Основы языка Java и обзор библиотек

Содержание

Общие сведения о технологиях Java

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 (сейчас — Oracle), и эта компания во многом определяет развитие технологий Java, но в нем активно участвуют и другие компании — IBM, Intel,

Hewlett-Packard, SAP, Bea и пр. Кроме того, отдельные разработчики и исследователи также оказывают большое влияние на развитие технологий Java за счет публикации своих открытых проектов, содержащих инфраструктурные и технологические новшества.

Этот набор содержит языки Java и JVM, а также множество так называемых «технологий», представляющих собой техники для решения определенных задач и поддерживающие их инструменты и библиотеки. Каждая платформа из следующего списка включает поддержку некоторого подмножества этих техник и соответствующий набор инструментов.

- Платформа *Java Platform Standard Edition (J2SE)*.
  Предназначена для разработки обычных, в основном, однопользовательских приложений.
- Платформа *Java Platform Enterprise Edition (J2EE)*.
  Предназначена для разработки распределенных Web-приложений уровня предприятия.
- Платформа *Java Platform Micro Edition (J2ME)*.
  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.

В настоящем курсе рассматриваются элементы платформ J2EE и J2SE, необходимых для разработки Web-приложений.
Среди Java-технологий можно отметить следующие.

- Java RMI (Java Remote Method Invocation), удаленный вызов методов на языке Java. Включает библиотеки java.rmi, инструмент построения каркасов и заглушек rmic, инструмент регистрации rmi-служб registry.
- JNDI (Java Naming and Directory Interface), Java-интерфейс для служб именования и каталогов. Это библиотеки javax.jndi, включающие несколько реализаций предлагаемых интерфейсов, в том числе простую реализацию службы каталогов LDAP (Lightweight Directory Access Protocol, простой протокол доступа к службе каталогов).
- JNI (Java Native Interface), интерфейс для работы из Java с библиотеками на языке C. Включает правила трансляции интерфейсов и типов данных между Java и C, а также набор библиотек на C, содержащих декларации типов и базовые функции для работы с объектами Java.
- JAXP (Java API for XML Processing), Java-интерфейс для работы с XML. Состоит из множества библиотек для обработки XML-документов — javax.xml, org.w3c, org.xml.
- JAXB (Java Architecture for XML Binding), механизм привязки Java-объектов к XML. Представляет собой набор библиотек для трансляции Java-объектов в XML-

документы и обратно (javax.xml.bind), также включает инструмент для такой трансляции XJC и генератор XML-схем schemagen.

- JMX (Java Management Extensions), программный интерфейс для мониторинга использования различных ресурсов, например, для использования в профилировщиках или контроля доступа (javax.management).
- JDBC (Java Database Connectivity), Java-интерфейс для связи с базами данных. Включает библиотеки java.sql, javax.sql и набор правил их использования.
- JavaBeans — набор библиотек (java.beans) и правил организации кода (образцов, patterns), в совокупности определяющих простую компонентную модель для создания и использования программных компонентов на Java, а также для построения инструментов сборки приложений из таких компонентов.
- EJB (Enterprise Java Beans) — исходно это была специфическая компонентная модель (сильно отличающаяся от JavaBeans) в рамках платформы J2EE, а именно — набор правил организации кода компонентов, предназначенных для представления данных и бизнес-служб в приложениях J2EE. В настоящее время постепенно выходит из употребления, заменяется так называемой моделью POJO (Plain Old Java Object, старые простые Java-объекты), которая предписывает реализовать объекты-данные и объекты-службы как обычные Java-объекты, «навешивая» на них связь с базой данных или транзакционные атрибуты с помощью аннотаций или аспектов.
- JSP (Java Server Pages), серверные страницы Java. Набор библиотек и инструментов, поддерживающих специализированный язык для удобного описания динамически генерируемых Web-страниц в рамках платформы J2EE. Язык JSP представляет собой сплав Java, HTML и специализированных конструкций для связи между кусками кода на этих двух языках.

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 и выполняющийся в Web-браузере.

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.

Язык Java за счет его интерпретируемости обладает рядом характеристик, которые делают его очень удобным для реализации компонентных технологий, стремящихся

максимально ослабить зависимости между компонентами, реализующими независимые функции, но в то же время упростить сборку приложений из таких независимых компонентов.

К таким характеристикам относятся следующие.

