Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 10. Основные конструкции языков Java и C#

Аннотация

Рассматриваются базовые элементы технологий Java и .NET и основные конструкции языков Java и C#. Рассказывается о лексике, базовых типах, выражениях и инструкциях обоих языков, а также о правилах описания пользовательских типов.

Ключевые слова

Java, .NET, J2SE, J2EE, C#, Unicode, пакет, пространство имен, примитивный тип, выражение, инструкция, исключение, ссылочный тип, тип значений, класс, интерфейс, массив, перечислимый тип, делегатный тип.

Текст лекции

Платформы Java и .NET

На данный момент наиболее активно развиваются две конкурирующие линии технологий создания ПО на основе компонентов — технологии Java и .NET. В этой и следующих лекциях мы рассмотрим несколько элементов этих технологий, являющихся ключевыми в создании широко востребованного в настоящее время и достаточно сложного вида приложений. Это Web-приложения, т.е. распределенное программное обеспечение, использующее базовую инфраструктуру Интернет для связи между различными своими компонентами, а стандартные инструменты для навигации по Web — браузеры — как основу для своего пользовательского интерфейса.

Технологии Java представляют собой набор стандартов, инструментов и библиотек, предназначенных для разработки приложений разных типов и связанных друг с другом использованием языка программирования Java. Торговая марка Java принадлежит компании Sun Microsystems, и эта компания во многом определяет развитие технологий Java, но в нем активно участвуют и другие игроки — IBM, Intel, Oracle, Hewlett-Packard, SAP, Bea и пр.

В этот набор входят следующие основные элементы.

- Платформа *Java Platform Standard Edition (J2SE)* [1].
  Предназначена для разработки обычных, в основном, однопользовательских приложений.
- Платформа *Java Platform Enterprise Edition (J2EE)* [2].
  Предназначена для разработки распределенных Web-приложений уровня предприятия.
- Платформа *Java Platform Micro Edition (J2ME)* [3].
  Предназначена для разработки встроенных приложений, работающих на ограниченных ресурсах, в основном, в мобильных телефонах и компьютеризированных бытовых устройствах.
- Платформа *Java Card* [4].
  Предназначена для разработки ПО, управляющего функционированием цифровых карт. Ресурсы, имеющиеся в распоряжении такого ПО, ограничены в наибольшей степени.

С некоторыми оговорками можно считать, что J2ME является подмножеством J2SE, а та, в свою очередь, подмножеством J2EE. Java Card представляет собой, по существу, особый набор средств разработки, связанный с остальными платформами только поддержкой (в сильно урезанном виде) языка Java.

Язык Java — это объектно-ориентированный язык программирования, который транслируется не непосредственно в машинно-зависимый код, а в так называемый *байт-код*, исполняемый

специальным интерпретатором, *виртуальной Java машиной (Java Virtual Machine, JVM)*. Такая организация работы Java-программ позволяет им быть переносимыми без изменений и одинаково работать на разных платформах, если на этих платформах есть реализация JVM, соответствующая опубликованным спецификациям виртуальной машины.

Кроме того, интерпретация кода позволяет реализовывать различные политики безопасности для одних и тех же приложений, выполняемых в разных средах, — к каким ресурсам (файлам, устройствам и пр.) приложение может иметь доступ, а к каким нет, можно определять при запуске виртуальной машины. Таким способом можно обеспечить запускаемое пользователем вручную приложение (за вред, причиненный которым, будет отвечать этот пользователь) б***о***льшими правами, чем аплет, загруженный автоматически с какого-то сайта в Интернет.

Режим интерпретации приводит обычно к более низкой производительности программ по сравнению с программами, оттранслированными в машинно-специфический код. Для преодоления этой проблемы JVM может работать в режиме *динамической компиляции (just-in-time-compilation, JIT)*, в котором байт-код на лету компилируется в машинно-зависимый, а часто исполняемые участки кода подвергаются дополнительной оптимизации.

В настоящем курсе мы рассмотрим ряд элементов платформ J2EE и J2SE, имеющих большое значение для разработки Web-приложений. Читателей, интересующихся деталями устройства и использования J2ME и Java Card, мы отсылаем к документации по этим платформам [3] и [4].

.NET [5] представляет собой похожий набор стандартов, инструментов и библиотек, но разработка приложений в рамках .NET возможна с использованием различных языков программирования. Основой .NET являются виртуальная машина для *промежуточного языка (Intermediate Language, IL,* иногда встречается сокращение *MSIL, Microsoft IL)*, в который транслируются все .NET программы, также называемая *общей средой выполнения (Common Language Runtime, CLR)*, и общая библиотека классов (.NET Framework class library), доступная из всех .NET приложений.

Промежуточный язык является полноценным языком программирования, но он не предназначен для использования людьми. Разработка в рамках .NET ведется на одном из языков, для которых имеется транслятор в промежуточный язык —Visual Basic.NET, C++, C#, Java (транслятор Java в .NET называется J#, и он не обеспечивает одинаковой работы программ на Java, оттранслированных в .NET, и выполняемых на JVM) и пр. Однако разные языки достаточно сильно отличаются друг от друга, и чтобы гарантировать возможность из одного языка работать с компонентами, написанными на другом языке, необходимо при разработке этих компонентов придерживаться *общих правил (Common Language Specifications, CLS)*, определяющих, какими конструкциями можно пользоваться во всех .NET языках без потери возможности взаимодействия между результатами. Наиболее близок к промежуточному языку C# — этот язык был специально разработан вместе с платформой .NET.

Некоторым отличием от Java является то, что код на промежуточном языке в .NET не интерпретируется, а всегда выполняется в режиме динамической компиляции (JIT).

Компания Microsoft инициировала разработку платформы .NET и принятие стандартов, описывающих ее отдельные элементы (к сожалению, пока не все), и она же является основным поставщиков реализаций этой платформы и инструментов разработки. Благодаря наличию стандартов возможна независимая реализация .NET (например, такая реализация разработана в рамках проекта Mono [6]), но, в силу молодости платформы и опасений по поводу монопольного влияния Microsoft на ее дальнейшее развитие, реализации .NET не от Microsoft используются достаточно редко.

Прежде чем перейти к более детальному рассмотрению компонентных технологий Java и .NET, ознакомимся с языками, на которых создаются компоненты в их рамках.

Для Java технологий базовым языком является Java, а при изучении правил построения компонентов для .NET мы будем использовать язык C#. Он наиболее удобен при работе в этой среде и наиболее похож на Java.

Оба этих языка сейчас активно развиваются — в сентябре 2004 года вышла версия 1.5, она же была объявлена версией 5 платформы J2SE и языка Java. При переходе между версиями 1.4 и 1.5 Java претерпела наиболее серьезные изменения за всю свою историю (достаточно полное описание этих изменений стало доступно только в середине 2005 года). Во время написания данной лекции готовится выход J2EE 1.5, пока доступной только в виде предварительных спецификаций.

Существенное обновление C# также должно произойти в версии 2.0, выходящей в ноябре 2005 года. Поэтому мы сопоставляем характеристики уже имеющейся на момент написания версии Java 5 с готовящейся к выходу версией C# 2.0. Это представляется обоснованным еще и потому, что эти версии двух языков достаточно близки по набору поддерживаемых ими конструкций.

Данная лекция дает лишь базовую информацию о языках Java и C#, которая достаточна для понимания приводимого далее кода компонентов и общих правил, регламентирующих их разработку и может служить основой для дальнейшего их изучения. Оба языка достаточно сложны, и всем их деталям просто невозможно уделить внимание в рамках двух лекций. Для более глубокого изучения этих языков (особенно необходимого при разработке инструментов для работы с ними) рекомендуется обратиться к соответствующим стандартам [7] и [8] (ссылки приведены на последние версии стандартов на момент написания этой лекции, кроме того некоторые элементы C# 2.0 не вошли в [8], они описываются согласно [9]).

Оба языка имеют мощные выразительные возможности объектно-ориентированных языков последнего поколения, поддерживающих автоматическое управление памятью и работу в многопоточном режиме. Они весьма похожи, но имеют большое число мелких отличий в деталях. Наиболее существенны для построения программ различия, касающиеся наличия в C# невиртуальных методов, возможности объявления и использования пользовательских типов значений и делегатных типов в C# и возможности передачи значений параметров в C# по ссылке. Обсуждение и одновременно сравнение характеристик языков мы будем проводить по следующему плану.

1. Лексика
2. Общая структура программ
3. Базовые типы и операции над ними
4. Инструкции и выражения
5. Пользовательские типы
6. Средства создания многопоточных программ

Общие черты Java и C# описываются далее обычным текстом, а особенности — в колонках.

В левой колонке будут описываться особенности Java.	В правой колонке будут описываться особенности C#.

Лексика

Программы на обоих рассматриваемых языках, C# и Java, могут быть написаны с использованием набора символов Unicode, каждый символ в котором представляется при помощи 16-ти бит. Поскольку последние версии стандарта Unicode [10] определяют более широкое множество символов, включая символы от U+10000 до U+10FFFF (т.е. имеющие коды от 2^{16} до $2^{20}+2^{16}-1$), такие символы представляются в кодировке UTF-16, т.е. двумя 16-битными символами, первый в интервале U+D800–U+DBFF, второй — U+DC00–U+DFFF.

Лексически программы состоят из разделителей строк (символы возврата каретки, перевода строки или их комбинация), комментариев, пустых символов (пробелы и табуляции), идентификаторов, ключевых слов, литералов, операторов и разделительных символов.

В обоих языках можно использовать как однострочный комментарий, начинающийся с символов `//` и продолжающийся до конца строки, так и выделительный, открывающийся символами `/*` и заканчивающийся при помощи `*/`.

Идентификаторы должны начинаться с буквы (символа, который считается буквой в Unicode, или символа _) и продолжаться буквами или цифрами. В качестве символа идентификатора может использоваться последовательность `\uxxxx`, где `x` — символы `0-9`, `a-f` или `A-F`, обозначающая символ Unicode с шестнадцатеричным кодом `xxxx`. Корректными идентификаторами являются, например, `myIdentifier123`, `αρετη_μυσ`, `идентификатор765` (если последние два представлены в Unicode). Ключевые слова устроены также (без возможности использовать Unicode-последовательности в C#), но используются для построения деклараций, инструкций и выражений языка или для обозначения специальных констант.

В Java ключевые слова не могут использоваться в качестве идентификаторов.

Добавив в начало ключевого слова символ `@`, в C# можно получить идентификатор, посимвольно совпадающий с этим ключевым словом. Этот механизм используется для обращения к элементам библиотек .NET, написанным на других языках, в которых могут использоваться такие идентификаторы — `@class`, `@int`, `@public` и пр.

