КОНСПЕКТ ПО ПРОГЕ (3 СЕМЕСТР)

**ЛЕКЦИЯ 1**

**ИСТОРИЯ**

Первая версия - весна 1995

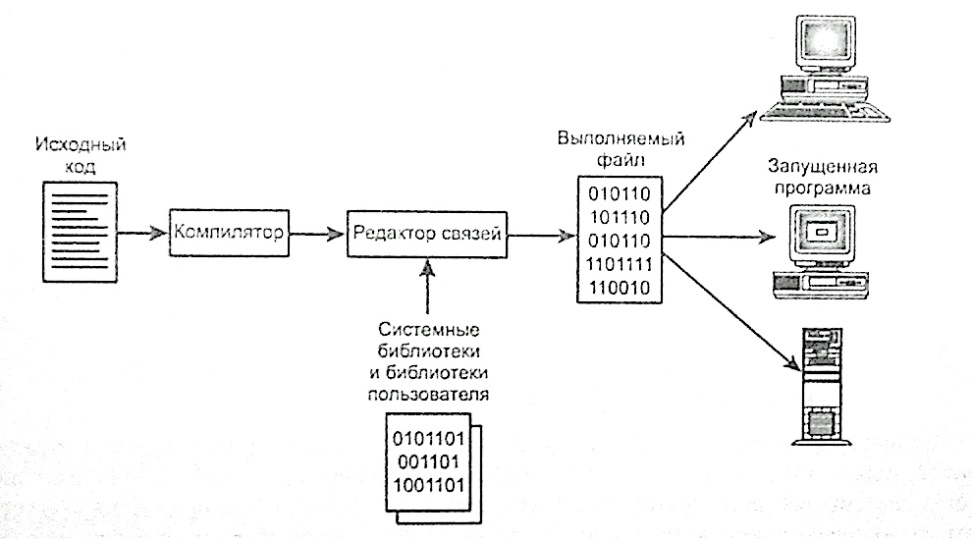
Основная цель создания – разработка независимого от платформы языка для выполнения на различных процессорах в различных средах, в первую очередь для программирования микропроцессоров в бытовых приборах (микроволновые печи, стиральные машины, дистанционные пульты и т.п.)

Вторая по значимости цель – язык программирования сетевых приложений для WWW.

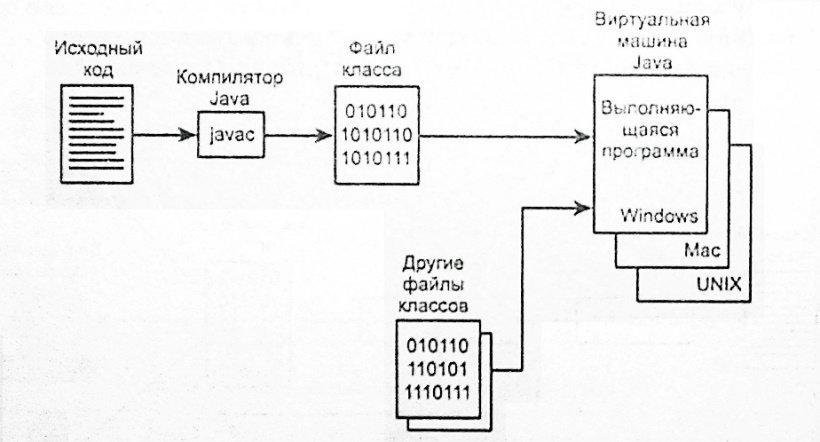
На сегодня – 2 поколения языка Java

Мы будем изучать версию J2SE8 – Java 2 Standard Edition version 8

**ПЕРЕНОСИМОСТЬ**

Переносимость программ Java достигнута решением компилировать Java в промежуточный код, или байт-код для выполнения на абстрактной виртуальной машине Java - JVM

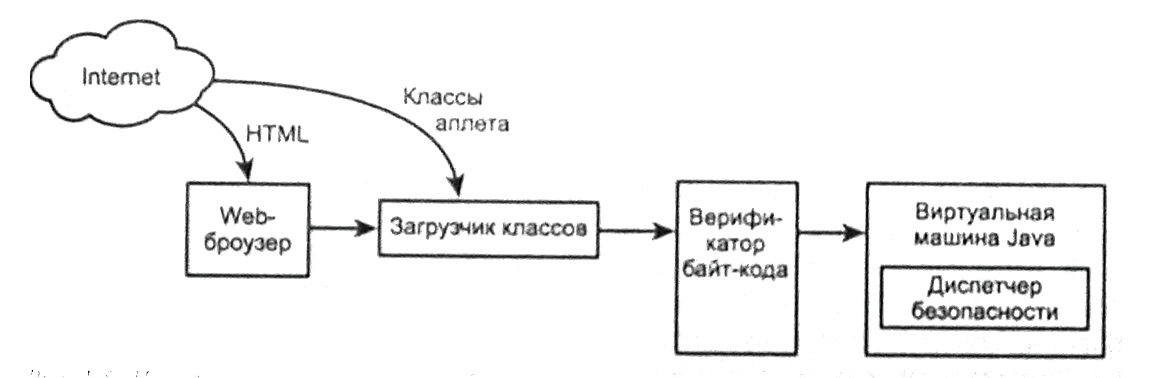
В традиционных языках программирования при вызове системных функций управление передаётся непосредственно ОС:



Результат работы компилятора Java интерпретируется JVM на каждой конкретной платформе:

**СЕТЕВЫЕ ПРИЛОЖЕНИЯ**

* Интернет работает по технологии клиент-сервер
* Клиент – веб-браузер, показывает HTML-страницы
* Сервер – вэб-сервер, источник HTML-страниц
* Апплет - особый вид программы Java предназначенный для передачи по интернету, выполняется совместимым с Java веб-браузером.
* Апплет выполняется как фрагмент HTML-страницы.

JVM ограничивает возможности апплета и сохраняет безопасность компьютера пользователя:

Сервлет - небольшая программа, выполняемая на сервере.

Сервлеты динамически расширяют возможности вэб-сервера, служат для создания динамически генерируемого содержимого для нужд клиента.

Сервлеты переносимы в различных серверных средах.

**СРАВНЕНИЕ С С/С++**

* Синтаксис Java унаследован от С
* Многие объектно-ориентированные свойства были перенесены в Java из С++
* Отличия от С/С++ связаны с необходимостью уменьшения размеров программ и требованиями к безопасности приложений работающих в сети

**ОТЛИЧИЯ ОТ С/С++**

* Нет указателей и оператора ->
* Нет структур и объединений (struct, union)
* Отсутствует перегрузка операторов
* Нет препроцессора
* Не выполняется автоматическое преобразование типов, результатом которого является потеря точности (например, от длинного целого к обычному)
* Нет глобальных переменных и глобальных функций
* Нет множественного наследования
* Отсутствуют деструкторы классов
* Нет операторов typedef, goto, delete
* Отсутствуют беззнаковые целые (нельзя объявить unsinged) (искл. Char, но для кода символа)
* Объекты передаются только по ссылке

**НОВЫЕ ВОЗМОЖНОСТИ**

* Введены понятия пакета, интерфейса и многопоточности.
* Пакеты определяют пространство имен и способ объединения вместе связанных классов
* Интерфейсы в чем-то аналогичны абстрактным классам в С++, используются для создания каркаса программы.
* Многопоточность разрешает двум или более частям одной и той же программы выполняться одновременно
* В Java реализован сборщик мусора, поддерживается ключевое слово new, но отсутствует delete
* В Java отсутствуют стандартные библиотеки. Они заменены собственным набором классов.
* Операторы break, continue поддерживают метки
* Оператор >>> выполняет беззнаковый сдвиг вправо
* Спецификаторы доступа public, private, protected воздействуют только на то, перед чем они стоят
* Тип char 16-битовый и поддерживает Unicode-символы

**ИСТРУМЕНТАЛЬНЫЕ СТРЕДСТВА**

* JRE – это Java Runtime Environment, среда выполнения Java, предназначена только для запуска готовых Java-приложений, а потому содержит лишь реализацию виртуальной машины и набор стандартных классов.
* JDK – это Java Development Kit, средство разработчика Java, включающее в себя набор утилит, стандартные библиотеки с их исходным кодом и набор демонстрационных примеров.
* Основные утилиты JDK включают в себя:
* java – реализация JVM (java virtual machine)
* javac – компилятор Java
* appletviewer – средство для запуска апплетов
* jar – архиватор формата JAR
* javadoc – утилита для автоматической генерации документации (позволяет из специальных комментариев составляет документацию)
* jdb - отладчик
* jvisualvm – профилировщик (для оптимизации скорости)

МЕТОД main И ПРИМЕР ПРОГРАММЫ

* Программа на Java, является набором классов.
* Требуется входная точка, с которой должно начинаться выполнение приложения. Такой входной точкой, по аналогии с языками C/C++, является метод main():
  + - **public static void main(String[] args){**
    - **}**
* Модификатор static позволяет вызвать метод main(), не создавая объектов.
* Метод не возвращает никакого значения, хотя в C есть возможность указать код возврата из программы. В Java для этой цели есть метод System.exit(), который закрывает виртуальную машину и имеет аргумент типа int.
* Аргументом метода main() является массив строк. Он заполняется дополнительными параметрами, которые были указаны при вызове программы.

**package test.first;**

**public class Test {**

**public static void main(String[] args) {**

**for (int i = 0; i < args.length; i++ ){**

**System.out.print( args[i]+" ");**

**}**

**System.out.println();**

**}**

**}**

* Для вызова программы виртуальной машине передается в качестве параметра имя класса, у которого объявлен метод main().
* Поскольку это имя класса, а не имя файла, то не должно быть указано никакого расширения (.class или .java)
* Расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя.
* Компилятору передается имя и путь к файлу.
* В ОС Windows, если вышеприведенный модуль компиляции сохранен в файле Test.java, который лежит в директории test \ first, то вызов компилятора записывается следующим образом:

**javac test\first\Test.java**

* Компилятору передается имя и путь к файлу.
* В ОС Windows, если вышеприведенный модуль компиляции сохранен в файле Test.java, который лежит в директории test \ first, то вызов компилятора записывается следующим образом:

**javac test\first\Test.java**

**КЛАСС Math**

Класс Math состоит из набора статических методов, производящих наиболее популярные математические вычисления и двух констант, имеющих особое значение в математике - это число Пи и экспонента. Часто этот класс еще называют классом-утилитой (Utility class).

Так как все методы класса статические нет необходимости создавать экземпляр этого класса - поэтому он и не имеет открытого конструктора.

Нельзя так же и унаследовать этот класс, поскольку он объявлен с атрибутом final.

Константы определены следующим образом:

**public static final double Math.PI**

**//- задает число Пи**

**public static final double Math.E**

**//- число e.**

тип констант double. При использовании этих констант в вычислениях, результат будет автоматически конвертирован в double, если его явно не привести к другому типу.

все методы класса, краткое описание:

**static double abs(double a)**

**//-Возвращает абсолютное значение типа double**

**static float abs(float a)**

**//-Возвращает абсолютное значение типа float**

**static int abs(int a)**

**//-Возвращает абсолютное значение типа int**

**static long abs(long a)**

**//-Возвращает абсолютное значение типа long**

**static double acos(double a)**

**//-Вернет значение арккосинуса угла в диапазоне от 0 до PI**

**static double asin(double a)**

**//-Вернет значение арксинуса угла от -PI/2 до PI/2**

**static double atan(double a)**

**//-Вернет значение арктангенса угла от -PI/2 до PI/2**

**static double ceil(double a)**

**//-Возвращает наименьшее целое число которое больше a.**

**static double floor(double a)**

**//-Возвращает целое число которое меньше a.**

**static double cos(double a)**

**//-Возвращает косинус угла**

**static double IEEEremainder(double a, double b)**

**//-Возвращает остаток от деления a/b по стандарту IEEE 754**

**static double sin(double a) //-Возвращает косинус угла**

**static double tan(double a) //-Возвращает тангенс угла**

**static double exp(double a) //-Возвращает e в степени a**

**static double log(double a)**

**//-Возвращает натуральный логарифм числа a**

**static double max(double a, double b)**

**//-Возвращает наибольшее из двух чисел типа double**

**static float max(float a, float b)**

**//-Возвращает наибольшее из двух чисел типа double**

**static long max(long a, long b)**

**//-Возвращает наибольшее из двух чисел типа long**

**static int max(int a, int b)**

**//-Возвращает наибольшее из двух чисел типа int**

**static double min(double a, double b)**

**//-Возвращает наименьшее из двух чисел типа double**

**static float min(float a, float b)**

**//-Возвращает наименьшее из двух чисел типа double**

**static long min(long a, long b)**

**//-Возвращает наименьшее из двух чисел типа long**

**static int min(int a, int b)**

**//-Возвращает наименьшее из двух чисел типа int**

**static double pow(double a, double b)**

**//-Возвращает а в степени b**

**static double random()**

**//-Возвращает случайное число от 0.0 до 1.0**

**static double rint(double a)**

**//-Возвращает int число, ближайшее к a**

**static long round(double a)**

**//-Возвращает значение типа long ближайшее к а.**

**static int round(float a)**

**//-Возвращает значение типа int ближайшее к а.**

**static double sqrt(double a)**

**//-Возвращает положительный квадратный корень a**

**static double toDegrees(double angrad)**

**//-Преобразует значение угла из радианов в градусы**

**static double toRadians(double angdeg)**

**//-Преобразует значение угла из градусов в радианы**

Следует обратить внимание, что:

* abs вернет значения типа int, если в качестве параметра будут переданы значения типа byte, short, char
* Угол задается в радианах

**ЛЕКСИКА, СЕМАНТИКА И ОСНОВНЫЕ УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА Java**

**ОСНОВНЫЕ ЭЛЕМЕНТЫ**

Программы на Java состоит из набора пробелов, комментариев, разделителей, ключевых слов, идентификаторов, литеральных констант и операторов.

**КОММЕНТАРИИ**

Java поддерживает два вида комментариев:

1. Традиционный многострочный комментарий (в стиле С): /\* Комментарий \*/
2. Комментарий до конца строки:

**// Комментарий**

Вложенные комментарии не поддерживаются:

/\* this comment /\* // /\*\* ends here: \*/ - пример единственного комментария

**РАЗДЕЛИТЕЛИ**

Используются для группировки и упорядочения программы

В Java к разделителям относятся следующие конструкции: () {} [] ; , .

* {} используются для ограничения тела классов, методов, блоков и других конструкций
* () используются в определении методов и для группировки выражений
* [] используются для доступа к элементам массива
* ; используется для ограничения ввода операторов
* , разделяет индексы многомерного массива и параметры методов класса
* . используются для доступа к переменным и методам класса

**ИДЕНТИФИКАТОРЫ**

* Идентификаторы дают имена классам, методам и переменным
* Идентификатор в Java – это неограниченная последовательность букв и цифр в формате UNICODE, причем первым символом должна быть буква. Java различает прописные и строчные буквы.
* Соглашения:
* Имена классов начинаются с прописной буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы: *ClassName*
* Имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы: *methodName*, *variableName*
* Имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания: *CONSTANT\_NAME*

**КЛЮЧЕВЫЕ СЛОВА**

* 50 слов зарезервированы в языке Java как ключевые, и не могут использоваться как идентификаторы: *abstract, continue, for, new, switch, assert, default, if, package, synchronized, boolean, do, goto, private, this, break, double, implements, protected, throw, byte, else, import, public, throws, case, enum, instanceof, return, transient, catch, extends, int, short, try, char, final, interface, static, void, class, finally, long, strictfp, volatile, const, float, native, super, while*
* Слова *const,* *goto* не используются в Java, несмотря на то, что зарезервированы.
* Слова *true, false, null* не являются ключевыми, это литералы для представления значений булевского типа и пустого объекта. Их тоже нельзя использовать как идентификаторы.

**ЛИТЕРАЛЫ**

Литерал – это запись в исходном коде программы, представляющая собой фиксированное значение.

Литералы в Java бывают следующих видов:

* целочисленный литерал;
* литерал значения с плавающей точкой;
* логический литерал;
* символьный литерал;
* строковый литерал;
* литерал для пустого объекта

**ПРИМЕРЫ**

**ЦЕЛОЧИСЛЕННЫЙ ЛИТЕРАЛ**

* Обычное целое(int) – *число* – 1234
* Длинное целое(long) – *число*l или *число*L – 1234L
* Шестнадцатеричное целое – 0x*число* – 0x12FF
* Восьмеричное целое – 0*число* – 077
* Знаковое – +*число* или -число – -9

**ЛИТЕРАЛ ЗНАЧИНИЯ С ПЛАВАЮЩЕЙ ТОЧКОЙ**

* Вещественное(double) - *число*.*число* или .*число –* 12.34
* Вещественное(float) - *число*f или *число*F - 12.34F (float pi = 3.14 ОШИБКА; flot pi = 3.14F ВЕРНО)
* Вещественное(double) - *число*d или *число*D - 12.34D
* Вещественное в экспоненциальной форме - *число*е*число* или *число*Е*число* - 1.23E4

**СИМВОЛЬНЫЙ ЛИТЕРАЛ**

* Один символ - '*символ*‘ - 'j‘
* Символ с кодом NNNN - \uNNNN - \u0013

**СПЕЦИАЛЬНЫЕ СИМВОЛЫ:**

* Символ «забоя» \b
* Символ табуляции \t
* Перевод строки \n
* Перевод страницы \f
* Возврат каретки \r
* Двойные кавычки \”
* Одинарные кавычки \’
* Обратная косая \\

**СТОРОКОВЫЙ ЛИТЕРАЛ**

* Строка символов – "*символ*“ – "abcdef"
* Пустая строка – "“ – “”

**ПЕРЕМЕННЫЕ**

Переменная – это основной компонент хранения данных в программе Java

Все переменные должны быть объявлены до их использования. Java допускает объявление переменных внутри любого блока {}. Блок задаёт область видимости и продолжительность существования переменной. В случае вложенных блоков, во внутреннем блоке нельзя объявлять переменные с тем же именем, что и во внешнем блоке.

Форма объявления переменной:

***тип идентификатор [= значение] [, идентификатор] [= значение] … ] ;***

Допускается динамическая инициализация

Пример объявлений:

***int x, y = 5, z = 3;***

***double d = Math.sqrt( y \* y + z \* z );***

**ТИПЫ ДАННЫХ, ОПЕРАЦИИ**

Java это язык со строгой типизацией. Это означает, что каждая переменная и каждое выражение имеет тип данных, который должен быть известен во время компиляции программы.

Типы ограничивают значения, которые может иметь переменная или результат исполнения выражения. Типы ограничивают операции, которые исполняются над величинами и определяют значения операции.

Строгая типизация позволяет обнаружить ошибку в момент компиляции программы.

Типы данных в Java делятся на две категории:

1. примитивные или простые типы данных;
2. ссылочные или сложные типы данных.

Ссылочные типы не поддерживают присваивания посредством оператора =. Вместо этого нужно передать желаемое значение как параметр конструктору объекта.

Объекты простых типов создаются сразу после объявления, а объекты сложных типов должны реализовываться с помощью оператора *new*.

**КЛАССЫ-ОБОЛОЧКИ**

* Для примитивных типов существуют соответствующие классы-оболочки. Они предназначены не для вычислений, а для действий, типичных при работе с классами - создания объектов, преобразования объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке
* Числовые классы имеют общего предка - абстрактный класс *Number*, в котором описаны шесть методов, возвращающих числовое значение, содержащееся в классе, приведенное к соответствующему примитивному типу: *byteValue(), doubleValue(), floatValue(), intValue(), longValue(), shortValue()*
* В классах-оболочках есть статические методы преобразования строки символов типа String, представляющей число в соответствующий примитивный тип: Byte.parseByte(), Double.parseDouble(), Float.parseFloat(), Integer.parseInt(), Long.parseLong(), Short.parseShort()**.** Исходная строка типа String задается как аргумент метода.
* В классе *Boolean* для получения логического значения используется метод *booleanValue().*
* Все эти методы принадлежат пакету java.lang.

**ТИПЫ ДАННЫХ, ОПЕРАЦИИ**

К простым типам данных относятся логический тип данных *boolean*, и числовые типы данных.

Числовые типы данных представляют собой:

* -целочисленные типы данных -*byte, short, int, long, char.*
* -типы данных с плавающей точкой - *float, double.*

К ссылочным типам данных относятся классы, интерфейсы и массивы.

**ОПЕРАЦИИ НАД ЦЕЛЫМИ ТИПАМИ**

Арифметические и операции сравнения производятся аналогично соответствующим операциям в С++.

Конструкторы, методы и константы для работы с целочисленными значениями находятся в стандартных Java классах-оболочках: *Integer*, *Long* и Character.

Встроенные в Java целочисленные операторы не сигнализируют о переполнении значений. Исключение составляют операторы деления (/) и нахождения остатка (%), вызывающее исключение если знаменатель равен нулю.

**ОПЕРАЦИИ НАД ВЕЩЕСТВЕННЫМИ ЧИСЛАМИ**

Арифметические и операции сравнения производятся аналогично соответствующим операциям в С++

Целые и вещественные значения можно смешивать в операциях. К обычным вещественным числам добавлены три значения: *POSITIVE\_INFINITY* – положительная бесконечность, *NEGATIVE\_INFINITY* – отрицательная бесконечность, *NaN* - "не число”

Конструкторы, методы и константы для работы со значениями с плавающей точкой находятся в стандартных Java классах-оболочках *Float*, *Double* и *Math*.

**ЛОГИЧЕСКИЙ ТИП ДАННЫХ**

Логический (булевский) тип данных служит для представления логической величины, имеющей одно из двух возможных значений: *true* – истина, *false* – ложь

0(ноль) не является эквивалентом для значения "ложь“

Над логическими данными можно выполнять операции присваивания, сравнение на равенство и неравенство, а также логические операции выполняемые аналогично соответствующим операциям в С++.

**ОПЕРАЦИИ СРАВНЕНИЯ**

*=,+=, - =,\*=,/=, %=, &=, |=, ^=,<<=, >>=, >>>=*

Выполняются аналогично соответствующим операциям в С++.

***int n = 0;***

***…***

***n -= 2; // n = n – 2***

**УСЛОВНАЯ ОПЕРАЦИЯ ?:**

Выполняются аналогично соответствующей операции в С++

***int y = 123;***

***…***

***int x = (y >= 0 ? 1 : -1);***

***boolean b = y < 0;***

**ВЫРАЖЕНИЕ**

Из констант и переменных, операций над ними, вызовов методов и скобок составляются выражения. При вычислении выражения выполняются четыре правила:

1. Операции одного приоритета вычисляются слева направо. Исключение: операции присваивания вычисляются справа налево
2. Левый операнд вычисляется раньше правого
3. Операнды полностью вычисляются перед выполнением операции
4. Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части

Пример.

***int a = 3, b = 5;***

***b += a += b += 7;***

В результате вычислений b получит значение 27.

**ПРИОРИТЕТ ОПЕРАЦИЙ**

Операции перечислены в порядке убывания приоритета:

* Постфиксные операции ++ и --.
* Префиксные операции ++ и --, дополнение ~ и отрицание !
* Приведение типа (тип)
* Умножение \*, деление / и взятие остатка %.
* Сложение + и вычитание - .
* Сдвиги <<, >>, >>>
* Сравнения >, <, >=, <=
* Сравнения ==, !=
* Побитовая конъюнкция &
* Побитовое исключающее ИЛИ ^
* Побитовая дизъюнкция |
* Конъюнкция &&
* Дизъюнкция ||
* Условная операция ?:
* Присваивания =,+=, -=, \*=, /=, %=, \&=, ^=, |=, <<=, >>=, >>>=

**ПРИВЕДЕНИЕ ТИПОВ**

При проведении компиляции Java программ происходит проверка соответствия типов данных и предотвращение неверных операций присваивания.

Все сомнительные программные конструкции приводят к выдаче компилятором сообщений об ошибке в приведении типов.

Java поддерживает неявное приведение значений целого типа данных в типы с плавающей точкой. Обратное преобразование некорректно и компилятор выдаст ошибку.