- Поддержка интроспекции (introspection или reflection) позволяет из программы обращаться к языковым свойствам объектов — узнавать их типы, реализуемые методы, поля, запускать методы по их сигнатуре, получать значения полей по их именам и пр.
- Поддержка декларативных описателей (annotations, аннотации) позволяет связать элементы кода (классы, методы, поля, параметры методов) с декларативными описателями, определяющими дополнительные характеристики этих элементов. Например, можно с помощью такого описателя указать, что данный метод должен всегда выполняться в рамках транзакции, или что он является тестовым методом, который должен быть выполнен в рамках тестирования, а значения его параметров при тестировании нужно брать из определенных коллекций.
- Поддержка динамического инструментирования кода позволяет в динамике реализовать некоторый интерфейс с помощью объекта, изначально для этого не предназначенного, вызвать дополнительные операции, например, измеряющие время работы или трассирующие используемые аргументы, перед или после вызова определенного метода и пр. С помощью динамического инструментирования кода и аннотаций могут быть реализованы основные элементы аспектно-ориентированного программирования.

Основы языка программирования Java

Лексика

Программы на Java могут быть написаны с использованием набора символов Unicode, каждый символ в котором представляется при помощи 16-ти бит. Поскольку последние версии стандарта Unicode 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.

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, но используются для построения деклараций, инструкций и выражений языка или для обозначения специальных констант.

В Java 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.

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.

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`.

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`.

Операторы и разделители Java:

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

Значение операторов описано ниже.

|	Побитовая дизъюнкция для числовых аргументов, длинная дизъюнкция для логических	||	Короткая дизъюнкция для логических аргументов
&	Побитовая конъюнкция для числовых аргументов, длинная конъюнкция для логических	&&	Короткая конъюнкция для логических аргументов
|=	(Побитовая) дизъюнкция с	&=	(Побитовая) конъюнкция с

	присваиванием		присваиванием
~	Побитовое отрицание числового аргумента	!	Отрицание логического аргумента
^	Побитовое исключающее «или» для числовых аргументов	^=	Исключающее «или» с присваиванием
<	Меньше	<=	Меньше или равно
>	Больше	>=	Больше или равно
+	Сложение чисел, унарный плюс, конкатенация строк	+=	Сложение с присваиванием
-	Вычитание, унарный минус	-=	Вычитание с присваиванием
++	Увеличение на один	--	Уменьшение на один
*	Умножение	*=	Умножение с присваиванием
/	Деление, для целых чисел — деление нацело	/=	Деление с присваивание м
%	Вычисление остатка	%=	Вычисление остатка с присваиванием
==	Сравнение на равенство	!=	Сравнение на неравенство
<<	Побитовый сдвиг влево	<<=	Сдвиг влево с присваиванием
>>	Побитовый сдвиг вправо с повторением знакового бита	>>=	Сдвиг вправо с присваиванием
>>>	Побитовый сдвиг вправо	>>>=	Сдвиг вправо с присваиванием
=	Присваивание	?:	Тернарный условный оператор

Общая структура программы

Программа на Java представляет собой набор пользовательских типов данных — в основном, классов и интерфейсов, с их методами. При запуске программы выполняется метод `main()` некоторого класса. 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)*. Имена пакетов 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.

Программный код компилируется в бинарный код, исполняемый виртуальной машиной. 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.

Пользовательский тип описывается полностью в одном файле.

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;
```

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).

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).

Результаты компиляции располагаются в 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 соответствующие 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.

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).

Компилятор Java и Java-машина располагаются в поддиректории `bin` той директории, в которую устанавливается набор для разработки Java Development Kit.

Пример простой программы на Java.