Можно получать идентификаторы, добавляя `@` и в начало идентификатора, но делать это не рекомендуется стандартом языка.

Другой способ получить идентификатор, совпадающий по символам с ключевым словом, — использовать в нем Unicode-последовательность вместо соответствующего символа ASCII.

Кроме того, в C# есть специальные идентификаторы, которые только в некотором контексте используются в качестве ключевых слов. Таковы **`add`**, **`alias`**, **`get`**, **`global`**, **`partial`**, **`remove`**, **`set`**, **`value`**, **`where`**, **`yield`**.

В обоих языках имеется литерал **`null`** для обозначения пустой ссылки на объект, булевские литералы **`true`** и **`false`**, символьные и строковые литералы, целочисленные литералы и литералы, представляющие числа с плавающей точкой.

Символьный литерал, обозначающий отдельный символ, представляется как этот символ, заключенный в одинарные кавычки (или апострофы). Так, например, можно представить символы `'a'`, `'#'`, `'Ы'`. Чтобы представить символы одинарной кавычки, обратного слэша и некоторые другие используются так называемые ESC-последовательности, начинающиеся с обратного слэша — `'\''` (одинарная кавычка), `'\\'` (обратный слэш), `'\"'` (обычная кавычка), `'\n'` (перевод строки), `'\r'` (возврат каретки), `'\t'` (табуляция). Внутри одинарных кавычек можно использовать и Unicode-последовательности, но осторожно — если попытаться представить так, например, символ перевода строки `\u000a`, то, поскольку такие последовательности заменяются соответствующими символами в самом начале лексического анализа, кавычки будут разделены переводом строки, что вызовет ошибку.

В Java можно строить символьные литералы в виде восьмеричных ESC-последовательностей из не более чем трех цифр — `'\010'`, `'\142'`, `'\377'`. Такая последовательность может представлять только символы из интервала U+0000–U+00FF.

В C# можно использовать шестнадцатеричные ESC-последовательности из не более чем четырех цифр для построения символьных литералов. Такая последовательность обозначает Unicode-символ с соответствующим кодом.

Строковые литералы представляются последовательностями символов (за исключением переводов строк) в кавычках. В качестве символов могут использоваться и ESC-последовательности, разрешенные в данном языке. Строковый литерал может быть разбит на несколько частей, между которыми стоят знаки `+`. Значения литералов `"Hello, world"` и `"Hello," + " world"` совпадают.

В C# можно строить *буквальные строковые литералы (verbatim string literals)*, в которых ESC-последовательности и Unicode-последовательности не преобразуются в их значения. Для этого нужно перед открывающей кавычкой поставить знак `@`. В такой строке могут встречаться любые символы, кроме `"`. Чтобы поместить туда и кавычку, надо повторить ее два раза.

Например, `"Hello \t world"` отличается от `@"Hello \t world"`, а `"\""` совпадает с `@""""`.

Целочисленные литералы представляют собой последовательности цифр, быть может, со знаком — `1234`, `-7654`. Имеются обычные десятичные литералы и шестнадцатеричные, начинающиеся с `0x` или `0X`. По умолчанию целочисленные литералы относятся к типу **`int`**. Целочисленные литералы, имеющие тип длинного целого числа **`long`**, оканчиваются на букву `l` или `L`.

В Java имеются также восьмеричные целочисленные литералы, которые начинаются с цифры `0`.

В C#, в отличие от Java, имеются беззнаковые целочисленные типы **`uint`** и **`ulong`**. Литералы этих типов оканчиваются на буквы `u` или `U`, и на любую комбинацию букв `u`/`U` и `l`/`L`, соответственно.

Литералы, представляющие числа с плавающей точкой, могут быть представлены в обычной записи (`3.1415926`) или экспоненциальной (`314.15926e-2` и `0.31415926e1`). По умолчанию такие литералы относятся к типу **`double`**, и могут иметь в конце символ `d` или `D`. Литералы типа `float` оканчиваются буквами `f` или `F`.

В Java литералы с плавающей точкой могут иметь шестнадцатеричное представление с двоичной экспонентой. При этом литерал начинается с `0x` или `0X`, экспонента должна быть обязательно и должна начинаться с буквы `p` или `P`.

В C# есть тип с плавающей точкой **`decimal`** для более точного представления чисел при финансовых расчетах. Литералы этого типа оканчиваются на букву `m` или `M`.

Операторы и разделители обоих языков:

```
(     )     {     }     [     ]     ;     ,     .     :     ?     ~
=     <     >     !     +     -     *     /     %     &     |     ^
==    <=    >=    !=    +=    -=    *=    /=    %=    &=    |=    ^=
&&    ||    ++    --    <<    >>    <<=   >>=
```

Дополнительные операторы Java:

```
>>>   >>>=
```

Дополнительные операторы C#:

```
->    ::    ??
```

В C#, помимо ранее перечисленных лексических конструкций, имеются директивы препроцессора, служащие для управления

компиляцией. Директивы препроцессора не могут находиться внутри кавычек, начинаются со знака # и пишутся в отдельной строке, эта же строка может заканчиваться комментарием.

Директивы #**define** и #**undef** служат для того, чтобы определять и удалять опции для условной компиляции (такая опция может быть произвольным идентификатором, отличным от **true** и **false**).

Директивы #**if**, #**elif**, #**else** и #**endif** служат для того, чтобы вставлять в код и выбрасывать из него некоторые части в зависимости от декларированных с помощью предыдущих директив опций. В качестве условий, проверяемых директивами #**if** и #**elif**, могут использоваться выражения, составленные из опций и констант **true** и **false** при помощи скобок и операций &&, ||, ==, !=.

Например

```
using System;
#define Debug
public class Assert
{
  public void Assert (bool x)
  {
    #if Debug
    if(!x) throw
      new Exception("Assert failed");
    #endif
  }
}
```

Директивы #**error** и #**warning** служат для генерации сообщений об ошибках и предупреждениях, аналогичных таким же сообщениям об ошибках компиляции. В качестве сообщения выдается весь текст, следующий в строке за такой директивой.

Директива #**line** служит для управления механизмом сообщений об ошибках с учетом строк. Вслед за такой директивой в той же строке может следовать число, число и имя файла в кавычках или слово **default**. В первом случае компилятор считает, что строка, следующая после строки с этой директивой, имеет указанный номер, во втором — помимо номера строки в сообщениях изменяется имя файла, в третьем компилятор переключается в режим по умолчанию, забывая об измененных номерах строк.

Директива **#pragma warning**, добавленная в

C# 2.0, служит для включения или отключения предупреждений определенного вида при компиляции. Она используется в виде

```
#pragma warning disable n_1, …, n_k
#pragma warning restore n_1, …, n_k
```

где `n_1, … , n_k` — номера отключаемых/включаемых предупреждений.

Общая структура программы

Программа на любом из двух рассматриваемых языков представляет собой набор пользовательских типов данных — в основном, классов и интерфейсов, с их методами. При запуске программы выполняется определенный метод некоторого типа. В ходе работы программы создаются объекты различных типов и выполняются их методы (операции над ними). Объектами особого типа представляются различные потоки выполнения, которые могут быть запущены параллельно.

Во избежание конфликтов по именам и для лучшей структуризации программ пользовательские типы размещаются в специальных пространствах имен, которые в Java называются *пакетами (packages)*, а в C# *пространствами имен (namespaces)*. Имена пакетов и пространств имен могут состоять из нескольких идентификаторов, разделенных точками. Из любого места можно сослаться на некоторый тип, используя его длинное имя, состоящее из имени содержащего его пространства имен или пакета, точки и имени самого типа.

В обоих случаях программный код компилируется в бинарный код, исполняемый виртуальной машиной. Правила размещения исходного кода по файлам несколько отличаются.

Код пользовательских типов Java размещается в файлах с расширением .java.

При этом каждый файл относится к тому пакету, чье имя указывается в самом начале файла с помощью декларации **`package`** `mypackage;`

При отсутствии этой декларации код такого файла попадает в пакет с пустым именем.

В одном файле может быть описан только один общедоступный (**`public`**) пользовательский тип верхнего уровня (т.е. не вложенный в описание другого типа), причем имя этого типа должно совпадать с именем файла без расширения.

В том же файле может быть декларировано сколько угодно необщедоступных типов.

Код пользовательских типов C# размещается в файлах с расширением .cs.

Декларация пространства имен начинается с конструкции **`namespace`** `mynamespace {` и заканчивается закрывающей фигурной скобкой. Все типы, описанные в этих фигурных скобках, попадают в это пространство имен. Типы, описанные вне декларации пространства имен, попадают в пространство имен с пустым именем.

Пространства имен могут быть вложены в другие пространства имен. При этом следующие декларации дают эквивалентные результаты.

```
namespace A.B { … }
namespace A
{
  namespace B { … }
}
```

В одном файле можно декларировать много типов, относящихся к разным пространствам имен, элементы одних и тех же пространств имен могут описываться в разных файлах.

Пользовательский тип описывается полностью в одном файле.	Пользовательский тип описывается целиком в одном файле, за исключением *частичных типов* (введены в C# 2.0), помеченных модификатором **`partial`** — их элементы можно описывать в разных файлах, и эти описания объединяются, если не противоречат друг другу.
Чтобы ссылаться на типы, декларированные в других пакетах, по их коротким именам, можно воспользоваться директивами импорта. Если в начале файла после декларации пакета присутствует директива **`import`** `java.util.ArrayList;` то всюду в рамках этого файла можно ссылаться на тип `ArrayList` по его короткому имени. Если же присутствует директива **`import`** `java.util.*;` то в данном файле можно ссылаться на любой тип пакета `java.util` по его короткому имени. Директива **`import static`** `java.lang.Math.cos;` (введена в Java 5) позволяет в рамках файла вызывать статический метод `cos()` класса `java.lang.Math` просто по его имени, без указания имени объемлющего типа. Во всех файлах по умолчанию присутствует директива **`import`** `java.lang.*;` Таким образом, на типы из пакета `java.lang` можно ссылаться по их коротким именам (если, конечно, в файле не декларированы типы с такими же именами — локально декларированные типы всегда имеют преимущество перед внешними).	Чтобы ссылаться на типы, декларированные в других пространствах имен, по их коротким именам, можно воспользоваться директивами использования. Директива **`using`** `System.Collections;` делает возможным ссылки с помощью короткого имени на любой тип (или вложенное пространство имен) пространства имен `System.Collections` в рамках кода пространства имен или типа, содержащего эту директиву или в рамках всего файла, если директива не вложена ни в какое пространство имен. Можно определять новые имена (синонимы или алиасы) для декларированных извне типов и пространств имен. Например, директива **`using`** `Z=System.Collections.ArrayList;` позволяет затем ссылаться на тип `System.Collections.ArrayList` по имени `Z`.
Файлы должны располагаться в файловой системе определенным образом. Выделяется одна или несколько корневых директорий, которые при компиляции указываются в опции `-sourcepath` компилятора. Файлы из пакета без имени должны лежать в одной из корневых директорий. Все остальные должны находиться в поддиректориях этих корневых директорий так, чтобы имя содержащего пакета, к которому файл относится, совпадало бы с именем содержащей сам файл директории относительно включающей ее корневой (с заменой точки на разделитель имен директорий).	Нет никаких ограничений на именование файлов и содержащихся в них типов, а также на расположение файлов в файловой системе и имена декларированных в них пространств имен.
Результаты компиляции располагаются в	Результат компиляции C# программы —

файлах с расширением .class, по одному типу на файл. Хранящие их директории организуются по тому же принципу, что и исходный код, — в соответствии с именами пакетов, начиная от некоторого (возможно другого) набора корневых директорий. Указать компилятору корневую директорию, в которую нужно складывать результаты компиляции, можно с помощью опции -d.

