Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 11. Основные конструкции языков Java и C# (продолжение)

Аннотация

Продолжается рассмотрение основных конструкций языков Java и C#. Рассказывается о правилах описания связей между типами, определения операций над ними и о создании многопоточных программ. Вкратце рассказывается об основных библиотеках Java и .NET.

Ключевые слова

Наследование типов, перегрузка операций, поле класса, операция, метод, конструктор, инициализатор, константа, свойство, индексированное свойство, событие, оператор, поток, синхронизация потоков.

Текст лекции

В этой лекции мы продолжаем рассмотрение элементов Java 5 и C# 2.0 на основе описывающих их стандартов [1] и [2,3].

Наследование

Отношение вложенности между типами определяется *наследованием*. Обычно говорят, что класс ***наследует*** другому классу или является его ***потомком***, ***наследником***, если он определяет более узкий тип, т.е. все объекты этого класса являются также и объектами наследуемого им. Второй класс в этом случае называют ***предком*** первого. Взаимоотношения между интерфейсами описываются в тех же терминах, но вместо «класс наследует интерфейсу» обычно говорят, что класс ***реализует*** интерфейс.

В обоих языках класс может наследовать только одному классу и реализовывать несколько интерфейсов. Интерфейс может наследовать многим интерфейсам.

Классы, которые не должны иметь наследников, помечаются в Java как **`final`**, а в C# как **`sealed`**.

В Java все классы (но не интерфейсы!) считаются наследниками класса `java.lang.Object`.

Примитивные типы не являются его наследниками, в отличие от своих классов-оберток.

В C# все классы, структурные, перечислимые и делегатные типы (но не интерфейсы!) рассматриваются как наследники класса `System.Object`, на который обычно ссылаются как на **`object`**. При этом, однако, типы значений (перечислимые и структурные типы, наследники `System.ValueType`) преобразуются к типу **`object`** с помощью упаковки, строящей каждый раз новый объект.

Структурный тип может реализовывать один или несколько интерфейсов, но не может наследовать классу или другому структурному типу.

При наследовании, т.е. сужении типа, возможно определение дополнительных полей и дополнительных операций. Возможно также определение в классе-потомке поля,

имеющего то же имя, что и некоторое поле в классе-предке. В этом случае происходит *перекрытие имен* — определяется новое поле, и в коде потомка по этому имени становится доступно только оно.

Если же необходимо получить доступ к соответствующему полю предка, нужно использовать разные подходы в зависимости от того, статическое это поле или нет, т.е. относится ли оно к самому классу или к его объектам. К статическому полю можно обратиться, указав его полное имя, т.е. `ClassName.fieldName`, к нестатическому полю из кода класса-потомка можно обратиться с помощью конструкций **`super`**`.fieldName` в Java и **`base`**`.fieldName` в C# (естественно, если оно не перекрыто в каком-то классе, промежуточном между данными предком и потомком). Конструкции **`super`** в Java и **`base`** в C# можно использовать и для обращения к операциям, декларированным в предке данного класса. Для обращения к полям и операциям самого объекта в обоих языках можно использовать префикс **`this`**, являющийся ссылкой на объект, в котором вызывается данная операция.

Основная выгода от использования наследования — возможность ***перегружать (override)*** реализации операций в типах-наследниках. Это значит, что при вызове операции с данной сигнатурой в объекте наследника может быть выполнена не та реализация этой операции, которая определена в предке, а совсем другая, определенная в точном типе объекта. Такие операции называют ***виртуальными (virtual)***. Для того чтобы определить новую реализацию некоторой виртуальной операции предка в потомке, нужно определить в потомке операцию с той же сигнатурой. При этом необходимо следовать общему принципу, обеспечивающему корректность системы типов в целом — *принципу подстановки (Liskov substitution principle)* [4,5]. Поскольку тип-наследник является более узким, чем тип-предок, его объект может использоваться всюду, где может использоваться объект типа-предка. Принцип подстановки, обеспечивающий это свойство, требует соблюдения двух правил.

- Во всякой ситуации, в которой можно вызвать данную операцию в предке, ее должно быть можно вызвать и в наследнике. Говоря по-другому, предусловие операции при перегрузке не должно усиливаться.
- Множество ситуаций, в которых система в целом может оказаться после вызова операции в наследнике, должно быть подмножеством набора ситуаций, в которых она может оказаться в результате вызова этой операции в предке. То есть, постусловие операции при перегрузке не должно ослабляться.

Статические операции, относящиеся к классу в целом, а не к его объектам, не виртуальны. Они не могут быть перегружены, но могут быть перекрыты, если в потомке определяются статические операции с такими же сигнатурами.

В Java все нестатические методы классов являются виртуальными, т.е. перегружаются при определении метода с такой же сигнатурой в классе-потомке.

Но в Java, в отличие от C#, можно вызывать статические методы и обращаться к статическим полям класса через ссылки на его объекты (в том числе, и через this). Поэтому работу невиртуальных методов можно смоделировать с помощью обращений к статическим операциям.

В C# нестатические операции (обычные методы и методы доступа к свойствам, индексерам и событиям) могут быть как виртуальными, т.е. перегружаемыми, так и невиртуальными.

Для декларации виртуального метода (свойства, индексера или события) необходимо пометить его модификатором **`virtual`**. Метод (свойство, индексер, событие), перегружающий его в классе-потомке надо в этом случае пометить модификатором **`override`**. Если же мы не хотим перегружать элемент предка, а хотим определить *новый* элемент с такой же

сигнатурой (т.е. перекрыть старый), то его надо пометить модификатором **new**.

Элемент, не помеченный как **virtual**, не является перегружаемыми — его можно только перекрыть. При вызове операции с такой сигнатурой в некотором объекте будет вызвана ее реализация, определяемая по декларированному типу объекта.

Приводимые ниже примеры на обоих языках иллюстрируют разницу в работе виртуальных и невиртуальных операций.

```
class A
{
  public void m()
  {
    System.out.println
      ("A.m() called");
  }

  public static void n()
  {
    System.out.println
      ("A.n() called");
  }
}

class B extends A
{
  public void m()
  {
    System.out.println
      ("B.m() called");
  }

  public static void n()
  {
    System.out.println
      ("B.n() called");
  }
}

public class C
{
  public static void main
    (String[] args)
  {
    A a = new A();
    B b = new B();
    A c = new B();

    a.m();
    b.m();
    c.m();
    System.out.println("-----");
    a.n();
    b.n();
    c.n();
  }
}
```

Представленный в примере код выдает

```
using System;

class A
{
  public virtual void m()
  {
    Console.WriteLine("A.m() called");
  }

  public void n()
  {
    Console.WriteLine("A.n() called");
  }
}

class B : A
{
  public override void m()
  {
    Console.WriteLine("B.m() called");
  }

  public new void n()
  {
    Console.WriteLine("B.n() called");
  }
}

public class C
{
  public static void Main()
  {
    A a = new A();
    B b = new B();
    A c = new B();

    a.m();
    b.m();
    c.m();
    Console.WriteLine("-----");
    a.n();
    b.n();
    c.n();
  }
}
```

Если в приведенном примере убрать

следующие результаты.

```
A.m() called
B.m() called
B.m() called
-----
A.n() called
B.n() called
A.n() called
```

модификатор **new** у метода n() в классе B, ошибки компиляции не будет, но будет выдано предупреждение о, возможно случайном, перекрытии имен.

Представленный в примере код выдает следующие результаты.

```
A.m() called
B.m() called
B.m() called
-----
A.n() called
B.n() called
A.n() called
```

Элементы типов

Элементы или *члены (members)* пользовательских типов могут быть методами, полями (в классах) и вложенными типами. В классе можно также объявлять конструкторы, служащие для создания объектов этого класса. В обоих языках описание конструктора похоже на описание метода, только тип результата не указывается, а вместо имени метода используется имя самого класса.

В обоих языках поля можно только перекрывать в наследниках, а методы можно и перегружать. Вложенные типы, как и поля, могут быть перекрыты.

У каждого элемента класса могут присутствовать модификаторы, определяющие доступность этого элемента из разных мест программы, а также его *контекст* — относится ли он к объектам этого класса (нестатический элемент) или к самому классу (статический элемент, помечается как **static**).

Для указания доступности в обоих языках могут использоваться модификаторы **public**, **protected** и **private**, указывающие, что данный элемент доступен везде, где доступен содержащий его тип, доступен только в описаниях типов-наследников содержащего типа, или только в рамках описания самого содержащего типа. Доступность по умолчанию, без указания модификатора трактуется в рассматриваемых языках различно.

Нестатические методы в обоих языках (а также свойства, индексированные свойства и события в C#) могут быть объявлены *абстрактными (abstract)*, т.е. не задающими реализации соответствующей операции. Такие методы (а также свойства, индексированные свойства и события в C#) помечаются модификатором **abstract**. Вместо кода у абстрактного метода сразу после описания полной сигнатуры идет точка с запятой.

Методы (свойства, индексированные свойства и события в C#), которые не должны быть перегружены в наследниках содержащего их класса, помечаются в Java как **final**, а в C# как **sealed**.

В обоих языках можно использовать операции, реализованные на других языках. Для этого в C# используются стандартные механизмы .NET — класс реализуется на одном из языков, поддерживаемых .NET, с учетом ограничений на общие библиотеки этой среды и становится доступен из любого другого кода на поддерживаемом .NET языке.

В Java для этого предусмотрен механизм Java Native Interface, JNI [6,7]. Класс Java может иметь ряд внешних методов, помеченных модификатором **native**. Вместо кода у таких методов сразу после описания полной сигнатуры идет точка с запятой. Они по определенным правилам реализуются в виде функций на языке C (или на другом языке, если можно в результате компиляции получить библиотеку с интерфейсом на C). Внешние методы, а также свойства, индексированные свойства, события и операторы,

привязываемые по определенным правилам к функциям, имеющим интерфейс на C, даже не вложенным в среду .NET, есть и в C# — там такие операции помечаются как **`extern`**.

В Java, помимо перечисленных членов типов, имеются *инициализаторы*. Их описание приведено ниже.

Инициализаторы относятся только к тому классу, в котором они определены, их нельзя перегрузить.

В C# члены типов, помимо методов, полей, конструкторов и вложенных типов, могут быть *константами*, *свойствами*, *индексированными свойствами*, *событиями* или *операторами*. Кроме этого, в типе можно определить *деструктор* и *статический конструктор*.

Нестатические свойства, индексированные свойства и события можно перегружать в наследниках. Остальные из перечисленных элементов относятся только к тому классу, в котором описаны.