Явное приведение типов осуществляется при помощи оператора:

***(тип) значение\_другого\_типа***

Результат арифметической операции имеет тип *int*, кроме случая, когда один из операндов типа long. В этом случае результат будет типа *long*. Перед выполнением арифметической операции всегда происходит повышение типов byte, short, char в тип *int*, а может быть, и в тип *long*, если другой операнд типа *long*

Нельзя преобразовать тип данных с плавающей точкой в логический тип данных и наоборот.

При преобразовании типа данных с большей точностью представления значений (например, *double*) к типу данных, поддерживающих меньшую точность представления (например, *float*) возможна потеря данных.

**УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ**

Управляющие конструкции предназначены для управления последовательностью исполнения операторов.

В Java существуют следующие управляющие конструкции:

* условие if-else
* переключатель switch
* цикл с предусловием while
* цикл с послесловием do-while
* цикл для определенного диапазона for
* обработчик исключений try-catch-finally

**ОПЕРАТОРЫ И БЛОКИ**

Символом завершения оператора в Java является знак "точка с запятой".

Для группировки операторов в блок используются фигурные скобки:

***{***

***оператор1;***

***оператор2;***

***}***

Блок является составным оператором и используется так же, как и отдельный оператор.

**УСЛОВИЕ if-else**

Условие if-else служит базовой управляющей конструкцией любого алгоритмического языка и в Java имеет вид:

***if (логическое\_Выражение) [then]***

***оператор1***

***[else***

***оператор2];***

Вначале работы этой управляющей конструкции вычисляется *логическое\_Выражение* и если оно верно, то управление передается *оператору1*, в противном случае управление передается *оператору2*.

**ПЕРЕКЛЮЧАТЕЛЬ switch**

Переключатель *switch* сначала вычисляет *целочисленноеВыражение*, после чего происходит поиск метки *case* с соответствующим вычисленным значением. Если такая метка найдена, то управление передается ей, иначе управление передается метке *default*. Если метка *default* отсутствует, то весь блок операторов управляющей конструкции *switch* игнорируется. Операторы, следующие за найденной меткой, исполняются последовательно, пока не встретится ключевое слово *break*, после чего происходит выход из управляющей конструкции:

***switch (целочисленноеВыражение)***

***{***

***case значение1: оператор1;***

***case значение2: оператор2;***

***...***

***default:***

***оператор0;***

***}***

**ЦИКЛ С ПРЕДУСЛОВИЕМ while**

Цикл с предусловием имеет вид:

***while (логическоеВыражение) оператор;***

Вначале работы цикла проверяется значение логического выражение, и если оно истинно, то выполняется оператор, а затем управление передается в начало цикла, и снова происходит проверка логического выражения.

Цикл исполняется до тех пор, пока логическое выражение имеет значение true.

**ЦИКЛ С ПОСЛЕУСЛОВИЕМ do-while**

Цикл с послеусловием имеет вид:

***do***

***оператор***

***while (логическоеВыражение);***

Цикл с послеусловием аналогичен циклу с предусловием, за исключением того, что сначала выполняется оператор, а потом проверяется истинность логического выражения, и если оно истинно, то управление снова передается в начало цикла. В противном случае происходит выход из цикла.

**ЦИКЛ for**

Цикл for используется для исполнения цикла тогда, когда задан диапазон изменения значений переменных и имеет вид:

***for (инициализацияПеременных; логическоеЗначение; приращение) оператор;***

**где *инициализацияПеременных* имеет вид:**

***[тип] имяПеременной1 = начальноеЗначение1***

***[, имяПеременной2 = начальноеЗначение2 [, ...]]***

Все переменные, разделенные запятой, должны принадлежать одному и тому же типу.

Приращение имеет вид:

***значение1 [, значение2 [, ...]]***

Существует вариант бесконечного цикла for:

***for (;;) оператор;***

из которого можно выйти только при помощи выражения прерывания цикла *break*, выражения выхода из метода *return* или посредством возбуждения исключения.

**ВЫХОД ИЗ ЦИКЛА break**

Для принудительного выхода из управляющих конструкций циклов *while, do - while, for* используется ключевое слово *break*.

**ПЕРЕХОД К НАЧАЛУ ЦИКЛА continue**

Для того, чтобы в процессе исполнения цикла *while*, *do-while* или *for* вернуться к его началу, проигнорировав дальнейшую последовательность операторов, применяется ключевое слово *continue*.

**МЕТКИ**

Определение метки имеет вид:

***имяМетки: цикл***

где *цикл* представляет собой цикл *while, do - while* или for того уровня, из которого следует выйти при помощи выражения *break имяМетки* или к началу которого следует перейти при помощи выражения *continue имяМетки*.

Использование ключевого слова *break* имеет вид:

***break имяМетки;***

Использование ключевого слова *continue* имеет вид:

***continue имяМетки;***

**ВОЗВРАТ ИЗ МЕТОДА return**

Выражение для выхода из метода имеет следующий вид:

***return значение;***

где тип значение совпадает с возвращаемым типом, определенным для соответствующего метода.

Если метод объявлен, как не возвращающий значения, то есть *void*, или если ключевое слово *return* используется в теле конструктора, то выражение для выхода из метода имеет вид:

***return;***

**ЕДИНИЦА КОМПИЛЯЦИИ (compilation unit)**

Единица компиляции Java хранится в текстовом .java-файле и является единичной порцией входных данных для компилятора. Каждый .java-файл состоит из трех частей:

1. объявление пакета;
2. import-выражения;
3. объявления верхнего уровня

Объявление пакета одновременно указывает, какому пакету будут принадлежать все объявляемые ниже типы. Это выражение может отсутствовать, что означает, что эти классы располагаются в безымянном пакете (другое название - пакет по умолчанию).

Пример:

***package java.lang;***

import-выражения позволяют обращаться к типам из других пакетов по их простым именам, "импортировать" их. Эти выражения также необязательны.

Пример:

***import java.io;***

Наконец, объявления верхнего уровня содержат объявления одного или нескольких типов. Название "верхнего уровня" противопоставляет эти классы и интерфейсы, располагающиеся в пакетах, внутренним типам, которые являются элементами и располагаются внутри других типов.

Эта часть также является необязательной, в том смысле, что компилятор не выдаст ошибки в случае ее отсутствия. Однако, никаких .class-файлов сгенерировано тоже не будет.

Пакеты представляют собой структуры, в которых хранятся скомпилированные байт-коды классов. Физически пакет представляет собой каталог на диске, в котором записаны скомпилированные коды классов.

**ЛЕКЦИЯ 2**

**КЛАССЫ И ОБЪЕКТЫ**

Данные и соответствующие им методы объединены в одну структуру, которая в ООП называется объектом.

Объект - это программная модель объектов реального мира или абстрактных понятий, представляющая собой совокупность переменных, задающих состояние объекта, и связанных с ними методов, определяющих поведение объекта.

Для описания объектов в Java используется понятие класс. Все коды в языке Java находятся внутри классов.

Класс – это шаблон по которому создаётся объект, он задаёт прототип, описывающий переменные и методы. Конструирование объекта на основе некоторого класса называется созданием экземпляра этого класса (instance).

**СОЗДАНИЕ КЛАССА**

Объявление класса в языке Java имеет вид:

**[модификаторы] class имя\_класса**

**[extends суперКласс]**

**[implements СписокИнтерфейсов]**

**{**

**переменные и методы класса**

**}**

Объекты создаются с помощью команды new.

В качестве модификаторов класса могут использоваться следующие ключевые слова:

* public – элементы класса доступны вне текущего пакета, которому принадлежит данный класс
* (по умолчанию) - элементы класса доступны из пакета, в который входит данный класс
* abstract - класс не имеет экземпляров класса
* final - класс не имеет подклассов

**ДОСТУП К ЭЛЕМЕНТАМ КЛАССА**

Все поля и методы доступны для методов данного класса. Для того, чтобы ограничить к ним доступ из других классов, используются спецификаторы доступа к полям и методам класса:

* public – открытый - к элементам класса можно обращаться из любого другого класса
* protected – защищенный - к элементам класса можно обращаться только из класса–потомка
* (по умолчанию) – пакетный - к элементам класса можно обращаться только из классов, принадлежащих тому же самому пакету, что и данный класс
* private – закрытый - к элементам класса можно обращаться только из самого данного класса

**ПОЛЯ КЛАССА**

Объявление поля класса имеет вид:

**[модификаторы] типДанных поле [= начальноеЗначение]**

Поля делятся на два типа:

* статические (поля класса) - поля, совместно используемые всеми экземплярами одного класса;
* динамические (поля экземпляра класса) - поля, уникальные для каждого экземпляра класса.

Статические поля помечаются модификатором static. Они используются для синхронизации данных в различных экземплярах одного и того же класса.

Полям класса или экземпляра класса можно присваивать инициализирующие значения, соответствующие их типу данных.

Значение динамических полей инициализируются при создании экземпляра класса.

Если поле объявляется с модификатором final, то значение такого поля в программе не может быть изменено - в противном случае компилятор выдаст сообщение об ошибке. Такие финальные переменные исполняют в Java роль констант.

Обращение к переменной класса (статическому полю) имеет вид:

**класс.статическоеПоле**

Обращение к переменной экземпляра класса (динамическому полю) имеет вид:

**экземплярКласса.поле**

**ОБЪЯВЛЕНИЕ И СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА**

Объявление экземпляра класса (то есть объекта, принадлежащего соответствующему типу класса) имеет вид:

**класс экземплярКласса**

Объявление объекта создает не сам объект, а только ссылку на этот объект. Доступ к объекту осуществляется только по его ссылке. Объекты создаются при помощи оператора new, за которым следует конструктор класса:

**экземплярКласса = new класс(параметры)**

Класс в языке Java связно хранит внутри себя кроме полей и методов также блоки инициализации, конструкторы и другие классы, встроенные в текущий класс.

НАПРИМЕР

**// объявление класса**

**class Point {**

**int x, y;**

**boolean isVisible;**

**// блок инициализатор:**

**{**

**x = 0; y = 0;**

**isVisible = false;**

**}**

**}**

**// объявление объекта**

**Point p;**

**// создание объекта**

**p = new Point();**

**p.x = 100;**

**p.y = 200;**

**p.isVisible = true;**

**ВЛОЖЕННЫЕ, ЛОКАЛЬНЫЕ И АНОНИМНЫЕ КЛАССЫ**

Вложенный класс – это класс, объявленный внутри объявления другого класса.

**public class EnclosingClass {**

**...**

**public class NestedClass {**

**...**

**}**

**}**

Локальный класс – это класс, объявленный внутри блока кода. Область видимости локального класса ограничена тем блоком, внутри которого он объявлен.

**public class EnclosingClass {**

**...**

**public void enclosingMethod(){**

**// Этот класс доступен только внутри enclosingMethod()**

**public class LocalClass {**

**...**

**}**

**}**

**}**

Анонимный класс – это локальный класс без имени.

**// Параметр конструктора – экземпляр анонимного класса,**

**// реализующего интерфейс Runnable**

**new Thread(**

**new Runnable() {**

**public void run() {...}**

**}**

**).start();**

**КОНСТРУКТОРЫ**

Для инициализации объекта при его создании используются конструкторы.

Конструктор имеет список параметров и имя, совпадающее с именем класса. В классе может быть несколько конструкторов, имеющих разные списки параметров.

Конструктор не возвращает значения и вызывается при создании нового экземпляра. Вызов конструктора происходит после инициализации по умолчанию полей объекта и вызова блоков инициализации полей экземпляра класса.

Создание конструктора класса имеет вид:

**[модификаторДоступа] класс (списокПараметров)[throws исключение]**

**{**

**телоКонструктора**

**}**

В качестве спецификатораДоступа можно использовать одно из следующих ключевых слов:

* public – доступ из любого другого класса
* protected - доступ из классов-потомков
* (по умолчанию) – доступ из классов пакета
* private – доступ из данного класса, обычно используется для запрета создания экземпляра текущего класса

Конструкторы имеют место только для экземпляров класса.

Для создания кода инициализации статических переменных, принадлежащих всему классу, используются блоки статической инициализации.

Инициализацию объекта можно производит с помощью конструктора по умолчанию. Этот конструктор не содержит параметров при его вызове для создания экземпляра данного класса.

Если в классе определен хотя бы один конструктор с непустым списком параметров, то для инициализации экземпляра класса без передачи параметров необходимо указать дополнительный конструктор с пустым списком параметров.

Последовательность инициализации полей класса при вызове конструктора с параметрами определяется следующим образом:

* + сначала инициализируются поля класса (по умолчанию или явно);
  + исполняются блоки инициализации;
  + происходит инициализация полей класса при помощи конструктора.

**МЕТОДЫ**

Метод объявляется следующим образом:

**[модификаторы]Возвращаемый\_тип имя\_метода() программный\_блок**

или

**[модификаторы]Возвращаемый\_тип имя\_метода(тип имя\_параметра, тип имя\_параметра, ...) программный\_блок**

Модификаторы определяют способы последующего использования класса.

Перед определением метода можно задать следующие ключевые слова:

* abstract - этот метод абстрактный, т. е. в программе нельзя создавать его экземпляры;
* static - метод класса;
* native - метод реализован на языке, зависящем от платформы;
* final - метод не может переопределяться в классах потомков;
* synchronized - разрешена работа только одного потока команд данного метода;
* public, private или protected - спецификаторы доступа.

Внутри метода или в теле конструктора могут использоваться следующие выражения:

* this - ссылка на текущий объект;
* super - ссылка на базовый объект;
* super имя\_метода() - метод базового класса;
* this(...) - вызов конструктора данного класса;
* super(...) - вызов конструктора базового класса;
* return или return значение - возвращает управление вызвавшей программе и передает ей указанное значение.

Метод может быть перегруженным, т.е. существует несколько его версий с одним и тем же именем, но с различным списком параметров.

Статические методы могут перегружаться нестатическими и наоборот.

Простые типы данных всегда передаются через параметры методов по значению.

Ссылочные типы данных и массивы всегда передаются через параметры методов по ссылке.

**МЕТОДЫ С ПЕРЕМЕННЫМ ЧИСОМ ПАРАМЕТРОВ**

Объявление:

**[модификаторы]Возвращаемый\_тип имя\_метода(тип... имя\_параметра) программный\_блок**

При передаче параметров в метод из них автоматически создается массив.

**ПАКЕТЫ. ИМПОРТИРОВАНИЕ**

Классы группируются в пакеты классов.

Пакеты представляют собой структуры, в которых хранятся скомпилированные байт-коды классов. Физически пакет представляет собой каталог на диске, в котором записаны скомпилированные коды классов.

Все стандартные объекты Java находятся в пакетах.

Для использования классов пакета в программе необходимо выполнить операцию подключения данного пакета. Такая процедура носит название импортирование пакета и имеет следующий синтаксис:

**import имя\_пакета.{имя\_класса | \* };**

В случае \* подключаются все классы пакета.

Статический импорт: Константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт, как это показано в примерах:

**import static java.lang.Math.\*;**

**import static java.lang.Math.E;**

**ООП СРЕДСТВАМИ JAVA.**

**ИНКАПСУЛЯЦИЯ**

Инкапсуляция - сокрытие данных внутри объекта, и обеспечение доступа к ним с помощью общедоступных методов.

**public class Point {**

**private int x, y;**

**public void move(x1,y1)**

**{**

**x = x1;**

**y = y1;**

**}**

**public int getX() {**

**return x;**

**}**

**public int getY() {**

**return y;**

**}**

**}**

**НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ**

Класс (подкласс) может наследовать переменные и методы другого класса (суперкласса), используя ключевое слово extends. Подкласс имеет доступ ко всем открытым переменным и методам (кроме private) родительского класса, как будто они находятся в подклассе.

В то же время подкласс может иметь методы с тем же именем, параметрами и возвращаемым значением, что и методы суперкласса. В этом случае подкласс переопределяет методы родительского класса. Это часть механизма ООП, который называется полиморфизмом.

Способность Java делать выбор метода, исходя из типа времени выполнения, называется динамическим полиморфизмом. Поиск метода происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут Object – суперкласс для всех классов.

Все методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется). Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения.

Cтатические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затрагивает объекты.

**ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ**

Отношением называется связь между классами. В объектно-ориентированном проектировании особое значение имеют четыре типа отношений: зависимости, обобщения, ассоциации и реализации.

**ЗАВИСИМОСТЬ**

Зависимостью (Dependency) называется отношение использования, определяющее, что изменение состояния объекта одного класса может повлиять на объект другого класса, который его использует, причем обратное в общем случае неверно.

Зависимости применяются тогда, когда экземпляр одного класса использует экземпляр другого, например в качестве параметра метода.

**ОБОБЩЕНИЕ**

Обобщение (Generalization) означает, что объекты подкласса могут использоваться всюду, где встречаются объекты суперкласса, но ни в коем случае не наоборот.

Определение суперкласса является более общим, чем определение подкласса.

Подкласс наследует свойства родителя (атрибуты и методы).

**РЕАЛИЗАЦИЯ**

Реализацией (Realization) называется отношение между классификаторами (классами, интерфейсами), при котором один из них описывает контракт (интерфейс сущности), а другой гарантирует его выполнение.

**АССОЦИАЦИЯ**

Ассоциации (Association) показывают, что объекты одного класса связаны с объектами другого класса и отражают некоторое отношение между ними. В этом случае можно перемещаться (с помощью вызова методов) от объектов одного класса к объектам другого.

**АГРЕГАЦИЯ**

Агрегация – ассоциация, моделирующая взаимосвязь “часть/целое” между классами, которые в тоже время могут быть равноправными. Оба класса при этом находятся на одном концептуальном уровне, и ни один не является более важным, чем другой.

**МЕТАКЛАСС**

Класс Class - Экземпляры этого класса описывают классы, интерфейсы, массивы и примитивные типы данных работающего приложения. У этого класса нет конструкторов, объекты создаются либо динамически виртуальной машиной Java, либо с помощью метода getClass() любого объекта.

Методы:

* forName(String className) -возвращает объект Class для заданного имени
* getName() -возвращает имя класса
* newInstance() -создает новый экземпляр класса
* getSuperclass() -возвращает суперкласс
* isInterface() -определяет, является ли объект
* интерфейсом
* getInterfaces() -возвращает интерфейсы класса
* isArray() -определяет, является ли объект массивом
* isPrimitive() -определяет, является ли тип примитивным

**КЛАССЫ-ОБОЛОЧКИ**

Используются для объектного представления примитивных типов данных.

Реализуют методы преобразования из примитивных типов и обратно, а также в cтроковое представление и обратно.

К классам-оболочкам относятся: Boolean, Character, Byte, Short, Integer, Long, Float, Double, Void.

Автоупаковка и автораспаковка:

Допускается присваивать переменным примитивного типа значения соответствующих классов-оболочек и наоборот — в этом случае вызов метода преобразования будет автоматически добавлен компилятором:

**Integer answer = 42; // Integer answer = new Integer(42);**

**int i = answer; // int i = answer.intValue();**

**КЛАСС OBJECT**

Класс Object лежит в основе всей иерархии классов Java.

Методы класса:

**public final native Class getClass()**

// -возвращает класс объекта

**public final native void notify()**

// -пробуждает поток, ожидающий монитор объекта

**public final native void notifyAll()**

// -пробуждает все потоки, ожидающие монитор объекта

**public final native void wait()**

// -ждет оповещения другим потоком

**public native int hashCode()**

// -возвращает хэш-код объекта

**public boolean equals(Object obj)**

// -сравнивает объекты на равенство

**public native Object clone()**

// -возвращает копию объекта

**public String toString()**

// -преобразует объект в строку символов

**protected void finalize()**

// -вызывается сборщиком мусора при разрушении объекта

**АБСТРАКТНЫЕ КЛАССЫ**

Абстрактные классы объявляются с ключевым словом abstract и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах.

Объекты абстрактных классов создать нельзя, но можно создать объекты подклассов, которые реализуют эти методы.

Абстрактные классы могут содержать и полностью реализованные методы.

НАПРИМЕР

**abstract class Square {**

**abstract int squareIt(int i); //абстрактный метод**

**public void show() {**

**System.out.println("обычный метод");**

**}**

**}**

**//squareIt() должен быть реализован подклассом Square**

**class SquareReal extends Square {**

**public int squareIt(int i) {**

**return i\*i;**

**}**

**}**

**public class AbstractDemo {**

**public static void main(String[] args) {**

**// Square ob1 = new Square(); нельзя создать объект!**

**Square ob2 = new SquareReal();**

**System.out.println("10 в квадрате равно “ + ob2.squareIt(10));**

**ob2.show();**

**}**

**}**

**ИНТЕРФЕЙСЫ**

Интерфейсы представляют полностью абстрактные классы: ни один из объявленных методов не может быть реализован внутри интерфейса.

Все объявленные методы автоматически трактуются как public и abstract, а все атрибуты – как public, static и final.

Класс может реализовывать любое число интерфейсов, указываемых после ключевого слова implements, дополняющего определение класса.

Определение интерфейса имеет вид:

**[public] interface имя [extends I1,I2,…,IN]**

**{/\*реализация интерфейса\*/}**

Пример:

**interface Displayable {**

**void hide();**

**void show();**

**}**

**СБОРКА МУСОРА**

Java автоматически выполняет освобождение памяти, занимаемой объектом, с помощью механизма “сборки мусора”.

Когда никаких ссылок на объект не существует, то есть все ссылки на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена.

“Сборка мусора” происходит нерегулярно во время выполнения программы. Форсировать “сборку мусора” невозможно, можно лишь “рекомендовать” ее выполнить вызовом метода gc(), но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным.

Вызов метода System.runFinalization() приведет к запуску метода finalize() для объектов утративших все ссылки.

**МЕТОД FINALIZE()**

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций используется механизм finalization.

Чтобы использовать finalization, необходимо определить метод finalize().

Непосредственно перед освобождением памяти для объекта вызывается метод finalize()

Метод finalize() имеет следующую сигнатуру:

**protected void finalize(){**

**// код завершения**

**}**

Ключевое слово protected запрещает доступ к finalize() кодам, определенным вне этого класса.

Невозможно определить, когда finalize() будет выполнен. В принципе, этот метод может быть вообще не выполнен.

**ДЕМОНСТРАЦИЯ СБОРКИ МУСОРА**

**class Demo {**

**private int a;**

**public Demo(int a) {**

**this.a = a;**

**}**

**protected void finalize() {**

**System.out.println("объект удален, a=" + a);**

**}**

**}**

**public class FinalizeDemo {**

**public static void main(String[] args) {**

**Demo d1 = new Demo(1);**

**d1 = null;**

**Demo d2 = new Demo(2);**

**Object d3 = d2; //1**

**//Object d3 = new Demo(3); //2**

**d2 = d1;**

**System.gc();//просьба выполнить “сборку мусора”**

**}**

**}**

В результате выполнения этого кода перед вызовом метода gc() без ссылки останется только один объект.

**объект удален, a=1**

Если закомментировать строку 1 и снять комментарий со строки 2, то перед выполнением gc() ссылку потеряют уже два объекта.

**объект удален, a=1**

**объект удален, a=2**

**ПЕРЕЧИСЛИМЫЕ ТИПЫ**

Перечисляемый тип (enum) — это тип, значения которого ограничены набором констант.

**public enum Season {**

**WINTER,**

**SPRING,**

**SUMMER,**

**AUTUMN**

**}**

**public class TestSeason {**

**public static void main(String[] args) {**

**Season s = Season.SUMMER;**

**System.out.println("Current season is " + s);**

**}**

**}**

**ПОЛЯ И МЕТОДЫ В ПЕРЕЧИСЛЕНИЯХ. ПРИМЕР**

**enum Coin {**

**PENNY(1), NICKEL(5), DIME(10), QUARTER(25);**

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

**private final int value;**

**public int value() { return value; }**

**}**

**class Test {**

**public static void main(String[] args) {**

**for (Coin c : Coin.values())**

**System.out.println(c + "\t\t" +**

**c.value() + "\t" + color(c));**

**}**

**private enum CoinColor { COPPER, NICKEL, SILVER }**

**private static CoinColor color(Coin c) {**

**switch(c) {**

**case PENNY:**

**return CoinColor.COPPER;**

**case NICKEL:**

**return CoinColor.NICKEL;**

**case DIME: case QUARTER:**

**return CoinColor.SILVER;**

**default:**

**throw new AssertionError("Unknown coin: " + c);**

**}**

**}**

**}**

Результат:

**PENNY 1 COPPER**

**NICKEL 5 NICKEL**

**DIME 10 SILVER**

**QUARTER 25 SILVER**

**INSTANCEOF**

Оператор instanceof возвращает значение true, если объект является экземпляром данного класса, например:

**Font obj = new Font("Courier", 1, 18);**

**if (obj instanceof java.awt.Font) {**

**/\*операторы\*/**

**}**

**ВНУТРЕННИЕ (INNER) КЛАССЫ**

Нестатические вложенные классы принято называть внутренними (inner) классами.

Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Объект внутреннего класса всегда скрыто хранит ссылку создавшего его объекта внешнего класса.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса.

Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (final static).

Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.

Внутренние классы позволяют окончательно решить проблему множественного наследования, когда требуется наследовать свойства нескольких классов.

**Пример (После компиляции объектный модуль, соответствующий внутреннему классу,получит имя Ship$Engine.class):**

**public class Ship {**

**// поля и конструкторы**

**// abstract, final, private, protected - допустимы**

**public class Engine { // определение внутреннего класса**

**// поля и методы**

**public void launch() {**

**System.out.println("Запуск двигателя");**

**}**

**}// конец объявления внутреннего класса**

**public void init() {// метод внешнего класса**

**// объявление объекта внутреннего класса**

**Engine eng = new Engine();**

**eng.launch();**

**}**

**}**

При таком объявлении объекта внутреннего класса Engine в методе внешнего класса Ship нет реального отличия от использования какого-либо другого внешнего класса, кроме объявления внутри класса Ship.

Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

**Ship.Engine obj = new Ship().new Engine();**

Если внутренний класс наследуется обычным образом другим классом (после extends указывается ИмяВнешнегоКласса.ИмяВнутреннегоКласса), то он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

**public class Motor extends Ship.Engine {**

**public Motor(Ship obj) {**

**obj.super();**

**}**

**}**

В данном случае конструктор класса Motor должен быть объявлен с параметром типа Ship, что позволит получить доступ к ссылке на внутренний класс Engine, наследуемый классом Motor.

**ВЛОЖЕННЫЕ (NESTED) КЛАССЫ**

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим.

Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.

При объявлении такого внутреннего класса присутствует служебное слово static, и такой класс называется вложенным (nested).

Пример:

**public class Ship {**

**private int id;**

**// abstract, final, private, protected - допустимы**

**public static class LifeBoat {**

**public static void down() {**

**System.out.println("шлюпки на воду!");**

**}**

**public void swim() {**

**System.out.println("отплытие шлюпки");**

**}**

**}**

**}**

**public class RunnerShip {**

**public static void main(String[] args) {**

**// вызов статического метода**

**Ship.LifeBoat.down();**

**// создание объекта статического класса**

**Ship.LifeBoat lf = new Ship.LifeBoat();**

**// вызов обычного метода**

**lf.swim();**

**}**

**}**

**ИСКЛЮЧЕНИЯ**

Java предоставляет языковую конструкцию, которая носит название механизм исключений, и позволяет осуществлять обработку ошибок, возникающих в процессе выполнения программы.

Исключения возникают когда возникшая проблема не может быть решена в текущем контексте и невозможно продолжение работы программы.

Исключение представляет собой событие, происходящее в процессе выполнения Java-программы в результате нормального хода выполнения команд.

При этом процесс обработки исключения называется перехватом исключения.

Процесс обработки ошибок в Java сводится к перехвату возбужденных исключений.

Исключения представляют собой механизм взаимодействия между кодом, служащим для обнаружения ошибки и кодом, обрабатывающим ошибку.

Исключение возбуждается (throws) тогда, когда возникает определенная ситуация.

После возбуждения исключение перехватывается (catch).

Если исключение не перехватывается явным образом, то вызывается обработчик исключительных ситуаций, определенный в Java по умолчанию, который выводит информацию о стеке вызова текущего метода.

Все исключения являются наследниками суперкласса Throwable и его подклассов Error и Exception из пакета java.lang.

Исключения делятся на две категории: проверяемые и непроверяемые.

К непроверяемым исключениям относятся исключения, унаследованные от классов Error и RunTimeException. Эти исключения возбуждаются ядром виртуальной машины Java.

Проверяемые исключения описываются явно при определении метода или конструктора с помощью ключевого слова throws:

**[модификаторы] тип метод (списокПараметров) throws списокИсключений**

**{**

**телоМетода**

**}**

**или**

**[модификаторы] конструктор (списокПараметров) throws списокИсключений**

**{**

**телоКонструктора**

**}**

Для создания проверяемых исключений расширяется класс Exception,

например:

**class MyException extends Exception {...}**

Если при объявлении метода или конструктора класса указано обрабатываемое исключение, то программист при использовании этого метода или конструктора обязан исполнить одно из трех действий:

1. Перехватить и возбудить данное исключение при помощи ключевого слова throws.

2. Перехватить и обработать исключение при помощи управляющей конструкции try-catch.

3. В текущем методе или конструкторе не перехватывать исключение и при этом обязательно объявить текущий метод или конструктор с ключевым словом throws.

Для перехвата проверяемых исключений, которые были возбуждены, используется следующая конструкция:

**try**

**{**

**операторы, в работе которых может возбудиться исключение**

**}**

**[catch типИсключения1 переменная1**

**оператор1 ]**

**[catch типИсключения2 переменная2**

**оператор2 ]**

**[...]]]**

**[finally**

**{**

**операторN**

**}]**

Оператор после ключевого слова try исполняется до тех пор, пока он не будет успешно исполнен или до тех пор, пока не будет возбуждено исключение.

В случае возбуждения исключения будут просмотрены все условия catch для поиска подходящего исключения или одного из родительских классов этого исключения и исполнены соответствующие операторы после этих условий.

Исполнение оператора после ключевого слова finally всегда происходит независимо от того, было ли возбуждено исключение или нет.

Это условие обычно используется для установки состояния объекта или для освобождения внешних ресурсов, например, для закрытия файла.

Если исключение перехвачено определенным catch - выражением, то управление программой передается операторам этого catch - блока. Обычно в таких catch - блоках выполняются специальные действия, необходимые для устранения ошибок работы программы, вызванных возбужденным исключением.

По завершении работы операторов catch – блока управление все равно передается блоку операторов finally. Однако это не означает обязательность использования finally - блока в конструкции перехвата исключений.

**ОПЕРАТОР THROW**

Исключительную ситуацию можно создать с помощью оператора throw, если объект-исключение уже существует, или инициализировать его прямо после этого оператора. Для этого может быть использован объект класса Throwable или объект его подкласса, а также ссылки на них.

**throw объектThrowable;**

Объект-исключение может уже существовать или создаваться с помощью оператора new:

**throw new IOException();**

Инициализация объекта-исключения без оператора throw никакой исключительной ситуации не вызовет!

**ОТЛАДОЧНЫЙ МЕХАНИЗМ ASSERTION**

При помощи assertion можно сформулировать требования к входным, выходным и промежуточным данным методов классов в виде некоторых логических условий.

Инструкция assert:

**assert (boolexp): expression;**

**assert (boolexp);**

Пример:

**int age = ob.getAge();**

**assert (age >= 0): "NEGATIVE AGE!!!";**

**// реализация**

**ЛЕКЦИЯ 3**

**МАССИВЫ И СТРОКИ**

**МАССИВЫ**

Массивом в Java называется упорядоченный набор элементов. В качестве элементов массива могут выступать как данные примитивных типов, так и ссылки на объекты, включая ссылки на другие массивы.

Массив объявляется и создается при помощи выражения:

**ТипЭлементов[] имяМассива =**

**new типЭлементов [размерМассива]**

или, эквивалентно:

**ТипЭлементов имяМассива[] =**

**new типЭлементов [размерМассива]**

Для создания массива необходимо выполнить следующие действия:

* Объявить массив - задать имя массива и тип элементов.
* Выделить для массива память - задать количество его элементов в вызове оператора new.
* Инициализировать массив, поместив в его элементы данные

Массивы являются объектами. Размер массива хранится в поле length объекта. Если массив имеет длину n, то корректными значениями индекса являются числа от 0 до n-1.

При обращении к массиву по некорректному индексу возбуждается исключение IndexOutOfBounds.

Доступ к элементам массива имеет вид:

**массив[индекс]**

Доступ к полям массива имеет вид:

**массив.поле**

Доступ к полям и методам экземпляров класса, являющихся элементами массива, имеет вид:

**массив[индекс].поле**

**массив[индекс].метод(параметрыМетода)**

Можно создавать массив, элементы которого являются массивами. Объявление и создание многомерных массивов имеет вид:

**типЭлемента [][] ...имяМассива =**

**new типЭлементов [размер1][размер2]...**

Например:

**int [][][] narr = new int [2][3][4];**

При создании многомерного массива обязательно требуется указывать первый размер слева. Другие размеры для вложенных массивов можно указывать позже при помощи оператора new.

Например:

**int [][][] narr = new int [2][][];**

**...**

**narr[0] = new int [3][];**

**narr[1] = new int [3][];**

Инициализирующие значения массива задаются в фигурных скобках сразу же после объявления массива, например:

**int [] nA ={1,2,3,4};**

**int [][] nB = {{1,0,0},{0,1,0},{0,0,1}};**

**String strArr[] ={"aaa", "bbb", "cde"+"xyz"};**

**int [][][] narr = {**

**{ {0}, {0, 1}, {0, 1, 2} },**

**{ {0, 1, 2}, {0, 1}, {0} }**

**};**

ПРИМЕР:

**public class ex1 {**

**static int [][][] narr = {**

**{ {0}, {0, 1}, {0, 1, 2} },**

**{ {0, 1, 2}, {0, 1}, {0} }**

**};**

**public static void main( String [] args ) {**

**System.out.println( narr.length );**

**for ( int i = 0; i < narr.length; i++ ) {**

**System.out.println( narr[i].length );**

**for ( int j = 0; j < narr[i].length; j++ )**

**System.out.print(**

**narr[i][j].length + " " );**

**System.out.println();**

**}**

**}**

**}**

**Результат:**

**2**

**3**

**1 2 3**

**3**

**3 2 1**

**МАССИВЫ – КЛАСС JAVA.UTIL.ARRAYS**

Класс Arrays содержит несколько удобных методов, предназначенных для работы с массивами.

**static String toString(type[] a);**

- возвращает строку с элементами а, заключенную в квадратные скобки и разделенную запятыми.

**static type[] copyOf(type[] a, int length);**

**static type[] copyOf(type[] a, int start, int end);**

-возвращает массив того же типа, что и а, длиной либо length, либо end - start, заполненный значениями из а.

**static void sort(type[] a);**

- cортирует массив, используя алгоритм быстрой сортировки

**static int binarySearch(type[] a, type v);**

**static int binarySearch(type[] a, int start, int end, type v);**

- использует алгоритм бинарного поиска для нахождения значения v; в случае успеха возвращается индекс найденного элемента; в противном случае возвращается отрицательное значение г; -г - 1 указывает на индекс позиции, куда должен быть вставлен искомый элемент, чтобы сохранился порядок сортировки.

**static void fill(type[] a, type v);**

- устанавливает все элементы массива в v.

**static boolean equals(type[] a, type[] b);**

- возвращает true, если массивы имеют равную длину и совпадают все их элементы в соответствующих позициях индекса.

**СТРОКИ**

Строки – это основной носитель текстовой информации. Строки не являются массивами символов типа char, это объекты соответствующего класса.

Пакет java.lang содержит классы String, StringBuilder и StringBuffer, поддерживающие работу со строками.

Эти классы объявлены как final, что лишает возможности создавать на их основе порожденные классы.

**СТРОКИ – КЛАСС STRING**

Каждая строка, создаваемая с помощью оператора new или с помощью литерала является объектом класса String.

Значение объекта String не может быть изменено после создания при помощи какого-либо метода класса. Любое изменение строки приводит к созданию нового объекта.

Если в выражении присутствует хотя бы один объект типа String, остальные объекты преобразуются в String с помощью метода toString().

Класс String поддерживает несколько конструкторов

**String()**

**String(String str)**

**String(byte asciichar[])**

**String(char[] unicodechar)**

**String(StringBuffer sbuf)**

**String(StringBuilder sbuild)**

**и др.**

Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект типа String, на который можно установить ссылку.

Пример:

**public class ex2 {**

**public static void main( String [] args ) {**

**String str = "123“, strSave = str;**

**str += "xyz";**

**boolean flag = str == strSave;**

**System.out.println(**

**"The vale of \"str == strSave\" is "**

**+ flag );**

**}**

**}**

Результат:

**The vale of "str == strSave" is false**

String содержит следующие методы для работы со строками:

**String concat(String s); // или “+”**

– слияние строк;

**boolean equals(Object ob);**

**boolean equalsIgnoreCase(String s);**

- сравнение строк с учетом и без учета регистра соответственно

**int compareTo(String s);**

**int compareToIgnoreCase(String s);**

- лексикографическое сравнение строк с учетом и без учета регистра, путём вычитания кодов символов вызывающей и передаваемой в метод строк

**boolean contentEquals(StringBuffer ob);**

-сравнение строки и содержимого объекта типа StringBuffer;

**String substring(int n);**

- извлечение из строки подстроки, начиная с позиции n;

**String substring(int n, int m);**

- извлечение из строки подстроки длины m-n, начиная с позиции n. Нумерация символов в строке начинается с нуля;

**int length();**

- длина строки

**boolean isEmpty();**

- проверяет длину строки на 0

**char charAt( int i );**

- символ в позиции

**String trim();**

- удаление пробельных символов вначале и в конце строки

**int indexOf( char c );**

**int indexOf( char c, int fromIdx );**

**int indexOf( String s );**

**int indexOf( String s, int fromIdx );**

- поиск в прямом направлении символа или строки начиная с указанного индекса; возвращает индекс или -1

**int lastIndexOf( char c );**

**int lastIndexOf( char c, int fromIdx );**

**int lastIndexOf( String s );**

**int lastIndexOf( String s, int fromIdx );**

- поиск в обратном направлении символа или строки начиная с указанного индекса; возвращает индекс или -1

**String toLowerCase();**

**String toUpperCase();**

- изменение регистра символов

**СТРОКИ – КЛАСС STRINGBUFFER**

Классы StringBuilder и StringBuffer являются “близнецами” и по своему предназначению близки к классу String, но, содержимое и размеры объектов классов StringBuilder и StringBuffer можно изменять.

Единственным отличием StringBuffer от StringBuilder является потокобезопасность StringBuffer. StringBuilder следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

Объекты этих классов можно преобразовать в объект класса String методом toString() или с помощью конструктора класса String.

Методы:

**int length();**

- размер строки

**int capasity();**

- размер буфера

**void setLength( int n );**

- изменение размера строки

**void ensureCapacity( int n );**

- изменение размера буфера

**char charAt( int i );**

- символ в позиции

**void setCharAt( int i, char ch );**

- изменение символа в позиции

**StringBuffer reverse();**

- обращение содержимого объекта

**StringBuffer append( String s );**

**StringBuffer append( int i );**

**StringBuffer append( Object obj );**

- добавление аргумента в конец строки

**StringBuffer insert( int idx, String s );**

**StringBuffer insert( int idx, char c );**

**StringBuffer insert( int idx, Object obj );**

- добавление аргумента в указанную позицию строки

**StringBuffer delete(int idx, int idxLast);**

- удаление подстроки

**StringBuffer replace(int idx, int idxLast, String str);**

- замена подстроки

**StringBuffer deleteCharAt(int pos);**

- удаление символа

**String substring( int idx );**

**String substring( int idx, int idxLast );**

- выделение подстроки

**int indexOf( String s );**

**int indexOf( String s, int fromIdx );**

- прямой поиск подстроки

**int lastIndexOf( String s );**

**int lastIndexOf( String s, int fromIdx );**

- обратный поиск подстроки

При создании объекта StringBuffer конструктор резервирует некоторый объем памяти, что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе.

Если длина строки StringBuffer после изменения превышает его размер, то ёмкость объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений.

Если метод, вызываемый объектом StringBuffer, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта String, а изменяет текущий объект StringBuffer.

Пример:

**public class DemoStringBuffer {**

**public static void main(String[] args) {**

**StringBuffer sb = new StringBuffer();**

**sb.append("Java");**

**System.out.println("строка ->" + sb);**

**System.out.println("длина ->" + sb.length());**

**System.out.println("размер ->" + sb.capacity());**

**System.out.println("реверс ->" + sb.reverse());**

**}**

**}**

Результат:

**строка ->Java**

**длина ->4**

**размер ->16**

**реверс ->avaJ**

Пример:

**public class RefStringBuffer {**

**public static void changeStr(StringBuffer s) {**

**s.append(" Microsystems");**

**}**

**public static void main(String[] args) {**

**StringBuffer str = new StringBuffer("Sun");**

**changeStr(str);**

**System.out.println(str);**

**}**

**}**

Результат:

**Sun Microsystems**

**СТРОКИ – КЛАСС STRINGTOKENIZER**

Используется для разбиения строки на лексемы.

Конструкторы:

**public StringTokenizer(String string);**

**public StringTokenizer(String string, String delimiters);**

Методы:

**public boolean hasMoreTokens();**

**public String nextToken();**

**public String nextToken(String newDelimiters);**

Пример:

**import java.util.StringTokenizer;**

**public class ex3{**

**public static void main( String [] args ){**

**String sentence = "It\'s a sentence, “+ “it can be tokenized.";**

**StringTokenizer st = new StringTokenizer (sentence, " ,.!?;-\n\r");**

**while ( st.hasMoreTokens()) {**

**System.out.println(st.nextToken());**

**}**

**}**

**}**

Результат:

**It's**

**a**

**sentence**

**it**

**can**

**be**

**tokenized**

**ИНТЕРНАЦИОНАЛИЗАЦИЯ**

Интернационализация - это процесс создания приложений таким образом, чтобы они легко адаптировались для различных языков и регионов без внесения конструктивных изменений.

Характеристики интернационализированного приложения:

* один и тот же код может работать в любой местности при условии добавления данных о локализации;
* приложение отображает текст на родном языке конечного пользователя;
* текстовые элементы не являются частью кода, а хранятся отдельно и запрашиваются динамически;
* поддержка новых языков не требует перекомпиляции;
* данные, зависящие от местности, такие как даты и денежные единицы, отображаются в соответствии с регионом и языком конечного пользователя;
* приложение может быть быстро и легко локализовано.

**ЛОКАЛИЗАЦИЯ**

Локализация - это процесс адаптации программного обеспечения для определенного региона или языка путем добавления специфических для данной местности компонентов и перевода текста.

Данные, зависящие от местности:

* текст;
* числа;
* денежные единицы;
* дата и время;
* изображения;
* цвета;
* звуки.

Класс java.util.Locale позволяет учесть особенности региональных представлений алфавита, символов и проч. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять. Для некоторых стран региональные параметры устанавливаются с помощью констант, например: Locale.US, Locale.FRANCE. Для других стран объект Locale нужно создавать с помощью конструктора:

**Locale myLocale = new Locale("bel", "BY");**

Получить доступ к текущему варианту региональных параметров можно следующим образом:

**Locale current = Locale.getDefault();**

Для создания приложений, поддерживающих несколько языков можно использовать возможности классов java.util.ResourceBundle и Locale.

Класс ResourceBundle предназначен для работы с текстовыми файлами свойств (расширение .properties).

Чтобы выбрать определенный объект ResourceBundle, следует вызвать метод ResourceBundle.getBundle(параметры). Следующий фрагмент выбирает text объекта ResourceBundle для объекта Locale, который соответствует английскому языку, стране Канаде и платформе UNIX.

**Locale currentLocale = new Locale("en", "CA", "UNIX");**

**ResourceBundle rb = ResourceBundle.getBundle("text", currentLocale);**

Если объект ResourceBundle для заданного объекта Locale не существует, то метод getBundle() извлечет наиболее общий. В случае если общее определение файла ресурсов не задано, то метод getBundle() генерирует исключительную ситуацию MissingResourceException. Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов:

**text.properties**

В файлах свойств информация должна быть организована по принципу:

**key1 = value1**

**key2 = value2**

**…**

Пример программы:

**import java.util.\*;**

**public class IntTest {**

**static public void main(String args[]) {**

**if (args.length != 2) {**

**System.out.println("Format: java IntTest lang country");**

**System.exit(-1);**

**}**

**String language = new String(args[0]);**

**String country = new String(args[1]);**

**Locale loc = new Locale(language, country);**

**ResourceBundle messages =**

**ResourceBundle.getBundle("MessagesBundle", loc);**

**System.out.println(messages.getString("greeting"));**

**System.out.println(messages.getString("inquiry"));**

**System.out.println(messages.getString("farewell"));**

**}**

**}**

Файлы ресурсов:

MessageBundle.properties:

**greeting = Hello!**

**inquiry = How are you?**

**farewell = Goodbye!**

MessageBundle\_ru\_RU.properties:

**greeting = Привет!**

**inquiry = Как дела?**

**farewell = До свидания!**

**КЛАСС LOCALE**

Представляет определенный географический, политический или культурный регион (местность).

Конструкторы:

**public Locale (String language, /\* ISO 639\*/ String country); // ISO 3166**

**public Locale (String language, String country, String variant);**

Пример:

**Locale current = new Locale("en", "US");**

**Locale loc = new Locale("ru", "RU", "koi8r");**

Методы:

**public String getLanguage();**

**public String getCountry();**

**public String getVariant();**

**public static Locale getDefault();**

**public static void setDefault(Locale loc);**

Метод Locale.getDefault() возвращает значение Locale, используемое по умолчанию. Установить его можно следующим образом:

* с помощью системных свойств user.language и user.region
* с помощью метода Locale.setDefault()

Получить список возможных комбинаций языка и страны можно с помощью статического метода getAvailableLocales() различных классов, которые используют форматирование с учетом местных особенностей.

Например:

**Locale list[] = DateFormat.getAvailableLocales();**

**КЛАСС ResourceBundle**

Абстрактный класс, предназначенный для хранения наборов зависящих от местности ресурсов.

Используется один из его потомков:

**ListResourceBundle**

или

**PropertyResourceBundle**

Это набор связанных классов с единым базовым именем, и различающихся суффиксами, задающими язык, страну и вариант.

Например:

**MsgBundle**

**MsgBundle\_ru**

**MsgBundle\_en\_US**

**MsgBundle\_fr\_CA\_UNIX**

Основные методы:

**public static final ResourceBundle getBundle(String name)**

**throws MissingResourceException;**

**public static final ResourceBundle getBundle(String name, Locale locale)**

**throws MissingResourceException;**

- возвращают объект одного из подклассов ResourceBundle с базовым именем name и местностью, заданной объектом locale или взятой по умолчанию. При отсутствии ресурса осуществляется поиск подходящего путем последовательного исключения суффиксов, при неудаче инициируется исключение: **MissingResourceException.**

**КЛАСС ListResourceBundle**

Абстрактный класс, управляющий ресурсами с помощью списка. Используется созданием набора классов, расширяющих ListResourceBundle, для каждой поддерживаемой местности и определения метода getContents().

Пример:

**public class MsgBundle\_ru extends ListResourceBundle {**

**public Object[][] getContents() {**

**return contents;**

**}**

**public Object[][] contents = {**

**{ "greeting", "Привет!" },**

**{ "inquiry", "Как дела?" },**

**{ "farewell", "До свидания!" }**

**};**

**}**

Массив **contents** содержит список пар ключ-значение, причем ключ должен быть объектом типа **String**, а значение - **Object**.

Объект класса **ListResourceBundle** можно получить вызовом статического метода **ResourceBundle.getBundle().**

**ResourceBundle messages = ResourceBundle.getBundle( "MsgBundle", new Locale("ru", "RU"));**

Поиск классов осуществляется в следующей последовательности:

1. **MsgBundle\_ru\_RU.class**
2. **MsgBundle\_ru.class**
3. **MsgBundle.class**

Для получения значения объекта используется метод getObject:

**String s = (String)**

**messages.getObject("greeting");**

КЛАСС PropertyResourceBundle

Абстрактный класс, управляющий ресурсами с помощью набора свойств. Используется в случаях, когда локализуемые объекты имеют тип String. Ресурсы хранятся отдельно от кода, поэтому для добавления новых ресурсов не требуется перекомпиляция.