Чтобы эти типы были доступны при компиляции других, корневые директории, содержащие соответвующие им .class файлы, должны быть указаны в опции компилятора `-classpath`.

В этой же опции могут быть указаны архивные файлы с расширением .jar, в которых много .class файлов хранится в соответствии со структурой пакетов.

динамически загружаемая библиотека (с расширением .dll в системе Windows) или исполняемый файл (.exe), имеющие особую структуру. Такие библиотеки называются *сборками (assembly)*.

Для того чтобы использовать типы, находящиеся в некоторой сборке с расширением .dll, достаточно указать ее файл компилятору в качестве внешней библиотеки.

Входной точкой программы является метод

```
public static void main (String[])
```

одного из классов. Его параметр представляет собой массив строк, передаваемых как параметры командной строки при запуске.

При этом полное имя класса, чей метод `main()` выбирается в качестве входной точки, указывается в качестве параметра виртуальной машине при запуске (параметры командной строки следуют за ним).

Входной точкой программы является метод

```
public static void Main ()
```

одного из классов. Такой метод может также иметь параметр типа `string[]` (представляющий параметры командной строки, как и в Java) и/или возвращать значение типа **`int`**.

Класс, чей метод выбирается в качестве входной точки, указывается в качестве стартового класса при сборке исполняемого файла. Собранный таким образом файл всегда будет запускать метод `Main()` указанного класса.

Ниже приведены программы на обоих языках, вычисляющие и печатающие на экране значение факториала неотрицательного целого числа (0! = 1, n! = 1·2·…·n), передаваемого им в качестве первого аргумента командной строки. Также приводятся командные строки для их компиляции и запуска.

В отличие от Java, параметры компилятора и способ запуска программ в C# не стандартизованы. Приведена командная строка для компилятора, входящего в состав Microsoft Visual Studio 2005 Beta 2. Предполагается, что директории, в которых находятся компиляторы, указаны в переменной окружения `$path` или `%PATH%`, и все команды выполняются в той же директории, где располагаются файлы с исходным кодом.

Компилятор Java и Java-машина располагаются в поддиректории `bin` той директории, в которую устанавливается набор для разработки Java Development Kit. Компилятор C# располагается в поддиректории `Microsoft.NET\Framework\v<номер версии установленной среды .NET>` системной директории Windows (обычно `Windows` или `WINNT`).

```
public class Counter
{
  public int factorial(int n)
  {
    if(n == 0) return 1;
```

```
using System;

public class Counter
{
  public int Factorial(int n)
  {
    if (n == 0) return 1;
```

```
    else if(n > 0)
      return n * factorial(n - 1);
    else
      throw new IllegalArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void main(String[] args)
  {
    int n = 2;
    if(args.length > 0)
    {
      try
      {
        n = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e)
      {
        n = 2;
      }
    }

    if(n < 0) n = 2;
    Counter f = new Counter();
    System.out.println(f.factorial(n));
  }
}
```

Компиляция

```
javac Counter.java
```

Выполнение

```
java Counter 5
```

Результат

```
120
```

```
    else if (n > 0)
      return n * Factorial(n - 1);
    else
      throw new ArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void Main(string[] args)
  {
    int n = 2;
    if (args.Length > 0)
    {
      try
      {
        n = Int32.Parse(args[0]);
      }
      catch (Exception)
      {
        n = 2;
      }
    }

    if (n < 0) n = 2;
    Counter f = new Counter();
    Console.WriteLine(f.Factorial(n));
  }
}
```

Компиляция

```
csc.exe Counter.cs
```

Выполнение

```
Counter.exe 5
```

Результат

```
120
```

Базовые типы и операции над ними

В обоих рассматриваемых языках имеются ссылочные типы и типы значений. Объекты ссылочных типов имеют собственную идентичность, на такой объект можно иметь ссылку из другого объекта, они передаются по ссылке, если являются аргументами или результатами методов. Объекты типов значений представляют собой значения, не имеющие собственной идентичности, — все равные между собой значения неотличимы друг от друга, никак нельзя сослаться только на одно из них.

В обоих языках есть примитивные типы, являющиеся типами значений, для представления простых данных: логических, числовых и символьных.

В Java только примитивные типы являются типами значений, все другие типы — ссылочные, являются наследниками класса `java.lang.Object`.

Для каждого примитивного типа есть класс-обертка, который позволяет представлять значения этого типа в виде объектов.

Между значениями примитивного типа и

В C# есть возможность декларировать пользовательские типы значений — структурные типы и перечисления. Ссылочные типы называются классами и интерфейсами. Структурные типы, так же как и ссылочные, наследуют классу `System.Object`, который также можно использовать под именем **`object`**.

Для каждого примитивного типа есть структурный тип-обертка. Преобразования между ними производятся неявно, компилятор считает их различными именами одного и того

объектами соответствующего ему класса-обертки определены преобразования по умолчанию — *упаковка* и *распаковка* (*autoboxing* и *auto-unboxing*, введены в Java 5), позволяющие во многих случаях не создавать объект по значению и не вычислять значение по объекту явно. Но можно производить их и явно.

Однако вызывать методы у значений примитивных типов нельзя.

же типа.

Поэтому все элементы класса **`object`** имеются во всех примитивных типах — у их значений можно, как у обычных объектов, вызывать методы.

Вполне законны, например, выражения `2.Equals(3)` и `(-175).ToString()`.

Логический тип

В Java он назван **`boolean`**, а его обертка — `java.lang.Boolean`.

В C# он назван **`bool`**, а его обертка — `System.Boolean`.

Значения этого типа — логические значения, их всего два — **`true`** и **`false`**. Нет никаких неявных преобразований между логическими и целочисленными значениями. Над значениями этого типа определены следующие операции.

- `==` и `!=` — сравнения на равенство и неравенство.
- `!` — отрицание.
- `&&` и `||` — условные (короткие) конъюнкция и дизъюнкция ('и' и 'или'). Второй аргумент этих операций не вычисляется, если по значению первого уже ясно, чему равно значение выражения, т.е., в случае конъюнкции — если первый аргумент равен **`false`**, а в случае дизъюнкции — если первый аргумент равен **`true`**. С помощью условного оператора `?:` их можно записать так: `(x && y)` — `((x)?(y):false)`, `(x || y)` — `((x)?true:(y))`. Напомним, что означает условный оператор — выражение `a?x:y` вычисляет значение `a`, если оно **`true`**, то вычисляется и возвращается значение `x`, иначе вычисляется и возвращается значение `y`.
- `&` и `|` — (длинные) конъюнкция и дизъюнкция ('и' и 'или'). У этих операций оба аргумента вычисляются всегда.
- `^` — исключающее 'или' или сумма по модулю 2.
- Для операций `&`, `|`, `^` имеются соответствующие операторы присваивания `&=`, `|=`, `^=`. Выражение `x op= y`, где `op` — одна из операций `&`, `|`, `^`, имеет тот же эффект, что и выражение `x = ((x) op (y))`, за исключением того, что значение `x` вычисляется ровно один раз.

Целочисленные типы

В обоих языках имеются следующие целочисленные типы.

- Тип байтовых целых чисел, называемый в Java **`byte`**, а в C# — **`sbyte`**. Его значения лежат между -2^7 и (2^7-1) (т.е. между -128 и 127)
- **`short`**, чьи значения лежат в интервале -2^{15} – $(2^{15}-1)$ (-32768 – 32767)
- **`int`**, чьи значения лежат в интервале -2^{31} – $(2^{31}-1)$ (-2147483648 – 2147483647)
- **`long`**, чьи значения лежат в интервале -2^{63} – $(2^{63}-1)$ (-9223372036854775808 – 9223372036854775807)

В C# имеются беззнаковые аналоги всех перечисленных выше типов:

свой тип **`byte`** со значениями от 0 до $(2^8-1 = 255)$.

`ushort` со значениями от 0 до $(2^{16}-1) = 65535$

uint со значениями от 0 до $(2^{32}-1) = 4294967295$

ulong со значениями от 0 до $(2^{64}-1) = 18446744073709551615$

Классы-обертки целочисленных типов называются так:

```
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
```

Типы-обертки целочисленных типов называются так:

```
System.SByte
System.Byte
System.Int16
System.UInt16
System.Int32
System.UInt32
System.Int64
System.UInt64
```

Минимальные и максимальные значения примитивных типов можно найти в их типах-обертках в виде констант (**static final** полей) `MIN_VALUE` и `MAX_VALUE`.

Минимальные и максимальные значения примитивных типов можно найти в их типах-обертках в виде констант `MinValue` и `MaxValue`.

Над значениями целочисленных типов определены следующие операции.

- `==, !=` — сравнение на равенство и неравенство.
- `<, <=, >, >=` — сравнение на основе порядка.
- `+, -, *, /, %` — сложение, вычитание, умножение, целочисленное деление, взятие остатка по модулю.
- `++, --` — увеличение и уменьшение на единицу. Если такой оператор написан до операнда, то значение всего выражения совпадает с измененным значением операнда, если после — то с неизмененным.
  В результате выполнения последовательности действий
  `x = 1; y = ++x; z = x++;`
  значение `x` станет равно 3, а значения `y` и `z` — 2.
- `~, &, |, ^` — побитовые операции дополнения, конъюнкции, дизъюнкции и исключающего ‘или’.
- `<<, >>` — операторы, сдвигающие биты своего первого операнда влево и вправо на число позиций, равное второму операнду.

В Java оператор >> сдвигает вправо биты числа, дополняя его слева значением знакового бита — нулем для положительных чисел и единицей для отрицательных.

Специальный оператор >>> используется для сдвига вправо с заполнением освобождающихся слева битов нулями.