Для многих из дополнительных разновидностей членов типов, имеющихся в C#, есть аналогичные идиомы в компонентной модели JavaBeans [8,9], предназначенной для построения элементов пользовательского интерфейса и широко используемой в рамках Java технологий для создания компонентов, структуру которых можно анализировать динамически на основе предлагаемых JavaBeans соглашений об именовании методов. Далее вместе с примерами кода на C# в правом столбце в левом приводится аналогичный код, написанный в соответствии JavaBeans.

Константы в Java принято оформлять в виде полей с модификаторами **`final`** **`static`**. Модификатор **`final`** для поля означает, что присвоить ему значение можно только один раз, и сделать это нужно либо в статическом инициализаторе класса (см. ниже), если поле статическое, либо в каждом из конструкторов, если поле нестатическое.

Константы являются один раз вычисляемыми и неизменными далее значениями, хранящимися в классе или структуре.

Пример объявления константы приведен ниже.

```
public class A2
{
  public static final double PHI =
    1.61803398874989;
}
```

```
public class A2
{
  public const double Phi =
    1.61803398874989;
}
```

Компонентная модель JavaBeans определяет ***свойство (property)*** класса `A`, имеющее имя *`name`* и тип *`T`*, как набор из одного или двух методов, декларированных в классе `A` — *`T`* `get`*`Name`*`()` и `void set`*`Name`*`(`*`T`*`)`, называемых ***методами доступа (accessor methods)*** к свойству.

Свойства представляют собой «виртуальные» поля. Каждое свойство имеет один или оба *метода доступа* **`get`** и **`set`**, которые определяют действия, выполняемые при чтении и модификации этого свойства. Оба метода доступа описываются внутри декларации свойства. Метод **`set`** использует специальный идентификатор **`value`** для ссылки на устанавливаемое значение свойства.

Свойство может быть доступным только для чтения, если имеется только метод `get`, и только для записи, если имеется только метод `set`.

Обращение к свойству — чтение (возможно, только если у него есть метод **`get`**) или изменение значения свойства (возможно, только если у него есть метод **`set`**) — происходит так же, как к полю.

Если свойство имеет логический тип, для метода чтения этого свойства используется имя `is`*`Name`*`()`.

Эти соглашения широко используются в

разработке Java программ, и такие свойства описываются не только у классов, предназначенных стать компонентами JavaBeans.

Они и стали основанием для введения специальной конструкции для описания свойств в C#.

```
public class MyArrayList
{
  private int[] items = new int[10];
  private int size = 0;

  public int getSize()
  {
    return size;
  }

  public int getCapacity()
  {
    return items.Length;
  }

  public void setCapacity(int value)
  {
    int[] newItems = new int[value];
    System.arraycopy
      (items, 0, newItems, 0, size);
    items = newItems;
  }

  public static void main
                    (String[] args)
  {
    MyArrayList l = new MyArrayList();
    System.out.println(l.getSize());
    System.out.println
              (l.getCapacity());
    l.setCapacity(50);
    System.out.println(l.getSize());
    System.out.println
              (l.getCapacity());
  }
}
```

JavaBeans определяет ***индексированное свойство (indexed property)*** класса `A`, имеющее имя *`name`* и тип *`T`*, как один или пару методов *`T`* `get`*`Name`*`(int)` и `void set`*`Name`*`(int, `*`T`*`)`.

Свойства могут быть индексированы только одним целым числом. В дальнейшем предполагалось ослабить это ограничение и разрешить индексацию несколькими параметрами, которые могли бы иметь разные типы. Однако с 1997 года, когда появилась последняя версия спецификаций JavaBeans [9], этого пока сделано не было.

При перегрузке свойства в наследниках перегружаются методы доступа к нему.

Пример объявления свойств и их использования приведен ниже.

```
using System;

public class MyArrayList
{
  private int[] items = new int[10];
  private int size = 0;

  public int Size
  {
    get { return size; }
  }

  public int Capacity
  {
    get { return items.Length; }
    set
    {
      int[] newItems = new int[value];
      Array.Copy
        (items, newItems, size);
      items = newItems;
    }
  }


  public static void Main()
  {
    MyArrayList l = new MyArrayList();
    Console.WriteLine( l.Size );
    Console.WriteLine( l.Capacity );
    l.Capacity = 50;
    Console.WriteLine( l.Size );
    Console.WriteLine( l.Capacity );
  }
}
```

Индексированное свойство или ***индексер (indexer)*** — это свойство, зависящее от набора параметров.

В C# может быть определен только один индексер для типа и данного набора типов параметров. Т.е. нет возможности определять свойства с разными именами, но одинаковыми наборами индексов.

Обращение к индексеру объекта (или класса, т.е. статическому) производится так, как будто этот объект (класс) был бы массивом, индексированным набором индексов соответствующих типов.

```
public class MyArrayList
{
  int[] items = new int[10];
  int size = 0;

  public int getItem(int i)
  {
    if (i < 0 || i >= 10) throw new
       IllegalArgumentException();
    else return items[i];
  }

  public void setItem
              (int i, int value)
  {
    if (i < 0 || i >= 10) throw new
       IllegalArgumentException();
    else items[i] = value;
  }

  public static void main
                    (String[] args)
  {
    MyArrayList l = new MyArrayList();
    l.setItem(0, 23);
    l.setItem(1, 75);
    l.setItem(1, l.getItem(1)-1);
    l.setItem(0,
      l.getItem(0) + l.getItem(1));
    System.out.println (l.getItem(0));
    System.out.println (l.getItem(1));
  }
}
```

События (events) в модели JavaBeans служат для оповещения набора *объектов-наблюдателей (listeners)* о некоторых изменениях в состоянии *объекта-источника (source)*.

При этом класс *`EventType`* объектов, представляющих события определенного вида, должен наследовать `java.util.EventObject`. Все объекты-наблюдатели должны реализовывать один интерфейс *`EventListener`*, в котором должен быть метод обработки события (обычно называемый так же, как и событие) с параметром типа *`EventType`*. Интерфейс *`EventListener`* должен наследовать

При перегрузке индексера в наследниках перегружаются методы доступа к нему.

Индексеры должны быть нестатическими.

Обращение к индексеру класса-предка в индексере наследника организуется с помощью конструкции **`base`**`[…]`.

Пример декларации и использования индексера приведен ниже.

```
using System;

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

  public int this[int i]
  {
    get
    {
      if (i < 0 || i >= 10) throw new
         IndexOutOfRangeException();
      else return items[i];
    }
    set
    {
      if (i < 0 || i >= 10) throw new
         IndexOutOfRangeException();
      else items[i] = value;
    }
  }

  public static void Main()
  {
    MyArrayList l = new MyArrayList();
    l[0] = 23;
    l[1] = 75;
    l[0] += (--l[1]);
    Console.WriteLine(l[0]);
    Console.WriteLine(l[1]);
  }
}
```

Событие (event) представляет собой свойство специального вида, имеющее делегатный тип. У события, в отличие от обычного свойства, методы доступа называются **`add`** и **`remove`**, и предназначены они для добавления или удаления обработчиков данного события, являющихся делегатами (это аналоги различных реализаций метода обработки события в интерфейсе наблюдателя в JavaBeans) при помощи операторов `+=` и `-=`.

Событие может быть реализовано как поле делегатного типа, помеченное модификатором event. В этом случае декларировать соответствующие методы **`add`** и **`remove`** необязательно — они

интерфейсу `java.util.EventListener`.

Класс источника событий должен иметь методы для регистрации наблюдателей и их удаления из реестра. Эти методы должны иметь сигнатуры

```
public void add_EventListener
          (EventListener)
public void remove_EventListener
          (EventListener).
```

Можно заметить, что такой способ реализации обработки событий воплощает образец проектирования «Подписчик».

В приведенном ниже примере все **public** классы и интерфейсы должны быть описаны в разных файлах.

```
public class MouseEventArgs { ... }
```

```
public class MouseEventObject
  extends java.util.EventObject
{
  MouseEventArgs args;

  MouseEventObject
  (Object source, MouseEventArgs args)
  {
    super(source);
    this.args = args;
  }
}
```

```
public interface MouseEventListener
  extends java.util.EventListener
{
  void mouseUp(MouseEventObject e);
  void mouseDown(MouseEventObject e);
}
```

```
import java.util.ArrayList;
```

```
public class MouseEventSource
{
  private
ArrayList<MouseEventListener>
    listeners = new ArrayList
    <MouseEventListener >();

  public synchronized void
  addMouseEventListener
    (MouseEventListener l)
  { listeners.add(l); }

  public synchronized void
  removeMouseEventListener
    (MouseEventListener l)
  { listeners.remove(l); }
```

автоматически реализуются при применении операторов += и -= в к этому полю как к делегату.

Если же программист хочет реализовать какое-то специфическое хранение обработчиков события, он должен определить методы **add** и **remove**.

В приведенном ниже примере одно из событий реализовано как событие-поле, а другое — при помощи настоящего поля и методов **add** и **remove**, дающих совместно тот же результат.

При перегрузке события в наследниках перегружаются методы доступа к нему.

```
public class MouseEventArgs { ... }
```

```
public delegate void MouseEventHandler
  (object source, MouseEventArgs e);
```

```
public class MouseEventSource
{
  public event
    MouseEventHandler MouseUp;

  private MouseEventHandler mouseDown;
  public event
    MouseEventHandler MouseDown
  {
    add
    {
      lock(this)
      { mouseDown += value; }
    }
    remove
    {
      lock(this)
      { mouseDown -= value; }
    }
  }
```

```
 protected void notifyMouseUp
    (MouseEventArgs a)
  {
    MouseEventObject e =
      new MouseEventObject(this, a);
    ArrayList<MouseEventListener> l;
    synchronized(this)
    {
      l =
      (ArrayList<MouseEventListener>)
        listeners.clone();
      for(MouseEventListener el : l)
        el.mouseUp(e);
    }
  }

  protected void notifyMouseDown
    (MouseEventArgs a)
  {
    MouseEventObject e =
      new MouseEventObject(this, a);

    ArrayList<MouseEventListener> l;
    synchronized(this)
    {
      l =
       (ArrayList<MouseEventListener>)
        listeners.clone();
      for(MouseEventListener el : l)
        el.mouseDown(e);
    }
  }
}
```

```
public class HandlerConfigurator
{
  MouseEventSource s =
    new MouseEventSource();

  MouseEventListener listener =
    new MouseEventListener()
  {
    public void mouseUp
     (MouseEventObject e) { ... }
    public void mouseDown
     (MouseEventObject e) { ... }
  };

  public void configure()
  {
    s.addMouseEventListener(listener);
  }
}
```

```
  protected void
    OnMouseUp(MouseEventArgs e)
  {
    MouseUp(this, e);
  }

  protected void
    OnMouseDown(MouseEventArgs e)
  {
    mouseDown(this, e);
  }
}
```

```
public class HandlerConfigurator
{
  MouseEventSource s =
    new MouseEventSource();

  public void UpHandler
    (object source, MouseEventArgs e)
  { ... }
  public void DownHandler
    (object source, MouseEventArgs e)
  { ... }

  public void Configure()
  {
    s.MouseUp += UpHandler;
    s.MouseDown += DownHandler;
  }
}
```

Методы доступа к свойствам, индексерам или событиям в классах-наследниках могут перегружаться по отдельности, т.е., например, метод чтения свойства перегружается, а метод записи — нет.