Пример файлa Msg\_ru.properties:

**greeting = Привет!**

**inquiry = Как дела?**

**farewell = До свидания!**

Объект класса PropertyResourceBundle можно получить вызовом статического метода ResourceBundle.getBundle():

**ResourceBundle messages = ResourceBundle.getBundle( "Msg", new Locale(“ru", “RU"));**

Если getBundle() не может найти соответствующий класс, производится поиск файлов с расширением .properties в той же последовательности, как и для ListResourceBundle:

1. **Msg\_ru\_RU.properties**
2. **Msg\_ru.properties**
3. **Msg.properties**

Для получения значения свойства используется метод getString:

**String s = messages.getString("greeting");**

**КЛАСС NumberFormat**

Абстрактный класс, позволяющий форматировать числа, денежные единицы, проценты в соответствии с форматом, принятым в определенной местности.

Форматирование осуществляется в 2 этапа:

1. получение требуемого экземпляра класса с помощью одного из методов getNumberInstance, getCurrencyInstance, getPercentInstance.
2. вызов метода format() для получения отформатированной строки.

Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода

**public Locale[] NumberFormat.getAvailableLocales();**

Числа:

**NumberFormat formatter = NumberFormat.getNumberInstance(Locale.GERMANY);**

**String result = formatter.format(123456.789);**

Денежные единицы:

**NumberFormat formatter = NumberFormat.getCurrencyInstance(Locale.FRANCE);**

**String result = formatter.format(4999.99);**

Проценты:

**NumberFormat formatter = NumberFormat.getPercentInstance(Locale.US);**

**String result = formatter.format(.75);**

**КЛАСС DecimalFormat**

Позволяет создавать собственные форматы для чисел, денежных единиц и процентов.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента

**String pattern = "###,##0.##";**

**DecimalFormat formatter =**

**new DecimalFormat(pattern);**

1. Вызывается метод format() для получения отформатированной строки

**String s = formatter.format(123123.456);**

Значения символов шаблона:

***Символ Значение***

0 цифра

# цифра, или пробел в случае нуля

. десятичный разделитель

, групповой разделитель

; разделитель форматов

- префикс отрицательного числа

% процент (значение умножается на 100)

? промилле (значение умножается на 1000)

¤ заменяется обозначением денежной единицы

(международным если удвоен) и в формате вместо десятичного будет использован денежный разделитель

X любой другой символ в префиксе или суффиксе

' используется для экранирования специальных символов в префиксе или суффиксе

**КЛАСС DecimalFormatSymbols**

Для изменения значения используемых по умолчанию разделителей в классе DecimalFormat используется класс DecimalFormatSymbols.

Конструкторы:

**public DecimalFormatSymbols();**

**public DecimalFormatSymbols(Locale locale);**

Методы:

**public void setZeroDigit(char c);**

**public void setGroupingSeparator(char c);**

**public void setDecimalSeparator(char c);**

**public void setPerMill(char c);**

**public void setPercent(char c);**

**public void setDigit(char c);**

**public void setNaN(char c);**

**public void setInfinity(char c);**

**public void setMinusSign(char c);**

**public void setPatternSeparator(char c);**

Имеются соответствующие методы get() для получения установленных значений.

Для передачи значений разделителей объект **DecimalFormatSymbols** передается конструктору класса **DecimalFormat** в качестве аргумента:

**DecimalFormatSymbols symbols = new DecimalFormatSymbols();**

**symbols.setGroupingSeparator(" ");**

**DecimalFormat formatter = new DecimalFormat("#,##0.00", symbols);**

**String s = formatter.format(4.50);**

**КЛАСС DateFormat**

Абстрактный класс, позволяющий форматировать дату и время в соответствии с форматом, принятым в определенной местности.

Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода

**public Locale[] DateFormat.getAvailableLocales();**

Стили форматирования определяются константами:

**DateFormat.DEFAULT,**

**DateFormat.SHORT,**

**DateFormat.MEDIUM,**

**DateFormat.LONG,**

**DateFormat.FULL**

Форматирование осуществляется в 2 этапа:

1. Получение требуемого экземпляра класса с помощью одного из методов

* + - **getDateInstance**
    - **getTimeInstance**
    - **getDateTimeInstance**

2. Вызов метода format() для получения отформатированной строки.

Дата:

**DateFormat fmt= DateFormat.getDateInstance(DateFormat.SHORT, Locale.UK);**

**String result = fmt.format(new Date());**

Время:

**DateFormat fmt= DateFormat.getTimeInstance(DateFormat.LONG, Locale.FRANCE);**

**String result = fmt.format(new Date());**

Дата и время:

**DateFormat f= DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL, Locale.US);**

**String result = f.format(new Date());**

**КЛАСС SimpleDateFormat**

Позволяет создавать форматы для даты и времени.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента:

**SimpleDateFormat f= new SimpleDateFormat( "K:mm EEE MMM d ''yy");**

2. Вызывается метод format() для получения отформатированной строки:

**String s = f.format(new Date());**

Шаблоны SimpleDateFormat

Символ Значение Тип Пример

G обозначение эры текст AD

y год число 1996

M месяц года текст/число July или 07

d число месяца число 23

h часы (1-12) число 5

H часы (0-23) число 22

m минуты число 45

s секунды число 31

S миллисекунды число 978

E день недели текст Tuesday

D номер дня в году число 189

F день недели в месяце число 2 (2nd Wed in July)

w неделя в году число 27

W неделя в месяце число 2

a знак AM/PM текст PM

k часы (1-24) число 24

K часы (0-11) число 0

z временная зона текст GMT

**КЛАСС DateFormatSymbols**

Используется для изменения названий месяцев, дней недели и других значений в классе SimpleDateFormat.

Конструкторы:

**public DateFormatSymbols();**

**public DateFormatSymbols(Locale locale);**

Методы:

**public void setEras(String newValue[]);**

**public void setMonths(String newValue[]);**

**public void setShortMonths(String newValue[]);**

**public void setWeekDays(String newValue[]);**

**public void setShortWeekDays(String newValue[]);**

**public void setAmPmStrings(String newValue[]);**

**public void setZoneStrings(String newValue[]);**

**public void setPatternChars(String newValue[]);**

Имеются соответствующие методы get() для получения установленныхзначений.

Для передачи значений разделителей объект DateFormatSymbols передается конструктору класса SimpleDateFormat в качестве аргумента:

**DateFormatSymbols symbols = new DateFormatSymbols();**

**String weekdays[] = {"Пн","Вт","Ср","Чт","Пт","Сб","Вс"};**

**symbols.setShortWeekDays(weekdays);**

**DecimalFormat f= new SimpleDateFormat("E", symbols);**

**String s = f.format(new Date());**

**КЛАСС MessageFormat**

Используется для выдачи сообщений на различных языках с включением изменяющихся объектов.

Использование класса:

1. Выделение переменных объектов в сообщении:

**At 1:15 PM on April 13, 1998, we detected 7 spaceships on the planet Mars.**

2. Помещение шаблона сообщения в ResourceBundle:

**ResourceBundle messages = ResourceBundle.getBundle("MessageBundle", currentLocale);**

cодержимое файла **MessageBundle.properties**:

**template = At {2,time,short} on {2,date,long}, we detected {1,number,integer} spaceships on the planet {0}.**

**planet = Mars**

3. Установка аргументов сообщения:

**Object[] args = {**

**messages.getString("planet"),**

**new Integer(7),**

**new Date()**

**}**

4. Создание объекта MessageFormat:

**MessageFormat formatter = new MessageFormat(messages.getString("template"));**

**formatter.setLocale(currentLocale);**

5. Форматирование сообщения:

**String s = formatter.format(args);**

Синтаксис аргументов MessageFormat:

**{ индекс аргумента, [ тип, [ стиль ] ] }**

Индекс задает порядковый индекс аргумента в массиве объектов (0-9).

Типы и стили аргументов:

**Возможные типы Возможные стили**

**number currency, percent, integer,**

**шаблон числа**

**date short, long, full, medium,**

**шаблон даты**

**time short, long, full, medium,**

**шаблон времени**

**КЛАСС ChoiceFormat**

Используется для задания возможности выбора различных элементов в зависимости от значения параметров.

Использование класса:

1. Выделение переменных объектов в сообщении:

**There are no files on disk C.**

**There is one file on disk C.**

**There are 3 files on disk C.**

2. Помещение шаблона сообщения в ResourceBundle:

cодержимое файла MessageBundle.properties:

**template = There {0} on disk {1}.**

**no = are no files**

**one = is one file**

**many = are {2} files**

3. Создание объекта ChoiceFormat:

**double limits[] = {0,1,2}**

**String choices[] = { messages.getString("no"), messages.getString("one"), messages.getString("many") }**

**ChoiceFormat choice = new ChoiceFormat(limits, choices);**

4. Создание объекта MessageFormat:

**MessageFormat formatter = new MessageFormat(messages.getString("template"));**

**formatter.setLocale(currentLocale);**

**Format[] formats = { choice, null, NumberFormat.getInstance()};**

**formatter.setFormats(formats);**

5. Установка аргументов сообщения:

**Object[] args = { 1, "C", 1 };**

6. Форматирование сообщения

**String s = formatter.format(args);**

**ЛЕКЦИЯ 4**

**ВВОД/ВЫВОД**

**КЛАСС File**

Для работы с физическим файлами и каталогами на внешних носителях, в Java используются классы из пакета java.io.

Класс File предназначен для хранения и обработки каталогов и имен файлов. Не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Примеры создания объектов класса File:

**File myFile = new File(”\\com\\myfile.txt”);**

**File myDir = new File( ”c:\\jdk1.7.0\\src\\java\\io”);**

**File myFile = new File(myDir, ”File.java”);**

**File myFile = new File(”c:\\com”, ”myfile.txt”);**

**File myFile = new File( new URI(”htmp://www.bsu.by/index.html”));**

При создании объекта класса File любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе File:

**public static final String separator;**

**public static final char separatorChar;**

С помощью этих полей можно задать путь, универсальный в любой системе:

**File myFile = new File( File.separator + ”com” + File.separator + ”myfile.txt” );**

Предусмотрен еще один тип разделителей – для директорий переменной PATH:

**public static final String pathSeparator;**

**public static final char pathSeparatorChar;**

К примеру, для ОС Unix значение pathSeparator=”:”, а для Windows – pathSeparator=”;”.

Пример:

**/\* FileTest.java \*/**

**package fpack;**

**import java.io.\*;**

**import java.util.\*;**

**public class FileTest {**

**public static void main(String[] args) {**

**//c объектом типа File ассоциируется файл на диске FileTest2.java**

**File fp = new File("fpack"+ File.separator+ "FileTest2.java");**

**if (fp.exists()) {**

**System.out.println(fp.getName() + " существует");**

**if(fp.isFile()) { //если объект - дисковый файл**

**System.out.println("Путь к файлу:\t"+ fp.getPath());**

**System.out.println("Абсолютный путь:\t"+ fp.getAbsolutePath());**

**System.out.println("Размер файла:\t"+ fp.length());**

**System.out.println("Последняя модификация :\t"+ new Date(fp.lastModified()));**

**System.out.println("Файл доступен для чтения:\t"+ fp.canRead());**

**System.out.println("Файл доступен для записи:\t"+ fp.canWrite());**

**System.out.println("Файл удален:\t"+ fp.delete());**

**}**

**} else**

**System.out.println("файл "+ fp.getName()+ " не существует");**

**try {**

**if ( fp.createNewFile())**

**System.out.println("Файл "+ fp.getName()+ " создан");**

**} catch(IOException e) {**

**System.err.println(e);**

**}**

**//в объект типа File помещается каталог\директория**

**// в корне проекта должен быть создан каталог com.learn**

**// с несколькими файлами**

**File dir = new File("com"+ File.separator+ "learn");**

**if (dir.exists() && dir.isDirectory())**

**/\*если объект является каталогом и если этот каталог существует\*/**

**System.out.println("каталог "+ dir.getName()+ " существует");**

**File[] files = dir.listFiles();**

**for(int i = 0; i < files.length; i++) {**

**Date date = new Date(files[i].lastModified());**

**System.out.print("\n"+ files[i].getPath()+ " \t| "+ files[i].length()+ "\t| "+ date.toString());**

**}**

**// метод listRoots() возвращает доступные корневые каталоги**

**File root = File.listRoots()[1];**

**System.out.printf("\n%s %,d из %,d свободно.", root.getPath(), root.getUsableSpace(), root.getTotalSpace());**

**}**

**}**

В результате файл FileTest2.java будет очищен, а на консоль выведено:

**FileTest2.java существует**

**Путь к файлу: pack\FileTest2.java**

**Абсолютный путь: D:\workspace\pack\FileTest2.java**

**Размер файла: 2091**

**Последняя модификация : Fri Mar 31 12:26:50 EEST 2010**

**Файл доступен для чтения: true**

**Файл доступен для записи: true**

**Файл удален: true**

**Файл FileTest2.java создан**

**каталог learn существует**

**com\learn\bb.txt | 9 | Fri Mar 24 15:30:33 EET 2010**

**com\learn\byte.txt| 8 | Thu Jan 26 12:56:46 EET 2010**

**com\learn\cat.gif | 670 | Tue Feb 03 00:44:44 EET 2011**

**C:\ 3 665 334 272 из 15 751 376 896 свободно.**

Каталог класса File имеет дополнительное свойство - просмотр списка имен файлов с помощью методов list(), listFiles(), listRoots().

**ПОТОКИ ВВОДА-ВЫВОДА**

Поток данных (stream) – это абстрактный объект для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

**ИЕРАРХИЯ ПОТОКОВ В JAVА**

Потоки реализуются классами пакета java.io.

Делятся на две больших группы — потоки ввода, и потоки вывода. Потоки ввода связаны с источниками данных, потоки вывода — с приемниками данных.

Кроме того, потоки можно разделить на байтовые и символьные. Единицей обмена для байтовых потоков является байт, для символьных — символ Unicode.

Базовые потоки ввода:

* **InputStream** – байтовый
* **Reader** -- символьный

Базовые потоки вывода:

* **OutputStream** – байтовый
* **Writer** -- символьный

Кроме этих основных потоков, в пакет входят специализированные потоки, предназначенные для работы с различными источниками или приемниками данных, а также преобразующие потоки, предназначенные для преобразования информации, поступающей на вход потока, и выдачи ее на выход в преобразованном виде.

**КЛАСС InputStream**

Класс InputStream – это абстрактный входной поток байтов, предок для всех входных байтовых потоков.

**Конструктор:**

**InputStream();**

Создает входной байтовый поток.

**Методы:**

* **abstract int read() throws IOException;**
  + - Читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение.
* **int read(byte[] buf);**
  + - Читает данные в буфер и возвращает количество прочитанных байтов.
* **int read(byte[] buf, int offset, int len);**
  + - Читает не более len байтов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных байтов
* **void close();**
  + - Закрывает поток.
* **int available();**
  + - Возвращает количество доступных на данный момент байтов для чтения из потока.
* **long skip(long n);**
  + - Пропускает указанное количество байтов из потока.
* **boolean markSupported();**
  + - Проверка на возможность повторного чтения из потока.
* **void mark(int limit);**
  + - Устанавливает метку для последующего повторного чтения. limit – размер буфера для операции повторного чтения.
* **void reset();**
  + - Возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки.

**КЛАСС OutputSrteam**

Класс OutputSrteam – это абстрактный выходной поток байтов, предок для всех выходных байтовых потоков.

**Конструктор:**

**OutputSrteam();**

Создает выходной байтовый поток.

**Методы:**

* **abstract void write(int n) throws IOException;**
  + - Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.
* **void write(byte[] buf);**
  + - Записывает в поток данные из буфера.
* **void write(byte[] buf, int offset, int len);**
  + - Записывает в поток len байтов из буфера, начиная со смещения offset.
* **void close();**
  + - Закрывает поток.
* **void flush();**
  + - Заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

**КЛАСС Reader**

Абстрактный входной поток символов, предок для всех входных символьных потоков.

**Конструктор:**

**Reader();**

Создает входной символьный поток

**Методы:**

* **abstract int read() throws IOException;**
  + - Читает очередной символ Unicode из входного потока. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение.
* **int read(char[] buf);**
  + - Читает данные в буфер и возвращает количество прочитанных символов.
* **int read(char[] buf, int offset, int len);**
  + - Читает не более len символов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных символов
* **void close();**
  + - закрывает поток
* **int available();**
  + - возвращает количество доступных на данный момент символов для чтения из потока
* **long skip(long n);**
  + - пропускает указанное количество символов из потока
* **boolean markSupported();**
  + - проверка на возможность повторного чтения из потока
* **void mark(int limit);**
  + - устанавливает метку для последующего повторного чтения. limit – размер буфера для операции повторного чтения
* **void reset();**
  + - возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки.

**КЛАСС Writer**

Абстрактный выходной поток символов, предок для всех выходных символьных потоков.

**Конструктор:**

**Writer();**

создает выходной символьный поток

**Методы:**

* **abstract void write(int n) throws IOException;**
  + - Записывает очередной символ Unicode в выходной поток. Значащими являются 16 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.
* **void write(char[] buf);**
  + - Записывает в поток данные из буфера.
* **void write(char[] buf, int offset, int len);**
  + - Записывает в поток len символов из буфера, начиная со смещения offset
* **void close();**
  + - закрывает поток
* **void flush();**
  + - заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

**СПЕЦИАЛИЗИРОВАННЫЕ ПОТОКИ**

Конструкторы этих потоков в качестве аргумента принимают ссылку на источник или приемник данных - файл, массив, строку. Методы для чтения и записи данных - read() для входных потоков, write() -для выходных потоков.

Конвейер имеет особенность, что источником данных для входного конвейера является выходной конвейер, и наоборот. Обычно конвейеры используются для обмена данными между двумя потоками выполнения (Thread).

В пакет java.io входят потоки для работы со следующими основными типами источников и приемников данных:

Файл:

* **FileInputStream** - входной байтовый поток
* **FileOutputStream** - выходной байтовый поток
* **FileReader** - входной символьный поток
* **FileWriter** - выходной символьный поток

Массив:

* **ByteArrayInputStream** - входной байтовый поток
* **ByteArrayOutputStream** - выходной байтовый поток
* **CharArrayReader** - входной символьный поток
* **CharArrayWriter** - выходной символьный поток

Строка:

* **StringReader** - входной символьный поток
* **StringWriter** - выходной символьный поток

Конвейер:

* **PipedInputStream** - входной байтовый поток
* **PipedOutputStream** - выходной байтовый поток
* **PipedReader** - входной символьный поток
* **PipedWriter** - выходной символьный поток

Пример чтения данных из файла:

**FileReader f = new FileReader(“myfile.txt”);**

**char[] buffer = new char[512];**

**f.read(buffer);**

**f.close();**

**ПРЕОБРАЗУЮЩИЕ ПОТОКИ**

Этот тип потоков выполняет некие преобразования над данными других потоков. Конструкторы таких классов в качестве аргумента принимают поток данных.

Классы **BufferedInputStream**, **BufferedOutputStream**, **BufferedReader** и **BufferedWriter** предназначены для буферизации ввода-вывода. Они позволяют читать и записывать данные большими блоками. При этом обмен данными со стороны приложения ведется с буфером, а по мере необходимости в буфер из источника данных подгружается новая порция данных, либо из буфера данные переписываются в приемник данных.

Класс **BufferedReader** имеет дополнительный метод **readLine()** для чтения строки символов, ограниченной разделителем строк.

Класс **BufferedWriter** имеет дополнительный метод **newLine()** для вывода разделителя строк.

Классы **InputStreamReader** и **OutputStreamWriter** предназначены для преобразования байтовых потоков в символьные и наоборот. Кодировка задается в конструкторе класса. Если она опущена, то используется системная кодировка, установленная по умолчанию).

В конструктор класса **InputStreamReader** передается как аргумент объект класса **InputStream**, а в конструктор класса **OutputStreamWriter** – объект класса **OutputStream**. Методы **read()** и **write()** этих классов аналогичны методам классов **Reader** и **Writer**.

Пример использования:

Вариант 1:

**FileInputStream f = new FileInputStream(“myfile.txt”);**

**InputStreamReader isr = new InputStreamReader(f);**

**BufferedReader br = new BufferedReader(isr);**

**br.readLine();**

Вариант 2:

**BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream(“myfile.txt”)));**

**br.readLine();**

Классы **DataInputStream** и **DataOutputStream** предназначены для записи и чтения примитивных типов данных и содержат методы **readBoolean(), readInt(), readDouble(), writeFloat(), writeByte()** и другие подобные методы.

Для успешного чтения таких данных из потока **DataInputStream** они должны быть предварительно записаны с помощью соответствующих методов **DataOutputStream** в том же порядке.

Классы **PrintStream** и **PrintWriter** предназначены для форматированного вывода в поток вывода. В них определено множество методов **print()** и **println()** с различными аргументами, которые позволяют напечатать в поток аргумент, представленный в текстовой форме (с использованием системной кодировки).

В качестве аргумента может использоваться любой примитивный тип данных, строка и любой объект. Методы **println** добавляют в конце разделитель строк.

**СТАНДАРТНЫЕ ПОТОКИ**

Класс **java.lang.System** содержит 3 поля, представляющих собой стандарные консольные потоки:

* + - **InputStream System.in** – стандартный поток ввода
    - **PrintStream System.out** - стандартный поток вывода
    - **PrintStream System.err** - стандартный поток ошибок

Имеется возможность перенаправлять данные потоки с помощью методов **System.setIn, System.setOut, System.setErr**.

Пример чтения данных с клавиатуры и вывода в окно терминала:

**BufferedReader br = new BufferedReader(new InputStreamReader(System.in));**

**String s = br.readLine();**

**System.out.println(“Введена строка : “ + s);**

**System.out.println(“Длина строки : “ + s.length);**

**java.lang.AutoClosable и java.io.Closable**

**public interface AutoCloseable {**

**void close() throws Exception;**

**}**

**public interface Closeable extends AutoCloseable {**

**public void close() throws IOException;**

**}**

**TRY C РЕСУРСАМИ**

Для классов реализующих интерфейс AutoClosable допустима специальная форма оператора try:

**try ( спецификация ресурса**

**[; спецификация ресурса] ) {**

**}**

**[блоки catch и finally]**

До выхода из блока try, метод close() вызывается автоматически для ресурсов объявленных в try.

Пример:

**try (FileInputStream fin = new FileinputStream(args[0]);**

**FileOutputStream fout = new FileOutputStream(args[1]))**

**{**

**while( true ) {**

**int i = fin.read();**

**if ( i == -1 ) break;**

**fout.write( i );**

**}**

**} // files are closed here**

**СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ**

Сериализация объектов - запись объекта со всеми полями и ссылками на другие объекты в виде последовательности байтов в поток вывода с последующим воссозданием (десериализацией) копии этого объекта путем чтения последовательности байтов сохраненного объекта из потока ввода.

**ИНТЕРФЕЙС java.io.Serializable:**

Интерфейс-метка, указывающий на то, что реализующий его класс может быть сериализован. Поля класса, не требующие сериализации, должны иметь модификатор transient.

При использовании Serializable десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.

При сериализации объекта класса, реализующего интерфейс Serializable, учитывается порядок объявления полей в классе. Поэтому при изменении порядка десериализация пройдет некорректно. На стадии компиляции, в каждый класс реализующий Serializable добавляется поле

**private static final long serialVersionUID;**

Это поле содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса – полям, их порядку объявления, методам, их порядку объявления.

Это поле записывается в поток при сериализации класса. Это единственный случай, когда static-поле сериализуется.

При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение java.io.InvalidClassException. Соответственно, при любом изменении в классе это поле поменяет свое значение.

Вместо реализации интерфейса Serializable можно реализовать Externalizable.

