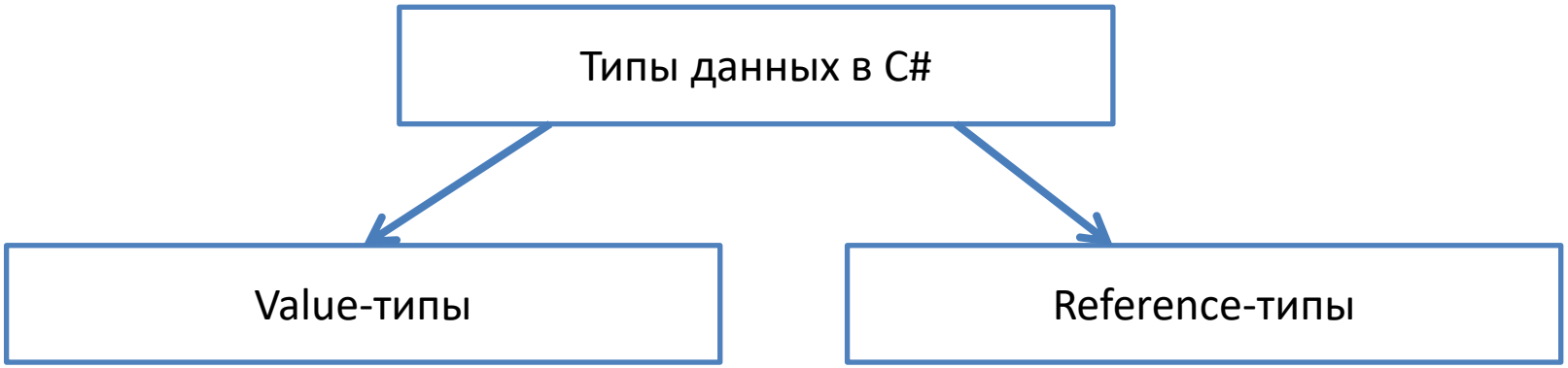
Лекция 8. Value-типы и Ссылочные типы. Символьный тип

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):

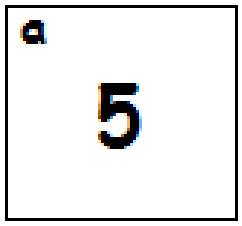
- string
- И другие типы

Value-типы

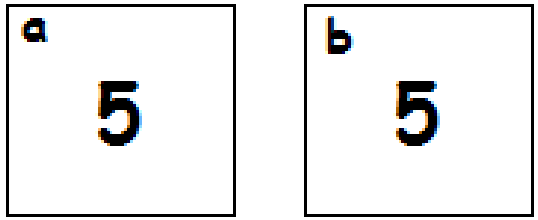
- Переменные **value-типов** хранят само значение типа
- При присваивании происходит копирование значения
- При передаче аргументов в функции, происходит копирование аргумента

Как работают value-типы

- int a = 5;



- int b = a;



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

Reference-типы

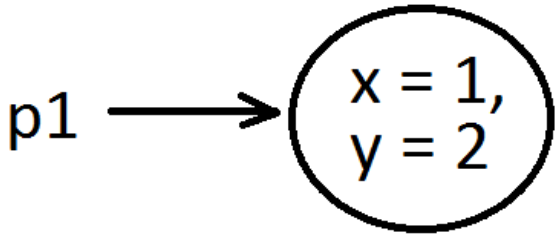
- Переменные **reference-типов** хранят не само значение, а **ссылку** на него
- Point p1 = new Point(1, 2);
- Здесь **p1** хранит не сам объект Point, а только **ссылку** на него
- А сам объект хранится отдельно в области оперативной памяти, называемой **кучей (heap)**
- **Куча (heap)** – это область памяти, выделенная ОС для использования приложением
- Все объекты создаются и хранятся именно в **куче**