В C# 2.0 введена возможность декларации различной доступности у таких методов. Например, метод чтения свойства можно

сделать общедоступным, а метод записи — доступным только для наследников.

Для этого можно описать свойство так:

```
public int Property
{
  get { … }
  protected set { … }
}
```

В Java никакие операторы переопределить нельзя.

Вообще, в этом языке имеются только операторы, действующие на значениях примитивных типах, сравнение объектов на равенство и неравенство, а также оператор + для строк (это объекты класса `java.lang.String`), обозначающий операцию конкатенации.

Оператор + может применяться и к другим типам аргументов, если один из них имеет тип `String`. При этом результатом соответствующей операции является конкатенация его и результата применения метода `toString()` к другому операнду в порядке следования операндов.

Некоторые операторы в C# можно переопределить (перекрыть) для данного пользовательского типа. Переопределяемый оператор всегда имеет модификатор **`static`**.

Переопределяемые унарные операторы (их единственный параметр должен иметь тот тип, в рамках которого они переопределяются, или *объемлющий тип*): +, -, !, ~ (в качестве типа результата могут иметь любой тип), ++,-- (тип их результата может быть только подтипом объемлющего), **`true`**, **`false`** (тип результата **`bool`**).

Переопределяемые бинарные операторы (хотя бы один из их параметров должен иметь объемлющий тип, а возвращать они могут результат любого типа): +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=. Для операторов сдвига << и >> ограничения более жесткие — первый их параметр должен иметь объемлющий тип, а второй быть типа **`int`**.

Можно определять также операторы приведения к другому типу или приведения из другого типа, причем можно объявить такое приведение неявным с помощью модификатора **`implicit`**, чтобы компилятор сам вставлял его там, где оно необходимо для соблюдения правил соответствия типов. Иначе надо использовать модификатор **`explicit`** и всегда явно приводить один тип к другому.

Некоторые операторы можно определять только парами — таковы **`true`** и **`false`**, == и !=, < и >, <= и >=.

Операторы **`true`** и **`false`** служат для неявного преобразования объектов данного типа к соответствующим логическим значениям.

Если в типе `T` определяются операторы & и |, возвращающие значение этого же типа, а также операторы **`true`** и **`false`**, то к

объектам типа можно применять условные логические операторы `&&` и `||`. Их поведение в этом случае может быть описано соотношениями `(x && y) = (T.false(x)? x : (x & y))` и `(x || y) = (T.true(x)? x : (x | y))`.

Аналогичным образом автоматически переопределяются составные операторы присваивания, если переопределены операторы `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<` или `>>`.

Ниже приведен пример переопределения и использования операторов. Обратите внимание, что оператор приведения типа `MyInt` в **`int`** действует неявно, а для обратного перехода требуется явное приведение.

Другая тонкость — необходимость приведения объектов типа `MyInt` к **`object`** в методе `AreEqual` — если этого не сделать, то при обращении к оператору `==` возникнет бесконечный цикл, поскольку сравнение `i1 == null` тоже будет интерпретироваться как вызов этого оператора.

```
using System;

public class MyInt
{
  int n = 0;

  public MyInt(int n) { this.n = n; }

  public override bool Equals
    (object obj)
  {
    MyInt o = obj as MyInt;
    if (o == null) return false;
    return o.n == n;
  }

  public override int GetHashCode()
  { return n; }

  public override string ToString()
  { return n.ToString(); }

  public static bool AreEqual
    (MyInt i1, MyInt i2)
  {
    if ((object)i1 == null)
      return ((object)i2 == null);
    else return i1.Equals(i2);
  }

  public static bool operator ==
    (MyInt i1, MyInt i2)
  { return AreEqual(i1, i2); }
```

```
  public static bool operator !=
    (MyInt i1, MyInt i2)
  { return !AreEqual(i1, i2); }

  public static bool operator true
    (MyInt i)
  { return i.n > 0; }

  public static bool operator false
    (MyInt i)
  { return i.n <= 0; }

  public static MyInt operator ++
    (MyInt i)
  { return new MyInt(i.n + 1); }

  public static MyInt operator --
    (MyInt i)
  { return new MyInt(i.n - 1); }

  public static MyInt operator &
    (MyInt i1, MyInt i2)
  { return new MyInt(i1.n & i2.n); }

  public static MyInt operator |
    (MyInt i1, MyInt i2)
  { return new MyInt(i1.n | i2.n); }

  public static implicit operator int
    (MyInt i)
  { return i.n; }

  public static explicit operator
    MyInt (int i)
  { return new MyInt(i); }

  public static void Main()
  {
    MyInt n = (MyInt)5;
    MyInt k = (MyInt)(n - 3 * n);
    Console.WriteLine("k = " + k +
      " , n = " + n);
    Console.WriteLine("n == n : " +
      (n == n));
    Console.WriteLine("n == k : " +
      (n == k));
    Console.WriteLine(
      "(++k) && (n++) : " +
      ((++k) && (n++)));
    Console.WriteLine("k = " + k +
      " , n = " + n);
    Console.WriteLine(
      "(++n) && (k++) : " +
       ((++n) && (k++)));
    Console.WriteLine("k = " + k +
      " , n = " + n);
  }
}
```

Аналогом деструктора в Java является метод **`protected void`** `finalize()`, который можно перегрузить для данного

Деструктор предназначен для освобождения каких-либо ресурсов, связанных с объектом и не освобождаемых

класса.

Так же, как и деструктор в C#, этот метод вызывается на некотором шаге уничтожения объекта после того, как тот был помечен сборщиком мусора как неиспользуемый.

автоматически средой .NET, либо для оптимизации использования ресурсов за счет их явного освобождения.

Деструктор вызывается автоматически при уничтожении объекта в ходе работы механизма управления памятью .NET. В этот момент объект уже должен быть помечен сборщиком мусора как неиспользуемый.

Деструктор оформляется как особый метод, без возвращаемого значения и с именем, получающимся добавлением префикса '~' к имени класса

```
public class MyFileReader
{
  java.io.FileReader input;

  public MyFileReader(String path)
    throws FileNotFoundException
  {
    input = new java.io.FileReader
     (new java.io.File(path));
  }

  protected void finalize()
  {
    System.out.println("Destructor");
    try { input.close(); }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

```
using System;

public class MyFileStream
{
  System.IO.FileStream input;

  public MyFileStream(string path)
  {
    input = System.IO.File.Open
      (path, System.IO.FileMode.Open);
  }


  ~MyFileStream()
  {
    Console.WriteLine("Destructor");
    input.Close();
  }
}
```

Инициализаторы представляют собой блоки кода, заключенные в фигурные скобки и расположенные непосредственно внутри декларации класса.

Эти блоки выполняются вместе с инициализаторами отдельных полей — выражениями, которые написаны после знака = в объявлениях полей — при построении объекта данного класса, в порядке их расположения в декларации.

Статические инициализаторы — такие же блоки, помеченные модификатором **`static`** — выполняются вместе с инициализаторами статических полей по тем же правилам в момент первой загрузки класса в Java-машину.

Статический конструктор класса представляет собой блок кода, выполняемый при первой загрузке класса в среду .NET, т.е. в момент первого использования этого класса в программе. Это аналог статического инициализатора в Java.

Оформляется он как конструктор с модификатором **`static`**.

```
public class A
{
  static
  {
    System.out.println("Loading A");
  }

  static int x = 1;

  static
  {
    System.out.println("x = " + x);
    x++;
  }

  static int y = 2;

  static
  {
    y = x + 3;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }

  public static void main
    (String[] args) {}
}
```

Приведенный выше код выдает результат

```
Loading A
x = 1
x = 2
y = 5
```

В Java нестатические вложенные типы трактуются очень специфическим образом — каждый объект такого типа считается привязанным к определенному объекту объемлющего типа. У нестатического вложенного типа есть как бы необъявленное поле, хранящее ссылку на объект объемлющего типа.

Такая конструкция используется, например, для определения классов итераторов для коллекций — объект-итератор всегда связан с объектом-коллекцией, которую он итерирует. В то же время, пользователю не нужно знать, какого именно типа данный итератор — достаточно, что он реализует общий интерфейс всех итераторов, позволяющий проверить, есть ли еще объекты, и получить следующий объект.

Получить этот объект внутри декларации вложенного типа можно с помощью конструкции `ClassName.this`, где `ClassName` — имя объемлющего типа.

При создании объекта такого вложенного класса необходимо указать объект объемлющего класса, к которому тот будет

```
using System;

public class A
{
  static A()
  {
    Console.WriteLine("Loading A");
    Console.WriteLine("x = " + x);
    x++;
    y = x + 3;
    Console.WriteLine("x = " + x);
    Console.WriteLine("y = " + y);
  }

  static int x = 1;

  static int y = 2;

  public static void Main() {}
}
```

Приведенный выше код выдает результат

```
Loading A
x = 1
x = 2
y = 5
```

В C# модификатор **`static`** у класса, все равно, вложенного в другой или нет, обозначает, что этот класс является контейнером набора констант и статических операций. Все его элементы должны быть декларированы как **`static`**.

привязан.

```
Public class ContainingClass
{
  static int counter = 1;
  static int ecounter = 1;
  int id = counter++;

  class EmbeddedClass
  {
    int eid = ecounter++;

    public String toString()
    {
      return «» +
        ContainingClass.this.id +
        '.' + eid;
    }
  }

  public String toString()
  {
    return «» + id;
  }

  public static void main
    (String[] args)
  {
    ContainingClass
       c  = new ContainingClass()
     , c1 = new ContainingClass();
    System.out.println(c);
    System.out.println(c1);

    EmbeddedClass
        e  = c.new EmbeddedClass()
      , e1 = c.new EmbeddedClass()
      , e2 = c1.new EmbeddedClass();
    System.out.println(e);
    System.out.println(e1);
    System.out.println(e2);
  }
}
```

В C# класс может определить различные реализации для операций (методов, свойств, индексеров, событий) с одинаковой сигнатурой, если они декларированы в различных реализуемых классом интерфейсах.

Для этого при определении таких операций нужно указывать имя интерфейса в качестве расширения их имени.