```
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));
  }
}
```

Базовые типы и операции над ними

В Java имеются ссылочные типы и типы значений. Объект ссылочного типа имеет собственную идентичность, на такой объект можно иметь ссылку из другого объекта, он передается по ссылке, если является аргументом или результатом метода. 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. Значения копируются при передаче аргументов в метод или результата метода.

В Java типами значений являются только примитивные типы, представляющие простые данные: логические, числовые и символьные. 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 объектами соответствующего ему класса-обертки определены преобразования по умолчанию — *упаковка* и *распаковка* (*autoboxing* и *auto-unboxing*), 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.

Логический тип

В Java логический тип назван **`boolean`**, а его обертка — `java.lang.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 Java имеются следующие целочисленные типы.

- Тип байтовых целых чисел, называемый **`byte`**. 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)

Классы-обертки целочисленных типов называются так: `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`.

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

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.
- 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. 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.

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).

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, хотя явные преобразования возможны.

Типы значений 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). 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 «отрицательный ноль» -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 называются POSITIVE_INFINITY, NEGATIVE_INFINITY и NaN.
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 Java 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;
- f(x) — оператор вызова метода с заданным набором аргументов;
- a[x] — оператор вычисления элемента массива;
- **new** — оператор создания нового объекта, используется вместе с обращением к одному из конструкторов типа — **new** MyType("Yes", 2), 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 1. 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 операции, связанные с типами — 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. Она имеет такой же приоритет, как операторы <, >, <=, >=.

Инструкции

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

- Допускается пустая инструкция `;`.
- Декларации локальных переменных устроены так: 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;
  - 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.
Значения, которые используются в метках **case**, должны быть константными выражениями.
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.

Пример.

```
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");
  }
}
```

- Циклы **while** и **do** устроены так.
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** также 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** имеется специальная конструкция для цикла, перебирающего элементы коллекции.
  for (**final**$_{opt}$ *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(); )
{
  final_opt 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++)
{
  final_opt 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)
      System.out.println((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). 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 относятся к особым типам — классам исключений, которыми в Java являются все наследники класса `java.lang.Throwable`. Только объекты таких классов могут быть выброшены в качестве исключений.
  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()`.
  - Иногда возникают цепочки «наведенных» исключений, если обработка одного вызывает созданиедругого. Каждый объект-исключение содержит ссылку на 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()`.
  - 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()`).

- 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.

Пример.

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

Инструкции, предназначенные для синхронизации работы нескольких потоков, рассматриваются далее, в разделе, посвященном разработке многопоточных программ.

Пользовательские типы

В Java 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 !=.

Можно создавать пользовательские ссылочные типы, определяя классы и интерфейсы. Кроме того, можно использовать массивы значений некоторого типа. 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. 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.

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, это может быть класс, реализующий данный интерфейс.

Из последней фразы может быть понятно, что в Java 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 Java могут быть ***шаблонными (generic)***, т.е. иметь типовые параметры. При создании объекта такого класса нужно указывать конкретные значения его типовых параметров.

Примеры деклараций классов и интерфейсов 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;
  }
  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));
  }
}
```

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}};
```

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

Есть возможность декларировать ***перечислимые типы (enums)***, объекты которых представляются именованными константами. 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;
}
```
Возможны декларации методов перечислимого типа и их отдельная реализация для каждой из констант.
```
public enum Operation
{
  ADD      { public int eval(int a, int b) { 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 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);
  }
}
```

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 Java 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`**.

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.

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, используются разные подходы в зависимости от того, статическое это поле или нет, т.е. относится ли оно к самому классу или к его объекту. 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` (естественно, если оно не перекрыто в каком-то классе, промежуточном между данными предком и потомком). Ключевое слово **`super`** в Java 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`** —ссылку 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)***. 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).* 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 вызов должен быть возможен 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.

В Java 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.

Приводимый ниже примеры иллюстрирует 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 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. Описание конструктора похоже на описание метода, только тип результата не указывается, а вместо имени метода используется имя самого класса.

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 Java могут использоваться модификаторы **`public`**, **`protected`** и **`private`**, указывающие, соответственно, 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 как пакетная: такой элемент типа доступен во всех типах того же пакета.

Нестатические методы могут быть объявлены *абстрактными (abstract)*, т.е. не задающими реализации соответствующей операции. Такие методы помечаются модификатором **`abstract`**. Вместо кода у абстрактного метода сразу после описания полной сигнатуры идет точка с запятой.

Методы, которые не должны быть перегружены в наследниках содержащего их класса, помечаются в Java как **`final`**.

Можно использовать операции, реализованные на других языках. В Java для этого предусмотрен механизм Java Native Interface, JNI. 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).

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.

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 могут быть декларированы в интерфейсах.

```
public class A2
{
  public static final 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.

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

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 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 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();
    System.out.println(l.getSize());
    System.out.println(l.getCapacity());
    l.setCapacity(50);
    System.out.println(l.getSize());
    System.out.println(l.getCapacity());

    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));
  }
}
```

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, этого пока сделано не было.

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 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); }


  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);
  }
}
```

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.

Аналогом деструктора в Java является метод **`protected void`** `finalize()`, который можно перегрузить для данного класса. Этот метод вызывается на некотором шаге уничтожения объекта после того, как тот был помечен сборщиком мусора как неиспользуемый.