Reference-типы

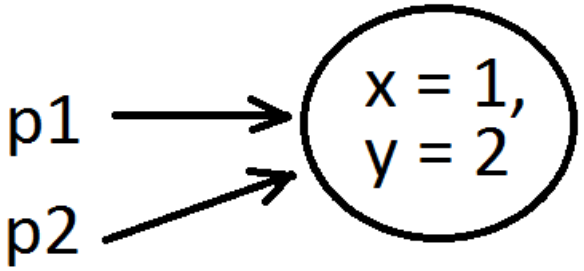
- **Ссылка (reference)** – это некоторая сущность, указывающая на данные, но не хранящая их
- По сути ссылка хранит адрес в памяти (в куче), начиная с которого начинается объект
- Этот адрес является целым числом
- Но у нас нет возможности работать со ссылкой как с числом – это от нас скрыто

Reference-типы

- Переменная **p1** хранит ссылку на объект:
- Point p1 = new Point(1, 2);



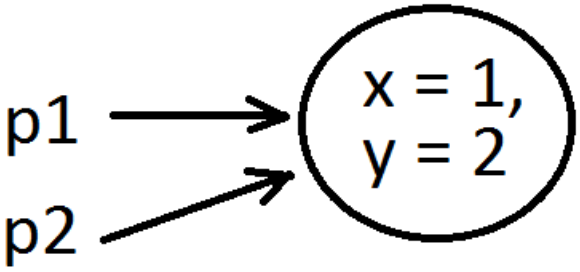
- При присваивании происходит копирование ссылки:
- Point p2 = p1;



- Т.е. получается, что к одному и тому же объекту можно обращаться по двум именам - **p1** и **p2**

Reference-типы

- Т.к. эти ссылки указывают на один и тот же объект, то мы получим измененное значение:



- p2.X = 3;
  Console.WriteLine(p1.X); // 3

- При передаче объекта в функцию, происходит копирование ссылки на него
- Аналогично при возврате объекта из функции

Зачем нужны ссылки?

- Объекты часто являются большими, и копировать их очень затратно по времени и памяти
- При использовании ссылок копируется только ссылка, которая по сути является целым числом. Это быстро и требует мало памяти

Структуры

- Если в этом же примере сделать Point структурой (при объявлении типа указать struct вместо class), а не классом, то результат будет такой:
- ```
  Point p1 = new Point(1, 2);
  Point p2 = p1;  // делается копия значения
  p2.X = 3;
  Console.WriteLine(p1.X);    // 1
  ```
- Здесь **p1** и **p2** независимы друг от друга – при присваивании создалась копия структуры
  ```

Указатели

- В некоторых языках, например, в C и C++, есть понятие **указатель (pointer)**
- **Указатель** – это переменная, которая может хранить адрес в памяти (целое число)
- В C# есть **указатели**, но их не используют в большинстве программ. Вместо них используются **ссылки**
- С указателями, в отличие от ссылок, можно работать как с числом
- Например, если к указателю прибавить 4, то мы сдвинемся на 4 байта и т.д.

Проверка объектов на равенство

- Для объектов нельзя использовать проверку через == и != для проверки равенства, 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);

Логика Equals по умолчанию

- Чтобы **Equals** работал для наших классов, нам нужно самим реализовать этот метод
- А по умолчанию **Equals** работает так же, как сравнение ссылок
- Переопределение **Equals** будет рассмотрено на курсе ООП
- У многих стандартных классов, например, у строк, **Equals** уже реализован правильно

Почему == и != работают для строк

- В C# есть возможность переопределять операторы
- Т.е. можно сделать так, чтобы при использовании некоторого оператора (например, ==) вызывалась некоторая наша функция
- Для строк в C# сделано так, что оператор == работает как вызов **Equals**, а != работает как отрицание от **Equals**
- Но для большинства стандартных типов операторы не переопределены, поэтому **Equals** – более общий вариант

Значение null

Значение null

- Переменные ссылочных типов могут принимать специальное значение null
- Пример: string s = null;
- Оно означает пустую ссылку, то есть адрес, который не указывает ни на какой объект

Падение кода при обращении к null

- Если вызвать функцию (или обратиться к полю) для переменной, которая имеет значение null, то произойдет ошибка NullReferenceException во время работы программы
- string s = null;
  int length = s.Length; // код упадет
- Код падает, т.к. объекта нет – здесь не у кого обращаться к свойству **Length**

Для чего используется null?

- Значение null используется, чтобы показать отсутствие данных
- Например, у нас есть некоторый объект Person, у которого есть поле типа Cat
- Но у некоторого человека может не быть кошки, тогда значение поля у него будет null

Для чего используется null?

- Также 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 используется 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() {
    string result = FindString(4);
    if (result == null) {
        Console.WriteLine("Ничего не найдено");
    } else {
        Console.WriteLine("Результат = " + result);
    }
}
```