В C# оператор >> сдвигает вправо биты числа, дополняя его слева для чисел со знаком значением знакового бита, а для беззнаковых чисел — нулем.

- Для операций `+, -, *, /, %, ~, &, |, ^, <<, >>` (и Java-специфичной операции >>>) имеются соответствующие операции присваивания. При этом выражение `x op= y`, где `op` — одна из этих операций, эквивалентно выражению `x = (T)((x) op (y))`, где `T` — тип `x`, за исключением того, что значение `x` вычисляется ровно один раз.

В Java результаты арифметических действий вычисляются в зависимости от типа этих результатов, с отбрасыванием битов, «вылезающих» за размер типа.

В C# результат арифметических действий над целочисленными данными, приводящих к переполнению, зависит от контекста, в котором эти действия производятся.

Таким образом, эти операции реализуют арифметику по модулю 2^n для n, подходящего для данного типа.

Арифметические операции над целыми числами приводят к созданию исключений только в трех случаях: при делении на 0 или вычислении остатка по модулю 0, при конвертации в примитивный тип ссылки на объект класса обертки, равной **`null`**, а также при исчерпании доступной Java-машине памяти, которое может случиться из-за применения операций `--` и `++` с одновременным созданием объектов классов-оберток.

Если действие происходит в unchecked контексте (т.е. внутри блока или выражения, помеченных ключевым словом **`unchecked`**), то вычисления производятся в арифметике по подходящему модулю 2^n.

Если же эти действия производятся в checked контексте (т.е. внутри блока или выражения, помеченных модификатором **`checked`**), то переполнение приводит к созданию исключения.

По умолчанию действия, производимые в ходе выполнения, происходят в unchecked контексте, а действия, которые выполняются над константами во время компиляции — в checked контексте. При этом создание исключения во время компиляции приводит к выдаче сообщения об ошибке.

Любые целочисленные типы можно явно приводить друг к другу, а неявные преобразования переводят из меньших типов в б*о*льшие, если при этом нет перехода от типа со знаком к беззнаковому (обратный переход возможен).

В обоих языках целочисленным типом считается и тип **`char`**, чьими значениями являются 16-битные символы (от `'\u0000'` до `'\uffff'`). Для него определен тот же набор операций, но преобразования между ним и другими типами по умолчанию не производятся (явные преобразования возможны).

Типы чисел с плавающей точкой

Представление типов значений с плавающей точкой, **`float`** и **`double`**, а также операции с ними, соответствуют стандарту на вычисления с плавающей точкой IEEE 754 (он же — IEC 60559) [11,12]. Согласно этому стандарту значение такого типа состоит из знакового бита, мантиссы и экспоненты (у значения **`float`** 23 бита отводятся на мантиссу и 8 на экспоненту, у **`double`** — 52 бита на мантиссу и 11 на экспоненту).

Помимо обычных чисел значения обоих типов включают -0.0 (кстати, написав так, вы получите обычный 0.0, поскольку этот текст будет воспринят как константа 0.0, к которой применен унарный оператор -; единственный способ получить -0.0 — конвертировать его битовое представление — в шестнадцатеричном виде для типа **`float`** он представляется как `0x80000000`, а для **`double`** — `0x8000000000000000`), положительные и отрицательные бесконечности (для типа **`float`** это `0x7f800000` и `0xff800000`, а для **`double`** — `0x7ff0000000000000` и `0xfff0000000000000`), а также специальное значение `NaN` (*Not-A-Number, не число*; оно может быть представлено любыми значениями, у которых экспонента максимальна, а мантисса не равна 0).

Для значений с плавающей точкой определены следующие операции.

- `==, !=` — сравнения на равенство и неравенство. В соответствии с IEEE 754 `NaN` не равно ни одному числу, в том числе самому себе. -0.0 считается равным 0.0.
- `<, <=, >, >=` — сравнения на основе порядка. +∞ больше, чем любой обычное число и -∞, а -∞ меньше любого конечного числа. `NaN` несравнимо ни с одним числом, даже с самим собой — это значит, что любая указанная операция возвращает **`false`**, если один из ее операндов — `NaN`. -0.0 считается равным, а не меньше, чем 0.0.
- `+, -, *, /, %` — сложения, вычитание, умножение, деление, взятие остатка по модулю, а также соответствующие операции присваивания с одновременным выполнением одного из этих действий. Все эти операции действуют согласно IEEE 754, кроме операции

вычисления остатка, которая реализована так, чтобы при всех конечных `a` и `b` (`b != 0`) выполнялось `a%b == a - b*n`, где `n` — самое большое по абсолютной величине целое число, не превосходящее `|a/b|`, знак которого совпадает со знаком `a/b`. По абсолютной величине `a%b` всегда меньше `b`, знак `a%b` совпадает со знаком `a`.
Согласно стандарту IEEE 754 все арифметические операции определены для бесконечных аргументов «естественным» образом: 1.0/0.0 дает +∞, -1.0/0.0 дает -∞, 0.0/0.0 — `NaN`, конечное `x` в сумме с +∞ дает +∞, а +∞+(-∞) — `NaN`. Если один из операндов `NaN`, то результат операции тоже `NaN`.

- `++`, `--` — увеличение и уменьшение на единицу. Для бесконечностей и `NaN` результат применения этих операторов совпадает с операндом.

В Java в классах `java.lang.Float` и `java.lang.Double` есть константы, равные максимальному конечному значению типа, минимальному положительному значению типа, положительной и отрицательной бесконечностям и `NaN`.

`Float.MAX_VALUE` $= (2-2^{-23})\cdot 2^{127}$

`Float.MIN_VALUE` $= 2^{-149}$

`Double.MAX_VALUE` $= (2-2^{-59})\cdot 2^{1023}$

`Double.MIN_VALUE` $= 2^{-1074}$

Бесконечности и `NaN` в обоих случаях называются `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` и `NaN`.

В C# соответствующие классы `System.Single` и `System.Double` также хранят эти значения в виде констант `MaxValue`, `Epsilon`, `PositiveInfinity`, `NegativeInfinity` и `NaN`.

В C# есть еще один тип для представления чисел с плавающей точкой — **`decimal`** (тип-обертка для него называется `System.Decimal`).

Значения этого типа представляются 128 битами, из которых один используется для знака, 96 — для двоичной мантиссы, еще 5 — для представления десятичной экспоненты, лежащей от 0 до 28. Остальные биты не используются.

Представляемое знаком s (+1 или -1), мантиссой m (0–(2^{96}-1)) и экспонентой e (0–28) значение равно $(-1)^{s}\cdot m\cdot 10^{-e}$.

Таким образом, значения этого типа могут, в отличие от стандартных типов **`float`** и **`double`**, представлять десятичные дроби с 28-ю точными знаками и используются для финансовых вычислений. Такая точность необходима, поскольку в этих вычислениях ошибки округления в сотые доли процента при накоплении за счет больших сумм и большого количества транзакций в течение нескольких лет могут привести к значительным суммам убытков для разных сторон.

Для типа **`decimal`** определены все те же операции, что и для типов с плавающей точкой, однако при выходе их результатов за рамки

значений типа создается исключительная ситуация. Этот тип не имеет специальных значений -0.0, `NaN` и бесконечностей.

В Java классы, методы и инициализаторы могут быть помечены модификатором **`strictfp`**. Он означает, что при вычислениях с плавающей точкой в рамках этих деклараций все промежуточные результаты должны быть представлены в рамках того типа, к которому они относятся, согласно стандарту IEEE 754.

Иначе, промежуточные результаты вычислений со значениями типа **`float`** могут быть представлены в более точном виде, что может привести к отличающимся итоговым результатам вычислений.

Инструкции и выражения

Выражения

В обоих языках выражения строятся при помощи применения операторов к именам и литералам. Условно можно считать, что имеется следующий общий набор операторов.

- `x.y` — оператор уточнения имени, служит для получения ссылки на элемент пространства имен или типа, либо для получения значения поля (или свойства в C#);
- `f(x)` — оператор вызова метода (а также делегата в C#) с заданным набором аргументов;
- `a[x]` — оператор вычисления элемента массива (а также обращения к индексеру в C#);
- **`new`** — оператор создания нового объекта (или значения в C#), используется вместе с обращением к одному из конструкторов типа — **`new`** `MyType("Yes", 2)` (в Java с его помощью нельзя создавать значения примитивных типов);
- `++`, `--` — префиксные и постфиксные унарные операторы увеличения/уменьшения на 1;
- `(T)x` — оператор явного приведения к типу T;
- `+`, `-` — унарные операторы сохранения/изменения знака числа;
- `!` — унарный оператор логического отрицания;
- `~` — унарный оператор побитового отрицания;
- `*`, `/`, `%`, `+`, `-` — бинарные операторы умножения, деления, взятия остатка по модулю, сложения и вычитания;
- `<<`, `>>` — бинарные операторы побитовых сдвигов влево/вправо;
- `<`, `>`, `<=`, `>=` — бинарные операторы сравнения по порядку;
- `==`, `!=` — бинарные операторы сравнения на равенство/неравенство;
- `&`, `|`, `^` — бинарные операторы логических или побитовых операций: конъюнкции, дизъюнкции, сложения по модулю 2;
- `&&`, `||` — бинарные операторы условных конъюнкции и дизъюнкции, `(x && y)` эквивалентно `(x?y:false)`, а `(x || y)` — `(x?true:y)`;
- `?:` — тернарный условный оператор, выражение `a?x:y` вычисляет значение `a`, если оно **`true`**, то вычисляется и возвращается значение `x`, иначе вычисляется и возвращается значение `y`;

- `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=` — бинарные операторы присваивания, все они, кроме первого, сначала производят некоторую операцию над старым значением левого операнда и значением правого, а затем присваивают полученный результат левому операнду.

Операторы	**Ассоциативность**
`x.y`, `f(x)`, `a[x]`, **`new`**, `x++`, `x--`	
`+`, `-`, `!`, `~`, `++x`, `--x`, `(T)x`	
`*`, `/`, `%`	левая
`+`, `-`	левая
`<<`, `>>`	левая
`<`, `>`, `<=`, `>=`	левая
`==`, `!=`	левая
`&`	левая
`^`	левая
`|`	левая
`&&`	левая
`||`	левая
`?:`	правая
`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=`	правая

Таблица 10. Приоритет и ассоциативность операторов.

В Таблице 10 операторы перечисляются сверху вниз в порядке уменьшения их приоритета, а также приводится ассоциативность всех операторов. Оператор `op` называется ***левоассоциативным***, если выражение `(x op y op z)` трактуется компилятором как `((x op y) op z)`, и ***правоассоциативным***, если оно трактуется как `(x op (y op z))`.

