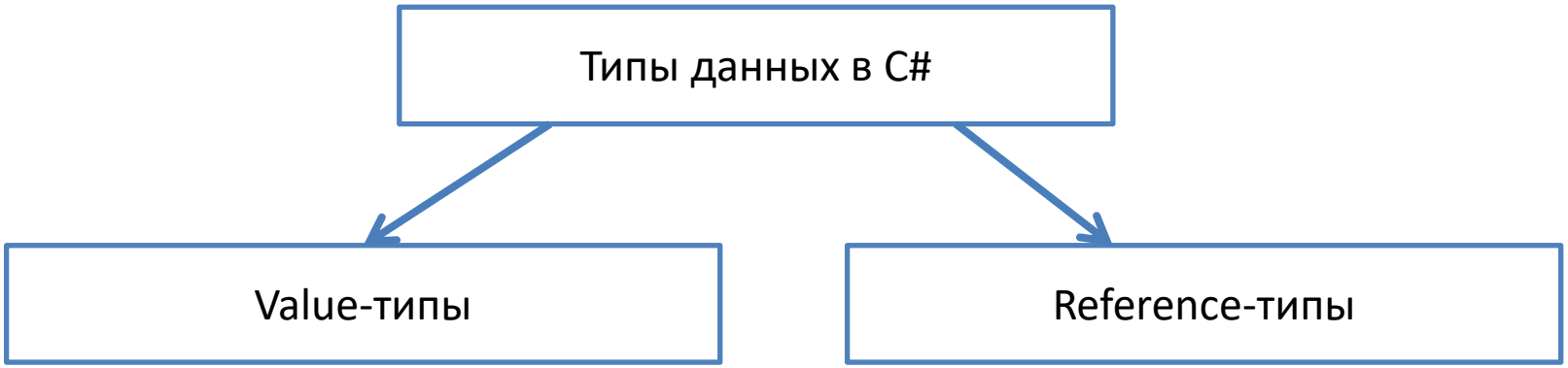
Лекция 2.
Value и reference-типы.
Статический конструктор
Enum’ы.
Перегрузка методов

Типы данных в C#

- Все типы в C# можно разделить на две категории: **value-типы (типы значений)** и **reference-типы (ссылочные типы)**
- Типы из данных категорий ведут себя по-разному
- К value-типам относятся, например числа, а к reference-типам относятся строки

Типы данных в C#



Структуры (объявлены как struct):

- Все числовые типы
- bool, char
- DateTime, TimeSpan
- Nullable-типы

Енумы (enum)

Классы (объявлены как class):

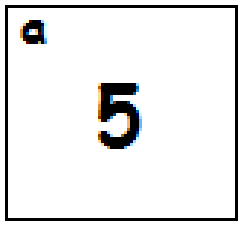
- string
- И другие типы

Value-типы

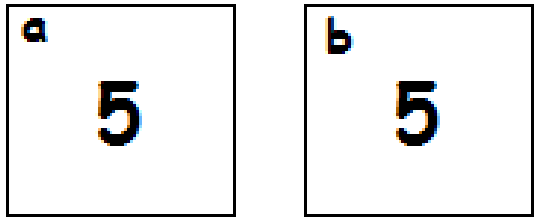
- Переменные value-типов хранят само значение типа
- При присваивании происходит копирование значения
- При передаче аргументов в функции, происходит копирование аргумента

Как работают value-типы

- int a = 5;



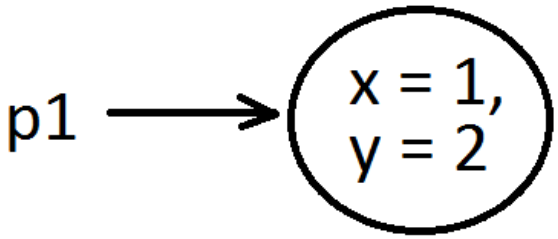
- int b = a;



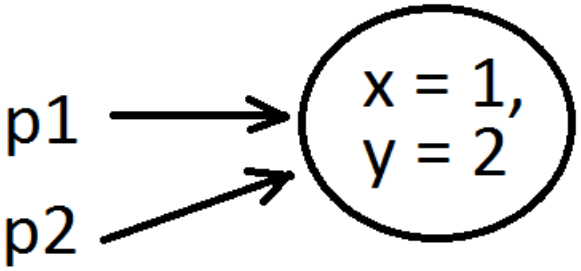
- Если изменить a или b, то это не повлияет на другую переменную