```
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();
    }
  }
}
```

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.

```
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 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 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);
  }
}
```

О правилах, определяющих выполнения инициализаторов полей и конструкторов классов-предков и наследников при построении объектов в Java, 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("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();
  }
}
```

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 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 10, в разделе о типах с плавающей точкой.

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

Шаблонные типы и операции

В Java есть шаблонные, т.е. имеющие типовые параметры, типы и операции.

Ниже приводятся примеры декларации шаблонного метода и его использования. В последнем вызове 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 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 );
  }
}
```

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 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 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. 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 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 типа `T` или более общего типа, либо вызывающий метод тоже должен указать тип `T` или более общий 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)
{
  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.

Описание метаданных

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)***. 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 Java 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 {}
@interface SimpleParameterAnnotation{}
@interface ComplexClassAnnotation
{
  int id();
  String author() default "Victor Kuliamin";
  String date();
}

@interface AdditionalMethodAnnotation
{
  String value() default "";
}

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

  @SimpleMethodAnnotation
  @AdditionalMethodAnnotation( value = "123" )
  public void n (@SimpleParameterAnnotation 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.

Аннотационный тип декларируется с 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.

Средства создания многопоточных программ

В Java 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) 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). 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()`.

```
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();
  }
}
```

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, которые реализуют 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.