Помимо перечисленных выше операторов имеются также общие для обоих языков операции, которые выполняются при помощи различных конструкций — это получение объекта, представляющего тип, который задан по имени, и проверка принадлежности объекта или значения типу. В каждом из языков есть также несколько операторов, специфических для данного языка.

Получение объекта, представляющего тип, связано с механизмом ***рефлексии (reflection)***, имеющимся в обоих языках. Этот механизм обеспечивает отображение сущностей языка (типов, операций над ними, полей их данных и пр.) в объекты самого языка. В обоих языках операция получения объекта, представляющего тип, входит в группу операций с высшим приоритетом.

Любой тип Java однозначно соответствует некоторому объекту класса `java.lang.Class`, любой метод описывается с помощью одного из объектов класса `java.lang.reflect.Method`, любое поле — с помощью одного из объектов класса `java.lang.reflect.Field`.

Получить объект типа `Class`, представляющий тип `T` (даже если `T` = **`void`**), можно с помощью конструкции `T.`**`class`**.

Для проверки того, что выражение `x` имеет тип `T`, в Java используется конструкция `(x` **`instanceof`** `T)`, возвращающая значение логического типа.

В C# типы представляются объектами класса `System.Type`, методы — объектами `System.Reflection.MethodInfo`, а поля — объектами `System.Reflection.FieldInfo`.

Объект типа `System.Type`, представляющий тип `T`, можно получить при помощи конструкции **`typeof`**`(T)`.

Для проверки того, что выражение `x` имеет тип `T`, в C# используется конструкция `(x` **`is`** `T)`, имеющая логический тип.

В обоих языках операция проверки типа имеет такой же приоритет, как операторы <, >, <=, >=.

Эта проверка использует естественные преобразования типов (подтипа в более общий тип или наоборот, если точный тип объекта является подтипом `T`) и автоупаковку/распаковку, не затрагивая определенных пользователем неявных преобразований.

В C# имеется и другой оператор, связанный с преобразованием типа.

Для преобразования объекта `x` к заданному ссылочному типу `T` можно использовать конструкцию
`(x` **`as`** `T)`,

тип результата которой — `T`.

Если в результате естественных преобразований типов и автоупаковки/распаковки, значение `x` не преобразуется к типу `T`, то результат этого выражения — **`null`**.

Приоритет этого оператора такой же, как у оператора **`is`**, а ассоциативность — левая.

В Java есть дополнительный оператор сдвига числового значения вправо >>>, заполняющий освобождающиеся слева биты нулями.

Он имеет такой же приоритет, как и остальные операторы сдвига, и левую ассоциативность.

Соответствующий оператор присваивания >>>= имеет такой же приоритет, как и другие операторы присваивания, и правую ассоциативность.

В C# можно строить выражения, в рамках которых переполнения при арифметических действиях вызывают (или не вызывают) исключения при помощи оператора
`checked``(x)` (**`unchecked`**`(x)`),
где `x` — выражение, контекст вычисления которого мы хотим определить (см. раздел о целочисленных типах).

Оба этих оператора входят в группу операторов с высшим приоритетом.

Выражение **`default`**`(T)` используется в C# 2.0 для получения значения типа `T` по умолчанию.

Для ссылочных типов это **`null`**, для числовых типов — 0, для логического типа — **`false`**, а для остальных типов значений определяется на основе их структуры.

Это выражение используется для инициализации данных в шаблонных типах, зависящих от типового параметра, который может быть как ссылочным типом, так и типом

значений.

Этот оператор входит в группу с высшим приоритетом.

Оператор `??` *(null coalescing operator)* используется в C# 2.0 в следующем смысле. Выражение `(x??y)` эквивалентно `((x == null)?y:x)`, только значение `x` вычисляется однократно, т.е., если значение `x` не равно **`null`**, то результатом этой операции является `x`, а иначе `y`.

Этот оператор имеет приоритет меньший, чем приоритет условной дизъюнкции `||`, но больший, чем приоритет условного оператора `?:`. Он правоассоциативен.

В C# 2.0 введен дополнительный оператор `::` для разрешения контекста имен в рамках глобального пространства имен или определенных синонимов.

Дело в том, что в C# при разрешении имен, построенных с помощью точек, разделяющих идентификаторы, возможны многочисленные проблемы, связанные с непредвиденными модификациями библиотек.

Например, если мы написали директиву **`using`** `System.IO`;, чтобы использовать класс `FileStream` с коротким именем, и одновременно определяем в этом контексте класс `EmptyStream`, то, если в будущем в `System.IO` появится класс `EmptyStream`, полученный код перестанет компилироваться.

Эту ситуацию можно разрешить при помощи синонимов, определив, например, для `System.IO` синоним `SIO`, а для нашей собственной библиотеки, куда входит `EmptyStream`, синоним `MIO`, и используя имена классов только вместе с синонимами — `SIO.FileStream`, `MIO.EmptyStream`. Однако если в одной из используемых библиотек будет введено пространство имен `MIO`, проблема возникнет вновь.

Чтобы однозначно отделить типы из внешних библиотек от внутрипрограммных, можно использовать оператор `::`. При этом левый аргумент такого оператора может иметь два вида. Либо он имеет значение **`global`** — тогда имя, заданное правым аргументом, ищется в глобальном пространстве имен и конфликтует с внутренними именами. Либо он является именем синонима — тогда имя, заданное правым аргументом, ищется только в

пространстве имен, определяемом этим синонимом.

Этот оператор входит в группу с высшим приоритетом.

Тип или часть его операций, или даже отдельный блок в C# могут быть помечены модификатором **`unsafe`**. При этом содержимое помеченного типа, операции или блока попадает в *небезопасный контекст (unsafe context)*. В рамках небезопасного контекста можно использовать указатели и операции над указателями, в частности, доступ к элементу данных по указателю с помощью оператора `->`, построение указателей на данные и разыменование указателей, арифметические действия над указателями.

В данном курсе мы не будем больше касаться правил написания небезопасного кода в C#, предоставляя заинтересованному читателю самому разобраться в них с помощью [8].

Инструкции

Большинство видов инструкций в Java и C# являются общими и заимствованы из языка C. В обоих языках есть понятие *блока* — набора инструкций, заключенного в фигурные скобки.

- Пустая инструкция `;` допускается в обоих языках.
- Декларации локальных переменных устроены совершенно одинаково — указывается тип переменной, затем ее идентификатор, а затем, возможно, инициализация. Инициализировать переменную можно каким-то значением ее типа. Использование неинициализированных переменных во многих случаях определяется компилятором и считается ошибкой (но не всегда). Однако даже при отсутствии инициализации переменной, ей все равно будет присвоено значение по умолчанию для данного типа. Массивы могут быть инициализированы с помощью специальных выражений, перечисляющих значения элементов массива, например
  ```
  int[][] array = new int[][]{{0, 1}, {2, 3, 4}};
  ```
- Инструкция может быть помечена с помощью метки, которая стоит перед самой инструкцией и отделяется от нее с помощью двоеточия.
- Инструкция может быть построена добавлением точки с запятой в конец выражения определенного вида. Такое выражение должно быть одним из следующих:
  - присваиванием;
  - выражением, в котором последним оператором было уменьшение или увеличение на единицу (`++`, `--`), все равно, префиксное или постфиксное;
  - вызовом метода в объекте или классе (в C# — еще и вызовом делегата);
  - созданием нового объекта.
- Условная инструкция имеет вид
  `if``(`*`expression`*`)` *`statement`*
  или
  `if``(`*`expression`*`)` *`statement`* **`else`** *`statement1`*
  где *`expression`* — выражение логического типа (или приводящегося к логическому), а *`statement`* и *`statement1`* — инструкции.

- Инструкция выбора имеет вид
  switch(*expression*) { … }
  Внутри ее блока различные варианты действий для различных значений выражения *expression* описываются с помощью списков инструкций, помеченных либо меткой **case** с возможным значением выражения, либо меткой **default**. Группа инструкций, помеченная **default**, выполняется, если значение выражения выбора не совпало ни с одним из значений, указанных в метках **case**. Один набор инструкций может быть помечен несколькими метками. Наборы инструкций могут отделяться друг от друга инструкциями **break;**
  Тип *expression* может быть целочисленным или приводящимся к нему, либо перечислимым типом. В C# допускается использование для выбора выражений типа **string**.
  Значения, которые используются в метках **case**, должны быть константными выражениями.

В Java группа инструкций для одного значения может оканчиваться инструкцией **break**, а может и не оканчиваться. Во втором случае после ее выполнения управление переходит на следующую группу инструкций.

```
public class A
{
  public static void main(String[] args)
  {
    if(args.length > 0)
    {
      int n = Integer.parseInt(args[0]);
      switch(n)
      {
        case 0:
          System.out.println("n = 0");

        case 1:
          System.out.println
            ("n = 0 or n = 1");
          break;
        case 2:case 3:
          System.out.println
            ("n = 2 or n = 3");
          break;
        default:
          System.out.println
            ("n is out of [0..3]");

      }
    }
    else
      System.out.println("No arguments");
  }
}
```

В C# группа инструкций для одного значения (включая и группу, помеченную **default**) всегда должна оканчиваться либо **break**, либо **goto default**, либо **goto case** *value* с каким-то из значений, указанных в рамках той же инструкции выбора.

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    if(args.Length > 0)
    {
      int n = Int32.Parse(args[0]);
      switch(n)
      {
        case 0:
          Console.WriteLine("n = 0");
          goto case 1;
        case 1:
          Console.WriteLine
            ("n = 0 or n = 1");
          break;
        case 2:case 3:
          Console.WriteLine
            ("n = 2 or n = 3");
          break;
        default:
          Console.WriteLine
            ("n is out of [0..3]");
          break;
      }
    }
    else
      Console.WriteLine("No arguments");
  }
}
```

- Циклы **while** и **do** в обоих языках устроены одинаково.
  while(*expression*) *statement*
  do *statement* **while**(*expression*);
  Здесь expression — логическое выражение, условие цикла, statement — тело цикла. Правила выполнения этих циклов фактически заимствованы из языка C. Первый на каждой

итерации проверяет условие и, если оно выполнено, выполняет свое тело, а если нет — передает управление дальше. Второй цикл сначала выполняет свое тело, а потом проверяет условие.

- Цикл **`for`** в обоих языках заимствован из языка C.
  `for``(A; B; C)` *`statement`*
  выполняется практически как
  `A;` **`while`**`(B) {` *`statement`* `C; }`
  Любой из элементов `A`, `B`, `C` может отсутствовать, `B` должно быть выражением логического типа (при отсутствии оно заменяется на **`true`**), `A` и `C` должны быть наборами выражений (`A` может включать и декларации переменных), разделенных запятыми.
  Помимо обычного **`for`** в обоих языках имеется специальная конструкция для цикла, перебирающего элементы коллекции.

В Java синтаксис цикла перебора элементов коллекции такой

```
for ( finalopt type id : expression )
  statement
```

При этом выражение *`expression`* должно иметь тип `java.lang.Iterable` или тип массива.

В первом случае такой цикл эквивалентен следующему (`T` далее обозначает тип результат метода `iterator()` у *`expression`*, `v` — нигде не используемое имя).