```
Using System;

public interface I1
{
  void m();
}

public interface I2
{
  void m();
```

```
}

public class A : I1, I2
{
  public void m()
  {
    Console.WriteLine(«A.m() called»);
  }

  void I1.m()
  {
    Console.WriteLine
      ("I1.m() defined in A called");
  }

  void I2.m()
  {
    Console.WriteLine
     ("I2.m() defined in A called");
  }

  public static void Main()
  {
    A f = new A();
    I1 i1 = f;
    I2 i2 = f;

    f.m();
    i1.m();
    i2.m();
  }
}
```

Результат работы приведенного выше примера следующий.

```
A.m() called
I1.m() defined in A called
I2.m() defined in A called
```

Последовательность выполнения инициализаторов полей и конструкторов классов-предков и наследников при построении объектов в Java и C# различается достаточно сильно.

О правилах, определяющих эту последовательность, можно судить по результатам работы следующих примеров.

```
public class A
{
  static
  {
    System.out.println
      ("Static initializer of A");
  }

  {
    System.out.println
      ("Initializer of A");
  }

  static int sinit()
  {
    System.out.println
```

```
using System;

public class A
{
  static A()
  {
    Console.WriteLine
      ("Static constructor of A");
  }

static int sinit()
  {
    Console.WriteLine
```

```
    ("Static field initializer of A");
    return 0;
  }

  static int init()
  {
    System.out.println
      ("Field initializer of A");
    return 0;
  }

  static int sf = sinit();
  int f = init();

  public A()
  {
    System.out.println
      ("Constructor of A");
  }
}

public class B extends A
{
  static int sf = sinit();
  int f = init();

  static
  {
    System.out.println
      ("Static initializer of B");
  }

  {
    System.out.println
      ("Initializer of B");
  }

  static int sinit()
  {
    System.out.println
    ("Static field initializer of B");
    return 0;
  }

  static int init()
  {
    System.out.println
      ("Field initializer of B");
    return 0;
  }

  public B()
  {
    System.out.println
      ("Constructor of B");
  }
}

public class C
{
  public static void main
    (String[] args)
  {
    B b = new B();
```

```
    ("Static field initializer of A");
    return 0;
  }

  static int init()
  {
    Console.WriteLine
      ("Field initializer of A");
    return 0;
  }

  static int sf = sinit();
  int f = init();

  public A()
  {
    Console.WriteLine
      ("Constructor of A");
  }
}

public class B : A
{
  static int sf = sinit();
  int f = init();

  static B()
  {
    Console.WriteLine
      ("Static constructor of B");
  }

static int sinit()
  {
    Console.WriteLine
    ("Static field initializer of B");
    return 0;
  }

  static int init()
  {
    Console.WriteLine
      ("Field initializer of B");
    return 0;
  }

  public B()
  {
    Console.WriteLine
      ("Constructor of B");
  }
}

public class C
{
  public static void Main()
  {
    B b = new B();
  }
```

```
  }
}
```

Результат работы приведенного примера такой.

```
Static initializer of A
Static field initializer of A
Static field initializer of B
Static initializer of B
Initializer of A
Field initializer of A
Constructor of A
Field initializer of B
Initializer of B
Constructor of B
```

В Java элемент типа, помимо доступности, указываемой с помощью модификаторов **`private`**, **`protected`** и **`public`**, может иметь пакетную доступность. Такой элемент может использоваться в типах того же пакета, к которому относится содержащий его тип.

Именно пакетная доступность используется в Java по умолчанию.

`protected` элементы в Java также доступны из типов того пакета, в котором находится содержащий эти элементы тип, т.е. **`protected`**-доступность шире пакетной.

Типы, не вложенные в другие, могут быть либо **`public`** (должно быть не более одного такого типа в файле), либо иметь пакетную доступность.

В Java для полей классов дополнительно к модификаторам доступности и контекста могут использоваться модификаторы **`final`**, **`transient`** и **`volatile`**.

Модификатор **`final`** обозначает, что такое поле не может быть изменено во время работы, но сначала должно быть инициализировано, статическое — в одном из статических инициализаторов, нестатическое — к концу работы каждого из конструкторов. В инициализаторах или конструкторах такое поле может модифицироваться несколько раз.

Модификатор **`final`** у локальных

```
}
```

Результат работы приведенного примера такой.

```
Static field initializer of B
Static constructor of B
Field initializer of B
Static field initializer of A
Static constructor of A
Field initializer of A
Constructor of A
Constructor of B
```

В C# доступность элемента типа по умолчанию — **`private`**.

Имеется дополнительный модификатор доступности — **`internal`**. Элемент, имеющий такую доступность, может быть использован всюду в рамках того же файла, где находится определение этого элемента.

Кроме того, в C# можно использовать комбинированный модификатор **`protected internal`** для указания того, что к данному элементу имеют доступ как наследники содержащего его типа, так и типы в рамках содержащего его файла.

Типы, не вложенные в другие, могут быть либо **`public`**, либо иметь доступность по умолчанию — **`private`**, что означает, что они могут использоваться только в рамках содержащего их пространства имен.

В C# все элементы типов могут быть помечены модификатором **`new`** для точного указания на то, что такой элемент — новый и никак не соотносится с элементами предков данного типа с тем же именем или той же сигнатурой.

Поля классов или структурных типов в C# могут иметь в качестве дополнительных модификаторов **`readonly`** и **`volatile`**.

Модификатор **`readonly`** по своему значению аналогичен модификатору **`final`** в Java. **`readonly`** поля должны инициализироваться к концу работы конструкторов и дальше не могут быть изменены.

Такие поля используются для представления постоянных в ходе работы программы объектов и значений, типы которых не допустимы для

переменных и параметров методов может использоваться примерно в том же значении — невозможность модификации их значений после инициализации.

Поля, помеченные модификатором **`transient`**, считаются не входящими в состояние объекта или класса, подлежащее хранению или передаче по сети.

В Java имеются соглашения о том, как должен быть оформлен интерфейс класса, объекты которого могут быть сохранены или переданы по сети — эти соглашения можно найти в документации по интерфейсу `java.io.Serializable`, который должен реализовываться таким классом. Имеются и другие подобные наборы соглашений, привязанные к определенным библиотекам или технологиям в рамках Java.

Стандартный механизм Java, обеспечивающий сохранение и восстановление объектов, реализующих интерфейс `java.io.Serializable`, по умолчанию сохраняет значения всех полей, кроме помеченных модификатором **`transient`**.

Методы классов в Java могут быть дополнительно помечены модификаторами **`strictfp`** (такой же модификатор могут иметь инициализаторы) и **`synchronized`**.

Значение модификатора **`strictfp`** описано в Лекции 9, в разделе о типах с плавающей точкой.

Значение модификатора **`synchronized`** описывается ниже, в разделе, посвященном многопоточным приложениям.

поддерживаемых языком констант, а также таких, значение которых не может быть вычислено во время компиляции.

Шаблонные типы и операции

В последних версиях обоих языков введены шаблонные, т.е. имеющие типовые параметры, типы и операции.

Ниже приводятся примеры декларации шаблонного метода и его использования в Java и C#. В последнем вызове в обоих примерах явное указание типового аргумента у метода `getTypeName()` необязательно, поскольку он вычисляется из контекста вызова. Если вычислить типовые аргументы вызова метода нельзя, их нужно указывать явно.

```
public class A
{
  public static <T> String getTypeName
    (T a)
  {
    if(a == null) return "NullType";
    else return
```

```
using System;

public class A
{
  public static string getTypeName<T>
    (T a)
  {
```

```
      a.getClass().getName();
  }

  public static void main
    (String[] args)
  {
    String  y = "ABCDEFG";

    System.out.println
      ( getTypeName(y) );
    System.out.println
      ( getTypeName(y.length()) );
    System.out.println
      ( A.<Character>getTypeName
          (y.charAt(1)) );
  }
}
```

В Java в качестве типовых аргументов могут использоваться только ссылочные типы.

Примитивный тип не может быть аргументом шаблона — вместо него нужно использовать соответствующий класс-обертку.

В Java типовые аргументы являются элементами конкретного объекта — они фактически представляют собой набор дополнительных параметров конструктора объекта или метода, если речь идет о шаблонном методе. Поэтому статические элементы шаблонного типа являются общими для всех экземпляров этого типа с разными типовыми аргументами.

```
public class A<T>
{
  public static int c = 0;
  public T t;
}

public class B
{
  public static void main
    (String[] args)
  {
    A.c = 7;

    System.out.println( A.c );
  }
}
```

```
    if(a == null) return "NullType";
    else return
      a.GetType().FullName;
  }

  public static void Main()
  {
    string y = "ABCDEFG";

    Console.WriteLine
      ( getTypeName(y) );
    Console.WriteLine
      ( getTypeName(y.Length) );
    Console.WriteLine
      ( getTypeName<char>(y[1]) );
  }
}
```

В C# любой тип может быть аргументом шаблона.

В C# каждый экземпляр шаблонного класса, интерфейса или структурного типа с определенными аргументами имеет свой набор статических элементов, которые являются общими для всех объектов такого полностью определенного типа.

```
using System;

public class A<T>
{
  public static int c = 0;
  public T t;
}

public class B
{
  public static void Main()
  {
    A<string>.c = 7;

    Console.WriteLine( A<int>.c );
    Console.WriteLine( A<string>.c );
  }
}
```

В C# можно определить и использовать шаблонные делегатные типы.

```
public delegate bool Predicate<T>
  (T value);

public class I
{
  public bool m(int i)
```

```
  { return i == 0; }

  public void f()
  {
    Predicate<int> pi = m;
    Predicate<string> ps =
      delegate(string s)
      { return s == null; };
  }
}
```

В обоих языках имеются конструкции для указания ограничений на типовые параметры шаблонных типов и операций. Такие ограничения позволяют избежать ошибок, связанных с использованием операций типа-параметра, точнее, позволяют компилятору обнаруживать такие ошибки.

Ограничения, требующие от типа-параметра наследовать некоторому другому типу, позволяют использовать операции и данные типа-параметра в коде шаблона.

В Java можно указать, что тип-параметр данного шаблона должен быть наследником некоторого класса и/или реализовывать определенные интерфейсы.

В приведенном ниже примере параметр `T` должен наследовать классу `A` и реализовывать интерфейс `B`.