```
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();
  }
}
```

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, который представляет 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 Java

Разработка программ на Java 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 охватывает очень много участников в силу открытости платформы и ее стандартизованности. Поэтому далеко не все часто используемые библиотеки классов распространяются в составе платформ J2SE и J2EE.

В данном разделе дается краткий обзор основных библиотек.

java.lang.Object

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

java.lang.System

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`.

Работа со строками

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 `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.
Класс `java.util.Scanner` реализует простой лексический анализатор текста.

Более гибкую работу с регулярными выражениями можно реализовать с помощью классов пакета `java.util.regex`.

Математические функции

В пакете `java.lang` находятся классы-обертки примитивных типов `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`.

Все числовые классы наследуют классу `java.lang.Number`.

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.

Для форматированного представления чисел используются классы `java.text.NumberFormat` и `java.text.DecimalFormat`.
Набор математических функций и констант реализован в виде элементов класса `java.lang.Math`.

Для генерации псевдослучайных чисел можно использовать как метод `Math.random()`, так и обладающий большей функциональностью класс `java.util.Random`.

Для более сложных вычислений можно использовать классы пакета `java.math` — `BigInteger` и `BigDecimal`, представляющие целые числа произвольной величины и десятичные дроби произвольной точности.

Потоки

`java.lang.Thread` — класс, объекты которого представляют потоки в Java машине.

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 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()`.

Базовые интерфейсы

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 быстрым поиском.

Поддержка интроспекции

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

Слабые ссылки

В Java имеется механизм, позволяющий 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.

Коллекции

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

Для манипуляций с массивами — поиска, сортировки, сравнения и пр. — используются методы класса `java.util.Arrays`.
`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 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, части обширного проекта Apache Jakarta Project.
Для определения линейного порядка на объектах типа `T` используется интерфейс `java.util.Comparator<T>`.

Наиболее часто применяется 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.

Работа с датами и временем

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

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.
Классы `java.util.Timer` и `java.util.TimerTask` служат выполнения определенных действий в заданное время или через определенное время.

Локализация

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 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, строки сообщений на разных языках, может храниться с помощью подклассов `ResourceBundle` — `ListResiourceBundle`, `PropertyResourceBundle`.

Работа с различными кодировками текста организуется при помощи классов пакета `java.nio.charset`.

Библиотеки для организации эффективного параллелизма

Пакет `java.util.concurrent` и его подпакет `locks` содержат набор классов, реализующих коллекции с эффективной синхронизацией работы нескольких потоков (например, разные потоки могут параллельно изменять значения по разным ключам отображения) и 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.

Ввод-вывод

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

JavaBeans

Интерфейс и классы пакета `java.util` `EventListener`, `EventListenerProxy`, `EventObject` используются для реализации образца «подписчик» в рамках спецификаций JavaBeans (см. предыдущую лекцию).
Классы и интерфейсы, лежащие в основе компонентной модели JavaBeans, находятся в пакете `java.beans`.

Библиотеки элементов GUI

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

Одна из этих библиотек (самая 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].

Шифрование

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.

Организация удаленного взаимодействия

Удаленное взаимодействие с помощью RMI

При описании взаимодействия между элементами программных систем инициатор взаимодействия, т.е. компонент, посылающий запрос на обработку, обычно называется ***клиентом***, а отвечающий компонент, тот, что обрабатывает запрос — ***сервером***. «Клиент» и «сервер» в этом контексте обозначают роли в рамках данного взаимодействия. В большинстве случаев один и тот же компонент может выступать в разных ролях — то клиента, то сервера — в различных взаимодействиях. Лишь в небольшом классе систем роли клиента и сервера закрепляются за компонентами на все время их существования.

