" происходит лишь время от времени по ходу выполнения программы.
- В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется **деструктором** и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта.
- Общая форма деструктора:

```
~имя_класса() {
  // код деструктора
}
```

где имя_класса означает имя конкретного класса.

Сборка мусора и деструкторы





Применение деструктора

```
using System;
class Destruct {
    public int x;
    public Destruct(int i) { x= i;   }
     // Вызывается при утилизации объекта.
    ~Destruct() { Console.WriteLine("Уничтожен объект № " + x);   }
    // Создает объект и тут же уничтожает его.
    public void Generator(int i) { Destruct o = new Destruct(i); }
}
class DestructDemo {
    static void Main() {
        int count;
        Destruct ob = new Destruct(38);
        /* А теперь создадим большое число объектов.
           В какой-то момент произойдет "сборка мусора".
        */
        for (count = 1; count < 100000; count++)
            ob.Generator(count);
        Console.WriteLine("Готово!");
    }
}
```

Ключевое слово this

```
using System;
class Rect {
    public int Width;
    public int Height;
    public Rect(int w, int h) {
        Width = w;
        Height = h;
    }
    public int Area() {
        return Width * Height;
    }
}
class UseRect {
    static void Main() {
        Rect r1 = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);
        Console.WriteLine("Площадь "
+ "прямоугольника r1: " + r1.Area());
        Console.WriteLine("Площадь "
+ "прямоугольника r2: " + r2.Area());
}   }
```

```
using System;
class Rect {
  public int Width;
  public int Height;
  public Rect(int Width, int Height) {
     this.Width = Width;
     this.Height = Height;
  }
  public int Area() {
     return this.Width * this.Height;
  }
}
class UseRect {
  static void Main() {
     Rect r1 = new Rect(4, 5);
     Rect r2 = new Rect(7, 9);
     Console.WriteLine("Площадь " +
"прямоугольника r1: " + r1.Area());
     Console.WriteLine("Площадь " +
"прямоугольника r2: " + r2.Area());
}  }
```