Reference-типы

- Переменные хранят не само значение, а **ссылку** на него (по сути – адрес в памяти)
- `Point p1 = new Point(1, 2);`



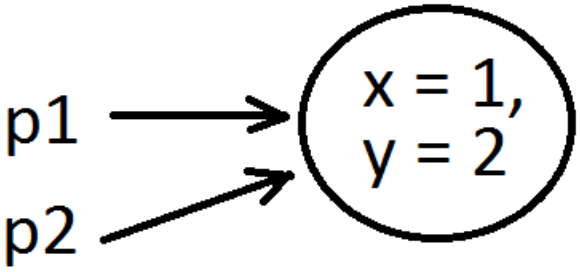
- При присваивании происходит копирование ссылки:
- `Point p2 = p1;`



- В Java все классы являются reference-типами

Reference-типы

- Если изменим объект,
  то все ссылки будут
  указывать на измененный объект



- p2.X = 3;
  Console.WriteLine(p1.X); // 3
- При передаче объекта в функцию, происходит копирование ссылки на него
- Зачем нужны ссылки? Чтобы более эффективно работать с памятью. Объекты часто являются большими, и копировать их очень затратно по времени и памяти

Проверка объектов на равенство

- Для объектов нельзя использовать проверку через == и !=, если эти операторы не переопределены
- Для объектов оператор == проверяет, что ссылки указывают на один и тот же объект в памяти или нет
- Аналогично != проверяет, что ссылки указывают на разные объекты
- Чтобы сравнить содержимое объектов, нужно использовать метод Equals
- bool x = o1.Equals(o2)

Значение null

- Переменные ссылочных типов могут принимать специальное значение null
- Пример: string s = null;
- Оно означает пустую ссылку, то есть адрес, который никуда не указывает
- Если вызвать функцию для переменной, которая имеет значение null, то произойдет ошибка NullReferenceException

Для чего полезен null?

- Значение null может быть полезно, если мы хотим показать, что функция отработала, но получить результат не удалось
- Например, мы написали функцию, которая ищет строку нужной длины среди некоторого набора строк
- Но такой строки не оказалось
- В этом случае функция может вернуть null, а вызывающий код проверить, что результат равен null и, например, напечатать сообщение, что ничего не найдено

Для чего полезен null?

```
public static string FindString(int length)
{
    // код, который делает return, если нашел строку

    // в конце делается return null если
    // ничего не найдено
    return null;
}
public static void Main()
{
    if (FindString(4) == null)
    {
        Console.WriteLine("Ничего не найдено");
    }
}
```

Структуры

Дисклэймер к этому разделу

- В этом месте лекции хочется собрать всю важную информацию по структурам
- Поэтому тут будут использоваться многие термины из следующих лекций – **reflection**, **наследование, методы Equals, GetHashCode** и др.
- Сейчас постарайтесь понять основную суть, а потом после прохождения отдельных тем можно будет перечитать этот раздел, и все будет понятно

Структуры

- **Структура** в C# - это класс, который является value-типом
- Структуры объявляются так же, как классы, но вместо слово class используется слово struct
- ```
  public struct Point {
      private double x;
      private double y;

      /* конструкторы, методы и т.д. */
  }
  ```
- Структуры могут содержать поля, константы, методы, конструкторы, свойства, события
- Но у структур есть ограничения и особенности по сравнению с классами
  ```

Ограничения структур

- В структурах нельзя объявить свой конструктор без аргументов – он генерируется сам и заполняет поля значениями по умолчанию
- Структуры не могут наследоваться и быть родительским типом
  - Все структуры наследуются от типа ValueType

Особенности структур

- У структур всегда есть автоматически генерируемый конструктор без аргументов – он заполняет поля значениями по умолчанию
- Структуры могут реализовывать интерфейсы
- У структур автоматически правильно определены методы **Equals** и **GetHashCode**
  - Но при этом они будут работать медленно (т.к. стандартная реализация через **Reflection**), поэтому если эти методы нужны, их следует переопределить самим

Зачем нужны структуры?

- Создание объектов в **куче** (**heap**), потом сборка мусора при уничтожении объекта – достаточно дорогие операции
- Структуры, являющиеся локальными переменными и аргументами методов, выделяются на **стеке вызовов**, поэтому они быстро создаются и уничтожаются
- Поэтому в некоторых случаях они могут улучшить производительность
- Но при этом со структурами легко и испортить производительность, если использовать их неправильно

Другие темы лекции

Перегрузка методов

- В одном классе можно создавать методы с одинаковыми именами, но разной **сигнатурой**
- Пример:
-
```
public class Summator
{
    public double Sum(double a, double b)
    {
        return a + b;
    }
    public int Sum(int a, int b)
    {
        return a + b;
    }
}
```

Создание в классе разных методов с одинаковыми именами называется **перегрузкой метода (overloading)**

Сигнатура метода для перегрузки

- В **сигнатуру** для перегрузки входят: название метода, количество, типы и порядок аргументов
  ```
  int F(int a, double b)
  int F(double a, int b)   // имеют разные сигнатуры
  double F(double a)
  ```
- В классе нельзя определить два метода с одинаковой сигнатурой
- **В сигнатуру для перегрузки не входит возвращаемый тип!**
- ```
  int F(double a)
  double F(double a)
  // ошибка, т.к. уже есть метод с такой сигнатурой
  ```
  ```