```
public class A
{
  public int m() { ... }
}

public interface B
{
  public String n();
}

public class C<T extends A & B>
{
  T f;

  public String k()
  {
    return f.n() + (f.m()*2);
  }
}
```

Кроме того, в Java можно использовать *неопределенные типовые параметры (wildcards)* при описании типов. Неопределенный типовой параметр может быть ограничен требованием наследовать определенному типу или, наоборот, быть предком определенного типа.

Неопределенные типовые параметры используют в тех случаях, когда нет никаких зависимостей между этими

В C# можно указать, что тип-параметр должен быть ссылочным, типом значения, наследовать определенному классу и/или определенным интерфейсам, а также иметь конструкторы с заданной сигнатурой.

В приведенном ниже примере параметр `T` должен быть ссылочным типом, параметр `V` — типом значений, а параметр `U` — наследовать классу `A`, реализовывать интерфейс `IList<T>` и иметь конструктор без параметров.

```
public class A { ... }

public class B<T, U, V>
  where T : class
  where U : A, IList<T>, new()
  where V : struct
{ ... }
```

параметрами, между ними и типами полей, типами результатов методов и исключений. В таких случаях введение специального имени для типового параметра не требуется, поскольку оно будет использоваться только в одном месте — при описании самого этого параметра.

В приведенном ниже примере первый метод работает с коллекцией произвольных объектов, второй — с коллекцией объектов, имеющих (не обязательно точный) тип `T`, третий — с такой коллекцией, в которую можно добавить элемент типа `T`.

```
public class A
{
  public void addAll
    (Collection<?> c)
  { ... }

  public <T> void addAll
    (Collection<? extends T> c)
  { ... }

  public <T> void addToCollection
    (T e, Collection<? super T> c)
  { ... }
}
```

Дополнительные элементы описания операций

В обоих языках (в Java — начиная с версии 5) имеются конструкции, позволяющие описывать операции с неопределенным числом параметров (как в функции `printf` стандартной библиотеки C). Для этого последний параметр нужно пометить специальным образом. Этот параметр интерпретируется как массив значений указанного типа. При вызове такой операции можно указать обычный массив в качестве ее последнего параметра, но можно и просто перечислить через запятую значения элементов этого массива или ничего не указывать в знак того, что он пуст. Детали декларации таких параметров несколько отличаются.

В Java нужно указать тип элемента массива, многоточие и имя параметра.

```
public class A
{
  public int f(int ... a)
  {
    return a.length;
  }

  public static void main
    (String[] args)
  {
    A a = new A();

    System.out.println
      ( a.f(new int[]{9, 0}) );
    System.out.println
      ( a.f(1, 2, 3, 4) );
  }
}
```

В C# нужно указать модификатор **`params`**, тип самого массива и имя параметра.

```
using System;

public class A
{
  public int f(params int[] a)
  {
    return a.Length;
  }

  public static void Main
  {
    A a = new A();

    Console.WriteLine
      ( a.f(new int[]{9, 0}) );
    Console.WriteLine
      ( a.f(1, 2, 3, 4) );
  }
```

В Java требуется указывать некоторые типы исключений, возникновение которых возможно при работе метода, в заголовке метода.

Точнее, все исключения делятся на два вида — *проверяемые (checked)* и *непроверяемые (unchecked)*. Непроверяемыми считаются исключения, возникновение которых может быть непреднамеренно — обращение к методу или полю по ссылке, равной **`null`**, превышение ограничений на размер стека или занятой динамической памяти, и пр. Проверяемые исключения предназначены для передачи сообщений о возникновении специфических ситуаций и всегда явно создаются в таких ситуациях.

Непроверяемое исключение должно иметь класс, наследующий `java.lang.Error`, если оно обозначает серьезную ошибку, которая не может быть обработана в рамках обычного приложения, или `java.lang.RuntimeException`, если оно может возникать при нормальной работе виртуальной машины Java. Если класс исключения не наследует одному из этих классов, оно считается проверяемым.

Все классы проверяемых исключений, возникновение которых возможно при работе метода, должны быть описаны в его заголовке после ключевого слова **`throws`**.

Если некоторый метод вызывает другой, способный создать проверяемое исключение типа `T`, то либо этот вызов должен быть в рамках **`try`**-блока, для которого имеется обработчик исключений этого или более общего типа, либо вызывающий метод тоже должен указать тип `T` среди типов исключений, возникновение которых возможно при его работе.

```
public void m(int x)
  throws MyException
{
  throw new MyException();
}

public void n(int x)
  throws MyException
{
  m(x);
}

public void k(int x)
}
```

В C# исключения, возникновение которых возможно при работе метода, никак не описываются.

```
{
  try { m(x); }
  catch(MyException e)
  { ... }
}
```

В Java все параметры операций передаются по значению.

Поскольку все типы, кроме примитивных, являются ссылочными, значения таких типов — ссылки на объекты. При выполнении операции можно изменить состояние объекта, переданного ей в качестве параметра, но не ссылку на него.

В C# можно определить параметры операций, передаваемые по ссылке и выходные параметры операций.

Параметры, передаваемые по ссылке, помечаются модификаторов **`ref`**. Выходные параметры помечаются модификатором **`out`**.

При вызове операции значения этих параметров должны быть помечены так же.

```
using System;

public class A
{
  public void f
    (int a, ref int b, out int c)
  {
    c = b - a;
    b += a;
  }

  public static void Main()
  {
    A a = new A();
    int n = 3, m = 0;

    Console.WriteLine
      ("n = " + n + " , m = " + m);
    a.f(1, ref n, out m);
    Console.WriteLine
      ("n = " + n + " , m = " + m);
  }
}
```

Описание метаданных

В обоих языках (в Java — начиная с версии 5) имеются встроенные средства для некоторого их расширения, для описания так называемых метаданных — данных, описывающих элементы кода. Это специальные модификаторы у типов, элементов типов и параметров операций, называемые в Java ***аннотациями (annotations)***, а в C# — ***атрибутами (attributes)***. Один элемент кода может иметь несколько таких модификаторов.

Такие данные служат для указания дополнительных свойств классов, полей, операций и параметров операций. Например, можно пометить специальным образом поля класса, которые должны записываться при преобразовании объекта этого класса в поток байтов для долговременного хранения или передачи по сети. Можно пометить методы, которые должны работать только в рамках транзакций или, наоборот, только вне транзакций.

Метаданные служат встроенным механизмом расширения языка, позволяя описывать простые дополнительные свойства сущностей этого языка в нем самом, не разрабатывая каждый раз специализированные трансляторы. Обработка метаданных должна, конечно, осуществляться дополнительными инструментами, но такие инструменты могут быть достаточно просты — им не нужно реализовывать функции компилятора исходного языка.

В обоих языках аннотации могут иметь структуру — свойства или параметры, которым можно присваивать значения. Эту структуру можно определить в описании специального аннотационного типа.

В приведенных ниже примерах определяются несколько типов аннотаций, которые затем используются для разметки элементов кода. Класс `A` помечен аннотацией, имеющей указанные значения свойств, оба метода помечены простой аннотаций (указывающей, например, что такой метод должен быть обработан особым образом), кроме того, метод `n()` помечен еще одной аннотацией. Параметр метода `n()` также помечен простой аннотацией.

```
@interface SimpleMethodAnnotation {}
```

```
class SimpleMethodAttribute
  : Attribute
{}
```

```
@interface SimpleParameterAnnotation{}
```

```
class SimpleParameterAttribute
  : Attribute
{}
```

```
@interface ComplexClassAnnotation
{

  int id();
  String author() default
    "Victor Kuliamin";
  String date();
}
```

```
class ComplexClassAttribute
  : Attribute
{
  public int id;
  public String author =
    "Victor Kuliamin";
  public String date;
}
```

```
@interface AdditionalMethodAnnotation
{

  String value() default "";
}
```

```
class AdditionalMethodAttribute
  : Attribute
{
  public String value = "";
}
```

```
@ComplexClassAnnotation
(
  id     = 126453,
  date   = "23.09.2005"
)
public class A
{
  @SimpleMethodAnnotation
  public void m() { ... }

  @SimpleMethodAnnotation
  @AdditionalMethodAnnotation
  ( value = "123" )
  public void n
    (@SimpleParameterAnnotation int k)
  { ... }
}
```

```
[ComplexClassAttribute
 (
    id     = 126453,
    date   = "23.09.2005"
 )]
public class A
{
  [SimpleMethodAttribute]
  public void m() { ... }

  [SimpleMethodAttribute,
   AdditionalMethodAttribute
   (value = "123")]
  public void n
  ([SimpleParameterAttribute] int k)
  { ... }
}
```

В Java аннотации могут помечать также пакеты (т.е. использоваться в директиве декларации пакета, к которому относится данный файл), декларации локальных переменных и константы перечислимых типов.

В C# могут быть описаны глобальные атрибуты, помечающие сборку, в рамках кода которой они встречаются. Такие атрибуты помещаются вне описаний пространств имен.

Также, атрибутами могут помечаться отдельные методы доступа к свойствам, индексерам и событиям.

Аннотационный тип декларируется с

Атрибутный тип описывается как класс,

модификатором **`@interface`** и неявно наследует интерфейсу `java.lang.annotation.Annotation`.

Такой тип не может иметь типовых параметров или явным образом наследовать другому типу.

Свойства аннотационного типа описываются как абстрактные методы без параметров, с возможным значением по умолчанию.

При определении значений свойств аннотации через запятую перечисляются пары <имя свойства> = <значение>.

Помимо свойств, в аннотационном типе могут быть описаны константы (**`public static final`** поля) и вложенные типы, в том числе аннотационные.

Свойство аннотационного типа может иметь примитивный тип, тип `String`, `Class`, экземпляр шаблонного типа `Class`, перечислимый тип, аннотационный тип или быть массивом элементов одного из перечисленных типов.

наследующий `System.Attribute` или его наследнику.
Такой тип не может иметь типовых параметров.

Свойства атрибутного типа могут быть *именованными параметрами* и *позиционными параметрами*.

Позиционные параметры определяются при помощи конструкторов атрибутного типа.

Именованные параметры определяются как доступные на чтение и запись нестатические поля и свойства атрибутного типа.

При определении значений свойств атрибута сначала через запятую перечисляются значения позиционных параметров, а затем пары <имя именованного параметра> = <значение>.

В атрибутном типе могут быть описаны такие же элементы, как и в обычном классе.

Ниже приведен пример использования позиционного параметра.