Синхронным (synchronous) называется такое взаимодействие между компонентами, при котором клиент, отослав запрос, блокируется и может продолжать работу только после получения ответа от сервера. По этой причине такой вид взаимодействия называют иногда ***блокирующим (blocking)***.

В рамках ***асинхронного (asynchronous)*** или ***неблокирующего (non blocking)*** взаимодействия клиент после отправки запроса серверу может продолжать работу, даже если ответ на запрос еще не пришел.

Обычное обращение к функции или методу объекта с помощью передачи управления по стеку вызовов является примером синхронного взаимодействия.

Примером асинхронного взаимодействия является электронная почта. Другой пример — распространение сообщений о новостях различных видов в соответствии с имеющимся на текущий момент реестром подписчиков, где каждый подписчик определяет темы, которые его интересуют.

Синхронное взаимодействие достаточно просто организовать, и оно гораздо проще для понимания. Человеческое сознание обладает единственным «потоком управления», представленным в виде фокуса внимания, и поэтому человеку проще понимать процессы, которые разворачиваются последовательно, поскольку не нужно постоянно переключать внимание на происходящие одновременно различные события. Код программы клиентского компонента, описывающей синхронное взаимодействие, устроен проще — его часть, отвечающая за обработку ответа сервера, находится непосредственно после части, в которой формируется запрос. В силу своей простоты синхронные взаимодействия в большинстве систем используются гораздо чаще асинхронных.

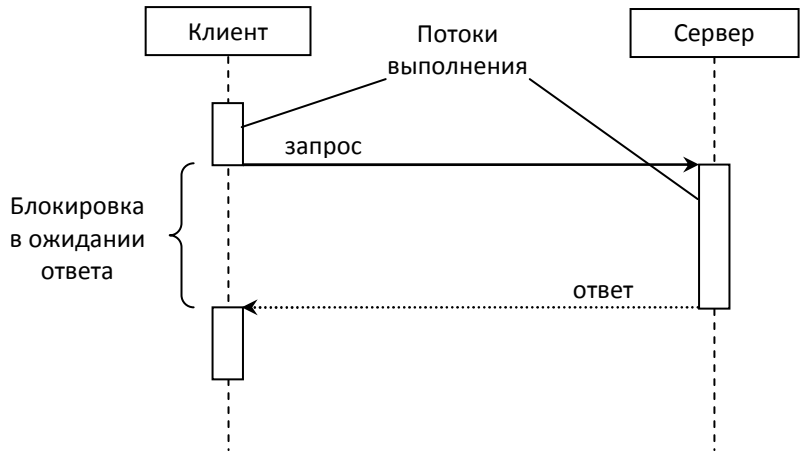

Рисунок 1. Синхронное взаимодействие.

Вместе с тем синхронное взаимодействие ведет к значительным затратам времени на ожидание ответа. Это время часто можно использовать более полезным образом: ожидая ответа на один запрос, клиент мог бы заняться другой работой, выполнить другие запросы, которые не зависят от еще не пришедшего результата. Поскольку все распределенные системы состоят из достаточно большого числа уровней, через которые проходят практически все взаимодействия, суммарное падение производительности, связанное с синхронностью взаимодействий, оказывается очень большим.

Наиболее распространенным и исторически первым достаточно универсальным способом реализации синхронного взаимодействия в распределенных системах является ***удаленный вызов процедур*** (***Remote Procedure Call, RPC***, вообще-то, по смыслу правильнее было бы сказать «дистанционный вызов процедур», но по историческим причинам закрепилась имеющаяся терминология). Его модификация для объектно-ориентированной среды называется ***удаленным вызовом методов (Remote Method Invocation, RMI)***. Удаленный вызов процедур определяет как способ организации взаимодействия между компонентами, так и методику разработки этих компонентов.

На первом шаге разработки определяется интерфейс процедур, которые будут использоваться для удаленного вызова. Это делается при помощи *языка определения интерфейсов (Interface Definition Language, IDL)*, в качестве которого может выступать специализированный язык или обычный язык программирования, с ограничениями, определяющимися возможностью передачи вызовов на удаленную машину.

Определение процедуры для удаленных вызовов компилируется компилятором IDL в описание этой процедуры на языках программирования, на которых будут разрабатываться клиент и сервер (например, заголовочные файлы на C/C++), и два дополнительных компонента — ***клиентскую*** и ***серверную заглушки (client stub*** и ***server stub)***.

Клиентская заглушка представляет собой компонент, размещаемый на той же машине, где находится компонент-клиент. Удаленный вызов процедуры клиентом реализуется как обычный, локальный вызов определенной функции в клиентской заглушке. При обработке этого вызова клиентская заглушка выполняет следующие действия.

1. Определяется физическое местонахождение в системе сервера, для которого предназначен данный вызов. Это шаг называется ***привязкой (binding)*** к серверу. Его результатом является адрес машины, на которую нужно передать вызов.
2. Вызов процедуры и ее аргументы упаковываются в сообщение в некотором формате, понятном серверной заглушке (см. далее). Этот шаг называется ***маршалингом (marshaling)***.
3. Полученное сообщение преобразуется в поток байтов (это *сериализация, serialization*) и отсылается с помощью какого-либо протокола, транспортного или более высокого уровня, на машину, на которой помещен серверный компонент.
4. После получения от сервера ответа, он распаковывается из сетевого сообщения и возвращается клиенту в качестве результата работы процедуры.

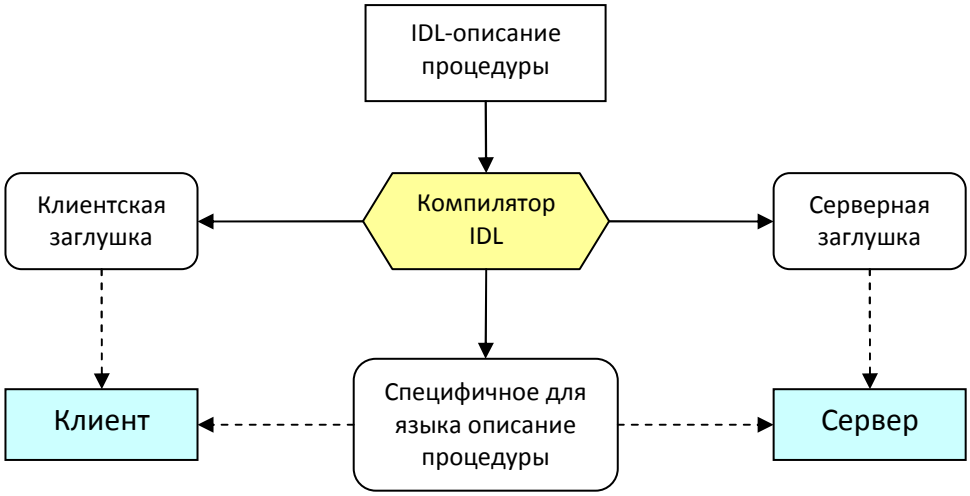