```
for(T v = expression.iterator();
    v.hasNext(); )
{
  finalopt type id = v.next();
  statement
}
```

Во втором случае, когда *`expression`* — массив типа `T[]`, эта конструкция эквивалентна следующей (`a`, `i` — нигде не используемые имена)

```
T[] a = expression;
for(int i = 0; i < a.length; i++)
{
  finalopt type id = v.next();
  statement
}
```

Пример использования перебора элементов коллекции:

```
public class A
{
  public static void main(String[] args)
  {
    int i = 1;
    for(String s : args)
```

В C# синтаксис цикла перебора элементов коллекции такой

```
foreach ( type id in expression )
  statement
```

Выражение *`expression`* должно быть массивом, или иметь тип `System.Collections.IEnumerable` или `System.Collections.Generic.IEnumerable<T>`, или же его тип должен иметь метод `GetEnumerator()`, результат которого, в свою очередь, должен иметь свойство `Current` и метод `MoveNext()`.

Тип результата метода `GetEnumerator()` во всех случаях, кроме массива, называется *типом итератора (enumerator type)*. Тип свойства `Current`, которое имеется у типа итератора, должен совпадать с *`type`*.

Пусть тип итератора `E`, а `e` — неиспользуемое имя. Тогда приведенная конструкция, с точностью до некоторых деталей, эквивалентна следующей.

```
E e = expression.GetEnumerator();
while(e.MoveNext())
{
  type id = (type)e.Current;
  statement
}
```

Опущенные детали касаются освобождения ресурсов, используемых итератором (см. далее описание инструкции **`using`**).

Пример использования перебора элементов коллекции:

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    int i = 1;
    foreach (string s in args)
```

```
      System.out.println((i++) +
        "-th argument is " + s);
  }
}
```

```
      Console.WriteLine((i++) +
        "-th argument is " + s);
  }
}
```

- Инструкции прерывания **`break`** и **`continue`** также заимствованы из C.
  Инструкция **`break`** прерывает выполнение самого маленького содержащего ее цикла и передает управление первой инструкции после него. Инструкция **`continue`** прерывает выполнение текущей итерации и переходит к следующей, если она имеется (т.е. условие цикла выполнено в сложившейся ситуации), иначе тоже выводит цикла.
  При выходе с помощью **`break`** или **`continue`** за пределы блока **`try`** (см. ниже) или блока **`catch`**, у которых имеется соответствующий блок **`finally`**, сначала выполняется содержимое этого блока **`finally`**.

В Java инструкция **`break`** используется для прерывания выполнения не только циклов, но и обычных блоков (наборов инструкций, заключенных в фигурные скобки).

Более того, после **`break`** (или **`continue`**) может стоять метка. Тогда прерывается выполнение того блока/цикла (или же начинается новая итерация того цикла), который помечен этой меткой. Этот блок (или цикл) должен содержать такую инструкцию внутри себя.

- Инструкция возврата управления **`return`** используется для возврата управления из операции (метода, оператора, метода доступа к свойству и пр., см. далее). Если операция должна вернуть значение некоторого типа, после **`return`** должно стоять выражение этого же типа.
- Инструкция создания исключительной ситуации **`throw`** используется для выброса исключительной ситуации. При этом после **`throw`** должно идти выражение, имеющее тип исключения.
  Исключение (exception) представляет собой объект, содержащий информацию о какой-то особой (исключительной) ситуации, в которой операция не может вернуть обычный результат. Вместо обычного результата из нее возвращается объект-исключение — при этом говорят, что исключение *было выброшено* из операции. Механизм этого возвращения несколько отличается от механизма возвращения обычного результата, и обработка исключений оформляется иначе (см. следующий вид инструкций), чем обработка обычных результатов работы операции.
- Исключения в обоих языках относятся к особым типам — классам исключений. Только объекты таких классов могут быть выброшены в качестве исключений. Классами исключений являются все наследники классов `java.lang.Throwable` в Java и `System.Exception` в C#.
- Объекты-исключения содержат, как минимум, следующую информацию.
  - Сообщение о возникшей ситуации (его должен определить автор кода операции, выбрасывающей это исключение).
    В Java это сообщение можно получить с помощью метода `String getMessage()`, а в C# — с помощью свойства **`string`** `Message`.
  - Иногда возникают цепочки «наведенных» исключений, если обработка одного вызывает выброс другого. Каждый объект-исключение содержит ссылку на другое исключение, непосредственно вызвавшее это. Если данное исключение не вызвано никаким другим, эта ссылка равна **`null`**.

В Java эту ссылку можно получить с помощью метода `Throwable getCause()`, а в C# — с помощью свойства `System.Exception.InnerException`.

- Для описания ситуации, в которой возникло исключение, используется состояние стека исполнения программы — список методов, которые вызывали друг друга перед этим, и указание на место в коде каждого такого метода. Это место обозначает место вызова следующего метода по стеку или, если это самый последний метод, то место, где и возникло исключение. Обычно указывается номер строки, но иногда он недоступен, если соответствующий метод присутствует в системе только в скомпилированном виде или является внешним для Java машины.
  Информация о состоянии стека на момент возникновения исключения, как и его сообщение, автоматически выводится в поток сообщений об ошибках, если это исключение остается необработанным в программе.
  В Java состояние стека для данного исключения можно получить с помощью метода `StackTraceElement[] getStackTrace()`, возвращающего массив элементов стека. Каждый такой элемент несет информацию о файле (`String getFileName()`), классе (`String getClassName()`) и методе (`String getMathodName()`), а также о номере строки (**`int`** `getLineNumber()`).
  В C# можно сразу получить полное описание состояния стека в виде одной строки с помощью свойства **`string`** `StackTrace`.

- Блок обработки исключительных ситуаций выглядит так.

```
try                  { statements   }
catch ( type_1 e_1 ) { statements_1 }
…
catch ( type_n e_n ) { statements_n }
finally              { statements_f }
```

  Если во время выполнения одной из инструкций в блоке, следующем за **`try`**, возникает исключение, управление передается на первый блок **`catch`**, обрабатывающий исключения такого же или более широкого типа. Если подходящих блоков **`catch`** нет, выполняется блок **`finally`** и исключение выбрасывается дальше.
  Блок **`finally`** выполняется всегда — сразу после блока **`try`**, если исключения не возникло, или сразу после обрабатывавшего исключение блока **`catch`**, даже если он выбросил новое исключение.
  В этой конструкции могут отсутствовать блоки **`catch`** или блок **`finally`**, но не то и другое одновременно. В C# разрешается опускать имя объекта-исключения в **`catch`**, если он не используется при обработке соответствующей исключительной ситуации.

```
public class A
{
  public static void main(String[] args)
  {
    try {
      if(args.length > 0)
        System.out.println
     ("Some arguments are specified");
      else throw new
        IllegalArgumentException
        ("No arguments specified");
    }
    catch(RuntimeException e)
    {
      System.out.println
        ("Exception caught");
```

```
using System;

public class A
{
  public static void Main(string[] args)
  {
    try {
      if(args.Length > 0)
        Console.WriteLine
     ("Some arguments are specified");
      else throw new
        ArgumentException
        ("No arguments specified");
    }
    catch(Exception e)
    {
      Console.WriteLine
        ("Exception caught");
```

```
      System.out.println
        ("Exception type is " +
           e.getClass().getName());
      System.out.println
        ("Exception message is \"" +
           e.getMessage() + "\"");
    }
    finally
    {
      System.out.println
        ("Performing finalization");
    }
  }
}
```

```
      Console.WriteLine
        ("Exception type is " +
          e.GetType().FullName);
      Console.WriteLine
        ("Exception message is \"" +
          e.Message + "\"");
    }
    finally
    {
      Console.WriteLine
        ("Performing finalization");
    }
  }
}
```

В Java, начиная с версии 1.4, появилась инструкция **`assert`**, предназначенная для выдачи отладочных сообщений.

Эта инструкция имеет один из двух видов:

`assert` *`expression`* `;`
`assert` *`expression`* `:` *`expression_s`* `;`

Выражение *`expression`* должно иметь логический тип, а выражение *`expression_s`* — произвольный.

Проверка таких утверждений может быть выключена. Тогда эта инструкция ничего не делает, и значения входящих в нее выражений не вычисляются.

Если проверка утверждений включена, то вычисляется значение *`expression`*. Если оно равно **`true`**, управление переходит дальше, иначе в обоих случаях выбрасывается исключение `java.lang.AssertionError`.

Во втором случае еще до выброса исключения вычисляется значение выражения *`expression_s`*, оно преобразуется в строку и записывается в качестве сообщения в создаваемое исключение.

В C# имеется возможность использовать инструкцию **`goto`**. Эта инструкция передает управления на инструкцию, помеченную меткой, которая следует за **`goto`**.

Как мы уже видели, помимо обычных меток, в **`goto`** могут использоваться метки **`case`** вместе со значениями и метка **`default`**. В этих случаях инструкция **`goto`** должна находиться внутри блока **`switch`**, в котором имеются эти метки.

При выходе с помощью **`goto`** из блока **`try`** или блока **`catch`**, у которых имеется соответствующий блок **`finally`**, сначала выполняется содержимое этого блока **`finally`**.

Ключевые слова **`checked`** и **`unchecked`** в C# могут помечать блок, определяя тем самым

контекст вычислений в рамках этого блока (см. раздел о целочисленных типах).

Инструкция **using** может быть использована в C#, чтобы выполнить действия, требующие захвата каких-либо ресурсов, без необходимости заботиться потом об их освобождении.

Эта инструкция имеет вид

using (*expression*) *statement* или
using (*declaration*) *statement*

где *declaration* — это декларация одной или нескольких переменных.

Первый вид этой инструкции сводится ко второму — если тип используемого выражения `T`, и имя `v` нигде не используется, то он эквивалентен
using (`T v =` *expression*) *statement*

Эта конструкция, в свою очередь, эквивалентна следующей.

```
{
  T v = expression;
  try { statement }
  finally { disposal }
}
```

Здесь *disposal* представляет вызов метода `Dispose()`, который должен быть у типа `T`, с возможной предварительной проверкой того, что переменная `v` не равна **null**, и приведением ее к типу `System.IDisposable`, если `T` является его подтипом.