```
class ClassAttribute : Attribute
{
  public ClassAttribute(int id)
  {
    this.id = id;
  }

  int id;
  public string value;
}

[ClassAttribute(4627, value = "")]
public class A { ... }
```

Свойство атрибутного типа может иметь один из типов **`bool`**, **`byte`**, **`char`**, **`double`**, **`float`**, **`int`**, **`long`**, **`short`**, **`string`**, **`object`**, `System.Type`, перечислимый тип или быть массивом элементов одного из таких типов.

У атрибутов может указываться цель, к которой они привязываются. Для атрибутов, помещаемых вне рамок пространств имен, указание такой цели — assembly — обязательно.

```
[assembly :
  MyAttribute
  (
    id = 4627,
    author = "Victor Kuliamin"
```

```
)]
```

В C# принято соглашение, согласно которому окончание `Attribute` в именах атрибутных типов может отбрасываться при использовании атрибутов такого типа.

Поэтому, имея атрибутный тип `ClassAttribute`, можно использовать в атрибутах как полное имя `ClassAttribute`, так и сокращенное `Class`.

В последнем случае компилятор будет пытаться найти атрибутный тип с именем `Class` или с именем `ClassAttribute`. При этом возможна неоднозначность — оба таких типа могут существовать в рамках сборки и доступных библиотек. Для ее разрешения можно использовать точное имя атрибутного типа с префиксом `@`. Увидев атрибут `@Class`, компилятор будет искать атрибутный тип в точности с именем `Class`.

Средства создания многопоточных программ

Как в Java, так и в C# возможно создание многопоточных приложений. Вообще говоря, каждая программа на этих языках представляет собой набор ***потоков (threads)***, выполняющихся параллельно. Каждый поток является исполняемым элементом, имеющим свой собственный поток управления и стек вызовов операций. Все потоки в рамках одного ***процесса*** (одной виртуальной машины Java или одного процесса среды .NET) имеют общий набор ресурсов, общую память, общий набор объектов, с которыми могут работать.

Каждый поток представляется в языке объектом некоторого класса (`java.lang.Thread` в Java и `System.Threading.Thread` в C#). Для запуска некоторого кода в виде отдельного потока необходимо определить особую операцию в таком объекте и выполнить другую его операцию.

В Java это можно сделать двумя способами.

Первый — определить класс-наследник `java.lang.Thread` и перегрузить в этом классе метод **`public void`** `run()`. Этот метод, собственно и будет выполняться в виде отдельного потока.

Другой способ — определить класс, реализующий интерфейс `java.lang.Runnable` и его метод **`void`** `run()`. После чего построить объект класса `Thread` на основе объекта только что определенного класса.

В обоих случаях для запуска выполнения потока нужно вызвать в объекте класса `Thread` (в первом случае — его наследника) метод **`void`** `start()`.

В C# также можно использовать два способа. С помощью первого можно создать обычный поток, с помощью второго — поток, которому при запуске нужно передать какие-то данные.

Для этого нужно определить метод, который будет выполняться в рамках потока. Этот метод должен иметь тип результата **`void`**. Список его параметров в первом случае должен быть пустым, во втором — состоять из одного параметра типа **`object`**.

В первом варианте на основе этого метода создается делегат типа `System.Threading.ThreadStart`, во втором — типа `System.Threading.ParameterizedThreadStart`.

Этот делегат передается в качестве аргумента конструктору объекта класса `System.Thread`.

Поток запускается выполнением метода `Start()` у объекта класса `Thread` в первом случае, или метода `Start(object)` во втором.

```
class T extends Thread
{
  int id = 0;
  public T(int id)
  { this.id = id; }

  public void run()
  {
    System.out.println
    ("Thread " + id + " is working");
  }
}

public class A
{
  public static void main
    (String[] args)
  {
    Thread th1 = new T(1),
           th2 = new T(2),
           th3 = new Thread(
      new Runnable() {
        public void run()
        {
          System.out.println
          ("Runnable is working");
        }
      });

    th1.start();
    th2.start();
    th3.start();
  }
}
```

```
using System;
using System.Threading;

class T
{
  int id;
  public T(int id)
  { this.id = id; }

  public void m()
  {
    Console.WriteLine
    ("Thread " + id + " is working");
  }
}

public class A
{
  static void m()
  {
    Console.WriteLine
    ("Nonparameterized thread" +
     " is working");
  }

  static void m(object o)
  {
    Console.WriteLine
    ("Thread with object " + o +
     " is working");
  }

  public static void Main()
  {
    Thread th1 = new Thread(
      new ThreadStart(m)),
           th2 = new Thread(
      new ThreadStart(new T(1).m)),
           th3 = new Thread(
    new ParameterizedThreadStart(m));

    th1.Start();
    th2.Start();
    th3.Start(2);
  }
}
```

При разработке приложений, основанных на параллельном выполнении нескольких потоков, большое значение имеют вопросы синхронизации работы этих потоков. Синхронизация позволяет согласовывать их действия и аккуратно передавать данные, полученные в одном потоке, в другой. И недостаточная синхронизация, и избыточная приводят к серьезным проблемам. При недостаточной синхронизации один поток может начать работать с данными, которые еще находятся в обработке у другого, что приведет к некорректным итоговым результатам. При избыточной синхронизации как минимум

производительность приложения может оказаться слишком низкой, а в большинстве случаев приложение просто не будет работать из-за возникновения *тупиковых ситуаций (deadlocks)*, в которых два или более потоков не могут продолжать работу, поскольку ожидают друг от друга освобождения необходимых им ресурсов.

В обоих языках имеются конструкции, реализующие синхронизационный примитив, называемый ***монитором (monitor)***. Монитор представляет собой объект, позволяющий потокам «захватывать» и «отпускать» себя. Только один поток может «держать» монитор в некоторый момент времени — все остальные, попытавшиеся захватить монитор после его захвата этим потоком, будут приостановлены до тех пор, пока этот поток не отпустит монитор.

Для синхронизации используется конструкция, гарантирующая, что некоторый участок кода в каждый момент времени выполняется не более чем одним потоком. В начале этого участка нужно захватить некоторый монитор, в качестве которого может выступать любой объект ссылочного типа, в конце — отпустить его. Такой участок помещается в блок (или представляется в виде одной инструкции), которому предшествует указание объекта-монитора с ключевым словом **`synchronized`** в Java или **`lock`** в C#.

```
public class PingPong extends Thread
{
  boolean odd;

  PingPong (boolean odd)
  { this.odd = odd; }

  static int counter = 1;
  static Object monitor =
    new Object();

  public void run()
  {
    while(counter < 100)
    {
      synchronized(monitor)
      {
        if(counter%2 == 1 && odd)
        {
          System.out.print("Ping ");
          counter++;
        }
        if(counter%2 == 0 && !odd)
        {
          System.out.print("Pong ");
          counter++;
        }
      }
    }
  }

  public static void main
    (String[] args)
  {
    Thread th1 = new PingPong (false),
           th2 = new PingPong (true);

    th1.start();
    th2.start();
```

```
using System;
using System.Threading;

public class PingPong
{
  bool odd;

  PingPong (bool odd)
  { this.odd = odd; }

  static int counter = 1;
  static object monitor =
    new object();

  public void Run()
  {
    while(counter < 100)
    {
      lock(monitor)
      {
        if(counter%2 == 1 && odd)
        {
          Console.Write("Ping ");
          counter++;
        }
        if(counter%2 == 0 && !odd)
        {
          Console.Write("Pong ");
          counter++;
        }
      }
    }
  }

  public static void Main()
  {
    Thread th1 = new Thread(
      new ThreadStart(
        new PingPong(false).Run)),
          th2 = new Thread(
      new ThreadStart(
        new PingPong(true).Run));
```