**ИНТЕРФЕЙС java.io.Externalizable**

Предназначен для реализации классами, которым требуется нестандартное поведение при сериализации. В интерфейсе описаны 2 метода:

**void writeExternal(ObjectOutput out);**

**void readExternal(ObjectInput in);**

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого должны быть переопределены методы writeExternal() и readExternal() интерфейса Externalizable. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении Externalizable-объекта экземпляр создается вызовом конструктора без аргументов, затем вызывается метод readExternal(), поэтому в классе должен быть пустой конструктор. Для сохранения состояния вызываются методы ObjectOutput, с помощью которых можно записать как примитивные, так и объектные значения.

Для чтения и записи в поток значений отдельных полей объекта можно использовать соответственно методы внутренних классов:

**ObjectInputStream.GetField**

**ObjectOutputStream.PutField.**

**КЛАСС java.io.ObjectOutputStream**

Предназначен для записи в поток вывода примитивных типов, подобно классу **DataOutputStream** и объектов (иерархически).

**Конструктор**:

**ObjectOutputStream(OutputStream o);**

Cоздает объект класса, связанный с выходным потоком o.

**Методы:**

* **void writeObject(Object obj);**
  + - Иерархически записывает в поток заданный объект.
* **void useProtocolVersion(int v);**
  + - Задает версию протокола сериализации (ObjectStreamConstants. PROTOCOL\_VERSION\_\*).
* **void defaultWriteObject();**
  + - Вызывается из метода writeObject сериализуемого класса для сохранения нестатических и нетранзитивых полей этого класса
* **writeBoolean, writeByte, writeShort, writeChar, writeInt, writeLong, writeFloat, writeDOuble, writeUTF**
  + - Методы, аналогичные методам класса **DataOutputStream** для записи в поток примитивных типов.

**КЛАСС java.io.ObjectInputStream**

Предназначен для получения из потока ввода примитивных типов, подобно классу **DataOutputStream** и объектов (иерархически), которые были предварительно записаны с помощью класса **ObjectOutputStream**.

**Конструктор:**

**ObjectInputStream(InputStream i);**

Создает объект класса, связанный с входным потоком i.

**Методы:**

* **Object readObject();**
  + - Получает из потока заданный объект и восстанавливает его иерархически.
* **void defaultReadObject();**
  + - Вызывается из метода readObject сериализуемого класса для восстановления нестатических и нетранзитивых полей этого класса.
* **readBoolean, readByte, readShort, readChar, readInt, readLong, readFloat, readDouble, readUTF**
  + - Методы, аналогичные методам класса DataInputStream для чтения из потока примитивных типов.

**СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ**

В случае, если стандартного поведения для сериализации объекта недостаточно, можно определить в сериализуемом классе методы

**private void writeObject(ObjectOutputStream oos);**

и

**private void readObject(ObjectInputStream ois);**

и определить в них необходимые действия по сериализации.

Пример сериализации объекта:

**ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(“file.dat”));**

**oos.writeObject(new Date());**

**oos.close();**

**ObjectInputStream ois = new ObjectInputStream(new FileInputStream(“file.dat”));**

**Date d = (Date) ois.readObject();**

**ois.close();**

**RANDOMACCESSFILE**

Позволяет осуществлять произвольный доступ к потокам как ввода, так и вывода.

Поток рассматривается как массив байтов. Класс позволяет перемещать указатель файла перед выполнением операции чтения из файла или записи в файл.

Конструкторы:

**RandomAccessFile(String name, String mode);**

**RandomAccessFile(File file, String mode);**

Параметр **mode** равен **"r"** для чтения или **"rw"** для чтения и записи.

**Методы:**

* **int read() throws IOException;**
  + - Читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение.
* **int read(byte[] buf) throws IOException;**
  + - Читает данные в буфер и возвращает количество прочитанных байтов.
* **int read(byte[] buf, int offset, int len) throws IOException;**
  + - Читает не более len байтов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных байтов
* **void close();**
  + - Закрывает поток.
* **long skipBytes(long n);**
  + - Пропускает указанное количество байтов из потока.
* **void write(int n) throws IOException;**
  + - Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.
* **void write(byte[] buf) throws IOException;**
  + - Записывает в поток данные из буфера.
* **void close();**
  + - Закрывает поток.
* **long skipBytes(long n);**
  + - Пропускает указанное количество байтов из потока.
* **void write(int n) throws IOException;**
  + - Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение.
* **void write(byte[] buf) throws IOException;**
  + - Записывает в поток данные из буфера.
* **void write(byte[] buf, int offset, int len);**
  + - Записывает в поток len байтов из буфера, начиная со смещения offset.
* **public long getFilePointer();**
  + - Возвращает позицию указателя файла отсчитывая от начала.
* **public void seek(long pos);**
  + - Устанавливает позицию указателя файла отсчитывая от начала.
* **writeBoolean, writeByte, writeShort, writeChar, writeInt, writeLong, writeFloat, writeDouble, writeUTF**
  + - Методы для записи в поток примитивных типов.
* **readBoolean, readByte, readShort, readChar, readInt, readLong, readFloat, readDouble, readUTF**
  + - Методы для чтения из потока примитивных типов

**Пример:**

**// RandFil.java**

**import java.io.\*;**

**public class RandFil {**

**public static void main(String[] args) {**

**long pos[] = { 0, 0, 0, 0 };**

**double dataDouble = 0.12345;**

**int dataInt = 6789;**

**boolean dataBool = false;**

**String dataString = "Test message.";**

**try (RandomAccessFile rf =**

**new RandomAccessFile("temp.txt", "rw") ) {**

**// запись в файл**

**pos[0] = rf.getFilePointer(); rf.writeDouble( dataDouble );**

**pos[1] = rf.getFilePointer(); rf.writeInt( dataInt );**

**pos[2] = rf.getFilePointer(); rf.writeBoolean( dataBool );**

**pos[3] = rf.getFilePointer(); rf.writeUTF( dataString );**

**// чтение в обратном порядке**

**rf.seek(pos[3]); System.out.println(rf.readUTF());**

**rf.seek(pos[2]); System.out.println(rf.readBoolean());**

**rf.seek(pos[1]); System.out.println(rf.readInt());**

**rf.seek(pos[0]); System.out.println(rf.readDouble());**

**} catch (IOException e) {**

**System.err.println(e);**

**}**

**}**

**}**

Результат работы программы:

**Test message.**

**false**

**6789**

**0.12345**

**АРХИВАЦИЯ**

Для хранения классов языка Java и связанных с ними ресурсов используются сжатые архивные jar-файлы.

Для работы с архивами в спецификации Java существуют два пакета – java.util.zip и java.util.jar соответственно для архивов zip и jar.

Различие форматов jar и zip заключается только в расширении архива zip. Пакет java.util.jar аналогичен пакету java.util.zip

Пакет java.util.jar позволяет считывать, создавать и изменять файлы форматов jar, а также вычислять контрольные суммы входящих потоков данных.

Для работы с jar-архивами используются классы:

* **JarEntry** – запись jar-файла,
* **JarFile** – даёт эффективный доступ к записям jar-файла при его чтении,
* **JarOutputStream** – выходной поток со сжатием выходных данных,
* **JarInputStream** – входной поток с распаковкой входных данных.

**АРХИВАЦИЯ – JarEntry**

Класс JarEntry используется для предоставления доступа к записям jar-файла. Наиболее важными методами класса являются:

* **void setMethod(int method);**

**//– устанавливает метод сжатия записи;**

* **int getMethod();**

**//– возвращает метод сжатия записи;**

* **void setComment(String comment);**

**//– устанавливает комментарий записи;**

* **String getComment();**

**//– возвращает комментарий записи;**

* **void setSize(long size);**

**//– устанавливает размер несжатой записи;**

* **long getSize();**

**//– возвращает размер несжатой записи;**

* **long getCompressedSize();**

**//– возвращает размер сжатой записи**;

**АРХИВАЦИЯ – JarOutputStream**

Класс JarOutputStream осуществляет запись данных в поток вывода в jar-формате. Он переопределяет метод write() таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались.

Основными методами данного класса являются:

* **void setLevel(int level) ;**

– устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;

* **void putNextEntry(ZipEntry e);**

– записывает в поток новую jar-запись. Этот метод переписывает данные из экземпляра JarEntry в поток вывода;

* **void closeEntry();**

– завершает запись в поток jar-записи и заносит дополнительную информацию о ней в поток вывода;

* **void write(byte b[], int off, int len);** – записывает данные из буфера b начиная с позиции off длиной len в поток вывода;
* **void finish();**

– завершает запись данных jar-файла в поток вывода без закрытия потока;

* **void close() ;**

– закрывает поток записи.

**АРХИВАЦИЯ – ЗАПИСЬ В JAR**

Пример:

**/\* создание jar-архива: PackJar.java \*/**

**import java.io.\*;**

**import java.util.jar.\*;**

**import java.util.zip.\*;**

**public class PackJar {**

**public static void pack( String[] filesToJar,**

**String jarFileName, byte[] buffer) {**

**try {**

**JarOutputStream jos =**

**new JarOutputStream(**

**new FileOutputStream( jarFileName ));**

**// метод сжатия**

**jos.setLevel(Deflater.DEFAULT\_COMPRESSION);**

**for (int i = 0; i < filesToJar.length; i++) {**

**System.out.println(i);**

**jos.putNextEntry(new JarEntry( filesToJar[i] ));**

**FileInputStream in =**

**new FileInputStream( filesToJar[i] );**

**int len;**

**while (( len = in.read( buffer )) > 0 )**

**jos.write(buffer, 0, len);**

**jos.closeEntry();**

**in.close();**

**}**

**jos.close();**

**} catch ( IllegalArgumentException e ) {**

**e.printStackTrace();**

**System.err.println("Некорректный аргумент");**

**} catch (FileNotFoundException e) {**

**e.printStackTrace();**

**System.err.println("Файл не найден");**

**} catch (IOException e) {**

**e.printStackTrace();**

**System.err.println("Ошибка доступа");**

**}**

**}**

**public static void main(String[] args) {**

**System.out.println("Создание jar-архива");**

**// массив файлов для сжатия**

**String[] filesToJar = new String[2];**

**filesToJar[0] = "RandFil.java";**

**filesToJar[1] = "RandFil.class";**

**byte[] buffer = new byte[1024];**

**// имя полученного архива**

**String jarFileName = "example.jar";**

**pack( filesToJar, jarFileName, buffer );**

**}**

**}**

Вывод программы:

**Создание jar-архива**

**0**

**1**

**АРХИВАЦИЯ – JarFile**

Класс JarFile обеспечивает доступ к записям, хранящимся в jar- файле. Доступ к данным осуществляется быстрее, чем при считывании каждой отдельной записи. Единственным недостатком является то, что доступ может осуществляться только для чтения.

Метод entries() извлекает все записи из jar-файла. Этот метод возвращает список экземпляров JarEntry – по одной для каждой записи в jar- файле.

Метод getEntry(String name) извлекает запись по имени.

Метод getInputStream() создает поток ввода для записи. Этот метод возвращает поток ввода, который может использоваться для чтения данных записи.

Класс JarInputStream читает данные в jar-формате из потока ввода. Он переопределяет метод read() таким образом, чтобы любые данные, считываемые из потока, предварительно распаковывались.

**АРХИВАЦИЯ – чтение из jar**

Пример:

**/\* чтение jar-архива: UPackJar.java \*/**

**import java.io.\*;**

**import java.util.\*;**

**import java.util.jar.\*;**

**public class UPackJar {**

**private File destFile;**

**// размер буфера для распаковки**

**public final int BUFFER = 2048;**

**public void unpack( String destinationDirectory,**

**String nameJar) {**

**File sourceJarFile = new File(nameJar);**

**try {**

**File unzipDestinationDirectory =**

**new File(destinationDirectory);**

**// открытие zip-архива для чтения**

**JarFile jFile = new JarFile(sourceJarFile);**

**Enumeration jarFileEntries = jFile.entries();**

**while ( jarFileEntries.hasMoreElements() ) {**

**// извлечение текущей записи из архива**

**JarEntry entry =**

**(JarEntry) jarFileEntries.nextElement();**

**String entryname = entry.getName();**

**System.out.println("Extracting: " + entry);**

**destFile =**

**new File( unzipDestinationDirectory, entryname);**

**// определение каталога**

**File destinationParent = destFile.getParentFile();**

**// создание структуры каталогов**

**destinationParent.mkdirs();**

**// распаковывание записи, если она не каталог**

**if (!entry.isDirectory()) {**

**writeFile(jFile, entry);**

**}**

**}**

**jFile.close();**

**} catch (IOException ioe) {**

**ioe.printStackTrace();**

**}**

**}**

**private void writeFile(JarFile jFile, JarEntry entry)**

**throws IOException {**

**BufferedInputStream is =**

**new BufferedInputStream(**

**jFile.getInputStream(entry));**

**int currentByte;**

**byte data[] = new byte[BUFFER];**

**// запись файла на диск**

**BufferedOutputStream dest =**

**new BufferedOutputStream(**

**new FileOutputStream(destFile), BUFFER);**

**while ((currentByte = is.read(data, 0, BUFFER)) > 0){**

**dest.write(data, 0, currentByte);**

**}**

**dest.flush();**

**dest.close();**

**is.close();**

**}**

**public static void main(String[] args) {**

**System.out.println(**

**"Извлечение данных из jar-архива");**

**// расположение и имя архива**

**String nameJar = "example.jar";**

**// куда файлы будут распакованы**

**String destination = "c:\\temp\\";**

**new UPackJar().unpack(destination, nameJar);**

**}**

**}**

**ЛЕКЦИЯ 5**

**ОБОБЩЕНИЯ**

**ОБОБЩЕНИЯ**

Обобщенное программирование (generic programming) – описание данных и алгоритмов в программе, которое можно применить к различным типам данных, не меняя при этом само описание.

Для такого описания используются специальные синтаксические конструкции, называемые шаблонами (дженериками).

**ШАБЛОНЫ (generics)**

Шаблоны – описание класса, метода, атрибута без использования конкретного типа данных.

Пример без шаблонов:

**//Объявление:**

**List l = new LinkedList();**

**l.add(new Integer(0));**

**//потенциальна ошибка (\*)**

**l.add(new Double(1.1));**

**//Использование:**

**for(…) {**

**Integer x = (Integer)**

**l.iterator().next();**

**}**

Пример с шаблонами:

**//Объявление:**

**List<Integer> l =**

**new LinkedList<Integer>();**

**l.add(new Integer(0));**

**//ошибка компиляции**

**l.add(new Double(1.1);**

**//Использование:**

**for (…) {**

**Integer x**

**=l.iterator().next();**

**}**

Шаблоны повышают наглядность кода и снижают количество явных преобразований типа и возможных ошибок от неявных преобразований.

В примере с шаблонами отсутствует явное приведение к типу Integer. Это исключает появление ошибки ClassCastException в момент работы программы (\* - при ошибочном добавлении в List элемента Double), а также упрощает визуальное восприятие доступа к элементам и делает проще замену типа данных Integer на, например, Double.

**Ограничения:**

* Синтаксические конструкции с использованием шаблонов запрещены в перечислениях, исключительных ситуациях и анонимных встроенных классах.
* Примитивный тип нельзя подставить вместо типа параметра.
* Нельзя объявить массив параметризованных типов.
* Основное применение шаблонов — коллекции.
* При компиляции программы происходит уничтожение информации о шаблонах (type erasure) и приведение всех обобщенных и параметризованных типов, формальных параметров типа к использованию только базового типа.

Пример:

**System.out.println("ArrayList<String> это "**

**+new ArrayList<String>().getClass());**

**System.out.println("ArrayList<Double> это "**

**+new ArrayList<Double>().getClass());**

**Выдаст:**

**ArrayList<String> это class java.util.ArrayList**

**ArrayList<Double> это class java.util.ArrayList**

**ТИПЫ С ШАБЛОНАМИ**

Обобщенный тип – это описание класса с использованием формальных параметров типа. Параметризованный тип - реализация обобщенного типа с использованием конкретного типа данных в качестве аргумента.

Описание класса без шаблонов:

**class Game {**

**int result;**

**int getResult();**

**void setResult(int result);**

**}**

Описание с использованием шаблонов:

**class Game<T> {**

**T result;**

**T getResult();**

**void setResult(T result);**

**}**

Game<T> - обобщенный тип

**Т** – формальный параметр типа

**Game<String> g = new Game<String>()** - параметризованный тип для представления результатов игры, например, в футбол ("2:0"), а **Game<Integer> g = new Game<Integer>()** в тетрис.

**ОПИСАНИЕ МЕТОДОВ С ШАБЛОНАМИ**

Метод с использованием формального описания типа называется шаблонным методом. Признаком того, что метод является шаблонным служит указание типа данных с которым работает метод. В нашем примере это символ <T>.

Объявление:

**public static <T> Set<T> emptySet() {**

**return new HashSet<T>();**

**}**

**Вызов:**

**// конкретный тип для подстановки**

**// выбирает компилятор по**

**// аргументам вызова метода или оператору присвоения**

**Set<String> x = Collections.emptySet();**

**// указан явно**

**Set<Integer> y = Collections.<Integer>emptySet(); //**

**ФОРМАЛЬНЫЕ ПАРАМЕТРЫ ТИПА (type parameter)**

Формальный параметр типа — это параметр, вместо которого при создании по шаблону параметризованного типа необходимо подставить конкретный тип данных.

**interface Comparable<E> {**

**int compareTo(E other);**

**}**

в приведенном примере E является формальным параметром типа. Формальных параметров может быть несколько — KeyValue<KEY,value>.

Формальные параметры могут быть ограниченными.

Ограниченный формальный параметр позволят задать возможные границы подстановки конкретных типов данных для получения параметризованного типа.

Ограничение могут быть с верхней и с нижней стороны дерева наследования:

Пример:

**<T extends Number>** - ограничивает T сверху — в качестве формального параметра могут быть использованы только потомки класса Number;

**<T super Integer>** - ограничивает T снизу — могут быть использованы только предки класса Integer.

В ограничении могут быть использованы символы '&' и '|' для объединения условий соответствующими логическими операциями.

**WILDCARD**

Wildcard (дословно джокер) или групповая подстановка – синтаксические конструкции с использованием символа '?' (означает любой тип), используемые для замены в шаблонах конкретного класса множеством возможных классов.

Групповая подстановка обычно используется для создания шаблонных классов:

**/\***

**\* Допустимые параметры:**

**\* Square s (Square extends Shape) — можно**

**\* String str — нельзя**

**\*/**

**static void drawAll(Collection<? extends Shape> c) {**

**for(Shape s : c) {**

**s.draw();**

**}**

**}**

**КОЛЛЕКЦИИ**

**КОЛЛЕКЦИИ**

Коллекции – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним.

Они представляют собой реализацию абстрактных типов (структур) данных, поддерживающих три основные операции:

* + - добавление нового элемента в коллекцию;
    - удаление элемента из коллекции;
    - изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: просмотреть элементы, подсчитать их количество и др..

Коллекции в языке Java объединены в библиотеке классов java.util и представляют собой контейнеры для хранения и манипулирования объектами.

До появления Java 2 эта библиотека содержала классы только для работы с простейшими структурами данных: Vector, Stack, Hashtable, BitSet, а также интерфейс Enumeration для работы с элементами этих классов.

Коллекции, появившиеся в Java 2, представляют общую технологию хранения и доступа к объектам. Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, следует использовать коллекции из Java 1.

Начиная с версии 5.0 коллекции стали типизированными.

Интерфейсы коллекций:

* Map<K,V> – карта отображения вида “ключ-значение”;
* Collection<E> – вершина иерархии остальных коллекций;
* List<E> – специализирует коллекции для обработки списков;
* Set<E> – специализирует коллекции для обработки множеств, содержащих уникальные элементы.

Все классы коллекций реализуют также интерфейсы Serializable, Cloneable (кроме WeakHashMap). Кроме того, классы, реализующие интерфейсы List<E> и Set<E>, реализуют также интерфейс Iterable<E>.

**КОЛЛЕКЦИИ - Collection<E>**

В интерфейсе Collection<E> определены методы, которые работают на всех коллекциях:

boolean add(E obj) – добавляет obj к вызывающей коллекции и возвращает true, если объект добавлен, и false, если obj уже элемент коллекции;

boolean addAll(Collection<? extends E> c) – добавляет все элементы коллекции к вызывающей коллекции;

* void clear() – удаляет все элементы из коллекции;
* boolean contains(Object obj) – возвращает true, если вызывающая коллекция содержит элемент obj;
* boolean equals(Object obj**)** – возвращает true, если коллекции эквивалентны;
* boolean isEmpty() – возвращает true, если коллекция пуста;
* Iterator<E> iterator() – извлекает итератор;
* boolean remove(Object obj) – удаляет obj из коллекции;
* int size() – возвращает количество элементов в коллекции;
* Object[] toArray() – копирует элементы коллекции в массив объектов;
* <T> T[] toArray(T a[]) – копирует элементы коллекции в массив объектов определенного типа.

**КОЛЛЕКЦИИ**

Для работы с элементами коллекции применяются следующие интерфейсы:

* Comparator<T> – для сравнения объектов;
* Iterator<E>, ListIterator<E>, Map.Entry<K,V> – для перечисления и доступа к объектам коллекции.

Интерфейс Iterator<E> используется для построения объектов, которые обеспечивают доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом iterator()**.** Такой объект позволяет просматривать содержимое коллекции последовательно, элемента за элементом.

**КОЛЛЕКЦИИ - Iterator<E>**

Позиции итератора располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса Iterator<E>:

* boolean hasNext() – проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает false. Итератор при этом остается неизменным;
* E next() – возвращает объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод next() генерирует исключение NoSuchElementException;
* void remove() – удаляет объект, возвращенный последним вызовом метода next().

**ЦИКЛ “for each”**

В Java SE 5.0 был реализован новый цикл, позволяющий перебирать все элементы набора данных не применяя счетчик, набор данных должен быть массивом или реализовывать интерфейс Iterable. Интерфейс Iterable объявлен в java.lang:

**public interface Iterable<T> {**

**Iterator<T> iterator();**

**}**

**Цикл получил название “for each”:**

**for( T element : IterableType ) {**

**// use element in expressions**

**}**

**Пример:**

**import java.util.Iterator;**

**public class DemoForEach {**

**static class Point**

**implements Iterable<Integer>, Iterator<Integer> {**

**protected Integer x, y;**

**public Point() { this(0,0); }**

**public Point ( int x, int y ) { this.x = x; this.y = y; }**

**//Iterable<Integer>:**

**private int idx = 0;**

**public void reset() { idx = 0; }**

**public Iterator<Integer> iterator() {**

**reset(); return this;**

**}**

**//Iterator<Integer>:**

**public boolean hasNext() {**

**return idx >= 0 && idx < 2;**

**}**

**public Integer next() {**

**return ++idx == 1 ? x : (idx == 2 ? y : null);**

**}**

**public void remove() {}**

**}**

**public static <T,E extends Iterable<T>> String toString( E data){**

**String str = "{";**

**Iterator<T> itr = data.iterator();**

**if ( itr.hasNext()) {**

**str += itr.next().toString();**

**while ( itr.hasNext()) {**

**str += ", " + itr.next().toString();**

**}**

**}**

**return str + "}";**

**}**

**public static <T, E extends Iterable<T>> void print( E[] arr){**

**for ( E e : arr ) {**

**System.*out*.println( *toString*( e ) );**

**}}**

**public static void main(String[] args) {**

**Point [] arr = {**

**new Point( 10, 10), new Point(20 ,15),**

**new Point(), new Point(12,34)**

**};**

**DemoForEach.*print*( arr );**

**for ( Point t : arr ) {**

**t.x -= 1; t.y += 1;**

**}**

**DemoForEach.*print*( arr );**

**}**

**}**

Результат:

**{10, 10}**

**{20, 15}**

**{0, 0}**

**{12, 34}**

**{9, 11}**

**{19, 16}**

**{-1, 1}**

**{11, 35}**

**КОЛЛЕКЦИИ - ListIterator<E>**

Интерфейс ListIterator<E> расширяет интерфейс Iterator<E>и предназначен в основном для работы со списками. Наличие методов

* + - E previous()**,** int previousIndex() **и** boolean hasPrevious() обеспечивает обратную навигацию по списку.
    - int nextIndex() возвращает номер следующего итератора.
    - void add(E obj) позволяет вставлять элемент в список текущей позиции. Вызов метода void set(E obj) производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

**КОЛЛЕКЦИИ - Map.Entry**

Интерфейс Map.Entry предназначен для извлечения ключей и значений карты с помощью методов K getKey() и V getValue() соответственно.