В версии 2.0 в C# введены две инструкции **yield**, предназначенные для более удобного построения итераторов.

Блок, содержащий инструкцию **yield**, называется *итерационным (iterator block)* и может быть телом метода, оператора или метода доступа к свойству и не должен быть блоком **finally**, **catch** или блоком **try**, у которого есть соответствующие **catch**-блоки. Этот блок порождает последовательность значений одного типа. Сам метод или оператор должен возвращать объект одного из четырех типов:

`System.Collections.IEnumerable`,

`System.Collections.IEnumerator`,
`System.Collections.Generic.IEnumerable<T>`,
`System.Collections.Generic.IEnumerator<T>`.

В первых двух случаях порождаются объекты типа **object**, во вторых двух — значения типа T.

Для возвращения одного из этой последовательности значений используется инструкция **yield return** *expression;* Выражение в ней должно иметь соответствующий тип, **object** или T.

Для указания на то, что порождаемая итерационным блоком последовательность значений завершилась, используется инструкция **yield break;**

Пример реализации итератора коллекции с использованием **yield** приведен ниже.

```
using System;

public class MyArrayList<T>
{
  T[] items = new T[10];
  int size = 0;

  public int Count
  { get { return size; } }

  public T this[int i]
  {
    get
    {
      if(i < 0 || i >= size) throw new
        IndexOutOfRangeException();
      else return items[i];
    }
    set
    {
      if(i < 0 || i > size) throw new
        IndexOutOfRangeException();
      else if (i == size)
      {
        T[] newItems =
          new T[size + 10];
        Array.Copy
          (items, newItems, size++);
      }
      items[i] = value;
    }
  }


  public IEnumerator<T> GetEnumerator
   ()
  {
    for(int i = 0; i < size; i++)
      yield return items[i];
  }
}

public class A
{
  public static void Main()
  {
    MyArrayList<string> l =
```

```
            new MyArrayList<string>();

        l[0] = "First";
        l[1] = "Second";
        l[2] = "Third";

        foreach (string s in l)
          Console.WriteLine(s);
      }
    }
```

Инструкции обоих языков, предназначенные для синхронизации работы нескольких потоков, рассматриваются в следующей лекции, в разделе, посвященном разработке многопоточных программ.

Пользовательские типы

В обоих рассматриваемых языках имеются ссылочные типы и типы значений. Объекты ссылочных типов имеют собственную идентичность, а значения такой идентичности не имеют. Объекты ссылочных типов можно сравнивать на совпадение или несовпадение при помощи операторов == и !=. В C# эти операторы могут быть перегружены, поэтому, чтобы сравнить объекты на идентичность, лучше привести их сначала к типу **`object`**.

В обоих языках можно создавать пользовательские ссылочные типы, определяя классы и интерфейсы. Кроме того, можно использовать массивы значений некоторого типа. В C# можно определять пользовательские типы значений, а в Java типами значений являются только примитивные.

Класс представляет собой ссылочный тип, объекты которого могут иметь сложную структуру и могут быть задействованы в некотором наборе операций. Структура данных объектов класса задается набором ***полей (fields)*** этого класса. В каждом объекте каждое поле имеет определенное значение, могущее быть ссылкой на другой или тот же самый объект.

Над объектом класса можно выполнять операции, определенные в этом классе. Термин «операция» будет употребляться для обозначения методов обоих языков, а также операторов, методов доступа к свойствам, индексерам и событиям в C# (см. ниже). Для каждой операции в классе определяются ее ***сигнатура*** и ***реализация***.

Полная сигнатура операции — это ее имя, список типов, значения которых она принимает в качестве параметров, а также тип ее результата и список типов исключений, которые могут быть выброшены из нее. Просто ***сигнатурой*** будем называть имя и список типов параметров операции — этот набор обычно используется для однозначного определения операции в рамках класса. Все операции одного класса должны различаться своими (неполными) сигнатурами, хотя некоторые из них могут иметь одинаковые имена. Единственное исключение из этого правила касается только C# и будет описано ниже.

Реализация операции представляет собой набор инструкций, выполняемых каждый раз, когда эта операция вызывается. ***Абстрактный класс*** может не определять реализации для некоторых своих операций — такие операции называются ***абстрактными***. И абстрактные классы, и их абстрактные операции помечаются модификатором **`abstract`**.

Поля и операции могут быть ***статическими (static)***, т.е. относиться не к объекту класса, а к классу в целом. Для получения значения такого поля достаточно указать класс, в котором оно определено, а не его объект. Точно так же, для выполнения статической операции не нужно указывать объект, к которому она применяется.

Интерфейс — ссылочный тип, отличающийся от класса тем, что он не определяет структуры своих объектов (не имеет полей) и не задает реализаций для своих операций. Интерфейс — это абстрактный тип данных, над которыми можно выполнять заданный набор операций. Какие конкретно действия выполнятся для данного объекта, зависит от его точного типа (в обоих

случаях это может быть класс, реализующий данный интерфейс, а в C# — еще и структурный тип, тоже реализующий данный интерфейс).

Из последней фразы может быть понятно, что и в Java, и в C# объект может относиться сразу к нескольким типам. Один из этих типов, самый узкий, — ***точный тип объекта***, а остальные (более широкие) являются классами-предками этого типа или реализуемыми им интерфейсами. Точным типом объекта не может быть интерфейс или абстрактный класс, потому что для них не определены точные действия, выполняемые при вызове (некоторых) их операций.

Классы и интерфейсы (а также отдельные операции) в обоих языках могут быть ***шаблонными (generic)***, т.е. иметь типовые параметры (соответствующие конструкции введены в Java 5 и C# 2.0). При создании объекта такого класса нужно указывать конкретные значения его типовых параметров.

Примеры деклараций классов и интерфейсов для обоих языков приведены ниже. В обоих случаях определяется шаблонный интерфейс очереди, которая хранит объекты типа-параметра, и класс, реализующий очередь на основе ссылочной структуры. Показан также пример использования такой очереди.

```
public interface Queue <T>
{
  void put (T o);
  T    get ();
  int  size();
}
public class LinkedQueue <T>
  implements Queue <T>
{
  public void put (T o)
  {
    if(last == null)
    {
      first = last = new Node <T> (o);
    }
    else
    {
      last.next = new Node <T> (o);
      last = last.next;
    }
    size++;
  }

  public T    get ()
  {
    if(first == null) return null;
    else
    {
      T result = first.o;
      if(last == first) last = null;
      first = first.next;
      size--;
      return result;
    }
  }

  public int  size()
  {
    return size;
  }
```

```
using System;

public interface IQueue <T>
{
  void Put (T o);
  T    Get ();
  int  Size();
}
public class LinkedQueue <T>
  : IQueue <T>
{
  public void Put (T o)
  {
    if(last == null)
    {
      first = last = new Node <T> (o);
    }
    else
    {
      last.next = new Node <T> (o);
      last = last.next;
    }
    size++;
  }

  public T    Get ()
  {
    if(first == null) return default(T);
    else
    {
      T result = first.o;
      if(last == first) last = null;
      first = first.next;
      size--;
      return result;
    }
  }

  public int  Size()
  {
    return size;
  }
```

```
  private Node <T> last  = null;
  private Node <T> first = null;
  private int      size  = 0;

  private static class Node <E>
  {

    E        o    = null;
    Node<E>  next = null;

    Node (E o)
    {
      this.o = o;
    }
  }
}
public class Program
{
  public static void main(String[] args)
  {
    Queue<Integer> q =
      new LinkedQueue<Integer>();

    for(int i = 0; i < 10; i++)
      q.put(i*i);

    while(q.size() != 0)
      System.out.println
        ("Next element + 1: " +
           (q.get()+1));
  }
}
```

```
  private Node <T> last  = null;
  private Node <T> first = null;
  private int      size  = 0;

  internal class Node <E>
  {

    internal E        o    = default(E);
    internal Node <E> next = null;

    internal Node <E> (E o)
    {
      this.o = o;
    }
  }
}
public class Program
{
  public static void Main()
  {
    Queue<int> q =
      new LinkedQueue<int>();

    for(int i = 0; i < 10; i++)
      q.Put(i*i);

    while(q.Size() != 0)
      Console.WriteLine
        ("Next element + 1: " +
          (q.Get()+1));
  }
}
```

Обе программы выдают на консоль текст

```
Next element + 1: 1
Next element + 1: 2
Next element + 1: 5
Next element + 1: 10
Next element + 1: 17
Next element + 1: 26
Next element + 1: 37
Next element + 1: 50
Next element + 1: 65
Next element + 1: 82
```

На основе пользовательского или примитивного типа можно строить *массивы* элементов данного типа. Тип массива является ссылочным и определяется на основе типа элементов массива. Количество элементов массива в обоих языках — это свойство конкретного объекта-массива, которое задается при его построении и далее остается неизменным. В обоих языках можно строить массивы массивов и пр.

В Java можно строить только одномерные массивы из объектов, которые, однако, сами могут быть массивами.

```
int[] array = new int[3];
String[] array1 =
  new String[]{"First","Second"};
int[][] arrayOfArrays = new int[][]
  {{1, 2, 3}, {4, 5}, {6}};
```

В C# есть возможность строить многомерные массивы в дополнение к массивам массивов.

```
int[] array = new int[3];
string[] array1 =
  new string[]{"First","Second"};
int[][] arrayOfArrays = new int[][]
  {{1, 2, 3}, {4, 5}, {6}};
int[,] twoDimensionalArray =
  new int[,] {{1, 2}, {3, 4}};
```

Количество элементов в массиве доступно как значение поля `length`, имеющегося в каждом типе массивов.

Любой тип массива наследует системному типу `System.Array`, и любой объект-массив имеет все свойства и методы этого типа.

Общее количество элементов в массиве (во всех размерностях) доступно как значение, возвращаемое свойством `Length`. Количество измерений в массиве — значение свойства `Rank`.

В обоих языках есть возможность декларировать ***перечислимые типы (enums)***, объекты которых представляются именованными константами. Однако реализована эта возможность по-разному.

В Java перечислимые типы (введены в Java 5) являются ссылочными, частным случаем классов. По сути, набор констант перечислимого типа — это набор статически (т.е. во время компиляции, а не в динамике, во время работы программы) определенных объектов этого типа.

Невозможно построить новый объект перечислимого типа — декларированные константы ограничивают множество его возможных значений. Любой его объект совпадает с одним из объектов-констант, поэтому их можно сравнивать при помощи оператора ==.

Пример декларации перечислимого типа приведен ниже.

```
public enum Coin
{
  PENNY  ( 1),
  NICKEY ( 5),
  DIME   (10),
  QUARTER(25);

  Coin(int value)
  { this.value = value; }

  public int value()
  { return value; }

  private int value;
}
```

Как видно из примеров, перечисления в Java устроены несколько сложнее, чем в C#.