```
  }
}
```

```
    th1.Start();
    th2.Start();
  }
}
```

Кроме того, в Java любой метод класса может быть помечен как **`synchronized`**. Это значит, что не более чем один поток может выполнять этот метод в каждый момент времени в рамках объекта, если метод нестатический и в рамках всего класса, если он статический.

Такой модификатор эквивалентен помещению всего тела метода в блок, синхронизированный по объекту **`this`**, если метод нестатический, а если метод статический — по выражению **`this`**`.getClass()`, возвращающему объект, представляющий класс данного объекта.

В Java также имеется стандартный механизм использования любого объекта ссылочного типа в качестве монитора для создания более сложных механизмов синхронизации.

Для этого в классе `java.lang.Object` имеются методы `wait()`, приостанавливающие текущий поток до тех пор, пока другой поток не вызовет метод `notify()` или `notifyAll()` в том же объекте, или пока не пройдет указанное время. Все эти методы должны вызываться в блоке, синхронизированном по данному объекту.

Однако это механизм достаточно сложен в использовании и не очень эффективен. Для реализации более сложной синхронизации лучше пользоваться библиотечными классами из пакетов `java.util.concurrent` и `java.util.concurrent.locks`, появившихся в JDK версии 5 (см. ниже).

Библиотеки

Обе платформы, как Java, так и .NET, в большой степени опираются на библиотеки готовых компонентов. Сообщество разработчиков ведет постоянную работу по выработке стандартных интерфейсов компонентов для решения различных задач в разных предметных областях. По достижении определенной степени зрелости такие интерфейсы включаются в стандартные библиотеки.

Для Java этот процесс начался значительно раньше и захватил гораздо больше участников в силу более раннего появления платформы и ее большей открытости и стандартизованности. Поэтому далеко не все часто используемые библиотеки классов

распространяются в составе платформ J2SE и J2EE. Разработка библиотек .NET ведется, в основном, силами программистов, работающих в Microsoft и ряде ее партнеров.

В данном разделе дается краткий обзор основных библиотек Java и .NET, подробное описание см. в [10] и [11].

Основные классы языка Java содержатся в пакете `java.lang`.

Базовым классом для всех ссылочных типов Java служит класс `java.lang.Object`.

Этот класс содержит следующие методы.

`boolean` `equals(Object)` — предназначен для сравнения объектов с учетом их внутренних данных, перегружается в наследниках. В классе `Object` сравнивает объекты на совпадение.

`int` `hashCode()` — возвращает хэш код данного объекта, используется в хэширующих коллекциях. Должен перегружаться одновременно с методом `equals()`.

`String toString()` — преобразует данный объект в строку, перегружается в наследниках. В классе `Object` выдает строку из имени класса и уникального кода объекта в JVM.

`Class<?` **`extends`** `Object> getClass()` — возвращает объект, представляющий класс данного объекта.

`protected void` `finalize()` — вызывается сборщиком мусора на одном из этапов удаления объекта из памяти. Может быть перегружен.

`protected` `Object clone()` — предназначен для построения копий данного объекта, перегружается в наследниках. В классе `Object` копирует поля данного объекта в новый, если класс данного объекта реализует интерфейс `java.lang.Cloneable`, иначе выбрасывает исключение.

`void` `wait()`, **`void`** `wait(`**`long`**` timeout)`, **`void`** `wait(`**`long`**` timeout, `**`int`**` nanos)` — методы, приостанавливающие выполнение текущего потока до вызова `notify()` или `notifyAll()` другим потоком в данном объекте или до истечения заданного интервала времени.

`void` `notify()`, **`void`** `notifyAll()` — методы, оповещающие потоки, ждущие

Основные классы C# содержатся в пространстве имен `System` в сборке mscorlib.

Базовым типом для всех типов C# служит класс `System.Object`, который также имеет имя **`object`**.

`bool` `Equals(`**`object`**`)` — аналог метода `equals()` в Java.

`static bool` `Equals(`**`object, object`**`)` — сравнивает два объекта с помощью `Equals()` или на равенство обеих ссылок **`null`**.

`static bool` `ReferenceEquals(`**`object, object`**`)` — сравнивает ссылки на заданные объекты.

`int` `GetHashCode()` — аналог метода `hashCode()` в Java. Должен перегружаться одновременно с методом `Equals()`.

`string` `ToString()` — аналог метода `toString()` в Java. В **`object`** выдает только имя типа данного объекта.

`System.Type GetType()` — возвращает объект, представляющий тип данного объекта.

`protected object` `MemebrwiseClone()` — создает копию данного объекта, имеющую те же значения всех полей.

оповещения по данному объекту. Первый метод «отпускает» только один из ждущих потоков, второй — все.

Класс System предоставляет доступ к элементам среды выполнения программы и ряд полезных утилит. Все его элементы — статические.

Поля in, out и err в этом классе представляют собой ссылки на стандартные потоки ввода, вывода и вывода информации об ошибках. Они могут быть изменены при помощи методов setIn(), setOut() и setErr().

Методы **long** currentTimeMillis() и **long** nanoTime() служат для получения текущего значения времени.

void exit(**int** status) — прекращает выполнение Java машины, возвращая указанное число внешней среде в качестве кода выхода.

void gc() — запускает сборку мусора. Время от времени сборка мусора запускается и самостоятельно.

Методы getenv(), getProperties() и getProperty() служат для получения текущих значений переменных окружения и свойств Java машины, задаваемых ей при запуске с опцией –d.

load(), loadLibrary() — служат для загрузки библиотек, например, реализующих native интерфейсы.

void arraycopy() — используется для быстрого копирования массивов.

int identityHashCode(Object) — возвращает уникальный числовой идентификатор данного объекта в Java машине.

Другой класс, содержащий методы работы со средой выполнения, — Runtime.

Для работы со строками используются классы String, StringBuffer и StringBuilder (последний появился в Java 5). Все они реализуют интерфейс последовательностей символов CharSequence.

Класс String представляет неизменяемые строки, два других класса — изменяемые. Отличаются они тем, что все операции StringBuffer синхронизованы, а операции

Данные о среде выполнения можно получить с помощью класса Environment.

В нем имеются методы GetEnvironmentVariables() и GetEnvironmentVariable() для получения значений переменных окружения, методы для получения командной строки, метод Exit(**int**) для прекращения работы текущего процесса, свойства с данными о машине и текущем пользователе, свойство TickCount, хранящее количество миллисекунд с момента запуска системы, и пр.

Управлять стандартным вводом-выводом можно с помощью класса Console.

Он содержит свойства In, Out, Err, методы для чтения из потока стандартного ввода и для записи в поток стандартного вывода, а также много других свойств консоли.

Для работы со строками используются классы System.String, представляющий неизменяемые строки, и System.Text.StringBuilder, представляющий изменяемые строки.

`StringBuilder` — нет. Соответственно, первый класс нужно использовать для представления строк, с которыми могут работать несколько потоков, а второй — для повышения производительности в рамках одного потока.	
В пакете `java.lang` находятся классы-обертки примитивных типов `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`.	Обертками примитивных типов C# служат следующие структурные типы из пространства имен `System`.
Все числовые классы наследуют классу `java.lang.Number`.	`Boolean`, `Byte`, `SByte`, `Int16`, `Int32`, `Int64`, `UInt16`, `UInt32`, `UInt64`, `Single`, `Double`, `Decimal`.
Эти классы содержат константы, представляющие крайние значения соответствующих типов, методы для преобразования значений соответствующих типов в строку и для получения этих значений из строк, а также методы для работы с битовым представлением значений числовых типов.	Они также содержат аналогичные константы и методы.
Для форматированного строкового представления числовых данных используются классы `java.text.NumberFormat` и `java.text.DecimalFormat`.	Для работы с форматированным строковым представлением чисел используются методы `Parse()` и `ToString()` тех же классов с дополнительными параметрами, реализующими интерфейс `IFormatProvider`. Чаще всего это объекты класса `System.Globalization.NumberFormatInfo`.
Набор математических функций и констант реализован в виде элементов класса `java.lang.Math`.	Набор математических функций и констант реализован в виде элементов класса `System.Math`.
Для генерации псевдослучайных чисел можно использовать как метод `Math.random()`, так и обладающий большей функциональностью класс `java.util.Random`.	Для генерации псевдослучайных чисел используется класс `System.Random`.
Для более сложных вычислений можно использовать классы пакета `java.math` — `BigInteger` и `BigDecimal`, представляющие целые числа произвольной величины и десятичные дроби произвольной точности.	
`java.lang.Thread` — класс, объекты которого представляют потоки в Java машине.	Классом, представляющим потоки .NET в C#, является `System.Threading.Thread`.
Он содержит методы, позволяющие прервать ожидание данным потоком синхронизационной операции (`interrupt()`), подождать конца работы данного потока (`join()`), запустить поток (`start()`), приостановить выполнение текущего потока на какое-то время (`sleep()`), получить текущий поток (`currentThread()`), а также получить	Текущий поток может быть получен с помощью его свойства `CurrentThread`. Этот класс тоже содержит методы `Interrupt()`, `Join()`, `Start()`, `Sleep()` и др.

различные характеристики данного потока (приоритет, имя, и пр.).

Класс `java.lang.ThreadLocal` служит для хранения значений, которые должны быть специфичны для потока. Т.е. значение, получаемое методом `get()` из объекта этого класса — то самое, которое текущий поток сохранил ранее с помощью метода `set()`.

В пакете `java.lang` находится и набор наиболее важных интерфейсов.

`CharSequence` — интерфейс последовательности символов.

`Cloneable` — маркирующий интерфейс объектов, для которых можно строить копии. Сам по себе он не требует реализации каких-либо методов, хотя для построения копий нужно перегрузить метод `clone()`, а лишь позволяет использовать функциональность этого метода в классе `Object`.

`Iterable<T>` — интерфейс итерируемых коллекций. Объекты, реализующие этот интерфейс, могут, наравне с массивами, использоваться в инструкции цикла по коллекции.

`Comparable<T>` — интерфейс, реализуемый классами, объекты которых линейно упорядочены, т.е. любые два объекта этого класса сравнимы по отношению «больше/меньше». Используется, например, для реализации коллекций с быстро работающими функциями поиска.

В пакете `java.lang` также находится ряд классов, объекты которых представляют элементы самого языка — `Class<T>`, представляющий классы, `Enum<T>`, представляющий перечислимые типы, и `Package`, представляющий пакеты.

Все эти классы, а также классы, находящиеся в пакете `java.lang.reflect` (`Field`, `Method`, `Constructor`) используются в рамках механизма ***рефлексии (reflection)*** для доступа к информации об элементах языка во время выполнения программы. На механизме рефлексии построены очень многие среды и технологии, входящие в платформу Java, например JavaBeans.

Для анализа структуры и элементов массивов во время работы программы можно использовать методы класса

Аналогичные интерфейсы в C# следующие.

`System.ICloneable` — требует реализации метода `Clone()` для создания копий.

`System.Collections.IEnumerable`, `System.Collections.Generic.IEnumerable<T>` — интерфейсы итерируемых коллекций.

`System.IComparable`, `System.IComparable<T>` — интерфейсы типов, объекты которых линейно упорядочены.

В C#, в отличии от Java, где шаблонные типы всегда имеют и непривязанный к типам-аргументам интерфейс, шаблоны с типизированным интерфейсом просто добавляются в библиотеки к уже существовавшим там до появления C# 2.0 нетипизированным классам и интерфейсам.

Для C# аналогичную роль в механизме рефлексии играют классы `System.Type`, `System.ValueType`, `System.Enum` и классы, входящие в пространство имен `System.Reflection` — `MemberInfo`, `MethodInfo`, `FieldInfo`, и т.д.

Аналогом обоих классов Java для работы с массивами служит класс `System.Array`.