Рисунок 2. Схема разработки компонентов, взаимодействующих с помощью RPC.

В результате для клиента удаленный вызов процедуры выглядит как обращение к обычной функции.

Серверная заглушка располагается на той же машине, где находится компонент-сервер. Она выполняет операции, обратные к действиям клиентской заглушки — принимает сообщение, содержащее аргументы вызова, распаковывает эти аргументы при помощи *десериализации (deserialization)* и *демаршалинга (unmarshaling)*, вызывает локально соответствующую функцию серверного компонента, получает ее результат, упаковывает его и посылает по сети на клиентскую машину.

Таким образом обеспечивается отсутствие видимых серверу различий между удаленным вызовом некоторой его функции и ее же локальным вызовом.

Определив интерфейс процедур, вызываемых удаленно, мы можем перейти к разработке сервера, реализующего эти процедуры, и клиента, использующего их для решения своих задач.

При удаленном вызове процедуры клиентом его обращение оформляется так же, как вызов локальной функции и обрабатывается клиентской заглушкой. Клиентская заглушка определяет адрес машины, на которой находится сервер, упаковывает данные вызова в сообщение и отправляет его на серверную машину. На серверной машине серверная заглушка, получив сообщение, распаковывает его, извлекает аргументы вызова, обращается к серверу с таким вызовом локально, дожидается от него результата, упаковывает результат в сообщение и отправляет его обратно на клиентскую машину. Получив ответное сообщение, клиентская заглушка распаковывает его и передает полученный ответ клиенту.

Эта же техника может быть использована и для реализации взаимодействия компонентов, работающих в рамках различных процессов на одной машине.

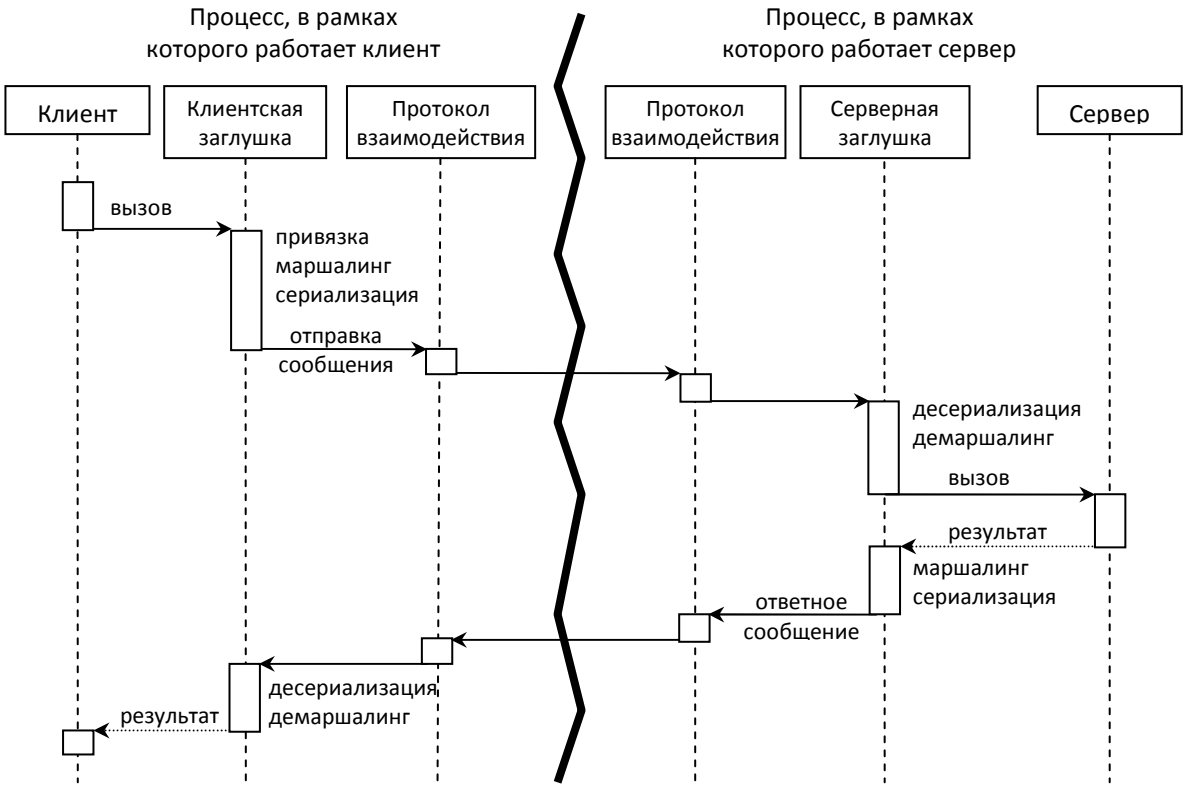

Рисунок 3. Схема реализации удаленного вызова процедуры.

При организации ***удаленного вызова методов*** в объектно-ориентированной среде применяются такие же механизмы. Отличия в его реализации связаны со следующими аспектами.

- Один объект-сервер может предоставлять несколько методов для удаленного обращения к ним.
  Для такого объекта генерируются клиентские заглушки, имеющие в своем интерфейсе все эти методы. Кроме того, информация о том, какой именно метод вызывается, должна упаковываться вместе с аргументами вызова и использоваться серверной заглушкой для обращения именно к этому методу.
  Серверная заглушка в контексте RMI иногда называется ***скелетом (skeleton)*** или ***каркасом***.
- В качестве аргументов удаленного вызова могут выступать объекты.
  Заметим, что передача указателей в аргументах удаленного вызова процедур практически всегда запрещена — указатели привязаны к памяти данного процесса и не могут быть переданы в другой процесс.
  При передаче объектов возможны два подхода, и оба они используются на практике.
  - Идентичность передаваемого объекта может не иметь значения, например, если сервер использует его только как хранилище данных и получит тот же результат при работе с правильно построенной его копией. В этом случае определяются методы для сериализации и десериализации данных объекта, которые позволяют сохранить их в виде потока байтов и восстановить объект с теми же данными на другой стороне.
  - Идентичность передаваемого объекта может быть важна, например, если сервер вызывает в нем методы, работа которых зависит от того, что это за объект. При этом используется особого рода ссылка на этот объект,

позволяющая обратиться к нему из другого процесса или с другой машины, т.е. тоже с помощью удаленного вызова методов.

RMI полностью реализован в рамках Java, при этом приняты следующие решения.

- Интерфейсы и классы, предоставляющие методы для удаленного вызова должны реализовывать интерфейс `java.rmi.Remote`. Этот интерфейс служит маркером — он не содержит каких-либо методов для реализации, а только помечает классы, на основе которых нужно будет создавать заглушки и каркасы.
- Методы для удаленного вызова должны декларировать среди возможных типов исключений `java.rmi.RemoteException` или его предков.
- Для передачи данных по сети используются два вида параметров.
    - Значения примитивных типов передаются в виде копий. Также передаются и объекты-значения, чья идентичность не важна. Классы таких объектов должны реализовывать интерфейс `java.io.Serializable`. При передаче по сети такой объект преобразуется в поток байтов, а на стороне сервера его копия восстанавливается из полученного потока.
    - Объекты, для которых важна идентичность, например, параметры, у которых сервер вызывает методы, возможно, изменяющие их состояние, передаются в виде ссылок на удаленные объекты. В этом случае по сети передается клиентская заглушка такого объекта, при обращении к которой вызывается исходный объект. Соответственно, такие объекты должны реализовывать интерфейс `java.rmi.Remote`.

Интерфейс для удаленного вызова может выглядеть так.