Полиморфизм

- **Полиморфизм** – свойство, при котором сущности с одинаковым интерфейсом ведут себя по-разному
- Здесь под **интерфейсом** понимается то, что выставляется наружу, то есть то, с чем можно взаимодействовать
- Например, если у класса есть публичные методы и поля, то они и составляют интерфейс класса. Это то, к чему могут обратиться другие
- Перегрузка методов является одним из вариантов полиморфизма – методы с одинаковым именем, а, возможно, и даже одинаковым числом аргументов, ведут себя по-разному в зависимости от порядка и типов переданных аргументов

Методы с произвольным количеством аргументов

- Можно создавать методы, которые принимают любое количество аргументов одного типа
-
```
public static double GetAverage(params double[] numbers)
{
    double sum = 0.0;
    foreach (double e in numbers)
    {
        sum += e;
    }
    return sum / numbers.Length;
}
```
- Как можно вызывать:

```
GetAverage(5);     // 5
GetAverage(2, 4); // 3
```

Переданные аргументы доступны как массив

Методы с произвольным количеством аргументов

- Использовать params в списке параметров метода можно только один раз
- При этом такой параметр обязательно должен быть последним в списке аргументов
- ```
  public static double F(int a, params double[] numbers)
  {
      // OK
  }
  ```
- ```
  public static double F(params double[] numbers, int a)
  {
      // Ошибка компиляции, params double[] numbers
      // должен быть последним параметром
  }
  ```

Методы с произвольным количеством аргументов

- На самом деле params это просто красивый синтаксис для методов, принимающих массив, который позволяет просто перечислить элементы массива при их передаче в метод
- Т.е. это аналогично:
- ```
  public static double GetAverage(double[] numbers)
  {
      // код
  }
  ```
- Если объявить такой метод и с params, то компилятор выдаст ошибку, что метод с аргументом double[] уже объявлен
  ```

Методы с произвольным количеством аргументов

- ```
  public static double GetAverage(params double[] numbers)
  {
      // код
  }
  ```

- Как и в метод, принимающий массив при помощи обычного синтаксиса, в методы с params можно передавать null и обычные массивы:
- GetAverage(); // вызов от пустого массива
- GetAverage(null); // вызов от null
- GetAverage(new double[] { 1.0, 3.0 }); // 2
  ```

Enums (енумы, перечисления)

- **Enum** – это специальный вид класса, который позволяет задать набор допустимых значений-констант
- Enum’ы помогают обеспечить контроль типов по сравнению с обычными статическими константами, т.е. не позволяют передать в метод что-то ещё, кроме значений этого enum’а
- Enum’ы следует использовать практически всегда когда какой-то из параметров метода может принимать значения только из определенного набора

Enums

- Хотим хранить направления движения – Север, Юг, Запад, Восток
- Мы могли бы воспользоваться типом string или int чтобы объявить константы:
- ```
  public class Direction
  {
      public const int South = 0;
      public const int West = 1;
      public const int East = 2;
      public const int North = 3;
  }
  ```
  ```

Enums

- ```
  public class Direction
  {
      public const int South = 0;
      public const int West = 1;
      public const int East = 2;
      public const int North = 3;
  }
  ```

- Если какому-то методу потребуется направление в качестве параметра, то мы объявим функцию так:
- ```
  public void Move(int direction, int offset)
  {
      // код
  }
  ```

Enums