Он содержит как методы анализа структуры

`java.lang.reflect.Array`, позволяющие определить тип элементов массива, получить их значения, создавать новые массивы и пр.

Для манипуляций с массивами — поиска, сортировки, сравнения и пр. — используются методы класса `java.util.Arrays`.

массива, так и операции для поиска, упорядочения, копирования и сравнения массивов.

В обоих языках имеются механизмы, позволяющие определять ***слабые ссылки***. Объект, хранящийся по слабой ссылке, считается сборщиком мусора недоступным по ней, и поэтому может быть уничтожен при очередном запуске процедуры сборки мусора, если обычных ссылок на него не осталось.

Слабые ссылки удобны для организации хранилищ объектов, которые не должны мешать их уничтожению, если эти объекты стали недоступны во всех других местах, т.е. ссылки на них остались только из такого хранилища.

Пакет `java.lang.ref` содержит классы для организации сложной работы со ссылками.

Например, класс `java.lang.ref.WeakReference` представляет слабые ссылки.

Другие классы представляют более хитрые виды ссылок.

В C# слабые ссылки реализуются при помощи класса `System.WeakReference`.

Пакет `java.util` содержит классы и интерфейсы, представляющие разнообразные коллекции объектов.

`Collection<T>` — общий интерфейс коллекций Java.

`Collections` — предоставляет набор общеупотребительных операций над коллекциями: построение коллекций с различными свойствами, поиск, упорядочение и пр.

`Set<T>`, `HashSet<T>`, `TreeSet<T>` — интерфейс множества объектов и различные его реализации. `TreeSet<T>` требует, чтобы объекты типа `T` были линейно упорядоченными, и предоставляет быстро работающие (за логарифмическое время от числа объектов в множестве) функции добавления, удаления и поиска.

`Map<K, V>`, `HashMap<K, V>`, `TreeMap<K, V>` — интерфейс ассоциативных массивов или отображений (maps) и его различные реализации. `TreeMap<K, V>` требует, чтобы объекты-ключи были линейно упорядоченными, и предоставляет быстрые операции с отображением.

`List<T>`, `ArrayList<T>`, `LinkedList<T>` — интерфейс расширяемого списка и

Набор интерфейсов и классов коллекций C# находится в пространствах имен `System.Collections` и `System.Collections.Generic`. В первом находятся интерфейсы и классы нетипизированных коллекций, во втором — шаблоны. Далее упоминаются только шаблонные типы, если их нетипизированный аналог называется соответствующим образом.

Базовые интерфейсы коллекций — `Generic.IEnumerable<T>`, `Generic.ICollection<T>`.

Интерфейс отображений и классы, реализующие отображения — `Generic.IDictionary<K, V>`, `Generic.Dictionary<K, V>`, `Hastable` (нетипизированный).

Интерфейс и классы списков — `Generic.IList<T>`, `Generic.List<T>`, `ArrayList` (нетипизированный).

`BitArray` — реализует расширяемый список битов.

различные его реализации.

`BitSet` — реализует расширяемый список флагов-битов.

`IdentityHashMap<K, V>` реализует отображение, сравнивающее свои ключи по их совпадению, а не с помощью метода `equals()`, как это делают остальные реализации `Map<K, V>` (и `Set<T>`).

`WeakHashMap<K, V>` хранит ключи с помощью слабых ссылок, что позволяет автоматически уничтожать хранящиеся в таком отображении пары ключ-значение, если на объект-ключ других ссылок не осталось.

Много полезных классов-коллекций можно найти вне стандартных библиотек, например, в библиотеке Jakarta Commons [12], части обширного проекта Apache Jakarta Project [13].

Для определения линейного порядка на объектах типа `T` используется интерфейс `java.util.Comparator<T>`.

Наиболее часто используется его реализация для строк — `java.text.Collator`, абстрактный класс, позволяющий создавать специфические объекты, сравнивающие строки в различных режимах, включая игнорирование регистра символов, использование национально-специфических символов и пр.

Аналогичный интерфейс в C# — `System.Collections.Generic.IComparer<T>`.

Его реализация для строк — абстрактный класс `System.StringComparer`.

Классы `java.util.Calendar`, `java.util.GregorianCalendar`, `java.util.Date` и `java.text.DateFormat` используются для работы с датами и временами.

Первые два класса используются для представления информации о календаре, объекты класса `Date` представляют даты и моменты времени, а последний класс используется для их конвертации в форматированный текст и обратно.

Класса для представления временных интервалов в стандартной библиотеке нет.

Гораздо более широкий набор средств для работы с датами и временами предоставляет библиотека Joda [14].

Для представления различных календарей используются подклассы `System.Globalization.Calendar`, тоже находящиеся в пространстве имен `System.Globalization`.

Для представления моментов времени — `System.DateType`.

Для интервалов времени — `System.TimeSpan`.

Для форматированного текстового представления дат и времени — `System.Globalization.DateTimeFormatInfo`.

Классы `java.util.Timer` и `java.util.TimerTask` служат выполнения

Аналоги в C# — `System.Threading.Timer` и делегатный тип

определенных действий в заданное время или через определенное время.	`System.Threading.TimerCallback`. Еще одни аналоги находятся в пространстве имен `System.Timers` сборки System.
Для построения и анализа форматированных строк, содержащих данные различных типов, полезен класс `java.util.Formatter`. С помощью реализации интерфейса `java.util.Formattable` можно определить разные виды форматирования для объектов пользовательских типов.	В C# преобразование в форматированную строку осуществляется методом `ToString()` с параметром типа `System.IFormatProvider`. Такой метод есть в типах, реализующих интерфейс `System.IFormattable`. Обычно в качестве объектов, задающих форматирование, используются объекты классов `System.Globalization.CultureInfo`, `System.Globalization.NumberFormatInfo` и `System.Globalization.DateTimeFormatInfo`.
В пакете `java.util` есть несколько классов, представляющих регионально- или национально-специфическую информацию С помощью объектов класса `Currency` представляют различные валюты. Информация о региональных или национально-специфических настройках и параметрах представляется в виде объектов класса `Locale`. Набор объектов, имеющих локализованные варианты, например, строк сообщений на разных языках, может храниться с помощью подклассов `ResourceBundle` — `ListResiourceBundle`, `PropertyResourceBundle`.	Аналогичную роль в C# играют классы пространства имен `System.Globalization` — `RegionInfo` и `CultureInfo`. Для хранения наборов объектов вместе с их аналогами для нескольких культур используются объекты класса `System.Resources.ResourceSet`.
Работа с различными кодировками текста организуется при помощи классов пакета `java.nio.charset`.	Работа с различными кодировками текста организуется при помощи классов пространства имен `System.Text`.
Интерфейс и классы пакета `java.util` `EventListener`, `EventListenerProxy`, `EventObject` используются для реализации образца «подписчик» в рамках спецификаций JavaBeans (см. предыдущую лекцию).	
Класс `java.util.Scanner` реализует простой лексический анализатор текста. Более гибкую работу с регулярными выражениями можно реализовать с помощью классов пакета `java.util.regex`.	В C# работа с регулярными выражениями может быть организована при помощи классов пространства имен `System.Text.RegularExpressions` в сборке System.
Пакет `java.util.concurrent` и его подпакет `locks` содержат набор классов, реализующих коллекции с эффективной синхронизацией работы нескольких	Аналогичные функции выполняют классы пространства имен `System.Threading`, расположенные как в сборке mscorlib, так и в System.

потоков (например, разные потоки могут параллельно изменять значения по разным ключам отображения), различные примитивы синхронизации потоков — барьеры, семафоры, события, затворы (latches), блокировки типа много читателей-один писатель и пр.

Пакет `java.util.concurrent.atomic` содержит классы, реализующие гарантированно атомарные действия с данными различных типов.

Пакет `java.io` содержит класс `File`, представляющий файлы и операции над ними, а также большое количество подклассов абстрактных классов `Reader` и `InputStream`, предназначенных для потокового ввода данных, и `Writer` и `OutputStream`, предназначенных для потокового вывода данных.

Аналогичные классы содержаться в пространстве имен `System.IO`.

Пакет `java.nio` содержат классы для организации более эффективного асинхронного ввода-вывода.

Аналоги классов из `java.nio` находятся в пространстве имен `System.Runtime.Remoting.Channels` в сборках mscorlib и System.Runtime.Remoting.

Классы и интерфейсы, лежащие в основе компонентной модели JavaBeans, находятся в пакете `java.beans`.

На основе этой модели реализованы библиотеки элементов управления графического пользовательского интерфейса (graphical user interface, GUI) Java.

Одна из этих библиотек (наиболее старая и не очень эффективная) размещается в пакете `java.awt`.

Другая, более новая и демонстрирующая б***о***льшую производительность — в пакете `javax.swing`.

Одной из наиболее эффективных библиотек графических элементов управления на Java на данный момент считается библиотека SWT (Standard Widget Toolkit [15]), на основе которой разрабатывается расширяемая среда разработки приложений Eclipse [16].

Классы и интерфейсы, лежащие в основе компонентной модели графических элементов управления .NET, находятся в пространстве имен `System.ComponentModel`.

Эти классы, в основном, расположены в сборке System.

Библиотека элементов GUI находится в пространстве имен `System.Windows.Forms` в рамках сборки `System.Windows.Forms`.

Библиотека классов общего назначения для работы с графикой находится в пространстве `System.Drawing` в сборке `System.Drawing`.

Интерфейсы и классы для разработки сетевого ПО и организации связи между приложениями, работающими на разных машинах, находятся в пакетах `java.net`, `javax.net`, `java.rmi`, `javax.rmi`.

Аналогичные классы и интерфейсы находятся в пространствах имен `System.Net` и `System.Runtime.Remoting` в рамках сборок mscorlib и System.

Пакеты `java.security`, `javax.crypto` и `javax.security` определяют основные интерфейсы и классы для поддержки обеспечения безопасных соединений, шифрования, использования различных протоколов безопасности и различных моделей управления ключами и сертификатами.	Аналогичные классы и интерфейсы находятся в пространстве имен `System.Security` в сборках mscorlib, System и System.Security.
Пакеты `java.sql` и `javax.sql` содержат основные интерфейсы и классы для организации работы с базами данных, образующие так называемый интерфейс JDBC (Java DataBase Connectivity).	Аналогичные библиотеки в .NET находятся в пространстве имен `System.Data` в сборке System.Data. Определенные там интерфейсы являются основой ADO.NET.
Пакет `javax.naming` содержит стандартный интерфейс служб директорий, называемый JNDI (Java Naming and Directory Interface) (см. следующие лекции).	Интерфейс и реализация аналогичной службы директорий ActiveDirectory on Microsoft находятся в пространстве имен `System.DirectoryServices` в сборках System.DirectoryServices и System.DirectoryServices.Protocols.

Литература к Лекции 11

[1] 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.

[2] 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.

[3] 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.

[4] Б. Лисков, Дж. Гатег. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.

[5] Б. Майер. Объектно-ориентированное программирование. Концепции разработки. М.: Русская редакция, 2004.

[6] Документация по JNI http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html.

[7] S. Liang. Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Professional, 1999.

[8] Страница технологии JavaBeans http://java.sun.com/products/javabeans/index.jsp.

[9] JavaBeans Specification 1.01. Доступна через страницу http://java.sun.com/products/javabeans/docs/spec.html.

[10] Документация по библиотекам J2SE http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[11] Страница разработчиков .NET http://www.msdn.microsoft.com/netframework/.

[12] Страница библиотеки Jakarta Commons http://jakarta.apache.org/commons/index.html.

[13] Страница Apache Jakarta Project http://jakarta.apache.org/.

[14] Страница библиотеки Joda http://www.joda.org/.

[15] Страница библиотеки SWT http://www.eclipse.org/swt/.

[16] Страница проекта Eclipse http://www.eclipse.org/.

Задания к Лекции 11