```
package samples.rmi.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
  String sendHello() throws RemoteException;
}
```

При реализации такого интерфейса необходимо объект-реализацию зарегистрировать в какой-нибудь службе именования или каталогов, чтобы его можно было найти при обработке удаленного вызова. В простейшем случае в этом качестве может использоваться служба именования `java.rmi.registry.Registry`.

```
package samples.rmi.hello;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
  public String sendHello() { return "Hello, world!"; }

  public static void main(String args[]) {
    System.setProperty("java.rmi.server.codebase", "<class dir>");
    try {
      Hello stub = (Hello) UnicastRemoteObject.exportObject(new Server(), 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.bind("Hello", stub);
    } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

В данном случае сервер предоставляет для удаленных вызовов объект с помощью `UnicastRemoteObject`. Другой способ — использовать службы активации и класс `java.rmi.activation.Activatable`.

Клиент для взаимодействия с созданным сервером может выглядеть так.

```
package samples.rmi.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
  public static void main(String[] args) {
    String host = (args.length < 1) ? null : args[0];
    try {
      Registry registry = LocateRegistry.getRegistry(host);
      Hello stub = (Hello) registry.lookup("Hello");
      System.out.println("response: " + stub.sendHello());
    } catch (Exception e) {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

Для выполнения приведенного примера нужно на серверной машине запустить регистр RMI с помощью команды `rmiregistry`; затем там же запустить сервер `java samples.rmi.hello.Server`. Затем на клиентской машине нужно запустить клиент, указав имя серверной машины в качестве параметра командной строки.

Построение заглушек в данном примере происходит динамически. В Java до версии 5 для этого требовалось использовать IDL-транслятор `rmic`.

Литература

[1] Документация по платформе Java SE, JDK 6. http://java.sun.com/javase/6/docs/.

[2] Документация по платформе Java EE 6. http://java.sun.com/javaee/sdk/resources.jsp.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification, Third Edition.

[4] I. Rabinovitch, S. Zakhour, S. Hommel, J. Royal, T. Risser. The Java Tutorial, Fourth Edition.

[5] Руководство АО Java EE 5. http://java.sun.com/javaee/5/docs/tutorial/doc/.