Вызов метода V setValue(V value) заменяет значение, ассоциированное с текущим ключом.

**КОЛЛЕКЦИИ - ArrayList<E>**

Класс ArrayList<E> – динамический массив объектных ссылок. Расширяет класс AbstractList<E> и реализует интерфейс List<E>**.**

ArrayList<E> лучше всего подходит для хранения неизменяемых списков.

Класс имеет конструкторы:

* + - ArrayList()
    - ArrayList(Collection <? extends E> c)
    - ArrayList(int capacity)

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов. Методы интерфейса List<E> позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

* void add(int index, E element) – вставляет element в позицию, указанную в index;
* void addAll(int index, Collection<? extends E> c) – вставляет в вызывающий список все элементы коллекции с, начиная с позиции index;
* E get(int index) – возвращает элемент в виде объекта из позиции index;
* E remove(int index) – удаляет объект из позиции index;
* E set(int index, E element) – заменяет объект в позиции index, возвращает при этом удаляемый элемент;
* List<E> subList(int fromIndex, int toIndex) – извлекает часть коллекции в указанных границах.

Пример:

**import java.util.\*;**

**public class DemoGeneric {**

**public static void main(String args[]) {**

**ArrayList<String> list = new ArrayList<String>();**

**// ArrayList<int> b = new ArrayList<int>(); //ошибка**

**list.add("Java");**

**list.add("Fortress");**

**String res = list.get(0);/\* компилятор "знает"тип значения \*/**

**// list.add(new StringBuilder("C#")); // ошибка компиляции**

**// компилятор не позволит добавить "посторонний" тип**

**System.out.print(list);**

**}**

**}**

**КОЛЛЕКЦИИ – СОРТИРОВКА**

Сортировка элементов коллекции в интерфейсе SortedMap и аналогичных производится при помощи естественного порядка сортировки, определяемого в элементе коллекции, либо при помощи интерфейса Comparator.

Естественный порядок сортировки (natural sort order) — естественный и реализованный по умолчанию (реализацией метода compareTo интерфейса java.lang.Comparable) способ сравнения двух экземпляров одного класса.

* **int compareTo(E other)** 
  + - - сравнивает this объект с other и возвращает: -1 если this < other, 0 — если они равны и 1 если this > other.

**КЛАСС Collections**

Collections — класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями.

* sort(List) - сортировать список
* binarySearch(List, Object) - бинарный поиск элементов в списке
* reverse(List) - изменить порядок элементов в списке на противоположный
* fill(List, Object) - заменить каждый элемент заданным
* copy(List dest, List src)- скопировать список src в dest

Методы:

Collections содержит статические методы для манипулирования коллекциями.

Для работы с коллекциями, разработанными в предыдущих версиях языка разработаны ряд методов с проверкой объектов коллекции на этапе выполнения.

* public static <E> Collection<E>
  + - checkedCollection( Collection<E> c, Class<E> type );
      * - создает коллекцию, проверяемую на этапе выполнения, в случае добавления “постороннего” объекта генерируется исключение ClassCastException

Пример использования:

**Collection c = Collections.checkedCollection(**

**new HashSet<String>(), String.class);**

**c.add(“Text string");**

**c.add(12345.67); // ошибка времени выполнения**

Аналогичные методы для создания специализированных для проверки конкретных типов коллекций:

* **checkedList(), checkedSortedMap(), checkedMap(), checkedSortedSet(), checkedSet()**
* **<T> boolean addAll( Collection<? super T> c, T...a );**
  + - - добавляет в параметризованную коллекцию соответствующие элементы
* **<T> void copy( List<? super T> dest, List<? extends T> src);**
  + - - копирует все элементы из одного списка в другой
* **boolean disjoint(Collection<?> c1, Collection<?> c2);**
  + - - возвращает true, если коллекции не содержат одинаковых элементов
* **<T> List<T> emptyList();**
* **<K,V> Map<K,V> emptyMap();**
* **<T> Set<T> emptySet();** 
  + - - возвращают пустой список, карту отображения и множество
* **<T> void fill(List<? super T> list, T obj);**
  + заполняет список заданным элементом
* **int frequency(Collection<?> c, Object o);**
  + - - возвращает количество вхождений в коллекцию заданного элемента
* **<T extends Object & Comparable <? super T>>**
* **T max( Collection<? extends T> coll );**
* **<T extends Object & Comparable <? super T>>**
* **T min( Collection<? extends T> coll );**
  + - - возвращают минимальный и максимальный элемент
* **<T> T max( Collection <? extends T> coll, Comparator<? super T> comp);**
* **<T> T min( Collection<? extends T> coll, Comparator<? super T> comp);**
  + - - возвращают минимальный и максимальный элемент соответственно, используя Comparator для сравнения
* **<T> List <T> nCopies(int n, T o);**
  + - - возвращает список из n заданных элементов
* **<T> boolean replaceAll( List<T> list, T oldVal, T newVal);**
  + - - заменяет все заданные элементы новыми
* **void reverse( List<?> list );**
  + “переворачивает” список
* **void rotate( List<?> list, int distance );**
  + - - циклически сдвигает список на заданное число элементов
* **void shuffle( List<?> list );**
  + - - перетасовывает элементы списка
* **<T> Set<T> singleton(T o);**
* **<T> List<T> singletonList(T o);**
* **<K,V> Map<K,V> singletonMap(K key, V value);**
  + - - создают множество, список и карту отображения, состоящие из одного элемента
* **<T extends Comparable<? super T>>** 
  + - **void sort( List<T> list );**
* **<T> void sort( List<T> list, Comparator<? super T> c );**
  + - - сортировка списка, естественным порядком и используя Comparator
* **void swap( List<?> list, int i, int j );**
  + - - меняет местами элементы списка стоящие на заданных позициях

Пример:

**import java.util.ArrayList;**

**import java.util.Collections;**

**import java.util.List;**

**import java.util.Comparator;**

**public class ColDemo {**

**public static class Compr<T>**

**implements Comparator<Integer> {**

**public int compare(Integer n, Integer m) {**

**return m.intValue() - n.intValue();**

**}**

**}**

**static void print(List<Integer> c) {**

**for ( int i : c )**

**System.out.print(i + " ");**

**System.out.println();**

**}**

**public static void main(String[] args) {**

**Compr<Integer> comp = new Compr<Integer>();**

**ArrayList<Integer> list =**

**new ArrayList<Integer>();**

**Collections.addAll(list, 1, 2, 3, 4, 5);**

**Collections.shuffle(list);**

**print(list);**

**Collections.sort(list, comp);**

**print(list);**

**Collections.reverse(list);**

**print(list);**

**Collections.rotate(list, 3);**

**print(list);**

**System.out.println("min: " +**

**Collections.min(list, comp));**

**System.out.println("max: " +**

**Collections.max(list, comp));**

**List<Integer>**

**singl = Collections.singletonList(71);**

**print(singl);**

**//singl.add(21);//ошибка времени выполнения**

**}}**

Результат:

**3 4 1 2 5**

**5 4 3 2 1**

**1 2 3 4 5**

**3 4 5 1 2**

**min: 5**

**max: 1**

**71**

**КЛАСС Arrays - методы**

java.util.Arrays содержит методы манипулирования содержимым массива.

* **int binarySearch( параметры );**

- перегруженный метод организации бинарного поиска значения в массивах примитивных и объектных типов; возвращает позицию первого совпадения

* **void fill( параметры );**

- перегруженный метод для заполнения массивов значениями различных типов и примитивами

* **void sort( параметры );**

- перегруженный метод сортировки массива или его части с использованием интерфейса Comparator и без него

* **static <T> T[] copyOf( T[] original, int newLength );**

- заполняет массив определенной длины, отбрасывая элементы или заполняя null при необходимости

* **static <T>**
* **T[] copyOfRange( T[] original, int from, int to );**

- копирует заданную область массива в новый массив

* **<T> List<T> asList( T...a );**

- метод, копирующий элементы массива в объект типа List<T>

Пример:

**import java.util.\*;**

**public class ArrDemo {**

**public static class Compr<T> implements Comparator<Integer> {**

**public int compare(Integer n, Integer m) {**

**return m.intValue() - n.intValue();**

**}**

**}**

**static void print( Integer[] arr) {**

**for ( int i : arr ) System.out.print(i + " ");**

**System.out.println();**

**}**

**public static void main(String[] args) {**

**Compr<Integer> comp = new Compr<Integer>();**

**Integer[] arr = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8 };**

**Arrays.sort( arr );**

**print( arr );**

**Arrays.sort( arr, comp );**

**print( arr );**

**}**

**}**

Результат:

**0 1 2 3 4 5 6 7 8 9**

**9 8 7 6 5 4 3 2 1 0**

**ИЕРАРХИЯ ИНТЕРФЕЙСОВ**



**ИНТЕРФЕЙС List**

Интерфейс java.util.List расширяет интерфейс Collection, описывает методы работы с упорядоченными коллекциями. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

Элементы коллекции List могут повторяться.







При выделении подсписка, необходимо помнить, что изменения вносимые в подсписок операциями add, remove, set отображаются также и в исходном списке:



Интерфейс ListIterator позволяет просматривать список в обратном направлении:



**КЛАСС LinkedList<E>**

Коллекция LinkedList<E> реализует связанный список, т.е. хранит объекты вместе со ссылками на следующее и предыдущее элементы списка.

В дополнение к методам интерфейса List, в LinkedList<E> реализованы методы для работы с первым и последним элементами списка:

**void addFirst(E ob), void addLast(E ob), E getFirst(),**

**E getLast(), E removeFirst(), E removeLast()**

Класс LinkedList<E> реализует интерфейс Queue<E>, что позволяет такому списку легко придать свойства очереди.

**ИНТЕРФЕЙС Queue<E>**

Расширяет интерфейс Collection, позволяет реализовать FIFO очередь (first-in-first-out).

Методы:

* E element();

- возвращает, но не удаляет головной элемент очереди

* boolean offer(E o);

- вставляет элемент в очередь, если возможно

* E peek();

- возвращает, но не удаляет головной элемент очереди, возвращает null, если очередь пуста;

* E poll();

- возвращает и удаляет головной элемент очереди, возвращает null, если очередь пуста

* E remove();

- возвращает и удаляет головной элемент очереди

Методы element() и remove() отличаются от методов peek() и poll() тем, что генерируют исключение, если очередь пуста.

**ИНТЕРФЕЙС Deque<E>**

Расширяет интерфейс Queue<E>. Позволяет использовать коллекцию как очередь (FIFO – first-in-first-out) или как стек (LIFO – last-in-first-out).

Методы:



Например:

**import java.util.\*;**

**public class DeqDemo {**

**public static void main(String[] args) {**

**Deque<Character> d = new ArrayDeque<Character>();**

**int i; String str = "Test string";**

**System.out.println( "Use as queue:" );**

**for ( i = 0; i < str.length(); i++ )**

**d.offer( str.charAt( i ));**

**for ( i = 0; i < str.length(); i++ )**

**System.out.print( d.poll() );**

**System.out.println();**

**System.out.println( "Use as stack:" );**

**for ( i = 0; i < str.length(); i++ )**

**d.push( str.charAt( i ));**

**for ( i = 0; i < str.length(); i++ )**

**System.out.print( d.pop() );**

**System.out.println();**

**}**

**}**

Результат:

**Use as queue:**

**Test string**

**Use as stack:**

**gnirts tseT**

**МНОЖЕСТВА**

Интерфейс Set<E> реализует интерфейс Collection<E> не добавляя никаких новых методов. Может содержать только уникальные объекты – дублирование не допускается.

Интерфейс SortedSet<E> наследует Set<E> и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса.

Интерфейс NavigableSet существенно облегчает поиск элементов.

**КЛАСС HashSet<E>**

Класс HashSet<E> наследуется от абстрактного суперкласса AbstractSet<E> и реализует интерфейс Set<E>, используя хэш-таблицу для хранения коллекции. Хэш-код используется вместо индекса что значительно ускоряет поиск.

Конструкторы класса:

* HashSet();
* HashSet(Collection <? extends E> c);
* HashSet(int capacity);
* HashSet(int capacity, float loadFactor);

где capacity – число ячеек для хранения хэш-кодов

Класс не гарантирует порядок элементов.

**КЛАСС TreeSet<E>**

Класс TreeSet<E> для хранения объектов использует бинарное дерево. При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что все добавляемые элементы реализуют интерфейсы Comparator и Comparable. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса:

* TreeSet();
* TreeSet(Collection <? extends E> c);
* TreeSet(Comparator <? super E> c);
* TreeSet(SortedSet <E> s);

Класс TreeSet<E> содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов E first() и E last()**.**

Методы

* SortedSet<E> subSet(E from, E to),
* SortedSet<E> tailSet(E from)
* и SortedSet<E> headSet(E to)
  + - * предназначены для извлечения определенной части множества.

**ИНТЕРФЕЙС Map**

Базовый интерфейс для всех структур данных основанных на паре ключ-значение.

Содержит три группы операций:

* редактирования (altering),
* проверки (query)
* представления (view).

Операции редактирования (altering) добавляют или удаляют пары ключ-значение:



Операции проверки (Query) – проверяют содержимое:



Операции представления (view) позволяют работать с ключами или значениями как с коллекцией объектов:



**КЛАССЫ РЕАЛИЗУЮЩИЕ Map**

Класс HashMap полностью реализует интерфейс Map, а также итератор типа Iterator.

Класс WeakHashMap отличается от класса HashMap только тем, что в его объектах неиспользуемые элементы, на которые никто не ссылается, автоматически исключаются из объекта.

Класс LinkedHashMap полностью реализует интерфейс Map. Реализация сделана в виде двунаправленного списка, а значит, его элементы хранятся в упорядоченном виде. Порядок элементов задается порядком их занесения в список.

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу enum, который должен быть определен при создании коллекции. Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

Пример использования:

**import java.util.\*;**

**enum User {**

**STUDENT, TUTOR, INSTRUCTOR, DEAN**

**}**

**class UserPriority {**

**private int priority;**

**public UserPriority(User k) {**

**switch (k) {**

**case STUDENT:**

**priority = 1; break;**

**case TUTOR:**

**priority = 3; break;**

**case INSTRUCTOR:**

**priority = 7; break;**

**case DEAN:**

**priority = 10; break;**

**default:**

**priority = 0;**

**}**

**}**

**public int getPriority() {**

**return priority;**

**}**

**}**

**public class EnmDemo {**

**public static void main(String[] args) {**

**EnumMap<User, UserPriority> faculty =**

**new EnumMap<User, UserPriority> (User.class);**

**for (User user : User.values()) {**

**faculty.put(user, new UserPriority(user));**

**}**

**for (User user : User.values()) {**

**System.out.println(user.name() +**

**"-> Priority:" + ((UserPriority)**

**faculty.get(user)).getPriority());**

**}**

**}**

**}**

Результат:

**STUDENT-> Priority:1**

**TUTOR-> Priority:3**

**INSTRUCTOR-> Priority:7**

**DEAN-> Priority:10**

**ИНТЕРФЕЙС SortedMap**

Интерфейс SortedMap, расширяющий интерфейс Map, описывает упорядоченную по ключам коллекцию Map.

Сортировка производится либо в естественном порядке возрастания ключей, либо в порядке, описываемом в интерфейсе Comparator.

Элементы не нумеруются, но есть понятия большего и меньшего из двух элементов, первого, самого маленького, и последнего, самого большого элемента коллекции. Эти понятия описываются следующими методами:

* Comparator comparator()
  + - * + — способ упорядочения коллекции;
* Object firstKey()
  + - * + — первый, меньший элемент коллекции;
* SortedMap headMap(Object toKey)
  + - * + — начало коллекции до элемента с ключом toKey исключительно;
* Object lastKey()
  + - * + — последний, больший ключ коллекции;
* SortedMap subMap(Object fromKey, Object toKey)
  + - * + — часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно;
* SortedMap tailMap(Object fromKey)
  + - * + — остаток коллекции, начинающийся от элемента fromKey включительно.

**КЛАССЫ РЕАЛИЗУЮЩИЕ SortedMap**

Класс TreeMap полностью реализует интерфейс SortedMap. Класс реализован как бинарное дерево поиска, что значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

Пример использования TreeMap



**ЛЕКЦИЯ 6**

**ПОТОКИ ВЫПОЛНЕНИЯ**

**ОСНОВНЫЕ ПОНЯТИЯ**

**Процесс** – это экземпляр выполняемой программы, состоит из двух компонентов:

* объект ядра со статистической информацией о приложении, через который операционная система им управляет
* адресное пространство, в котором содержится код и данные всех модулей приложения

Чтобы процесс что-нибудь выполнил, в нем нужно создать **поток выполнения**. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. В каждом процессе есть минимум один поток.

При создании процесса система автоматически создает его первый поток, называемый **главным потоком**.

Чтобы потоки работали, операционная система отводит каждому из них определенное процессорное время. Выделяя потокам отрезки времени (называемые **квантами**) по принципу карусели, она создает тем самым иллюзию одновременного выполнения потоков.

Если в машине установлено более одного процессора, операционная система стремится сбалансировать нагрузку между процессорами.

Ядро ОС управляет планированием исполнения потоков.

**Поток** (выполнения) состоит из двух компонентов:

* объект ядра, через который операционная система управляет потоком (там же хранится статистическая информация о потоке)
* стек потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Потоки исполняют код и манипулируют данными в адресном пространстве процесса, они могут исполнять один и тот же код и работать с одними и теми же данными.

**ПОТОЧНАЯ МОДЕЛЬ Java**

При однопоточном программировании в бесконечном цикле выполняется один поток управления, который опрашивает единую очередь событий и принимает решение, какое действие выполнять следующим.

При многопоточном программировании один поток может делать паузу, не прерывая выполнение других потоков - именно такой подход реализован в Java.

Поточная модель Java реализуется иерархией классов, описывающих потоки. Основу этой иерархии составляют класс Thread и интерфейс Runnable.

Для создания потока необходимо либо расширить класс Thread, либо реализовать интерфейс Runnable. При этом класс Thread инкапсулирует поток исполнения.

При запуске Java-программы начинает выполняться главный поток. Главным потоком можно управлять с помощью методов класса Thread. Ссылку на поток получают с помощью статического метода Thread.currentThread().

**ЖИЗНЕННЫЙ ЦИКЛ ПОТОКА**

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State:

* + - * + NEW – поток создан, но еще не запущен;
        + RUNNABLE – поток выполняется;
        + BLOCKED – поток блокирован;
        + WAITING – поток ждет окончания работы другого потока;
        + TIMED\_WAITING – поток некоторое время ждет

окончания другого потока;

* + - * + TERMINATED — поток завершен.

Получить значение состояния потока можно вызовом метода getState().

При создании потока он получает состояние “новый” (NEW) и не выполняется. Для перевода потока из состояния “новый” в состояние “работоспособный” (RUNNABLE) следует выполнить метод start(), который вызывает метод run() – основной метод потока.

Поток переходит в состояние “неработоспособный” (WAITING) вызовом метода wait() или методов ввода/вывода, которые предполагают задержку.

Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания (TIMED\_WAITING) с помощью методов sleep(long millis) и wait(long timeout), при выполнении которого может генерироваться прерывание InterruptedException.

Вернуть потоку работоспособность после вызова метода wait() можно методами notify() или notifyAll().

Поток переходит в “пассивное” состояние (TERMINATED), если вызван метод interrupt() или метод run() завершил выполнение. После этого, чтобы запустить поток еще раз, необходимо создать новый объект потока. Метод interrupt() успешно завершает поток, если он находится в состоянии “работоспособность”. Если же поток неработоспособен, то метод генерирует исключительные ситуации разного типа в зависимости от способа остановки потока.

**СОЗДАНИЕ ПОТОКА - РЕАЛИЗАЦИЕЙ ИНТЕРФЕЙСА Runnable**

Создать поток можно на базе любого класса, который реализует интерфейс Runnable. При реализации интерфейса Runnable достаточно определить всего один метод run(). Программный код этого метода — это тот код, который выполняется в рамках создаваемого потока. Говорят, что метод run() определяет точку входа в поток. Метод run() имеет следующую сигнатуру:

**public void run();**

Для начала выполнения потока вызывают метод start().

Общая последовательность действий при создании нового потока путем реализации интерфейса Runnable следующая.

* Определяется класс, реализующий интерфейс Runnable. В этом классе определяется метод run().
* В этом классе создается объект класса Thread. Конструктору класса передается два аргумента: объект класса, реализующего интерфейс Runnable, и текстовая строка — название потока.
* Для запуска потока из объекта класса Thread вызывается метод start().

Один экземпляр Runnable можно передать нескольким объектам Thread:

**MyRunnable r = new MyRunnable();**

**Thread foo = new Thread(r);**

**Thread bar = new Thread(r);**

**Thread bat = new Thread(r);**

Это означает что несколько потоков будут делать одну и ту же работу.

**Создание потока - реализацией интерфейса Runnable**

**Пример:**

**class NewThread implements Runnable {**

**Thread t;**

**NewThread() {**

**t = new Thread( this,"New thread" );**

**System.out.println( "Child thread: " + t );**

**t.start();**

**}**

**public void run() {**

**try {**

**for( int i = 5; i > 0; i-- ) {**

**System.out.println("Child thread: " + i );**

**Thread.sleep(500);**

**}**

**}**

**catch( InterruptedException e ) {**

**System.out.println(**

**"Child thread interruption!");**

**}**

**System.out.println("Child thread termination!");**

**}**

**}**

**public class RunableDemo{**

**public static void main(String args[]) {**

**new NewThread();**

**try {**

**for ( int i = 5; i > 0; i-- ) {**

**System.out.println("Main thread: " + i \* 100 );**

**Thread.sleep(1000);**

**}**

**}**

**catch( InterruptedException e ) {**

**System.out.println("Main thread interruption!");**

**}**

**System.out.println("Main thread termination!");**

**}**

**}**

**Создание потока - расширением класса Thread**

**Пример:**

**class NewThread extends Thread {**

**NewThread() {**

**super( "New thread" );**

**System.out.println( "Child thread: "**

**+ this );**

**start();**

**}**

**public void run() {**

**try {**

**for( int i = 5; i > 0; i-- ) {**

**System.out.println("Child thread: " + i );**

**Thread.sleep(500);**

**}**

**}**

**catch( InterruptedException e ) {**

**System.out.println(**

**"Child thread interruption!");**

**}**

**System.out.println("Child thread termination!");**

**}**

**}**

**public class ThreadDemo{**

**public static void main(String args[]) {**

**new NewThread();**

**try {**

**for ( int i = 5; i > 0; i-- ) {**

**System.out.println("Main thread: " + i \* 100 );**

**Thread.sleep(1000);**

**}**

**}**

**catch( InterruptedException e ) {**

**System.out.println("Main thread interruption!");**

**}**

**System.out.println("Main thread termination!");**

**}**

**}**

Результат:

**Child thread: Thread[New thread,5,main]**

**Main thread: 500**

**Child thread: 5**

**Child thread: 4**

**Child thread: 3**

**Main thread: 400**

**Child thread: 2**

**Child thread: 1**

**Main thread: 300**

**Child thread termination!**

**Main thread: 200**

**Main thread: 100**

**Main thread termination!**

**УПРАВЛЕНИЕ ПРИОРИТЕТАМИ И ГРУППЫ ПОТОКОВ**

Потоку можно назначить приоритет от 1 (константа MIN\_PRIORITY) до 10 (MAX\_PRIORITY) с помощью метода setPriority(int prior). Получить значение приоритета можно с помощью метода getPriority().

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

**ThreadGroup tg = new ThreadGroup(**

**"Группа потоков 1");**

**Thread t0 = new Thread(tg, "поток 0");**

Все потоки, объединенные группой, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод getThreadGroup().

Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения в группу значение его приоритета станет равным приоритету группы.

Поток со значением приоритета более низким, чем приоритет группы после включения в группу, значения своего приоритета не изменит.