Возможны декларации методов перечислимого типа и их отдельная реализация для каждой из констант.

```
public enum Operation
{
  ADD {
    public int eval(int a, int b)
```

В C# перечислимые типы являются типами значений, определяемыми на основе некоторого целочисленного типа (называемого *базовым;* по умолчанию это **`int`**). Каждая константа представляет собой некоторое значение базового типа. Однако можно искусственно построить другие значения перечислимого типа из значений его базового типа.

Пример декларации перечислимого типа приведен ниже.

```
public enum Coin : uint
{
  PENNY   =  1,
  NICKEY  =  5,
  DIME    = 10,
  QUARTER = 25
}
```

```
    { return a + b; }
  },
  SUBTRACT {
    public int eval(int a, int b)
    { return a - b; }
  },
  MULTIPLY {
    public int eval(int a, int b)
    { return a * b; }
  },
  DIVIDE {

    public int eval(int a, int b)
    { return a / b; }
  };

  public abstract int eval
                      (int a, int b);
}
```

В C# имеется возможность декларировать пользовательские типы значений, помимо перечислимых. Такие типы называются ***структурами***.

Структуры описываются во многом похоже на классы, они так же могут реализовывать интерфейсы и наследовать классам, но имеют ряд отличий при использовании, инициализации переменных структурных типов и возможности описания различных членов. Все эти особенности связаны с тем, что структуры — типы значений. Две переменных структурного типа не могут иметь одно значение — только равные. Значение структурного типа содержит все значения полей. При инициализации полей структуры используются значения по умолчанию для их типов (0 для числовых, **`false`** для логического типа, **`null`** для ссылочных типов и построенное так же значение по умолчанию для структурных).

Пример декларации структуры приведен ниже.

```
public struct Point2D
{
  private double x;
  private double y;

  public Point2D(double x, double y)
  {
    this.x = x;
    this.y = y;
  }


  public double X() { return x; }
  public double Y() { return y; }
}
```

Все структуры считаются наследующими

ссылочному типу **`object`**, поэтому возможно приведение значения структуры к этому типу. Наоборот, если известно, что данный объект типа **`object`** представляет собой значение некоторого структурного типа, он может быть приведен к этому типу.

При переводе значений между структурным типом и **`object`** производятся их преобразования, называющиеся ***упаковкой (autoboxing)*** — построение объекта, хранящего значение структурного типа, — и ***распаковкой (auto-unboxing)*** — выделение значения из хранящего его объекта.

Эти преобразования строят копии преобразуемых значений. Поэтому изменения значения, хранимого в объекте, никак не отражаются на значениях, полученных ранее из него распаковкой.

При присваивании переменной структурного типа некоторого значения также происходит копирование этого значения. При присваивании переменной ссылочного типа в нее копируется значение ссылки, а сам объект, на который указывает эта ссылка, не затрагивается.

В Java, помимо явно описанных типов, можно использовать *анонимные классы (anonymous classes)*.

Анонимный класс всегда реализует какой-то интерфейс или наследует некоторому классу. Когда объект анонимного класса создается в каком-то месте кода, все описание элементов соответствующего класса помещается в том же месте. Имени у анонимного класса нет.

Ниже приведен пример использования объекта анонимного класса, реализующего интерфейс стека.

```
interface Stack <T>
{
  void push(T o);
  T    pop ();
}

public class B
{
  public void m()
  {
    Stack<Integer> s =
      new Stack<Integer>() {
      final static int maxSize = 10;
      int[] values = new int[maxSize];
      int last = -1;

      public void push(Integer i) {
        if(last + 1 == maxSize)
```

```
          throw new
          TooManyElementsException();
        else values[++last] = i;
      }

      public Integer pop() {
        if(last - 1 < -1)
          throw new
            NoElementsException();
        else return values[last--];
      }
    };

    s.push(3);
    s.push(4);
    System.out.println(s.pop() + 1);
  }
}
```

В C# 2.0 есть специальная конструкция для ***обнуляемых (nullable)*** типов значений — переменная такого типа может иметь либо определенное значение, либо не иметь значения, что представляется как равенство **null**.

Эта возможность еще отсутствует в версии стандарта [8], о ней можно прочитать в [9].

В основе обнуляемого типа значений всегда лежит обычный тип значений — примитивный, структурный или перечислимый.

```
bool? mayBeNullFlag     = null;
int? mayBeNullNumber    = 5;
Point2D? mayBeNullPoint = null;
```

Обозначение `T?` является сокращением от `System.Nullable<T>`. Этот тип имеет свойства `HasValue`, возвращающее **true** тогда, когда его значение является значением типа `T`, а не **null**, и `Value`, возвращающее это значение, если оно не **null**.

Определено неявное преобразование значений типа `T` в `T?`.

Для обнуляемых типов, построенных на основе примитивных, определены все те же операции.

Арифметические действия над значениями обнуляемых типов возвращают **null**, если один из операндов равен **null**.

Сравнения по порядку (<, >, <=, >=) возвращают **false**, если один из операндов равен **null**.

Можно использовать сравнение значений обнуляемых типов на равенство или неравенство **null**.

Для типа **bool**? операции `&` и `|` возвращают не **null**, если их результату можно приписать логическое значение. Т.е. **false** `&` **null** ==

false, а **true | null == true**. Также выполнены равенства, получаемые при перестановке операндов в указанных примерах.

Имеется специальный оператор ??, применимый к объектам ссылочных типов или к значениям обнуляемых типов.

Значение a??b равно (a != **null**)?a:b.

Кроме перечисленных разновидностей типов, в C# имеется возможность определять ссылочные типы, являющиеся аналогами указателей на функцию в C — ***делегатные типы (delegate types)***.

Делегатный тип объявляется примерно так же, как абстрактный метод, не имеющий реализации.

Объект делегатного типа можно инициализировать с помощью подходящего по типам параметров и результата метода или с помощью *анонимного метода (anonymous method,* введены в C# 2.0*)* [9].

В приведенном ниже примере объявляется делегатный тип BinaryOperation и 6 объектов этого типа, инициализируемых различными способами.

Объекты op1 и op3 инициализируются при помощи статического метода A.Op1(), объекты op2 и op4 — при помощи метода Op2(), выполняемого в объекте a, объекты op5 и op6 — при помощи анонимных методов.

```
public delegate int BinaryOperation
                      (int x, int y);
public class A
{
  private int x = 0;
  public A(int x) { this.x = x; }

  public static int Op1(int a, int b)
  { return a + b; }

  public int Op2(int a, int b)
  { return x + a + b; }

  public static A a = new A(15);
  BinaryOperation op1 = A.Op1;
  BinaryOperation op2 = a.Op2;
  BinaryOperation op3 =
    new BinaryOperation(A.Op1);
  BinaryOperation op4 =
    new BinaryOperation(a.Op2);
  BinaryOperation op5 =
    delegate(int c, int d)
    { return c * d; };
  BinaryOperation op6 =
    delegate { return 10; };
}
```

По идее, объекты делегатных типов предназначены служить обработчиками некоторых событий. Т.е. при наступлении заданного события надо вызвать соответствующий делегат.

Обработчики событий часто надо изменять в ходе выполнения программы — добавлять в них одни действия и удалять другие.

Поэтому каждый объект-делегат представляет некоторый *список операций (invocation list)*. При этом пустой список представляется как **`null`**.

Добавлять элементы в конец этого списка можно при помощи операторов `+` и `+=`, применяемых к делегатам (и методам, которые неявно преобразуются в объекты делегатного типа, как видно из инициализации `op1` и `op2` в примере). При объединении двух делегатов список операций результата получается конкатенацией списков их операций — список операций правого операнда пристраивается в конце списка операций левого операнда. Списки операций операндов не меняются.

Удалять операции из делегатов можно при помощи операторов `-` и `-=`. При вычитании одного делегата из другого находится последнее вхождение списка операций второго операнда как подсписка в список операций первого операнда. Список операций результата получается как результат удаления этого подсписка из списка операций первого операнда. Если этот список пуст, результат вычитания делегатов равен **`null`**. Если такого подсписка нет, список операций результата совпадает со списком операций первого операнда.

Объект делегатного типа можно вызвать, передав ему набор аргументов. При этом вызываются друг за другом все операции из представляемого им списка. Если объект-делегат равен **`null`**, в результате вызова выбрасывается исключение типа `System.NullReferenceException`.

```
using System;

public class A
{
  delegate void D();

  static void M1()
  { Console.WriteLine("M1 called"); }
  static void M2()
  { Console.WriteLine("M2 called"); }
```

```
  public static void Main()
  {
    D d1 = M1, d2 = M2;
    d1 += M1;
    d2 = d1 + d2 + d1;

    d1();
    Console.WriteLine("------------");
    d2();
    Console.WriteLine("------------");
    (d1 + d2)();
    Console.WriteLine("------------");
    (d1 - d2)();
    Console.WriteLine("------------");
    (d2 - d1)();
  }
}
```

Программа из приведенного примера выдает следующий результат.

```
M1 called
M1 called
------------
M1 called
M1 called
M2 called
M1 called
M1 called
------------
M1 called
M1 called
M1 called
M1 called
M2 called
M1 called
M1 called
------------
M1 called
M1 called
------------
M1 called
M1 called
M2 called
```

В следующей лекции продолжается рассмотрение способов описания пользовательских типов в Java и C#.

Литература к Лекции 10

[1] Страница платформы J2SE http://java.sun.com/j2se/index.jsp.
[2] Страница платформы J2EE http://java.sun.com/j2ee/index.jsp.
[3] Страница платформы J2ME http://java.sun.com/j2me/index.jsp.
[4] Страница платформы Java Card http://java.sun.com/products/javacard/index.jsp.
[5] Страница для разработчиков на .NET http://www.microsoft.com/net/developers.mspx.
[6] Страница проекта Mono http://www.mono-project.com/Main_Page.
[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java Language Specification, 3-rd edition. Addison Wesley Professional, 2005.
Доступна как http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.
[8] C# Language Specification. Working Draft 2.7. ECMA, June 2004.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/standard.pdf.

[9] C# Langauge Specification 2.0, March 2005 Draft.
Доступна как http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/CSharp%202.0%20Specification.doc.
[10] The Unicode Consortium. The Unicode Standard, Version 4.0. Boston, MA, Addison-Wesley Developers Press, 2003.
[11] IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. Revised 1990, IEEE, New York, 1990.
[12] IEC 60559:1989 Binary floating-point arithmetic for microprocessor systems. (2-nd ed., previously designated IEC 559:1989) International Electrotechnical Commission, 1989.