- ```
  public class Direction
  {
      public const int South = 0;
      public const int West = 1;
      public const int East = 2;
      public const int North = 3;
  }
  ```
- ```
  public void Move(int direction, int offset)
  {
  }
  ```
- Проблема состоит в том, что те, кто использует наш код, могут передать в метод не только наши константы, но и просто число 10
- `Move(10, 1000);  // число 10 не является направлением!`

Enums

- public enum Direction
  {
  South, West, East, North
  }
- Теперь метод можно объявить так:
- public void Move(Direction direction, int offset)
  {
  // код
  }
- Туда не могут передать что-то недопустимое, можно передавать только значения enum’а Direction:
- Move(Direction.South, 100);

Элементы enum’а доступны глобально

Enum’ы в C#

- По факту enum в C# - это числовой тип
- enum Direction
  {
  South, West, East, North
  }
- Если объявить enum так, то South будет соответствовать числу 0, West – 1, East – 2, North – 3

Enum’ы в C#

- Значения констант можно задать явно:
- ```
  enum Direction
  {
      South = 0,
      West = 1,
      East = 2,
      North = 3
  }
  ```
- Рекомендую именно так и делать, потому что если кто-то добавит новое значение енума в середину, то последующие элементы перенумеруются
- Ещё крайне рекомендуется всегда иметь константу со значением 0
  ```

Enum’ы в C#

- По умолчанию в основе енума лежит int, но можно указывать и другие числовые типы, если хочется сэкономить память
- ```
  enum Direction : byte
  {
      South = 0,
      West = 1,
      East = 2,
      North = 3
  }
  ```
- В основе этого енума лежит тип byte
  ```

Enum’ы в C#

- Так как енумы – это числа, то между ними и числами можно делать преобразования типов:
- `int directionInt = (int) Direction.West;  // 1`
- `Direction direction = (Direction) 2;  // Direction.East`

- Конечно, делать такое не рекомендуется, но иногда в жизни пригождалось

Печать enum’ов

- Если просто печатать значение енума, то будет печататься имя
- Console.WriteLine(Direction.South); // South

- Если хотим распечатать число, то можно использовать функцию ToString с параметром формата “d”:
- Console.WriteLine(Direction.South.ToString(“d”));

Полезные методы enum’ов

- В классе System.Enum есть множество полезных статических методов для работы с енумами

- Преобразование строки в енум
- Direction d = (Direction)Enum.Parse(typeof(Direction), “South”);

- Оператор typeof(НазваниеТипа) – выдает объект Type для указанного типа
- Объект Type – содержит информацию о типе данных

Енумы-флаги

- В C# есть дополнительная возможность делать енумы-битовые флаги
- Тогда для них будет особая поддержка компилятора и есть полезные функции для работы с ними
- https://msdn.microsoft.com/ru-ru/library/cc138362.aspx

Статический конструктор

- Кроме полей, методов и конструкторов, объявление класса может содержать **статический конструктор**
- **Статический конструктор** – это метод, который выполняется 1 раз перед первым использованием класса, и который может работать только со статическими полями класса
- Обычно используется, чтобы инициализировать статические поля класса

Статический конструктор

- Обычно используется чтобы заполнить статические поля класса

- ```
  public class Currencies
  {
      private static readonly string[] currencies;

      static Currencies()
      {
          currencies = new string[] { “RUB”, “USD” };
          // либо можем загрузить из файла
      }
  }
  ```

Статический конструктор всегда без аргументов и без модификатора видимости
  ```

Статический конструктор

- Когда мы присваиваем статическому полю значение сразу при объявлении поля, то на самом деле оно выполняется в статическом конструкторе

- ```
  public class Currencies
  {
      private static readonly int CONSTANT = 1;
  }
  ```
  ```

Инициализация экземпляра

- Если мы сразу заполняем нестатические поля при объявлении, то этот код копируется в начало всех конструкторов
- ```
  public class MyClass
  {
      private string field = “Hello”;
  }
  ```
- Обычно, так стараются не делать, а все писать в конструкторах, чтобы весь код инициализации экземпляра был в одном месте
  ```

Домашнее задание «Shapes»

- Реализовать задачу Shapes

Домашнее задание «Vector»

- Начать делать Vector