Поток с более высоким приоритетом, как правило, монополизирует вывод на консоль.

Пример:

**class MyThread extends Thread {**

**MyThread( String name ) {**

**super(name);**

**}**

**public void run() {**

**for ( int i = 0; i < 20; i++) {**

**System.out.print( getName() + " " + i + "; ");**

**try {**

**sleep(1); //test with sleep(0)**

**} catch ( InterruptedException e ) {**

**System.err.print("Error" + e);**

**}**

**}**

**System.out.println( getName() + " finished." );**

**}**

**}**

**public class PriorThread {**

**public static void main( String[] args) {**

**Thread min = new MyThread("Min");//1**

**Thread max = new MyThread("Max");//10**

**Thread norm = new MyThread("Norm");//5**

**min.setPriority(Thread.MIN\_PRIORITY);**

**max.setPriority(Thread.MAX\_PRIORITY);**

**norm.setPriority(Thread.NORM\_PRIORITY);**

**min.start();**

**max.start();**

**norm.start();**

**}**

**}**

**УПРАВЛЕНИЕ ПОТОКАМИ**

Приостановить (задержать) выполнение потока можно с помощью метода sleep(время задержки) класса Thread. Альтернативный способ состоит в вызове метода yield(), который сообщает планировщику выделить квант времени потоку ожидающему в очереди.

Метод join() блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока.

Прервать работу выполняемого потока можно с помощью метода interrupt(). Данный метод отправляет запрос на прекращение работы потока.

В момент вызова метода interrupt(), устанавливается статус прерывания для потока. Это флаг типа Boolean, который обязательно присутствует в любом потоке.

Можно проверить, является ли поток прерванным с помощью методов isInterrupted() и interrupted(). Метод interrupted() является статическим и может применяться только к текущему потоку. Так же метод interrupted(), в отличие от isInterrupted(), сбрасывает статус прерывания потока.

В случае, когда для заблокированного потока (методами sleep() или wait()) вызывается метод interrupt(), работа потока прерывается и управление передается обработчику исключения InterruptedException().

Чтобы определить состояние потока используется метод isAlive().

**Thread ct = Thread.currentThread();**

**ct.isAlive();**

Если поток запушен или заблокирован, то возвращается значение true, а если поток является созданным (еще не запущенным) или остановленным, то возвращается значение false.

**ПОТОКИ-ДЕМОНЫ**

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон.

С помощью метода **setDaemon(boolean value),** вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод boolean **isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода main(), не обращая внимания на то, что поток-демон еще работает.

Необходимо также помнить, что потоки-демоны завершают свои методы run() без выполнения секции finally.

**СИНХРОНИЗАЦИЯ ПОТОКОВ**

Очень часто возникает ситуация, когда несколько потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу; более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток ввода-вывода.

Проблема, когда несколько потоков могут получить доступ к одному ресурсу и изменить его данные до того как операция будет завершена другим потоком, известна как “состязание потоков”.

Существует два способа создания синхронизированного кода:

создание синхронизированных методов;

создание синхронизированных блоков.

В обоих случаях используется ключевое слово synchronized. Если создается синхронизированный метод, ключевое слово synchronized указывается в его сигнатуре.

При вызове синхронизированного метода потоком другие потоки на этом методе блокируются — они не смогут его вызвать, пока работу с методом не завершит первый вызвавший его поток.

Можно синхронизировать объект в блоке команд. Для этого блок выделяется фигурными скобками, перед которыми указывается ключевое слово synchronized, а в скобках после этого слова - синхронизируемый объект.

Синхронизация происходит за счет блокировок. Каждый объект в Java имеет встроенную блокировку, которая работает только в том случае, если объект имеет синхронизированные методы.

Когда мы входим в синхронизированный нестатический метод, мы автоматически получаем блокировку, связанную с текущим экземпляром класса (текущий – это тот, код которого выполняется в данный момент).

Поскольку существует только одна блокировка на объект, то если один поток захватил блокировку, ни один другой поток не может ее захватить, пока первый не отпустит ее. Это означает, что ни какой другой поток не может войти в синхронизированный код пока блокировка не будет освобождена.

Ключевые моменты блокировки и синхронизации:

1. Только методы (или блоки) могут быть synchronized.
2. Каждый объект имеет только одну блокировку.
3. Не все методы в классе нуждаются в синхронизации. Класс может иметь как синхронизированные методы, так и не синхронизированные.
4. Можно синхронизировать не весь метод, а только блок кода.
5. Если два потока собираются выполнить синхронизированный метод в классе, и оба потока используют один и тот же экземпляр класса для вызова метода, то только один поток сможет выполнить метод. Другой поток будет ждать пока первый не закончит выполнение метода. Иными словами, как только поток получает блокировку на объект, ни один другой поток не может войти ни в один из синхронизированных методов класса (для этого объекта).
6. Если поток уходит спать, он удерживает все свои блокировки, а не освобождает их.
7. Если класс имеет и синхронизированные и не синхронизированные методы, то несколько потоков могут получить доступ к не синхронизированным методам класса. Если ваш метод не имеет доступа к данным, которые вы хотите защитить, то вам не нужно объявлять этот метод как синхронизированный. В некоторых случаях синхронизация может привести к взаимной блокировке, поэтому не стоит ей злоупотреблять.
8. Поток может захватить несколько блокировок. Например, поток может войти в синхронизированный метод, таким образом захватив блокировку, а затем сразу же вызвать синхронизированный метод другого объекта, получив также и его блокировку.

Поскольку синхронизация “вредит” параллелизму, необходимо синхронизировать наименьшее количество кода, которого достаточно для защиты данных. Уменьшить масштаб синхронизации можно использовать синхронизированный блок:

**class SyncTest {**

**public void doStuff() {**

**System.out.println("не синхронизированный");**

**synchronized (this) {**

**System.out.println("синхронизированный");**

**}**

**}**

**}**

Статические методы также могут быть синхронизированы. Существует только одна копия статических данных, которую нужно защитить, поэтому необходима только одна блокировка на класс для синхронизации статических методов.

Что используется в качестве блокировки при синхронизации статического метода?

- Каждый загруженный класс в Java имеет соответствующий экземпляр класса java.lang.Class, представляющий этот класс. Именно этот объект и используется для защиты статических синхронизированных методов класса.

**static int count;**

**public static synchronized int getCount() {**

**return count;**

**}**

Этот метод также можно переписать с помощью синхронизированного блока. Предположим, что метод getCount() находится в классе MyClass, тогда:

**public static int getCount() {**

**synchronized (MyClass.class) {**

**return count;**

**}**

**}**

или так:

**public static void classMethod() {**

**Class cl = Class.forName("MyClass");**

**synchronized (cl) {**

**// какой-то код**

**}**

**}**

Если поток пытается войти в синхронизированный метод, а блокировка уже захвачена, то, по сути, поток переходит в некий пул для конкретного объекта и ожидает там пока блокировка не снимется.

Когда блокировка освободится, этот поток снова перейдет в состояние runnable/running. То, что блокировка освобождена еще не означает, что ее получит какой-то конкретный поток. Одну блокировку могут ожидать, например, три потока и нет ни какой гарантии, что поток, который ждал дольше всех, получит блокировку первым.

Важно обращать внимание на то, какие объекты используются для блокировки:

1. потоки, вызывающие нестатические синхронизированные методы в одном классе будут блокировать другие потоки если они вызываются с помощью того же экземпляра класса.
2. Потоки, вызывающие статические синхронизированные методы в одном классе будут всегда блокировать друг друга если все они используют один и тот же экземпляр Class.
3. Статические синхронизированные методы и нестатические синхронизированные методы не будет блокировать друг друга, никогда. Статические методы блокируются на экземпляре класса Class в то время как нестатические методы блокируются на текущем экземпляре (this). Эти действия не мешают друг другу.
4. При использовании синхронизированных блоков, необходимо обращать внимание на то, какой объект используется для блокирования. Потоки, которые синхронизируются на том же объекте будут блокировать друг друга. Потоки, которые синхронизируются на разных объектах, не будут блокировать друг друга.

**ПОТОКО-БЕЗОПАСНЫЕ КЛАССЫ**

Когда класс тщательно синхронизирован, говорят что он является “потоко-безопасным” (“thread-safe”). Многие классы в Java API являются потоко-безопасными.

Например, StringBuffer и StringBuilder практически идентичные классы, однако все методы в StringBuffer синхронизированы, а в StringBuilder – нет, поэтому использование StringBuffer в многопоточной среде безопасно, а StringBuilder – нет.

**ВЗАИМНАЯ БЛОКИРОВКА ПОТОКОВ (Deadlock)**

Самое страшное что может случиться в Java-программе – это deadlock. Deadlock – это ситуация, когда два потока блокируются, при этом каждый ждет освобождения блокировки, которую захватил другой поток. Это означает, что потоки не получат ожидаемых блокировок никогда.

Простейший случай deadlock — это когда поток A получает блокировку ресурса AR1, затем пытается получить доступ к ресурсу AR2, который в это время уже захвачен потоком B, который в свою очередь пытается получить доступ к AR1.

**DEADLOCK**

Пример демонстрирует deadlock:

**public class DeadlockRisk {**

**private static class Resource {**

**public int value;**

**}**

**private Resource resourceA = new Resource();**

**private Resource resourceB = new Resource();**

**public int read() {**

**synchronized (resourceA) { // deadlock может быть здесь**

**synchronized (resourceB) {**

**return resourceB.value + resourceA.value;**

**}**

**}**

**}**

**public void write(int a, int b) {**

**synchronized (resourceB) {**

**// deadlock может быть здесь**

**synchronized (resourceA) {**

**resourceA.value = a;**

**resourceB.value = b;**

**}**

**}**

**}**

**}**

Предположим, что read() запускается первым потоком, а write() вторым. Если есть два разных потока значит чтение и запись могут вызываться независимо друг от друга, а следовательно в прокомментированных строках может возникнуть deadlock. Поток-читатель захватит resourceA, писатель – resourceB и оба остановятся в ожидании друг друга.

Приведенный пример легко исправить просто переставив местами строки synchronized(resourceB) и synchronized(resourceA) в методе write(), однако это всего лишь простой пример. Может возникнуть более сложная ситуация, которая потребует много времени для решения.

**ВЗАИМОДЕЙСТВИЕ ПОТОКОВ**

В классе Object есть три метода wait(), notify(), и notifyAll(), которые позволяют потоку сообщать информацию о своем состоянии другим, заинтересованным в этой информации, потокам.

Используя методы wait() и notify(), мы можем отправить один поток в состояние ожидания до того времени, пока другой поток не вызовет его.

Важный момент при использовании механизма wait/notify заключается в следующем: wait(), notify(), и notifyAll() должны вызываться из синхронизированного (synchronized) контекста, иначе получим исключительную ситуацию!

Методы wait() и notify() являются методами класса Object. Каждый объект имеет блокировку. Таким же образом, каждый объект может иметь список потоков, которые ждут от него сигнала (уведомления).

Поток попадает в список ожидающих при помощи вызова wait() целевого объекта. С этого момента он не выполняет ни каких инструкций пока не будет вызван метод notify(). Если один объект ждут несколько потоков, то после вызова notify(), планировщиком потоков будет выбран один поток (какой именно не известно), который приступит к исполнению. Если ожидающих потоков нет, то ничего и не произойдет.

Пример в котором один объект ждет уведомления от другого:

**public class ThreadA {**

**public static void main(String[] args) {**

**ThreadB b = new ThreadB();**

**b.start();**

**synchronized (b) {**

**try {**

**System.out.println("Ждем пока поток b выполнится...");**

**b.wait();**

**} catch (InterruptedException e) {**

**}**

**System.out.println("Total равно: " + b.total);**

**}**

**}**

**}**

**class ThreadB extends Thread {**

**int total = 0;**

**public void run() {**

**synchronized (this) {**

**for (int i = 0; i < 100; i++) {**

**total += i;**

**}**

**notify();**

**}**

**}**

**}**

Эта программа содержит два потока: ThreadA – основной поток (main) и ThreadB – поток, который вычисляет сумму всех чисел от 0 до 99. После того как выполнится строка b.start(), ThreadA продолжит свое выполнение и может попасть в строку System.out.println(“Total равно: ” + b.total) до того как ThreadB закончит свои расчеты. Чтобы предотвратить такое развитие событий, мы используем метод wait().

Обратите внимание на строку synchronized (b). Это необходимо для того, чтобы ThreadA получил блокировку на b. Чтобы вызвать методы wait() или notify(), поток должен иметь блокировку объекта, на котором вызываются эти методы.

Приведенный выше код уведомляет один поток, ожидающий на этом объекте. Обратите внимание на то, что если вызвать метод wait() из потока не имеющего блокировку на объект на котором вызывается wait(), то возникнет исключение IllegalMonitorStateException.

Это исключение не проверяемое (not a checked exception), поэтому в приведенном выше коде мы не ловим его в явном виде. Однако, блок кода, в котором у нас вызывается wait() обрамлен конструкцией try/catch и мы перехватываем исключение InterruptedException, т.к. ожидающий поток может быть прерван таким же образом, как и спящий (sleeping).

В большинстве случаев необходимо уведомить не один поток, ждущий конкретный объект, а все потоки. Для этого используется метод notifyAll(). notifyAll() выводит все потоки из состояния ожидания и переводит их в состояние runnable.

Это актуально если у вас есть несколько потоков, ожидающих один объект и вы хотите быть уверенным в том, что все потоки получат соответствующие уведомления.

При вызове метода notifyAll() все потоки получат уведомление и начнут конкурировать за право получить блокировку. Как уже было сказано ранее, объект может иметь множество ожидающих его потоков.

Метод notify() воздействует только на один из них. Какой поток получит уведомление – не известно, – это зависит от реализации JVM. Поэтому, когда имеет место ожидание объекта несколькими потоками, лучший способ уведомить все потоки – использовать метод notifyAll().

Рассмотрим пример, в котором в одном потоке выполняются вычисления и множество потоков-читателей ждут результата расчетов.

**public class Reader extends Thread {**

**Calculator c;**

**public Reader(Calculator calc) {**

**c = calc;**

**}**

**public void run() {**

**synchronized (c) {**

**try {**

**System.out.println("Вычисление...");**

**c.wait();**

**} catch (InterruptedException e) {**

**}**

**System.out.println(**

**"Total равно: " + c.total);**

**}**

**}**

**public static void main(String[] args) {**

**Calculator calculator = new Calculator();**

**new Reader(calculator).start();**

**new Reader(calculator).start();**

**new Reader(calculator).start();**

**calculator.start();**

**}**

**}**

**class Calculator extends Thread {**

**int total = 0;**

**public void run() {**

**synchronized (this) {**

**for (int i = 0; i < 100; i++) {**

**total += i;**

**}**

**notifyAll();**

**}**

**}**

**}**

Программа запускает три потока, которые ожидают получения результатов расчета, а затем запускает поток, который производит расчеты. Заметьте, что если в методе run() из класса Calculator вызвать notify() вместо notifyAll(), то будет уведомлён только один поток.

**ЛЕКЦИЯ 7**

**СЕТЕВЫЕ СРЕДСТВА Java**

**СЕТЬ – ОСНОВНЫЕ ПОНЯТИЯ**

**Одноранговая** сеть (peer-to-peer) – сеть основанная на равноправии участников.

**Клиент-сервер** сеть – сеть, в которой сетевая нагрузка распределена между поставщиками услуг (серверы) и заказчиками услуг (клиентами).

Сервер постоянно находится в состоянии ожидания, он **прослушивает** (listen) сеть, ожидая запросов от клиентов.

Клиент связывается с сервером и посылает ему **запрос** (request) с описанием услуги, например имя нужного файла.

Сервер обрабатывает запрос и отправляет **ответ** (response), в нашем примере — файл, или сообщение о невозможности оказать услугу.

После этого связь может быть разорвана или продолжится, организуя **сеанс связи**, называемый иногда **сессией** (session).

Запросы клиента и ответы сервера формируются по строгим правилам, совокупность которых образует **протокол** (protocol) связи. Все правила, образующие протокол, должны быть понятными, однозначными и короткими, чтобы не загружать сеть.

Все сетевые соединения базируются на трех основных понятиях: клиент, сервер и протокол.

Заметим, что клиент и сервер — понятия относительные. В одной сессии компьютер может быть сервером, а в другой - клиентом. Например, файл-сервер может послать принт-серверу файл на печать, становясь его клиентом.

Каждый компьютер или другое устройство, подключенное к сети называется **хост (host).**

Для обслуживания протокола: формирования запросов и ответов, проверок их соответствия протоколу, расшифровки сообщений, связи с сетевыми устройствами создается программа, состоящая из двух частей.

Одна часть программы работает на сервере, другая — на клиенте. Эти части так и называются — **серверной частью программы** и **клиентской частью программы**, или, короче, **сервером** и **клиентом**.

Обычно на одном компьютере-сервере работают несколько программ-серверов. Одна программа занимается приемом и отправкой электронной почты, другая – пересылкой файлов, третья предоставляет Web-страницы.

Для того чтобы различать программы- серверы, каждой программе-серверу назначается **номер порта** (port) – целое положительное число, которое указывает клиент, обращаясь к определенной программе-серверу. Число может быть любым, но наиболее распространенным протоколам даются стандартные номера, чтобы клиенты были твердо уверены, что обращаются к нужному серверу.

Например, стандартный номер порта электронной почты — 25, пересылки файлов — 21, Web-сервера — 80. Стандартные номера простираются до 1023. Числа, начиная с 1024 до 65535, можно использовать для своих собственных номеров портов.

В процессе передачи сообщения используется несколько протоколов. Протоколы работают друг с другом в **стеке –**  протокол работающий на уровне выше, работает «поверх» нижнего, т.е. каждый протокол только добавляет к сообщению свою информацию, не меняя его, ничего не зная о том, что сделано по предыдущему протоколу и что будет выполнено по правилам следующего протокола.

В современных глобальных сетях принят стек из четырех протоколов, называемый **стеком протоколов TCP/IP**.

Сначала мы пишем сообщение, пользуясь программой, реализующей **прикладной (application)** протокол: HTTP (80), SMTP (25), Telnet (23), FTP (21), POP3 (100) или другой протокол. В скобках записан стандартный номер порта.

Затем сообщение обрабатывается по **транспортному (transport)** протоколу. К нему добавляются, в частности, номера портов отправителя и получателя, контрольная сумма и длина сообщения. Наиболее распространены транспортные протоколы TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). В результате работы протокола TCP получается TCP-сегмент (segment), а протокола UDP — пакет (packet).

Протокол UDP работает как почтовая служба. Сообщение делится на прикладном уровне на части, из которых создаются отдельные пакеты, около килобайта каждый. Части сообщения нумеруются, как страницы в книге. Пакеты посылаются один за другим.

Связь по протоколу TCP больше напоминает телефонный разговор, чем почтовую связь. Сначала устанавливается соединение с получателем. Только после этого посылаются сегменты. Получение каждого сегмента подтверждается получателем, при ошибке посылка сегмента повторяется. Сообщение аккуратно собирается получателем. Для отправителя и получателя создается впечатление, что пересылаются не сегменты, а сплошной поток байтов, поэтому передачу сообщений по протоколу TCP часто называют передачей потоком. TCP обеспечивает надежную связь.

Далее сообщением занимается программа, реализующая **сетевой (network)** протокол. Чаще всего это протокол IP (Internet Protocol). Он добавляет к сообщению адрес отправителя и адрес получателя, и другие сведения. В результате получается IP-дейтаграмма.

Наконец, IP-дейтаграмма поступает к программе, работающей по **канальному (link)** протоколу ENET, SLIP, PPP (Network Access Leyer) и сообщение принимает вид, пригодный для передачи по сети (frame - кадр).

На стороне получателя сообщение проходит через эти четыре уровня протоколов в обратном порядке, освобождаясь от служебной информации, и доходит до программы, реализующей прикладной протокол.

Термины, применяемые для обозначения блока передаваемых данных, различны при использовании разных протоколов транспортного уровня.

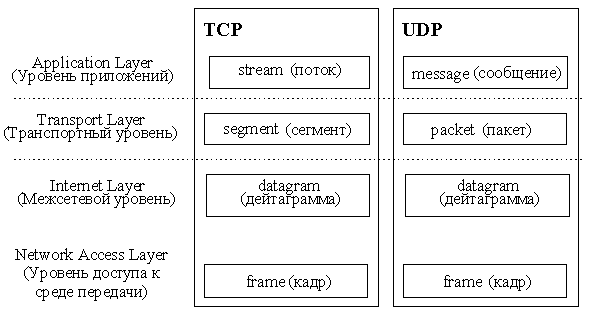


Рис.1 Названия блоков передаваемых данных.



Рис.2 Структура блоков передаваемых данных.

Какой адрес заносится в IP-заголовок? IP-адрес может быть представлен в двух форматах — IPv4 и IPv6.

Адрес формата IPv4 имеет длину 32 бита, и представляется в виде четырех десятичных чисел от 0 до 255, разделенных точками (192.168.0.1 или 92.123.155.81).

Адрес формата IPv6 имеет длину 128 бит, и представляется в виде восьми 16-ричных чисел от 0 до FFFF, разделенных двоеточиями (1080:0:0:0:8:800:200C:417A).

Пользователям удобнее иметь дело с именами хостов, представленных в алфавитно-цифровом виде.

Для преобразования цифровых адресов в алфавитно-цифровые имена используется служба имен DNS (Domain Name Service).

**ПАКЕТ java.net.**

Пакет java.net содержит классы, которые отвечают за различные аспекты сетевого взаимодействия.

Классы пакета можно поделить на 2 категории:

* + - * – низкоуровневый доступ (адреса, сокеты, интерфейсы)
      * – высокоуровневый доступ (URI, URL, соединения).

Классы, входящие в java.net, позволяют организовать передачу данных с помощью протоколов TCP, UDP, HTTP.

TCP и UDP являются протоколами транспортного уровня. Протоколы имеют следующие особенности.

Протокол TCP UDP

Установление соединения да нет

Подтверждение доставки

сообщений да нет

Протокол TCP имеет более высокую надежность доставки сообщений, а UDP – более высокую скорость передачи данных.

**java.net.InetAddress**

Для представления IP-адресов и доступа к DNS в Java используется класс InetAddress. Данный класс имеет два подкласса – Inet4Address и Inet6Address, но они используются редко, так как для большинства приложений хватает функциональности базового класса. Объект класса InetAddress содержит IP-адрес и имя хоста.

Экземпляр класса InetAddress можно получить с помощью статических методов класса:

* getLocalHost();
  + - -Возвращает локальный хост
* getByAddress(String host, byte[] addr);
  + - -Возвращает InetAddress с заданным IP-адресом и именем (корректность имени для данного адреса не проверяется)
* getByAddress(byte[] addr);
  + - -Возвращает InetAddress с заданным IP-адресом
* getByName(String host);
  + - -Возвращает InetAddress с заданным именем хоста (путем обращения к DNS)
* getAllByName(String host);
  + - -Возвращает массив IP-адресов хоста с заданным именем (путем обращения к DNS)

Основные методы класса InetAddress:

* **byte[] getAddress();**
  + - * + -Возвращает IP-адрес хоста
* **String getHostName();**
  + - * + -Возвращает имя хоста

**ПЕРЕДАЧА ПО ПРОТОКОЛУ TCP**

Для обеспечения передачи данных по протоколу TCP основным классом является java.net.Socket.

Конструктор класса Socket:

* Socket(InetAddress address, int port);
  + -создает соединение и подключает его к заданному порту по заданному IP-адресу

Методы класса Socket:

* InputStream getInputStream();

-возвращает входной поток данных

* OutputStream getOutputStream();

-возвращает выходной поток данных

* void setSoTimeout(int ms);

-устанавливает время ожидания завершения операции чтения извходного потока сокета в миллисекундах. По истечении данного времени выбрасывается исключение SocketTimeoutException.

* void close();

-закрывает сокет

Для реализации сервера, который ожидает запрос от клиента и отвечает на него, используется класс ServerSocket.

Конструктор класса ServerSocket:

* ServerSocket(int port);

-создает подключение на заданном порту

Методы класса ServerSocket:

* Socket accept();

-ожидает соединение и устанавливает его.