Проверка на null

- Чтобы код не падал, нужно обязательно проверять на null в тех местах, где он может быть
-
```
public static void Main() {
    string result = FindString(4);

    if (result == null) {
        Console.WriteLine("Ничего не найдено");
    } else {
        Console.WriteLine("Результат = " + result);
    }
}
```
- Проверять на null нужно через == и !=
- Код через **Equals** может падать, если вызывать его от null

Символьный тип char

Символьный тип char

- Кроме строкового типа, в C# есть символьный тип char
- Это value-тип
- Размер переменной – 2 байта
- Его переменные могут хранить один символ
- Литералы заключены в одинарные кавычки: ‘a’, ‘5’, ‘\\’, ‘\n’
- char lineSeparator = ‘\n’;

Символьный тип char

- У строк можно брать символ по порядковому номеру (отсчитывается от 0)
- `string s = "ABCDE";`
- `char secondSymbol = s[1];  // B`
- `char lastSymbol = s[5];`
  `// ошибка при исполнении программы –`
  `// выход за границы строки`
- Правильно:
  `char lastSymbol = s[s.Length – 1];`

Функции для работы с символами

- Статические методы типа char:
  - bool IsDigit(char c) – проверка что цифра
  - bool IsLetter(char c) – проверка что буква
  - bool IsLetterOrDigit(char c) – что буква или цифра
  - bool IsLower(char c) – что буква в нижнем регистре
  - bool IsUpper(char c) – что буква в верхнем регистре

- Пример:

```
bool isDigit = char.IsDigit('4');   // true
```

Пробельные символы

- bool char.IsWhiteSpace(char c) – проверка, что это пробельный символ
- **Пробельными символами** считаются пробел, табуляция и перевод строки

Функции работы с символами

- Статические методы типа char:
  - char ToUpper(char c) – перевод в верхний регистр
  - char ToLower(char c) – перевод в нижний регистр

- Если символ уже в этом регистре, или не буква, то выдается сам символ
- **Пример:**
- ```
  char lowerCaseChar1 = char.ToLower('A');   // a
  char lowerCaseChar2 = char.ToLower('a');   // a
  char upperCaseChar1 = char.ToUpper('A');   // A
  char upperCaseChar2 = char.ToUpper('a');   // A
  ```
  ```

Пример работы со строками

```
string name = Console.ReadLine();
if (char.IsLower(name[0]))
{
    Console.WriteLine(
      “Имя должно начинаться с заглавной буквы!”);
    return;
}
```

Проход по всем символам строки

- ```
  string s = Console.ReadLine();

  for (int i = 0; i < s.Length; ++i)
  {
      char c = s[i];
      // работаем с текущим символом c
  }
  ```
  ```

Задача на дом «Подсчет символов»

- Прочитать с консоли строку
- Вывести число букв в этой строке
- Вывести число цифр в этой строке
- Вывести число пробельных символов в этой строке
- Вывести число остальных символов в строке

Задача на курс «Макс. подстрока»

- Написать функцию, которая ищет в строке подстроку максимальной длины, состоящую из одного и того же символа, и выдает эту максимальную длину
- Например, есть строка "аааббдеггггв", должно выдаться число 4, потому что есть 4 подряд символа «г», и это максимальная подстрока, где подряд идет один и тот же символ
- Функция должна работать без учета регистра

Задача на курс «Палиндром»

- Объявить некоторую строковую переменную в программе
- Проверить, что данная строка является палиндромом – то есть читается одинаково слева направо и справа налево.
- При проверке не учитывать регистр символов, учитывать только буквы
- Пример палиндрома: «Аргентина манит негра»
- **Требование:** сделать без создания новой строки и без удаления символов из строки