* void setSoTimeout(int ms);

-устанавливает время ожидания установления соединения в миллисекундах. По истечении данного времени выбрасывается исключение SocketTimeoutException

* void close();

-закрывает сокет

Последовательность создания TCP-соединения на стороне клиента:

1. Получение объекта InetAddress:

**InetAddress ia = InetAddress.getLocalHost()**

1. Создание сокета. При этом задается адрес (объект InetAddress) и порт, к которым будет устанавливаться соединение:

**Socket soc = new Socket(ia, 8888);**

1. Получение входного и выходного потоков сокета:

**InputStream is = soc.getInputStream();**

**OutputStream os = oc.getOutputStream();**

1. Чтение данных из входного и запись данных в выходной поток:

**is.read();**

**os.write();**

1. Закрытие потоков:

**is.close();**

**os.close();**

1. Закрытие сокета:

**soc.close();**

Последовательность создания TCP-соединения на стороне сервера:

1. Создание объекта ServerSocket, который будет принимать соединения на заданный порт:
   * + - 1. ServerSocket ss = new ServerSocket(8888);
2. Ожидание соединения от клиента и получение сокета для коммуникации с клиентом:
   * + - 1. Socket soc = ss.accept();
3. Получение входного и выходного потоков сокета:
   * + - 1. InputStream is = soc.getInputStream();
         2. OutputStream os = soc.getOutputStream();
4. Чтение данных из входного и запись данных в выходной поток:
   * + - 1. is.read();
         2. os.write();
5. Закрытие потоков:
   * + - 1. is.close();
         2. os.close();
6. Закрытие сокета:
   * + - 1. soc.close();

**РАБОТА С PROXY-СЕРВЕРОМ**

В настоящее время выход в Интернет часто осуществляется через proxy-сервер, кэширующий полученную информацию и дающий возможность нескольким машинам выходить в Интернет с одним и тем же IP-адресом.

Для того чтобы создавать сокеты, работающие через proxy-сервер, начиная с JDK 5.0 в класс Socket добавлен конструктор:

**Socket(Proxy proxy);**

Этот конструктор использует ссылку на объект абстрактного класса Proxy, устанавливающего связь с proxy-сервером. Объект создается конструктором:

**Proxy(Proxy.Type type, SocketAddress address);**

в котором указывается тип proxy-сервера — одна из констант:

* DIRECT — соединение без proxy-сервера;
* HTTP — соединение с proxy-сервером, обслуживающим протокол HTTP или FTP;
* SOCKS — соединение через proxy-сервер, работающий по протоколу SOCKS4 или SOCKS5.

Класс SocketAddress, содержащий адрес и порт сервера, — это абстрактный класс. На практике при создании адреса используется его потомок InetSocketAddress.

Соберем все это вместе. Обычный способ создания сокета, работающего через proxy-сервер, выглядит так:

**Socket sock = new Socket( new Proxy(Proxy.Type.SOCKS,**

**new InetSocketAddress("socks.domain.com", 1080)));**

**ПЕРЕДАЧА ПО ПРОТОКОЛУ UDP**

При работе с UDP используются следующие основные классы: java.net.DatagramPacket (представляющий передаваемое сообщение - датаграмму) и java.net.DatagramSocket.

Конструкторы класса DatagramPacket:

* + **DatagramPacket(byte[] buf,int length);**
    - - создает пакет из заданного массива байтов с заданной длиной
  + **DatagramPacket(byte[] buf, int length, InetAddress addr, int port);**
    - - создает пакет из заданного массива байтов с заданной длиной, для отправления на заданный адрес по заданному порту

Методы класса DatagramPacket:

* **InetAddress getAddress();**
  + - * -возвращает адрес
* **int getPort();**
  + - * -возвращает порт
* **byte[] getData();**
  + - * - возвращает массив данных
* **int getLength();**
  + - * - возвращает длину данных
      * Конструкторы класса DatagramSocket:
* **DatagramSocket();**
  + - * - создает сокет с использованием первого доступного локального порта
* **DatagramSocket(int port);**
  + - * - создает сокет с использованием заданного порта

Методы класса DatagramSocket:

* **void send(DatagramPacket p);**
  + отсылает заданный пакет
* **void receive(DatagramPacket p);**
  + принимает данные в заданный пакет
* **void setSoTimeout(int ms);**
  + - устанавливает время ожидания завершения операции приема пакета в миллисекундах. По истечении данного времени создается исключение SocketTimeoutException.
* **void close();**
  + - закрывает сокет

Последовательность создания UDP-сокета на стороне сервера:

1. Создание объекта DatagramSocket, который будет принимать соединения на заданный порт и буферов для ввода и вывода:

**DatagramSocket ds = new DatagramSocket(7777);**

**byte[] ib = new byte[256];**

**byte[] ob = new byte[256];**

1. Получение пакета от клиента:

**DatagramPacket ip = new DatagramPacket(ib, ib.length);**

**ds.receive(ip);**

1. Формирование ответа клиенту в виде массива байтов ob
2. Формирование пакета и отсылка его клиенту (адрес и порт клиента получаются из полученного от клиента пакета):

**InetAddress addr = ip.getAddress();**

**int port = ip.getPort();**

**DatagramPacket op = new DatagramPacket(ob, ob.length, addr, port);**

**ds.send(dp);**

1. Закрытие сокета:

**ds.close();**

Последовательность создания UDP-сокета на стороне клиента:

1. Создание объекта DatagramSocket (без указания порта) и буферов для ввода и вывода:

**DatagramSocket ds = new DatagramSocket();**

**byte[] ib = new byte[256];**

**byte[] ob = new byte[256];**

1. Формирование запроса серверу в виде массива байтов ob:
2. Формирование пакета и отсылка его серверу:

**DatagramPacket op = new DatagramPacket( ob, ob.length, InetAddress.getByName(server\_name), 7777);**

**ds.send(dp);**

1. Получение ответного пакета от сервера:

**DatagramPacket ip = new DatagramPacket(ib,ib.length);**

**ds.receive(ip);**

1. Закрытие сокета:

**ds.close();**

**URL-СОЕДИНЕНИЯ**

URI (Unifrom Resource Identifier) – унифицированный идентификатор ресурса. Представляет собой символьную строку идентифицирующую какой-либо ресурс (обычно ресурс Интернет).

URL (Uniform Resource Locator) – унифицированный локатор ресурса. Представляет собой подкласс URI, который кроме идентификации дает информацию о местонахождении ресурса.

Формат URL в общем виде (элементы в квадратных скобках могут отсутствовать):

**[протокол:][//[логин[:пароль]@]хост[:порт]][путь] [?запрос][#фрагмент]**

Пакет java.net содержит классы URI и URL для представления URI и URL соответственно, класс URLConnection, использующийся для создания соединения с заданным ресурсом, и его подкласс HttpURLConnection, реализующий соединение с ресурсом по протоколу HTTP.

Последовательность работы с URL-соединениями:

1. Создать URI:

**URI uri = new URI(** [**http://joe:12345@mail.ru:8080/index.php**](http://joe:12345@mail.ru:8080/index.php)

**+ "?getMail=1&page=2#end");**

**URI uri2 = new URI("http", ”joe:12345”, ”mail.ru”,**

**8080, ”index.php”,”getMail=1&page=2”,”end”);**

1. Преобразовать URI в URL:

**URL url = uri.toURL();**

1. Открыть URL-соединение и поток данных:

**URLConnection uc = url.openConnection();**

**uc.connect();**

**InputStream is = uc.getInputStream();**

или открыть поток данных:

**InputStream is = url.openStream();**

1. Получить данные;
2. Закрыть поток и соединение:

**is.close();**

**uc.close();**

Использование URLConnection по сравнению с простым открытием потока из URL позволяет дополнительно устанавливать параметры соединения, такие как возможность взаимодействия с пользователем, разрешение записи и чтения, а также получать информацию о соединении, такую, как даты создания и модификации, тип, длину и кодировку содержимого.

Пример:

**import java.net.\*;**

**import java.io.\*;**

**public class SimpleURL{**

**public static void main(String[] args) {**

**try{**

**URL tut = new URL("http://www.tut.by/");**

**BufferedReader br = new**

**BufferedReader( new**

**InputStreamReader(**

**tut.openStream()));**

**String line = null;**

**while ((line = br.readLine()) != null)**

**System.*out*.println(line);**

**br.close();**

**} catch( MalformedURLException me ){**

**System.*err*.println(**

**"Unknown host: " + me);**

**System.*exit*(0);**

**} catch(IOException ioe) {**

**System.*err*.println(**

**"Input error: " + ioe);**

**}**

**}**

**}**

**МНОГОПОТОЧНОСТЬ**

Сервер должен поддерживать многопоточность для обработки нескольких соединений одновременно. Обычно сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток.

Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции.

В следующем примере создается класс ServerThread, расширяющий класс Thread, и используется затем для соединений с многими клиентами, каждый в своем потоке.

Многопоточность-пример, сервер

**import java.io.\*;**

**import java.net.\*;**

**public class NetServerThread {**

**public static void main(String[] args) {**

**try {**

**ServerSocket serv = new ServerSocket(8071);**

**System.*out*.println("initialized");**

**while (true) {**

**//ожидание клиента**

**Socket sock = serv.accept();**

**System.*out*.println(**

**sock.getInetAddress().getHostName()**

**+ " connected");**

**/\* создание отдельного потока для обмена данными**

**с соединившимся клиентом\*/**

**ServerThread server = new ServerThread(sock);**

**server.start();//запуск потока**

**}**

**} catch (IOException e) {**

**System.*err*.println(e);**

**}**

**}**

**}**

**class ServerThread extends Thread {**

**private PrintStream os;//передача**

**private BufferedReader is;//чтение**

**private InetAddress addr;//адрес клиента**

**public ServerThread(Socket s) throws IOException {**

**os = new PrintStream(s.getOutputStream());**

**is = new BufferedReader( new InputStreamReader(**

**s.getInputStream()));**

**addr = s.getInetAddress();**

**}**

**public void run() {**

**int i = 0;**

**String str;**

**try {**

**while ((str = is.readLine()) != null) {**

**if ("PING".equals(str))**

**os.println("PONG "+ ++i);**

**System.*out*.println("PING-PONG" + i + " with " +**

**addr.getHostName());**

**}**

**} catch (IOException e) {**

**//если клиент не отвечает, соединение с ним разрывается**

**System.*out*.println("Disconnect");**

**} finally {**

**disconnect();//уничтожение потока**

**}**

**}**

**public void disconnect() {**

**try {**

**System.*out*.println(addr.getHostName() +**

**" disconnected");**

**os.close();**

**is.close();**

**} catch (IOException e) {**

**e.printStackTrace();**

**} finally {**

**this.interrupt();**

**}**

**}**

**}**

**Многопоточность-пример, клиент**

**import java.io.\*;**

**import java.net.\*;**

**public class NetClientThread {**

**public static void main(String[] args) {**

**try {**

**// установка соединения с сервером**

**Socket s = new Socket( InetAddress.*getLocalHost*(), 8071 );**

**//или Socket s = new Socket("ИМЯ\_СЕРВЕРА", 8071);**

**PrintStream ps = new PrintStream(s.getOutputStream());**

**BufferedReader br = new BufferedReader(**

**new InputStreamReader( s.getInputStream() ));**

**//**

**for (int i = 1; i <= 10; i++) {**

**ps.println("PING");**

**System.*out*.println(br.readLine());**

**Thread.*sleep*(1000);**

**}**

**s.close();**

**} catch (UnknownHostException e) {**

**// если не удалось соединиться с сервером**

**System.*out*.println("адрес недоступен");**

**e.printStackTrace();**

**} catch (IOException e) {**

**System.*out*.println("ошибка I/О потока");**

**e.printStackTrace();**

**} catch (InterruptedException e) {**

**System.*out*.println( "ошибка потока выполнения");**

**e.printStackTrace();**

**}**

**}}**

**XML и Java**

**ВВЕДЕНИЕ В XML**

XML (Extensible Markup Language – расширяемый язык разметки) – свод общих синтаксических правил для описания с помощью тегов текстового представления данных, структурированных в виде дерева.

XML рекомендован к использованию W3C (разработчик стандартов интернета).

Предназначен для представления структурированной информации и обмена этой информацией между приложениями.

XML – это язык метаразметки. Это значит, что он не имеет фиксированного набора тегов и элементов для работы с любыми сферами интересов. XML позволяет разработчикам и создателям документов определять нужные им элементы по мере необходимости.

Он устанавливает для XML документов грамматику, которая регулирует, где могут встречаться теги и как они располагаются, какие имена элементов являются допустимыми, как атрибуты присоединяются к элементам и т. д.

Эта грамматика достаточно конкретна для разработки XML анализаторов, которые могут читать и понимать любой XML документ.

Документы, удовлетворяющие этой грамматике, называются **корректными** (well-formed). Документ, не являющийся корректным, не допустим.

Разработчики могут договариваться об употреблении лишь определенных тегов. Эти наборы тегов называются **XML-приложениями**.

Разметка в XML документе описывает его структуру. Она позволяет видеть, как элементы связаны друг с другом. В хорошо составленном XML документе разметка также описывает и его семантику.

Разметка, разрешенная для конкретного XML-приложения, как правило, документирована в определении **типа документа** (**DTD**, Document Type Definition).

В DTD перечислены допустимые теги разметки и указано, каким образом разметка включается в документ. Каждый конкретный случай сверяется с DTD.

Документ, который соответствует DTD, называется **действительным** (**valid**). Если он не соответствует DTD, то является **недействительным** (**invalid**).

Действительность зависит от DTD: документ будет действительным или недействительным в зависимости от того, с каким DTD выполняется сверка.

DTD в XML необязательны. Не все документы должны быть действительными. Для многих целей достаточно корректности документа.

Пример: с помощью тегов и атрибутов описывается цвет, размер, штрих-код, производитель и наименование продукта.

**<?xml version="1.0"?>**

**<product barcode="2394287410">**

**<manufacturer>Verbatim</manufacturer> <name>DataLife MF 2HD</name> <quantity>10</quantity>**

**<size>3.5"</size>**

**<color>черный</color>**

**<description>дискеты</description>**

**</product>**

Программы, которые пытаются понять содержание XML документа используют для чтения документа **XML-анализатор** (**XML-parser**).

Анализатор отвечает за выделение из документа отдельных элементов, атрибутов и других его частей. Если в каком-либо месте анализатор обнаруживает нарушение правил XML, он сообщает приложению об ошибке и прекращает анализ.

Отдельные XML приложения четко предписывают, где и какие элементы и атрибуты допускаются. Правила задаются с помощью DTD.

Документ содержит либо само DTD, либо указатель на URI, где находится DTD. Некоторые XML-анализаторы сверяют документ с DTD в процессе его чтения на предмет соответствия его указанным ограничениям. Такой анализатор называется **проверяющим анализатором** (**validating parser**).

Нарушение этих ограничений – это **нарушение действительности** (**validity error**), а весь процесс сверки документа с DTD называется **проверкой действительности** (**validation**).

**ОСНОВЫ XML**

XML документ всегда содержит текст, а не двоичные данные. Его может открыть любая программа, которая читает текстовые файлы.

Простой XML документ:

**<person>**

**Алан Тьюринг**

**</person>**

Обычно такой документ помещают в файл с именем, например, person.xml. Но, надо иметь ввиду, что XML анализатор не предъявляет каких-либо требований к именам файлов, файл может называться, например, person.txt или как-либо ещё.

Документ в примере состоит из одного элемента, который имеет **тип** person. Этот элемент ограничен **начальным тегом** <person> и **конечным тегом** </person>. Все, что находится между начальным и конечным тегами, называется **содержимым элемента**. Содержимым данного элемента является следующая текстовая строка:

**Алан Тьюринг**

Пробельные символы являются частью содержимого элемента, хотя многие приложения могут их игнорировать.

Теги **<person>** и **</person>** называются **разметкой**.

Строка Алан Тьюринг и окружающие ее пробельные символы представляют собой **символьные данные**.

Начальные теги начинаются с символа <, а конечные теги – с </. Далее следует имя элемента, и заканчиваются они символом >. В отличие от тегов HTML, по мере надобности можно создавать новые XML теги. (Имена тегов отражают тип содержимого внутри элемента, а не то, как это содержимое следует форматировать.)

Перекрывающиеся теги в XML запрещены.

Существует особый синтаксис для **пустых элементов** – элементов без содержимого. Эти элементы обозначаются тегами, начинающимися с < и заканчивающимися />.

Например, пустой тег person записывается так:

**<person/>**

что эквивалентно записи:

**<person></person>**

XML, в отличие от HTML, чувствителен к регистру символов.

Рассмотрим более сложный XML-документ:

**<person>**

**<name>**

**<first\_name>Алан</first\_name>**

**<last\_name>Тьюринг</last\_name>**

**</name>**

**<profession>специалист по информатике</profession>**

**<profession>математик</profession>**

**<profession>криптограф</profession>**

**</person>**

Этот XML-документ все еще состоит из одного элемента person. Однако теперь этот элемент не просто содержит символьные данные. Он включает четыре дочерних элемента: элемент name и три элемента profession. Элемент name содержит два собственных дочерних элемента: first\_name и last\_name.

Элемент person называется родительским элементом для элемента name и трех элементов profession. Элемент name является **родительским** по отношению к элементам first\_name и last\_name.

Элементы name и три элемента profession иногда называются **одноуровневыми** (**sibling**) по отношению друг к другу. Элементы first\_name и last\_name также являются одноуровневыми.

В каждом XML-документе есть один элемент без родителя. Это первый элемент документа, содержащий все остальные элементы.

В приведенных примерах эту роль выполняет элемент person, и он называется **корневым элементом документа**, а иногда просто **элементом документа**.

**Аттрибуты.** Элементы в XML могут иметь атрибуты. Атрибут – это пара имя-значение, присоединяемая к начальному тегу элемента. Имена отделяются от значений знаком равенства и, по желанию, пробельными символами. Значения заключаются в одинарные или двойные кавычки.

Например, добавим к тегу person аттрибуты born и died:

**<person born="1912/06/23" died="1954/06/07">**

**Алан Тьюринг**

**</person>**

Пример: XML-документ, описывающий личность с помощью атрибутов:

**<person>**

**<name first="Алан" last="Тьюринг"/>**

**<profession value="специалист по информатике"/>**

**<profession value="математик"/>**

**<profession value="криптограф"/>**

**</person>**

Подобный пример поднимает вопрос о том, в каком случае для хранения информации следует использовать дочерние элементы, а когда – атрибуты.

Необходимо учитывать, что каждый элемент может иметь не более одного атрибута с данным именем. Кроме того, атрибуты очень ограничены по своей структуре. Значение атрибута – это лишь текстовая строка. Следовательно, структура, основанная на элементах, является значительно более гибкой и расширяемой. При этом атрибуты, конечно, удобнее для некоторых приложений.

Если вы разрабатываете собственный словарь XML, то сами решаете, когда использовать элементы, а когда – атрибуты.

Имена элементов и другие XML-имена могут содержать любые алфавитно-цифровые символы. Эти символы включают стандартные латинские буквы от A до Z и от a до z, а также цифры от 0 до 9. XML-имена могут также содержать нелатинские буквы, числа и идеограммы, например ö, ç, ϑ и ψ. В них также могут присутствовать три символа пунктуации: \_ подчеркивание, - дефис и . точка.

Другие символы пунктуации, такие как кавычки, апострофы, символ доллара, знак вставки, символ процента и точка с запятой, XML-имена содержать не могут. Двоеточие разрешается, однако его использование зарезервировано для пространств имен, что обсуждается далее.

XML-имена не могут содержать также пробельные символы любого вида: пробелы, переводы строк, возвраты каретки и табуляции.

XML-имена начинаются только с букв, идеограмм и символа подчеркивания. Они не могут начинаться с цифр, дефиса или точки. Ограничение на длину имени элемента и других XML-имен отсутствует.

Символьные данные внутри элемента не могут содержать символы <>”’&. Если нужно использовать эти символы в тексте, используются ссылки (указатели) на символы, которые должны быть заключены между символами & и ; .

Особо распространенными указателями являются:

&lt; – символ <;

&gt; – символ >;

&amp; – символ &;

&apos; – символ апострофа ‘;

&quot; – символ двойной кавычки “.

Если в XML-документе имеются фрагменты исходного текста на XML или HTML, символы < и & в этих примерах должны быть заменены на &lt; и &amp;. Чем больше разделов символьного кода содержит документ и чем они длиннее, тем более утомительным может стать такое кодирование.

Чтобы облегчить эту работу, можно включить каждый случай такого кода в секцию CDATA. Секция CDATA отмечается с помощью <![CDATA[ и ]]>. Все, что находится между <![CDATA[ и ]]>, рассматривается только как символьные данные. Единственное, что не может присутствовать в секции CDATA, – это ее завершающий ограничитель]]>.

XML-документы могут включать комментарии. Комментарии в XML синтаксически аналогичны комментариям в HTML – они начинаются с <!-- и заканчиваются первым вхождением -->. Например:

<!–- Это комментарий -->

Запрещается закрытие комментария с тройным дефисом (--->).

Комментарии могут присутствовать в любом месте в символьных данных документа. Они также могут помещаться перед корневым элементом или после него. Но, комментарии не могут находиться внутри тега или другого комментария.

XML-анализаторы могут как передавать, так и не передавать информацию из комментариев. Не создавайте документы или приложения, зависящие от наличия комментариев.

В HTML комментарии иногда используют не по назначению – для поддержки нестандартных расширений. Например, содержимое элемента style иногда заключается в комментарий, чтобы избежать его отображения браузером, не поддерживающим сценарии.

В XML существуют инструкции обработки – альтернативное средство передачи информации конкретным приложениям, имеющим дело с документом.

Инструкция обработки начинается с <? и заканчивается ?>. Сразу за символами <? следует XML-имя, называющееся **целевым объектом** (**target**), которое может быть именем приложения, для которого предназначена эта инструкция, либо просто идентификатором данной инструкции обработки. Оставшаяся часть инструкции содержит текст в формате того приложения, которому предназначена инструкция.

Самая распространенная инструкция обработки, xml-stylesheet, присоединяет к документу таблицу стилей. Она всегда находится перед корневым элементом.

Пример: XML-документ с инструкцией xml-stylesheet:

**<?xml-stylesheet href="person.css" type="text/css"?>**

**<person>**

**Алан Тьюринг**

**</person>**

Инструкции обработки с названием xml, независимо от регистра букв (то есть xml, XML, Xml и т. д.), зарезервированы для нужд W3C. Во всех остальных случаях вы вольны назначить инструкции любое допустимое XML-имя.

XML-документ может, но не обязан, начинаться с XML-объявления. Оно выглядит как инструкция обработки с именем xml и атрибутами version, standalone и encoding. Но технически это не инструкция обработки, а лишь XML-объявление, не более того.

Пример:

**<?xml version="1.0" encoding="US-ASCII" standalone="yes"?>**

**<person>**

**Алан Тьюринг**

**</person>**

Наличие XML-объявления в XML-документах не обязательно. Однако, если XML-объявление все же присутствует, то документ должен начинаться именно с него. Перед XML-объявлением не может быть комментариев, пробельных символов или инструкций обработки.

Атрибут encoding задаёт кодировку текста документа. Атрибут encoding в XML-объявлении не обязателен. По умолчанию для XML-документов предполагается кодировка переменной длины UTF-8 c набором символов Unicode. Эта кодировка является строгим надмножеством ASCII, поэтому текстовые файлы в формате ASCII также являются документами UTF-8.

Если атрибут standalone имеет значение no, то приложению может потребоваться прочитать внешний DTD (то есть DTD из другого файла) для определения правильных значений частей документа. Например, DTD может содержать значения по умолчанию для атрибутов, которые необходимы анализатору для передачи приложению, но реально отсутствуют в самом документе.

Документы, не имеющие DTD обычно используют значение yes для атрибута standalone. Документы, имеющие DTD, могут также использовать это значение атрибута, если DTD не меняет их содержимого или если оно целиком содержится внутри документа.

Атрибут standalone не является обязательным для XML-объявления. Если он опущен, предполагается значение no.